Poster: Abstract Runtime Structure for Reasoning about Security

Marwan Abi-Antoun   Ebrahim Khalaj   Radu Vanciu   Ahmad Moghimi
Wayne State University, Department of Computer Science, Detroit, MI
{mabiantoun, mekhalaj, radu, amoghimi}@wayne.edu

ABSTRACT

We propose an interactive approach where analysts reason about the security of a system using an abstraction of its runtime structure, as opposed to looking at the code. They interactively refine a hierarchical object graph, set security properties on abstract objects or edges, query the graph, and investigate the results by studying highlighted objects or edges or tracing to the code. Behind the scenes, an inference analysis and an extraction analysis maintain the soundness of the graph with respect to the code.

CCS Concepts

•Software and its engineering → Object oriented architectures;

Keywords

object graphs; ownership type inference; graph query

1. INTRODUCTION

Reasoning about quality attributes such as security requires an abstraction of the system at runtime. For object-oriented code, abstract object graphs represent how objects communicate at runtime. However, flat object graphs convey little abstraction or high-level understanding.

One powerful organizing principle in software engineering is the notion of hierarchical decomposition. We follow this principle of *abstraction by hierarchy* and organize objects into a hierarchy. However, instead of making objects have child objects directly, we introduce a level of indirection, a *domain*, i.e., a conceptual group of objects. Domains express hierarchy: in the graph, each object contains child domains, and each domain in turn contains child objects. Domains are interesting for reasoning about security because they are related to the notion of trust boundary. A private domain expresses strict encapsulation where the owned object cannot be accessed without going through its owner. A public domain expresses logical containment where one object is conceptually part of another object, but is still accessible to other objects. Domains also express precision since the extraction analysis uses them to abstract the concrete, runtime objects to pairs of types and domains.

Since object hierarchy is not directly visible in mainstream object-oriented languages, analysts must provide richer type information such as ownership types [2]. Requiring analysts to manually add ownership types will not work in practice, so there is active work in ownership type inference. Fully automated inference can identify strict encapsulation. But fully automated inference cannot infer the design intent of logical containment, except for simple cases, so we rely on analysts to express that design intent.

Instead of having analysts think in terms of ownership types in the code, our approach enables them to think in terms of re-grouping abstract objects or modifying parent-child relationships, using *refinement* operators that move an abstract object from one domain to another, as long as the corresponding ownership types that typecheck can be inferred for the code as written.

```
class MutableClass {
    private MyDate d = new MyDate();

    public MyDate getDate() {
      // v1: return alias to field
      // return d;

      // v2: return copy/clone
      return d.clone();
    }
}
class MyDate {
    private Long value = new Long(0);

    // ... setter/getter elided...

    public MyDate clone() {
      final MyDate copy = new MyDate();
      // v2: shallow copy
      // copy.setValue(this.value);

      // v3: deep copy
      Long longVal = new Long(this.value.longValue());
      copy.setValue(longVal);

      return copy;
    }
}
```

Figure 1: Three versions of the code: v1 returns an alias to a private field. v2 and v3 return a copy. v2 creates a shallow copy, v3 creates a deep copy.

HotSoS '16, April 19-21, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4277-3/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2898375.2898377

2. MOTIVATING EXAMPLE

The CERT Oracle Secure Coding Standard for Java [3] has many rules for securing object-oriented code. For instance, *OBJ05* stipulates: *Do not return references to private mutable class members* [4]. Consider the following example to illustrate the above rule (Fig. 1). In the `MutableClass`, a non-compliant version (v1, line 6) of the method `getDate()` returns an alias to the `private` field `d` of type `MyDate`, thus exposing the weakness of visibility modifiers. A malicious client can then directly mutate the object.

Our approach. Our approach promotes reasoning based on a high-level representation, the abstract object graph, rather than the code. The graph conveys high-level understanding, enabling analysts to set security properties on selected abstract objects, run queries, and investigate unexpected sharing or communication.

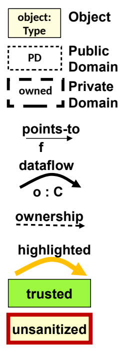
For example, a *tampering* query checks if any object that has the property *isSanitized=false* flows to any object that has the property *trustLevel=High*. Here, the analysts set the properties as follows (Fig. 2). The `Long` object is *unsanitized* (shown with a thick border), and the `MutableClass` is `trusted` (shown in green). The query uses the object hierarchy: it will find if a tampered object flows to a child of the trusted destination.

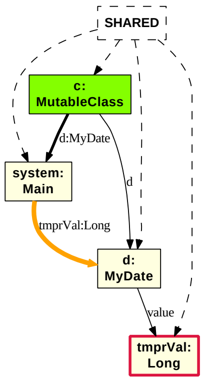
Act I. The analyst runs the tool, which extracts an initial object graph. She sets the properties as above and runs the predefined tampering query, which highlights a problematic edge on the first object graph (Fig. 2(b)). She then attempts a refinement: make the `MyDate` *owned-by* the `MutableClass` object. The tool, however, indicates that the code does not support such a refinement, and which expression in the code prevents the analysis from inferring valid types. The analyst then traces to the code and studies the issue.

Since timestamps are used to enforce many security properties, a compliant version must return a copy or a clone of the `MyDate` object (line 9). Malicious clients then mutate a copy rather than the internal representation of the object.

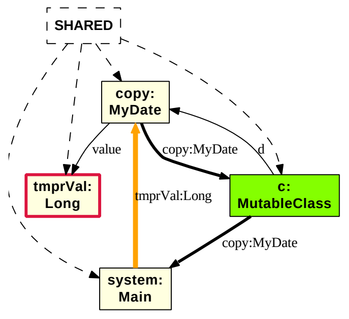
Act II. The analyst fixes the code, reruns the tool, and re-attempts the refinement, which now succeeds. The tampering query, however, still shows a problematic edge on the second object graph (Fig. 2(c)). The analyst traces to code and investigates further. She realizes that this code is still non-compliant since the `clone()` method of `MyDate` is returning a shallow copy (v2, line 19), where both objects of type `MyDate` share the same representation of type `Long`.

Act III. The analyst fixes the code to return a deep copy (v3, line 22), and reruns the tool. On the third object graph (Fig. 2(d)), the query no longer shows a problematic edge. The abstract object `c` of type `MutableClass` receives a `copy` of type `MyDate`, that is distinct from the object `d`, also of type `MyDate`.

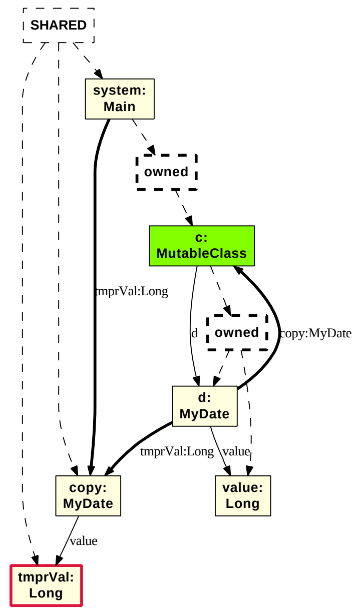




(a) Legend. (b) Alias to my `MyDate` (v1).



(c) Shallow copy of `MyDate`(v2).



(d) Deep copy of `MyDate` (v3).

Figure 2: Object graphs extracted from the code.

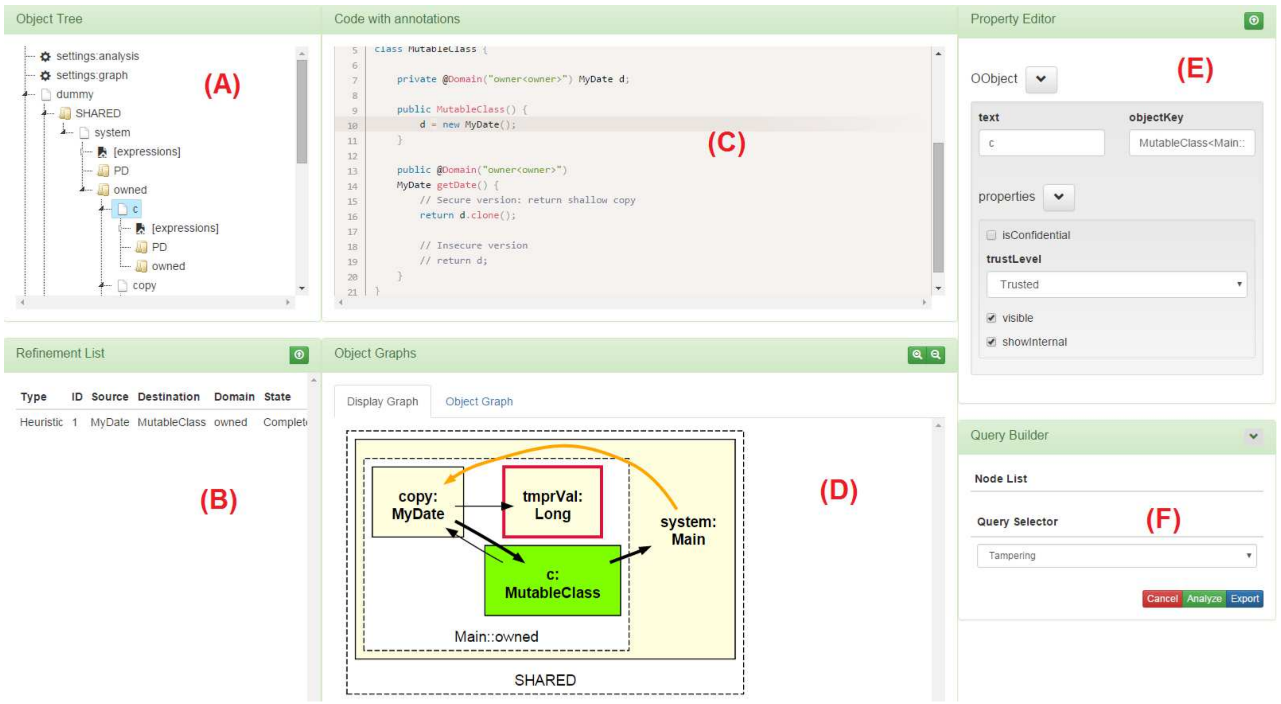
3. APPROACH OVERVIEW

We propose a multi-pronged approach for reasoning about the security of an object-oriented system, through the interactive refinement, extraction, and querying of its abstract object graph. The approach works as follows:

- Analysts use *refinement operators* to express their design intent related to strict encapsulation, logical containment and the grouping of objects;
- If the code supports this refinement, an inference analysis infers ownership types in the code, and saves them as annotations;
- Based on these inferred types and annotations, a static analysis extracts from the code an updated hierarchical abstract object graph, which can be refined further;
- Analysts set properties, set arguments to queries, and investigate the query results.

We discuss briefly each part in turn.

Interactive refinement. A user interface supports refinements by direct manipulation of the object tree or the object



```
class MutableClass {

    private @Domain("owner<owner>") MyDate d;

    public MutableClass() {
        d = new MyDate();
    }

    public @Domain("owner<owner>")
    MyDate getDate() {
        // Secure version: return shallow copy
        return d.clone();

        // Insecure version
        // return d;
    }
}
```

Type	ID	Source	Destination	Domain	State
Heuristic	1	MyDate	MutableClass	owned	Complet

text: c
objectKey: MutableClass<Main::
isConfidential ☐
trustLevel: Trusted
visible ✓
showInternal ✓
Query Selector: Tampering

Figure 3: Screenshot of the web-based interface: each panel (labeled A–F) is explained in Section 4.

graph. For example, the analyst can push an object underneath another object in the object tree using drag-n-drop operations. For example, the analyst makes the object `d` of type `MyDate` *owned-by* the object of type `MutableClass`. The list of refinements contains valuable design intent and can be replayed on evolving versions of the system.

Inference static analysis. Given a refinement as input, the inference analysis either infers valid ownership types thus confirming that the refinement is well-formed. Conversely, the analysis fails to infer valid types showing that the refinement is infeasible for the code, and identifying the problematic expressions for which it cannot infer valid types. If the analysts insist on that refinement, they must change the code and re-run the tool.

Extraction static analysis. From the code with the inferred types, the extraction static analysis [1] extracts a hierarchical abstract object graph that shows two types of node: abstract objects and domains. Several edge types can be shown including points-to and data-flow.

Querying of object graph. Analysts select abstract objects or edges, assign them properties, set the arguments to several possible predefined queries and investigate the results, by tracing to the corresponding lines of code [5].

The properties are displayed graphically. These properties are tied to abstract objects and can be reused across evolving versions of the system.

4. CONTRIBUTION

This poster presents a web-based user interface that integrates these above analyses into a tool for analysts (Fig. 3):

- **Object Tree (panels A-B):** analysts request refinements by direct manipulation (drag-and-drop) in the object tree. The list of refinements and their status is below the tree (panel B);
- **Code Display (panel C):** analysts trace from selected nodes or edges to the code and investigate highlighted edges identified by the queries;
- **Graph Display (panel D):** the object graph can be displayed as either a graph of objects and domains (Fig. 2(d)) or as nested boxes (middle of Fig. 3). The graphs in Fig. 2(d) are available under the *Object Graph* tab. Analysts can select nodes directly on the graph;
- **Property Editor (panel E):** edit the values of properties on selected objects or edges such as *isConfidential* or *trustLevel*;
- **Query Builder (panel F):** analysts select the source, intermediate, and destination objects, by clicking on the graph, then invoke a query from a list of predefined queries. Query results are shown as highlighted objects or edges on the graphs (panel D). Predefined queries for Information Disclosure and Tampering require that only properties be set.

Funding. This work is supported in part by the National Security Agency lablet contract #H98230-14-C-0140.

5. REFERENCES

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

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

[3] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley, 2011.

[4] SEI CERT Oracle Coding Standard for Java, 2016. www.securecoding.cert.org/confluence/display/java/.

[5] R. Vanciu and M. Abi-Antoun. Finding architectural flaws using constraints. In *ASE*, 2013.