Ownership Object Graphs with Dataflow Edges

Radu Vanciu Marwan Abi-Antoun
Department of Computer Science, Wayne State University, Detroit, Michigan, USA
Email: {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. Architectural risk analysis involves thinking about worst-case scenarios, and thus requires a sound object graph, which shows all possible objects and dataflow communication between them. Extracting a sound object graph that conveys architectural abstraction is challenging. One solution is to apply a hierarchy to the object graph to convey both high-level understanding and detail. Achieving soundness requires a static analysis, but architectural hierarchy is not available in general purpose programming languages.**

To achieve hierarchy, we annotate the program with ownership types and use abstract interpretation 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, prove its soundness, then show that the extracted edges are similar to those drawn by a security expert.

***Keywords*-architectural risk analysis; hierarchical object graph; sound static analysis; dataflow communication**

I. Introduction

For over a decade, architectural risk analysis, also known as threat modeling [1], has been one of the three pillars of building secure systems, together with code review and penetration testing [2, Chap. 5]. 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 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 prerequisite of architectural risk analysis is a documented architecture. Unfortunately, for many systems the architecture is missing, or when it is documented, it may be inconsistent with the code. Reasoning about security requires an architecture that focuses on the runtime structure, rather than a code structure that shows packages and classes.

A runtime architecture shows runtime components and connectors, uses hierarchical decomposition, and partitions a system into tiers [3]. The tools for extracting runtime architectures are immature [4], [5] compared to tools for code architecture [6]. While both architectural views are useful, architects require a runtime architecture to reason about quality attributes, such as security. Indeed, architectural risk analysis uses a particular style of runtime architecture, a Dataflow Diagram (DFD), which shows data sources, trust boundaries and processes and the type of communicated data [7]. A DFD allows architects to recognize unencrypted data communicated to an external interactor, which may lead to information disclosure. Architects may also focus on a dataflow edge that crosses a boundary from an untrusted to a trusted component, to ensure that data is validated, since an attacker might tamper with it.

Moreover, a security analysis must consider the worst—rather than the typical—case of possible component communication. The analysis results are valid only if they are based on a sound architecture that reveals all objects and relations that may exist at runtime—in any program run. A sound architecture shows a unique representative for every runtime object, and if a runtime relation exists between two runtime objects, the sound architecture must show an edge between their corresponding representatives. Achieving soundness requires a static analysis to extract the architecture. In contrast, dynamic analysis considers a finite number of executions [8], and may miss important objects or relations that arise only in uncovered executions.

In previous work [9], we showed how to extract a hierarchical object graph from object-oriented code with ownership annotations. Ownership annotations encode information about logical containment and strict encapsulation. A typechecker guarantees that annotations are consistent with the code and with each other. A sound, static analysis abstractly interprets the program with annotations 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 can be then abstracted into a standard runtime architecture. We specialized OOG to security architectures showing how an OOG maps to a DFD [10]. The OOG showed points-to edges, but during architectural risk analysis, architects require dataflow communication edges on the OOG.

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

- Formalizing the analysis and proving its soundness;
- Evaluating our analysis by comparing an OOG with dataflow edges to a DFD drawn by a security expert, and to an OOG with points-to edges.

In this paper, Section II defines dataflow communication and reviews ownership domains. Section III describes the challenges of statically extracting an object graph. Section IV formalizes our analysis. Section V evaluates the analysis on an extended example. Section VI discusses related work. Section VII concludes.

II. Background

We illustrate the kind of dataflow that might be extracted using a small system that follows a Document-View architecture [11], to which we refer to as Listeners. Since the analysis requires ownership domain annotations to extract the OOG, we review ownership domains using Listeners.

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 has fields of type `Model`, `PieChart`, and `BarChart`. `Model` extends `Listener` and contains the information displayed in the charts. `BarChart` and `PieChart` extend `BaseChart`, which subsequently extends `Listener`. `BaseChart` and `Model` have a field of type `List` with elements of type `Listener`.

As a 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". The program instantiates `Main`, which in turn instantiates `Model`, `BarChart`, and `PieChart`. A `model` object has a list of `BarChart` and `PieChart` objects, while `barChart` and pieChart have their own list of `Model` objects. These lists allow `model` to exchange with charts messages of type `MsgMtoV` and `MsgVtoM`.

A. Ownership hierarchy

An *ownership domain* is a named conceptual group of objects. 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 they are treated like fields in the sense that fresh domains are created for each instance of that class. As a notation, to distinguish between domains with the same name `D`, we refer to a domain as `o.D`, where `o` is the parent object. 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`.

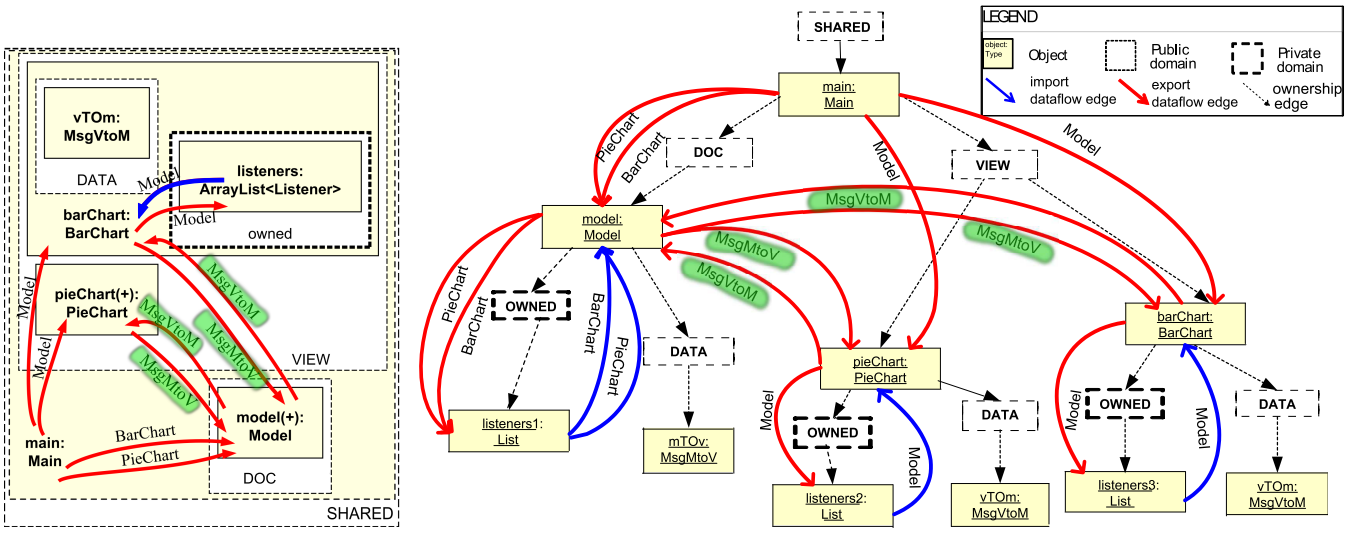
Domains define two kinds of object hierarchy. A public domain provides *logical containment* and makes an object conceptually *part of* another. Having access to an object gives the ability to access objects inside all its public domains. A private domain provides *strict encapsulation* and makes an 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 private field. For example, the class `Model` declares the `OWNED` private domain. Since the collection `listeners` is inside `OWNED`, a public method cannot return an alias

```
class Main<OWNER> {
  domain DOC, VIEW;
  BarChart<VIEW, DOC> barChart = new BarChart();
  PieChart<VIEW, DOC> pieChart = new PieChart();
  Model<DOC, VIEW> model = new Model();
  void run(){
   model.addListener(barChart);
   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> {
  List<OWNED, Listener<M>> listeners = new List();
}
class Model<OWNER, V> extends Listener<OWNER> {
  domain OWNED; //domain declaration
  List<OWNED, Listener<V>> listeners = new List();
  Msg<DATA> mTOv = new MsgMtoV();
  public Msg<DATA> getMsg(){
    return mTOv; //OK; DATA is a public domain
  }
  void addListener(Listener<V> l) {
   listeners.value = l;
  }
  void notifyObservers() {
   Listener<V> l = listeners.value;
   l.update(mTOv);
  }
}
class Listener<OWNER> {
  public domain DATA; //public domain
}
class List<OWNER, T<ELTS>> {//generic type T
  T<ELTS> value; // ELTS is a domain parameter
}
```

Figure 2. Listeners code fragments. The complete code is in [12].

to `listeners`. On the other hand, since messages should be available to other objects, the `mTOv:MsgMtoV` object is declared in `DATA`, a public domain inherited from `Listener`.

Ownership Domains are flexible. Domain parameters propagate ownership information of objects outside the current encapsulation, and allow objects to share state with each other. To bind the formal domain parameters of a class, the client code supplies actual domains when it instantiates the class. Each class declares a list of domain parameters, where the first one, `OWNER`, by convention binds to the owner domain. For example, `BarChart`, and `Model` declares a second domain parameter `M`, and `V`, respectively. `Main` instantiates `Model` and binds the actual domains `DOC`, and `VIEW` to the formal domain parameters `OWNER`, and V, respectively. `Model` instantiates `List` and binds the actual domains `OWNED` and `VIEW` to the formal domain parameters `OWNER` and `ELTS`. Similarly, when `Main` instantiates `BarChart`, and in turn `BaseChart` instantiates `List`, the actual domains that the formal domain parameters `OWNER`, and `ELTS` of `List` binds to are `barChart.OWNED`, and `main.DOC`, respectively. Using domain parameters, the declaration of `List` is flexible



(a) Collapsing Listeners OOG

(b) Internal representation of the Listeners OOG.

Figure 1. Listeners OOG. Interesting edges are highlighted.

and allows different instances of `List` to have elements in different domains as the client code supplies.

Finally, special annotations add expressiveness to the type system. For an object that is marked as `SHARED`, little reasoning can be done about it. For example, since the root class `Main` cannot take domain parameters, `main:Main` is declared in `SHARED`.

For readability, we use the language extensions of Ownership Domains [13]. Our analysis uses available language support for annotations, which leads to verbose code constructs, but enables reverse engineering of OOGs from legacy code. 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 [14, Appendix A].

B. Dataflow Communication

We adopt the same definition of dataflow communication as Spiegel [15], but we use abstract interpretation to add those dataflow edges on a hierarchical object graph.

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, field writes, field reads, or method invocations lead to dataflow communication. Since the flow can be bidirectional, we distinguish between *import* and *export* dataflow.

Import dataflow communication: *An import dataflow communication exists from the source object `b:B` to the destination object `a: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 object `a:A` to the destination object `b: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 field write `b.f = c` or `b.mb(c)`, where `c` is a field of `A`, an argument of `ma`, an object instantiated by `ma`, or an object returned by another method invoked by `ma`. In this paper, when the kind of the edge is clear, we refer to import and export dataflow communication edges simply as import and export edges.

In Listeners, `model.addListener(barChart)` leads to an export edge from `main` to `model`. The edge is labeled with the most precise type of the argument, in this case `BarChart`. The method invocation `l.update(mTOv)` in `Model` leads to export edges from `model` to `barChart` and to `pieChart`. In the code, the type of the receiver `l` is `Listener`. Instead of considering the receiver all the instances of classes that extend `Listener`, Ownership Domains provide more precise aliasing information. Since `V` maps to `VIEW` in the context of `model`, `l` aliases only objects in `VIEW`. Using the same information, the field write `listeners.value = l` leads to export edges from `model` to `listeners1`, and the field read `listeners.value` leads to import edges from `listeners1` to `model`, where the edge labels are `BarChart` and `PieChart`.

Discussion. The OOG shows architecturally significant objects such as `barChart`, `pieChart`, and `model` at higher levels, and collections and messages at lower levels (Fig. 1(b)). The OOG makes visually obvious the bidirectional communication from `model` to `barChart` and `pieChart`, where the messages exchanged are of type `MsgMtoV`, and `MsgVtoM`, (highlighted in Fig. 1(a)). An architect can learn from the above edges how data is transferred across tiers. The OOG with dataflow edges shows transient, usage edges. On the other hand, an OOG with points-to edges [9] shows persistent references between objects, and would not show the above communication, since `model` has no field of type `BarChart` or `PieChart`.

Collapsed OOG. 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 than an object has a collapsed substructure. While collapsing, the visualization also lifts any edge that might be incoming to or outgoing from a child object, i.e., the edge is recursively lifted to the nearest visible ancestor of the child object. The nested box visualization can show only the objects at the top level, which makes the graph less cluttered. We deal with the underlying representation of the OOG, the OGraph (Section IV), because this paper focuses on extracting additional edges. Lifting edges, collapsing objects are separate concerns discussed elsewhere [14, Sec. 2.4.3]

III. 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 all possible ROGs, for any program execution. We represent the extracted object graph as an OGraph, where nodes are OObjects and edges are OEdges (Fig. 4). 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 is an approximation because multiple runtime edges could have the same representative, or an OGraph might have some representatives that do not come from any real runtime edge. The OGraph is unsound if one runtime object or one runtime edge would have two representatives in the OGraph.

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 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. In particular, object soundness handles aliasing by enforcing the unique representatives invariant. For two variables that may alias and refer to the same runtime object, the analysis must create a single OObject.

Edge soundness. If we were able to observe the ROGs for all possible executions, and there is a runtime dataflow edge 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 show architecturally significant OObjects near the top of the hierarchy and OObjects representing implementation details further down.

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

Support for legacy code. To reverse engineer an OGraph from legacy code, an analysis should not require writing the program in a specific language, such as ArchJava [16] or reengineering legacy code, which is hard [17]. Instead, the analysis should take advantage of available language support for annotations.

IV. Formalization of the Analysis

We present the abstract syntax, the data type, and informally describe the most interesting parts of the analysis. We also describe key differences with our earlier work [9], [10].

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 [13]. 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 discussion. 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 $S[\ell]$; $S[\ell \mapsto C{<}\overline{\ell'.d}{>}(\overline{v})]$ denotes adding an entry for location ℓ to S; α and β range over formal domain parameters; m ranges over method names;

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

Figure 3. Simplified FDJ abstract syntax [13].

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 (OObjects) and domains (ODomains), and edges (OEdges) representing dataflow communication (Fig. 4). The OGraph is a triplet $G = \langle DO, DD, DE \rangle$, where DO is a set of OObjects, DD maps a pair $(O, C{::}d)$ to an ODomain D, and DE is a set of dataflow edges. Each E in DE is a directed edge from a source O_{src} to a destination O_{dst}. The label C is the class of the communicated object. The flag states whether the OEdge represents an import or an export dataflow communication. Multiple edges with different labels might exists 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 the program. 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 $C\texttt{<}\overline{D}\texttt{>}$. We follow the FDJ convention and consider an OObject's owning ODomain as the first element D_1 of $\overline{D}$. Our analysis relies on the precision about aliasing that Ownership Domains offer, and avoids merging object excessively. The Ownership Domains type system guarantees that two objects in different domains cannot alias. Our analysis 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 $C\texttt{<}\overline{D}\texttt{>}$ analyzed, to avoid non-termination of the analysis.

In addition to the OEdges that have source and destination OObjects, the OGraph has ownership edges. The OGraph representation is well-formed with respect to the ownership relations declared in the code using the annotations. The

$$
\begin{array}{lll}
G \in \mathsf{OGraph} & ::= \langle\ \textbf{Objects} = DO, & \\
 & \quad \textbf{DomainMap} = DD,\ \textbf{Edges} = DE\ \rangle & \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle & \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Type} = C\texttt{<}\overline{D}\texttt{>}\ \rangle & \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst}, & \\
 & \quad \textbf{Label} = C, \textbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle & \\
DD & ::= \emptyset\ \mid\ DD \cup \{\ (O, C{::}d) \mapsto D\ \} & \\
DO & ::= \emptyset\ \mid\ DO \cup \{\ O\ \} & \\
DE & ::= \emptyset\ \mid\ DE \cup \{\ E\ \} & \\
\Upsilon & ::= \emptyset\ \mid\ \Upsilon \cup \{\ C\texttt{<}\overline{D}\texttt{>}\ \} &
\end{array}
$$

Figure 4. Data type declarations for the OGraph.

data type declarations of the OGraph captures this hierarchy using the DD map without defining directly a set of ownership edges. An ownership edge states that an OObject $O = \langle C\texttt{<}\overline{D}\texttt{>} \rangle$ is a child of D_1, or that O owns a domain D. Given a mapping $\{(O, C'{::}d) \mapsto D\}$ in DD, D is a child of O. Since domains are inherited across classes [13], the class C of O can be a subclass of C' where d is declared.

Although a domain d is declared by a class C, each runtime instance of type C gets its own runtime domain $\ell.d$. For example, if there are two distinct object locations ℓ and ℓ' of class C, then $\ell.d$ and $\ell'.d$ are distinct. 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 and the fresh identifier D_{id} ensures that multiple ODomains can be created for the same domain declaration $C{::}d$.

During initialization, the analysis creates a global ODomain $D_{\texttt{SHARED}}$, the root of the OGraph. A developer picks a root class, C_{root}, and the analysis creates O_{root} in $D_{\texttt{SHARED}}$. The analysis also requires an initial context. We use a dummy OObject O_{world}, which does not correspond to an actual runtime object. Next, the analysis changes the context from O_{world} to O_{root}, and continues recursively with all the expressions in the methods of C_{root}.

Starting from the root class, the analysis uses abstract interpretation, and binds formal domain parameters to actual domains. For each kind of FDJ expression, we describe the analysis steps using transfer functions. Each transfer function $\langle [\![e]\!]\ O\ G \rangle$ takes an expression e, the context OObject O, and the OGraph G, and transforms G into G'. The analysis keeps track of the context O, which may change when the analysis encounters a `new` $C\texttt{<}\overline{p}\texttt{>}(\ldots)$ expression and pushes all expressions in the methods of C on stack.

To determine the instances of a type, the analysis uses the auxiliary $lookup$ function (Fig. 5). The analysis provides a context O, a type $C'\texttt{<}\overline{p}\texttt{>}$, and the maps DO and DD. Then, $lookup$ returns the set of the OObjects O_k in DO such that the class of O_k is C' or one of its subclasses. Also, each domain D_i of O_k is the domain that the pair (O, p_i) maps to in DD. The later condition increases the precision of our

$[\![e_0.m(\overline{e})]\!]\ O\ G$	$G' : \langle DO', DD', DE' \rangle \quad DO' = DO \quad DD' = DD$	
//method invocation	$e_0 : C\langle\overline{p}\rangle \quad mtype(m, C\langle\overline{p}\rangle) = \overline{T} \to T_R$	//receiver's type and meth. decl.
	$\{O_i \in DO \mid O_i = \langle C_i\langle\overline{D}\rangle\rangle, i = 1..sz\} = lookup(O, C\langle\overline{p}\rangle)$	//find receiver OObjects
	$\{O_j \in DO \mid O_j = \langle C_j\langle\overline{D'}\rangle\rangle, j = 1..sz'\} = lookup(O_i, T_R)$	//find labels C_j
	$DE' = DE \cup \bigsqcup_{i=1..sz, j=1..sz'} \{\langle O_i, O, C_j, Imp\rangle\}$	//create import edges
	$\forall e_k \in \overline{e},\ e_k : T'_k \quad T'_k <: T_k \quad T_k \in \overline{T}$	//for all arguments
	$\{O_{j'} \in DO \mid O_{j'} = \langle C_{j'}\langle\overline{D''}\rangle\rangle, j' = 1..sz''\} = lookup(O, T'_k)$	//get labels $C_{j'}$
	$DE' = DE' \cup \bigsqcup_{i=1..sz, j'=1..sz''} \{\langle O, O_i, C_{j'}, Exp\rangle\}$	//create export edges
	$[\![e_0]\!]\ O\ G' \quad \forall e_k \in \overline{e}\ [\![e_k]\!]\ O\ G'$	//recursive calls
$[\![e_0.f_k]\!]\ O\ G$	$G' : \langle DO', DD', DE' \rangle \quad DO' = DO \quad DD' = DD$	
//field read	$e_0 : C\langle\overline{p}\rangle \quad (C_k\langle\overline{p'}\rangle\ f_k) = fields(C\langle\overline{p}\rangle)[k]$	//receiver's type and field decl.
	$\{O_i \in DO \mid O_i = \langle C_i\langle\overline{D}\rangle\rangle, i = 1..sz\} = lookup(O, C\langle\overline{p}\rangle)$	//find receiver OObjects
	$\{O_j \in DO \mid O_j = \langle C_j\langle\overline{D'}\rangle\rangle, j = 1..sz'\} = lookup(O_i, C_k\langle\overline{p'}\rangle)$	//find labels C_j
	$DE' = DE \cup \bigsqcup_{i=1..sz, j=1..sz'} \{\langle O_i, O, C_j, Imp\rangle\}$	//create import edges
	$[\![e_0]\!]\ O\ G'$	//recursive calls
$[\![e_0.f_k := e_1]\!]\ O\ G$	$G' : \langle DO', DD', DE' \rangle \quad DO' = DO \quad DD' = DD$	
//field write	$e_0 : C\langle\overline{p}\rangle \quad (C_k\langle\overline{p'}\rangle\ f_k) = fields(C\langle\overline{p}\rangle)[k]$	//receiver's type and field decl.
	$e_1 : C_1\langle\overline{p''}\rangle \quad C_1\langle\overline{p''}\rangle <: C_k\langle\overline{p'}\rangle$	//ensures types are compatible
	$\{O_i \in DO \mid O_i = \langle C_i\langle\overline{D}\rangle\rangle, i = 1..sz\} = lookup(O, C\langle\overline{p}\rangle)$	//find receiver OObjects
	$\{O_j \in DO \mid O_j = \langle C_j\langle\overline{D'}\rangle\rangle, j = 1..sz'\} = lookup(O, C_1\langle\overline{p''}\rangle)$	//find labels C_j
	$DE' = DE \cup \bigsqcup_{i=1..sz, j=1..sz'} \{\langle O, O_i, C_j, Exp\rangle\}$	//create export edges
	$[\![e_0]\!]\ O\ G' \quad [\![e_1]\!]\ O\ G'$	//recursive calls
	$lookup(O, C'\langle p'\rangle) = \{O_k \in DO \mid O_k = \langle C\langle\overline{D}\rangle\rangle, C <: C',$	//for precision, use
	$\forall i \in 1..\lvert\overline{p'}\rvert\ D_i = DD[(O, p'_i)]\}$	//ownership domains

Figure 5. Transfer functions $\langle [\![e]\!]\ O\ G \rangle$ for the construction of the OGraph. They transform $G : \langle DO, DD, DE \rangle$ to $G' : \langle DO', DD', DE' \rangle$.

analysis, because $lookup$ returns only a subset of all the instances of class C' or its subclasses in DO.

For a method invocation, the analysis takes the type of the receiver, type of the arguments, and return type. The analysis calls $lookup$ for each of these types and finds: the receiver OObject, the OObjects passed as arguments, and the return OObject. The analysis creates import edges with the receiver as the source, O as destination, and the class of the return OObject as label. Next, for each argument, the analysis creates export edges with O as source, the receiver as destination, and the class of argument OObject as label.

For a field read, the analysis creates only import edges, from the receiver to O, where the label is a subtype of the field type, as $lookup$ returns. For a field write, the analysis creates only export edges, from O to the receiver. The label is a subtype of the right-hand-side expression type, as $lookup$ returns. For each of these expressions, DO and DD are unchanged. The last line in each transfer function represents a recursive call. The analysis continues until it covers all the expressions.

Precision. For a better intuition of how the analysis achieves precision, assume that the type available in the code for a receiver is a class C'. Without $lookup$, an imprecise analysis such as Spiegel's [15] would unfold C' into subclasses C, and create dataflow edges from O to all instances of C. If a dataflow edge is created for all these possible receivers, the resulted OGraph may have too many edges without a corresponding runtime edge. Instead, ownership domains improve the precision of our analysis, and enable it to distinguish between different instances of the same class in different domains, and include only a subset of all these edges in DE, thus reducing the number of false positives.

This paper includes only the transfer functions that create dataflow edges (Fig. 5). The companion technical report contains the transfer functions for `new` expressions, which show how the ownership hierarchy is created, alongside with the complete formalization of the analysis and the soundness theorem and proof [12]. To simplify proving soundness, we described the analysis using an equivalent constraint-based specification. Transfer functions are required by the framework we use to implement the analysis [18].

Differences with our previous work. Our analysis is similar to the one that extracts an OOG with points-to edges [9]. The two analyses create the same ownership hierarchy, but different types of edges. The key differences deal with generating dataflow edges and the soundness proof [14, Sec. 3.2, and 3.3]. Another previous work [10] made a simplistic assumption that dataflow edges can be approximated by reverting points-to edges, which turned out to be imprecise.

V. Evaluation

Our evaluation is similar in style to a code architecture analysis [6]. 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 on an OOG?

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 (DFD-L1) shows the main components of the system. A Level-2 DFD (DFD-L2) shows the components of a DFD-L1 broken into their constituent sub-components.

Subject System. For the subject system, we selected CryptoDB, a secure database system designed by a security expert [19]. 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 we manually added the annotations in the code [14, Sec. 7.8].

Tools. We implemented our analysis as an Eclipse plugin, on top of an existing static analysis framework [18]. We extracted two OOGs, one with dataflow edges and one with points-to edges. Since the nodes of the OOGs are the same, we overlay the two types of 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 edges. Also, since our analysis can extract multiple dataflow edges with different labels between the same objects, we post-processed the OOG to show only one edge having the merged set of labels (Fig. 6). The tools allowed us to automatically trace to code nodes and edges on the OOG.

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 labels. DFD-L1 showed 9 components and communication between them. Although the names in DFDs and OOGs do not match, by reading the documentation, we were able to establish the mapping between the DFD-L1 components and the 9 top-level objects in OOG-L1 (Table I). Next, this mapping at the top-level trigger the correspondence between the sub-components. DFD-L2 shows the sub-components of Crypto Consumer and Crypto Provider; in OOG-L1, we expanded the substructure of `mgr` and `provider` (OOG-L2) (Fig. 6(d)).

To answer RQ1, we compared a DFD to an OOG with dataflow edges as follows: (1) we extracted the OOG and collapsed it until it was at the same level of abstraction as the DFD; (2) using the mapping between names, we mapped the DFD components to the objects in OOG; (3) we compared the DFD edges to the dataflow edges by visual inspection; (4) if an edge appeared in both, we checked if we recognized the OOG labels in DFD edge labels; (5) if an edge appeared only in the OOG, we traced the dataflow edge to the corresponding lines of code using existing tools to confirm that the dataflow edge is a true positive.

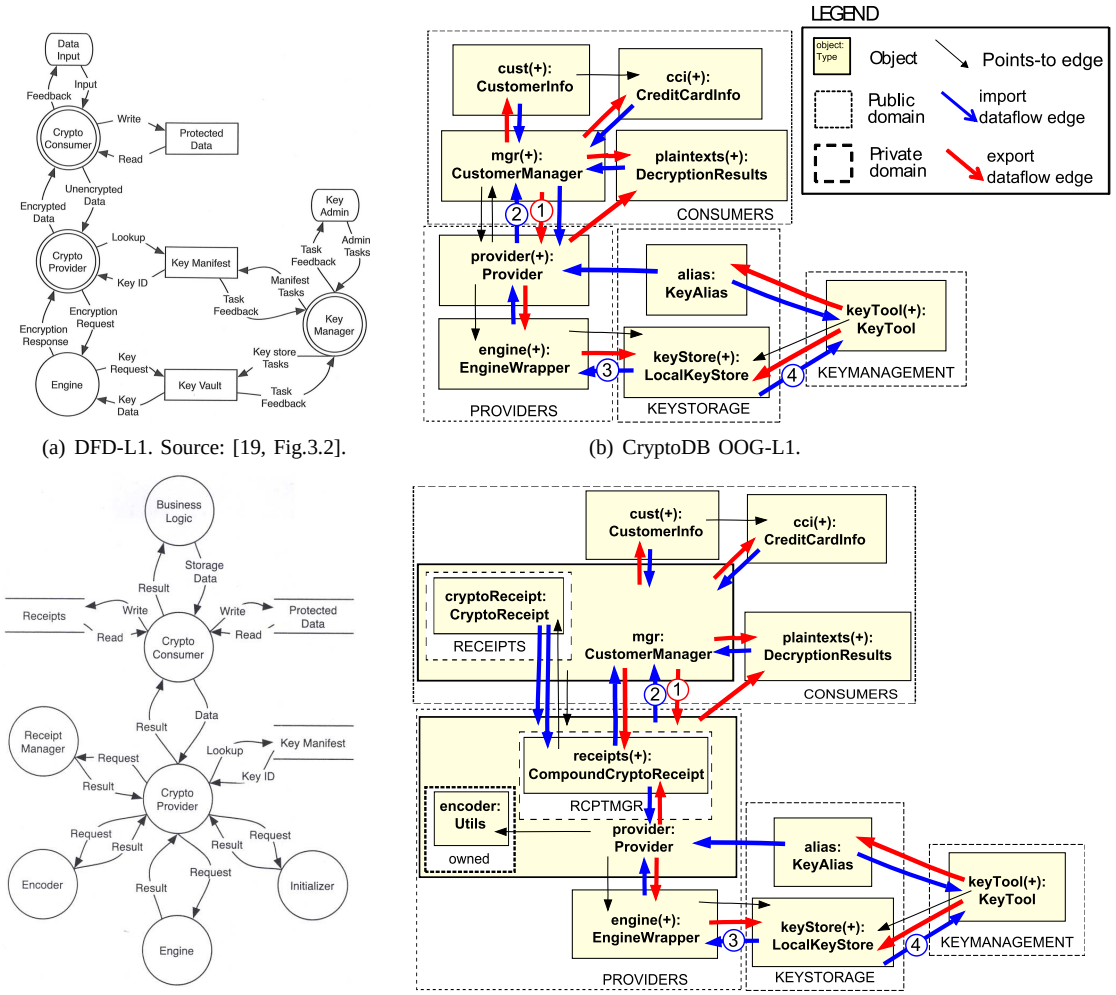
Table I
MAPPING BETWEEN DFD AND OOG.

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`

Observation 1: A cycle of 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` → `engine` → `keyStore` → `keyTool` → `alias` → `provider`. The same components were also linked by a cycle in the opposite direction, but in the OOG, the edge `provider`→`alias` was missing. Our inspection revealed that the edge was missing due to analysis limitations with static code.

Observation 2: We recognized the OOG edge labels in the corresponding DFD edge labels. We show the most interesting edge labels in Fig. 6(b) listed in Table II. DFD-L1 showed an edge from Key Vault to Engine labeled Key Data, while the corresponding edge label in OOG-L1 from `engine` to `keyStore` was `LocalKey` (3). While "Data" is a general term, the `LocalKey` indicated "what kind" of data `keyStore` provided. As another example, DFD-L1 showed the edge labels Unencrypted Data and Encrypted Data between Crypto Consumer and Crypto Provider. The corresponding labels in the OOG were five different classes (Table II). By tracing edges to code, we found for example that `CryptoReceipt` implemented encrypted data, while `CustomerInfo` and `CreditCardInfo` represented unencrypted data. The edge label `DecryptionResults` from `provider` to `mgr` indicated that `provider` also decrypts data. This information was unclear in DFD-L1, while in DFD-L2 the corresponding edge labels Data and Result were very general.

Observation 3: By expanding an OOG, we recognized the sub-components in DFD-L2. The OOG-L2 showed `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 communica-



(a) DFD-L1. Source: [19, Fig.3.2].

(b) CryptoDB OOG-L1.

(c) DFD-L2. Source: [19, Fig.6.1].

(d) CryptoDB OOG-L2 with `mgr` and `provider` expanded.

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

tion between `provider` and `receipts`, also shown in DFD-L2 as a Request – Result communication between Crypto Provider and Receipt Manager. We also found divergences and absences in the OOG. For example, DFD-L2 shows Initializer as a sub-component of Crypto Provider, but they were not clearly separated in the implementation. For a divergence example, OOG-L2 showed a dataflow edge from `cryptoReceipt` to `receipts`, while DFD-L2 had no edge between Receipts and Receipt Manager. By expanding the OOG, we observed that the implementation did not follow strictly the design. Our observation is also supported by the book, which states that the secondary functionalities were glossed over [19, Chap. 13].

To answer RQ2, we used the extraction tool to overlay the two types of edges, and we compared the set of dataflow edges (DE) against the set of points-to edges (PtE). To confirm divergences, we traced to code each edge. We repeated the procedure for OOG-L1 and OOG-L2 (Fig. 6).

Observation 4: Dataflow edges were different from points-to edges. The sets DE and PtE in OOG-L1 and OOG-L2 are different with little overlap (Table III). 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 those cases where objects were connected through dataflow edges only, which correspond to a method that creates an object, without storing it in a field as a persistent relation. We found for example that `CustomerManager` had a method `getCustomer` which instantiated the `cci` and `cust` objects, read the expiration date from the database and passed `cci` to the `cust` object. `CustomerManager` had only one field

Table II
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 III
POINTS-TO VS. DATAFLOW EDGES.

	$\#O$	PtE	DE	$DE \setminus PtE$	$PtE \setminus DE$	$DE \cap PtE$
OOG-L1	9	6	19	13	1	5
OOG-L2	12	7	24	20	3	4

of type `Provider`, so the analysis did not create points-to edges from `mgr` to `cci` or `cust`.

Observation 5: Rarely, import dataflow edges overlapped 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`.

Limitations. There are a few expressiveness challenges of the Ownership Domains type system. For example, all ownership type systems struggle with static code. 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 is annotated with `lent`, an additional analysis is needed to determine the actual domain. Currently, our analysis omits the corresponding dataflow edges. In CryptoDB, only 7% of the annotations were `lent`, so they had a limited influence on the results. One workaround is to manually resolve the `lent` annotation, and to use a more precise annotation.

This paper does not address the cost of adding ownership annotations, which was estimated to be 1 hour/KLOC [20]. Because the annotations implement a type system, they are amenable to type inference, which is a separate research problem and an active area of research [21], [22]. This paper focuses on the additional benefits of leveraging the annotations once they are in place. If the benefits are positive, in future work we can focus on reducing the cost.

VI. RELATED WORK

Murphy's Reflexion Models (RM) [6] inspired the evaluation style in this paper. In RM, a developer maps source-level entities to components in a high-level model. However, the mechanism of our approach is different since the OOG correspond to a runtime architecture, not a code architecture. Tools that work for code architecture are more mature, but they do not necessarily work for runtime architecture. For example, code architecture does not deal with aliasing. In a class diagram, a class is represented by one box, so an expert cannot reason about communication between difference instances of the same class. Also, in the presence of inheritance, two boxes in a class diagram may correspond to the one instance in an object graph.

Dataflow Communication. Andersen's static analysis extracts dataflow and points-to information from programs written in C [23], and was extended to object-oriented code [24], [25] including Java [15]. 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* [15]. However, the results of these analyses are flat graphs [26], [15]. For example, Spiegel builds a type graph which is then unfolded into an object graph. We use the same definition of dataflow edges, but a completely different technology, namely abstract interpretation. Not only is our analysis sound, it is also more precise due to $lookup$ which returns instances of a given type in a reachable domain, as opposed to instances of a given type in the whole program. Previous work [27] has shown that hierarchical object graphs can dramatically reduce the number of top-level objects in an OOG, and produce succinct diagrams that convey high-level comprehension of the runtime structure. For architectural risk analysis, half the battle is often extracting a "forest-level view" of the system structure [2, Chap. 5]).

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 different contexts that invoke the method. Object-sensitive analyses for points-to and dataflow edges addressed the aliasing and precision challenges [25], [24]. Such an analysis worked well for on-demand based approaches which refined the references analyzed [28], but might not scale for large number of references. Seeking a tradeoff between soundness and precision, our analysis considers domains as contexts and distinguishes objects of the same type but in different domains. So, our analysis is *domain-sensitive*, and object- and flow-insensitive.

Dynamic analyses. Object graphs were extracted by analyzing heap snapshots [29], [30], and execution traces [31]. 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 [31]. The OFG analysis addressed the 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 is not sound. 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 [13]. Ownership Domains also support logical containment and thus are more flexible in expressing arbitrary design intent. In addition, a dynamic analysis requires extensive graph summarization to obtain an abstracted graph [30], [32].

Annotation-based static analyses. Lam and Rinard [33] proposed a type system and a static analysis where 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, the extracted object model is non-hierarchical, thus with limited scalability. In addition, Lam and Rinard extract models for "subsystem access", "call/return interaction", and "heap interaction", which are similar to the dataflow information our analysis extracts. From the challenges we listed, they addressed aliasing, and precision supported by tokens. Our approach is different than Lam and Rinard's since it extracts hierarchical object graphs and supports object-oriented language constructs such as inheritance.

VII. 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. Ownership Domains improve the precision of our analysis and provide a hierarchical organization of objects. We evaluated our analysis on an extended example and showed that the reverse engineered edges are similar to the edges drawn by an architect who reasons about security and dataflow communication.

References

[1] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[2] G. McGraw, *Software Security: Building Security In*, 2006.

[3] 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.

[4] R. Koschke, "Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level," in *International Summer School on Software Engineering*, A. D. Lucia and F. Ferrucci, Eds., 2008.

[5] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *TSE*, vol. 35, no. 4, 2009.

[6] G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation" *TSE*, vol. 27, no. 4, 2001.

[7] P. Torr, "Demystifying the Threat-Modeling Process," *IEEE Security and Privacy*, vol. 3, no. 5, 2005.

[8] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering Architectures from Running Systems," *TSE*, vol. 32, no. 7, 2006.

[9] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," in *OOPSLA*, 2009.

[10] M. Abi-Antoun and J. M. Barnes, "Analyzing Security Architectures," in *ASE*, 2010.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] R. Vanciu and M. Abi-Antoun, "Extracting Dataflow Communication from Object-Oriented Code," WSU, Tech. Rep., 2011, www.cs.wayne.edu/~mabianto/tech_reports/VA11_TR.pdf.

[13] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism," in *ECOOP*, 2004.

[14] M. Abi-Antoun, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure," Ph.D. dissertation, CMU, 2010.

[15] A. Spiegel, "Automatic Distribution of Object-Oriented Programs," Ph.D. dissertation, FU Berlin, 2002.

[16] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," in *ICSE*, 2002.

[17] M. Abi-Antoun, J. Aldrich, and W. Coelho, "A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing," *J. Systems & Software*, vol. 80, no. 2, 2007.

[18] PLAID Research Group, "The Crystal Static Analysis Framework," 2009, http://code.google.com/p/crystalsaf. [Online]. Available: http://code.google.com/p/crystalsaf

[19] K. Kenan, *Cryptography in the Database*. Addison-Wesley, 2006, code: http://kevinkenan.blogs.com/downloads/cryptodb_code.zip.

[20] M. Abi-Antoun, N. Ammar, and Z. Hailat, "Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report," in *QoSA*, 2012.

[21] W. Dietl, M. Ernst, and P. Mller, "Tunable static inference for generic universe types," in *ECOOP*, 2011.

[22] W. Huang, W. Dietl, A. Milanova, and M. Ernst, "Inference and checking of object ownership," in *ECOOP*, 2012.

[23] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, University of Copenhagen, 1994.

[24] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized Object Sensitivity for Points-To Analysis for Java," *TOSEM*, vol. 14, no. 1, 2005.

[25] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[26] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," *TSE*, vol. 27, no. 2, 2001.

[27] R. Vanciu and M. Abi-Antoun, "Object Graphs with Ownership Domains: an Empirical Study," in *Springer LNCS State-of-the-Art Survey on Aliasing in Object-Oriented Programming*, 2012, to appear.

[28] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for Java," in *OOPSLA*, 2005.

[29] N. Mitchell, "The Runtime Structure of Object Ownership," in *ECOOP*, 2006.

[30] N. Mitchell, E. Schonberg, and G. Sevitsky, "Making Sense of Large Heaps," in *ECOOP*, 2009.

[31] A. Lienhard, S. Ducasse, and T. Grba, "Taking an object-centric view on dynamic information with object flow analysis," *COMLAN*, vol. 35, 2009.

[32] T. Hill, J. Noble, and J. Potter, "Scalable Visualizations of Object-Oriented Systems with Ownership Trees," *J. Visual Lang. and Comp.*, vol. 13, no. 3, 2002.

[33] P. Lam and M. Rinard, "A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information," in *ECOOP*, 2003.