Finding Architectural Flaws using Constraints

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 (ARA), security architects use a runtime architecture to look for security vulnerabilities that are architectural flaws rather than coding defects. The current ARA process, however, is mostly informal and manual. In this paper, we propose Scoria, a semi-automated approach for finding architectural flaws. Scoria uses a sound, hierarchical object graph with abstract objects and dataflow edges, where edges can refer to nodes in the graph. The architects can augment the object graph with security properties, which can express security information unavailable in code. Scoria allows architects to write queries on the graph in terms of the hierarchy, reachability, and provenance of a dataflow object. Based on the query results, the architects enhance their knowledge of the system security and write expressive constraints. The expressiveness is richer than previous approaches that check only for the presence or absence of communication or do not track a dataflow as an object. To evaluate Scoria, we apply these constraints to several extended examples adapted from the CERT standard for Java to confirm that Scoria can detect injected architectural flaws. Next, we write constraints to enforce an Android security policy and find one architectural flaw in one Android application.**

I. Introduction

Although 50% of security vulnerabilities are architectural flaws such as information disclosure [1], they receive less attention from security analyses that focus on coding defects such as a hard-coded password [2]. In object-oriented code, a local coding defect can be found by analyzing one class or one method at a time. On the other hand, architectural flaws require reasoning about non-local information such as the context of architectural components.

A security analysis focused on architectural flaws is Architectural Risk Analysis (ARA) also known as threat modeling [3]. During ARA, architects use a runtime architecture, which is a collection of runtime components and connectors. The current ARA process is informal and manual. One way to automate ARA is to execute machine-checkable *constraints* against the runtime architecture. For example, a constraint checks all connectors and restricts communication of components created in a given context. In addition, constraints may use *security properties* such as *TrustLevel* that architects assign to component and connector instances [4], [5].

In this paper, we target architectural flaws such as information disclosure and tampering, which may occur for connectors between trusted and untrusted components. Simply checking for the presence of communication is insufficient and may lead to false positives. Instead, architects need to reason about the content of a connector. For example, a connector that passes confidential information from a trusted source to an untrusted destination may lead to information disclosure. On the other hand, an untrusted connector that has a trusted destination and passes unsanitized information may lead to tampering [3].

In previous work [6], [7], we approximated a runtime architecture by using a static analysis to semi-automatically extract from object-oriented code with annotations a sound, hierarchical Ownership Object Graph (OOG) with dataflow communication edges such that the OOG approximates all possible runtime objects and communication. A component is represented by an abstract object and the objects in its substructure. A connector is represented by edges between abstract objects.

In this paper, we propose Scoria, a semi-automatic approach for finding architectural flaws during ARA. Scoria uses a security graph (SecGraph) that augments the OOG with security properties, queries, and constraints. The queries allow architects to deepen their understanding of an abstract runtime architecture as needed, and to assign security properties to sets of abstract objects and edges. The architects can then enforce a security policy as a set of constraints that are predicates on the sets returned by the queries.

The contributions of Scoria are queries that capture the thought process of how architects reason about the communication of objects in a runtime architecture. For security, it is important to identify the objects that edges refer to. Some objects that carry protected data may be part of some other object that seems to carry only unprotected data. In the SecGraph, a dataflow edge refers to an abstract object. This is similar to how a manually drawn runtime architecture used in ARA has messages rather than types or actions on the directed edges between component instances [8]. Scoria automatically tracks the content of the dataflow through *object hierarchy* and *object reachability*. Then, architects can write constraints by reasoning about the source, the destination, and the flow in terms of abstract objects. For example, in a communication from a trusted source to an untrusted destination, the dataflow object may not be confidential, but a confidential object could be reachable from it or be contained within its substructure.

Architects can also reason about *object provenance* when different dataflow edges refer to the same object. Object provenance uses only the structure of the SecGraph. In some cases, to supplement the information from the runtime architecture, architects use security properties to supply security-relevant design intent that is not directly available in the code or in an abstract object graph that is automatically extracted.

The queries also compute the *indirect communication* through object hierarchy and object reachability of the source

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA

and destination. Architects may miss such communication that is not explicit in the SecGraph and leads to architectural flaws.

To evaluate Scoria, we write machine-checkable constraints for several rules in the CERT Oracle Secure Coding Standard for Java [9] focusing on architectural flaws for which automated detection was previously unavailable. Our evaluation confirms that Scoria can detect architectural flaws in several extended examples. As a second evaluation, we formalize an Android security policy as a constraint and detect one architectural flaw in one Android application.

Contributions. Our contributions are:

- The Scoria approach of machine-checkable constraints on a security graph where dataflow edges refer to abstract objects. The constraints automatically track object provenance and indirect communication through object hierarchy and object reachability.
- An evaluation that the constraints find architectural flaws such as information disclosure and tampering in several extended examples adapted from the CERT standard for Java and in one Android application.

Outline. Section II identifies the requirements of ARA and describes the OOGs using a running example. Section III describes Scoria and defines the *SecGraph*. Section IV describes the evaluation and discusses the results. Section V describes related work, and Section VI concludes.

II. Background

A. Running example

We use Echo, a client-server application that highlights a potential information disclosure if the communication between the server and the client is not encrypted. To understand the communication, the architects need to distinguish between different objects of the same class `InputStream`. Echo is adapted from code fragments provided as examples in the CERT standard [9]. Echo consists of two objects, `client:EchoClient` and `server:EchoServer` that communicate via a message `ss:String` received from user input. The communication is established via a `sckt:Socket` object, which contains two streams of data, namely, `in:InputStream` and `out:OutputStream`. The object `sckt:Socket` does not provide encryption and should be used only if the data transmitted is not confidential or if the network is trusted. Otherwise, the implementation should use encryption protocols such as Secure Sockets Layer (SSL) to ensure that the channel is not vulnerable. The server supports both types of communication and has an object `sslSckt:SSLSocket` that encrypts data. However, to avoid the performance overhead associated with encryption, `client:EchoClient` uses `sckt:Socket` to communicate to `server:EchoServer`. To implement encryption in Java, developers should use the class `SSLSocket` instead of `Socket` as advised by the CERT rule [9, MSC00-J]. Otherwise, confidential messages may be disclosed to malicious applications that intercept messages from `sckt:Socket`. Such a vulnerability is an architectural flaw—not a coding defect—and according to the CERT standard, automated detection is unavailable.

B. Requirements on an ARA Solution

During ARA, architects use a runtime architecture rather than a code architecture. For object-oriented code, a runtime architecture shows objects as opposed to a code architecture that shows the code structure as packages and classes [1]. A runtime architecture can have multiple component instances of the same type that serve different conceptual purposes or have different security properties. For example, an object `in:InputStream` of standard input serves a different conceptual purpose from `in:InputStream` of a network, and the architecture shows them as two distinct components.

One limitation of the current ARA process is obtaining an initial runtime architecture that is consistent with the code. When developers manually document an architecture, they may miss some important components or connectors that exist at runtime in some program run. Since ARA is a worst-case analysis, it requires a *sound* architecture that represents all components and connectors that may exist at runtime because every component or connector can introduce vulnerabilities. For Echo, if `server:EchoServer` sends messages directly to `out:OutputStream`, the architecture should have such a connector even if it only occurs when an exception is thrown. In a sound runtime architecture, one component abstracts multiple runtime objects used in the same context, i.e., each runtime object is mapped to one component. Also, a sound runtime architecture does not map one runtime object to distinct components since architects could assign different property values to these architectural components and obtain misleading analysis results, when in fact these components are the same runtime entity.

Legacy code has high business value, and vulnerabilities in legacy code are expensive [10]. As a result, Scoria supports legacy code, requiring only annotations. In particular, it does not require the program to be implemented using specific programming languages [11] or libraries [12], or to require automated code generation as in SecureUML [13].

C. Extracting an Abstract Runtime Architecture

Scoria builds on a recent body of work that uses static analysis to extract an OOG with points-to edges [6] and dataflow edges [7]. In an OOG, a node is an abstract object that represents a possibly unbounded number of runtime objects. The OOG is sound such that each runtime object has exactly one abstract object as a representative. Soundness also means that every runtime edge has a corresponding abstract edge between the representatives of the source and the destination runtime objects. For brevity, we use *object* to refer to an abstract object.

The OOG is also hierarchical. An object does not have child object directly. Instead, an object contains domains, where a domain is a named conceptual group of objects where the name conveys design intent.

Notation. An object labeled `obj:T` indicates a reference `obj` of type `T`, which we then refer to as the "object `obj`" to mean "an instance of the `T` class". A dataflow communication has

the label obj:T to reflect that the same object obj:T (which could be a node in the graph) is flowing along the edge.

The OOG supports object identity where each abstract object has a unique identifier. In Echo, the architects can assign different property values to wrtr:OutputStream from the STREAMS parent domain of the sckt:Socket and to wrtr:OutputStream from the STREAMS parent domain of the sslSckt:SSLSocket (not shown) although both objects are of the same OutputStream class.

The architects can reason about object provenance because a dataflow edge refers to an object instead of a type. For example, the architects can use object identity to find that several dataflow edges refer to the same object such as ss:String in the DATA domain of m:Main. However, only some of these dataflow edges may lead to information disclosure. The architects can track the initial source of a dataflow transitively. The dataflow might be allowed as long as the *TrustLevel* value of the destination is *Trusted*. For example, a transitive communication of ss:String from in:InputStream to wrtr:OutputStream exists. The architects can reason that the object ss is confidential because the object in represents user input. Hence, a dataflow edge that refers to ss and has an untrusted destination may lead to information disclosure and should be investigated.

Since a hierarchical organization of objects is unavailable in plain Java code, the static analysis requires annotations to be added to the program. Another tool checks that the annotations are consistent with the code and with each other and convey design intent [6]. Using annotations, an object can be pushed in a domain underneath another object that has multiple domains in its substructure. Therefore, an OOG allows the architects to reason about the content of an abstract object that has a substructure. For Echo, the destination of a dataflow is wrtr:OutputStream in the substructure of sckt:Socket. The architects can reason that the wrtr object represents unencrypted data because Socket does not provide encryption, as opposed to the descendants of sslSckt:SSLSocket that may use encryption.

An OOG tracks object reachability through dataflow, points-to and creation edges. In Echo, the architects use the points-to edge from out:PrintWriter to wrtr:OutputStream to reason that a descendant of client:EchoClient has a persistent reference to a descendant object of sckt:Socket. If the object client were to pass out:PrintWriter as a dataflow object, it would also provide access to a descendant of sckt:Socket because the descendant object wrtr is reachable from out.

The OOG and the code are consistent and every node and edge of the OOG has traceability information. If the set of edges returned by a query satisfies a constraint, the architects can trace directly to the corresponding lines of code that may introduce an architectural flaw. The result of a query is thus directly actionable. This is in contrast to a manually drawn architecture that does not have traceability to code where the architects must spend additional effort reading the code and investigating each potential architectural flaw they find.

III. The Scoria Approach

Scoria is a semi-automated approach in which the architects supported by existing tools find architectural flaws as follows. First, the architects add annotations to code and typecheck them, fixing any of the typechecker warnings in the process. Second, the architects use a static analysis that automatically extracts a *SecGraph* from the code with annotations. Third, as an optional step, the architects assign security properties to objects and edges as needed using queries on the *SecGraph*. Fourth, the architects write and execute constraints on the sets returned by queries on the *SecGraph*. Finally, the architects trace to code from suspicious edges that constraints highlight and inspect potential architectural flaws. The process is incremental such that the architects can begin by assigning property values to only a few of the objects, while by default the rest have the ***Unknown*** value. In the following, we define several concepts informally introduced in the previous sections.

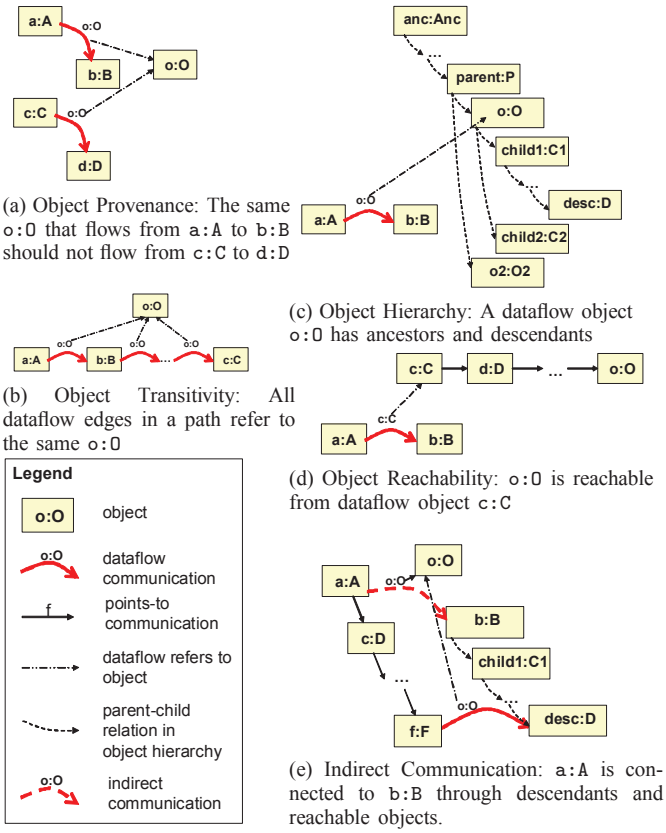
A. Definitions

Dataflow communication means that *an object a:A has a reference to an object o:O and passes it to an object b:B, or an object a:A has a reference to an object b:B and receives a reference to an object o:O* [14]. The objects a:A and b:B represent the source or destination objects, and o:O is a dataflow object that the dataflow communication refers to. To emphasize that a dataflow edge refers to an object, not a type, we use a thin dashed edge from a dataflow edge to the object that the edge refers to (Fig. 1). To capture the directionality of the flow, an object graph has import and export edges. An import dataflow edge exists due to the return values of method invocations or field reads. An export dataflow edge exists due to the arguments of method invocations or field writes. A dataflow edge is relevant for security because it may lead to vulnerabilities such as information disclosure or tampering [3].

Object identity *means that every abstract object in an object graph is uniquely identified.* As a result, object identity enables the architects to assign a value to a security property of an object o:O and enables comparison of references. Therefore, the property value can be accessed from all the dataflow edges that refer to o:O. Since o:O is uniquely identified, the property value is the same for all these dataflow edges. Also, two distinct abstract objects o1:C and o2:C can have different property values even if o1 and o2 are of the same class C.

Object Provenance *is a query that return the set of dataflow edges e_1 from a:A to b:B such that another dataflow edge e_2 exists from c:C to d:D and both e_1 and e_2 refer to the same object o:O* (Fig. 1a). For security, the architects can decide if the communication of the object o:O is suspicious by writing a constraint that checks if the set returned by an object provenance query is empty. When b:B and c:C are the same, the architects can reason about object transitivity.

Object Transitivity *is defined as the communication of an object o:O from a:A to c:C where a path of dataflow edges exists from a:A to c:C through some intermediate object b:B, where all the dataflow edges in the path refer to the*



(a) Object Provenance: The same `o:O` that flows from `a:A` to `b:B` should not flow from `c:C` to `d:D`

(c) Object Hierarchy: A dataflow object `o:O` has ancestors and descendants

(b) Object Transitivity: All dataflow edges in a path refer to the same `o:O`

(d) Object Reachability: `o:O` is reachable from dataflow object `c:C`

(e) Indirect Communication: `a:A` is connected to `b:B` through descendants and reachable objects.

Fig. 1: Information content available from communication

same object `o:O` (Fig. 1b). Object transitivity is important for security to reason about the initial source of a dataflow object. The object flowing along an edge is not just a simple object, it can have a substructure, or it can reach other objects.

Object Hierarchy *is defined as descendants and ancestors of an object `o:O` that are sets of objects such that a transitive parent-child relation exists between an object in the sets and `o:O`.* Object hierarchy is important for security because confidential information may be a descendant of a dataflow object, or a dataflow object can be part of the substructure of an object that represents confidential information (Fig. 1c).

Object reachability *is defined as a path of edges of various types from a source object `c:C` to a given object `o:O`* (Fig. 1d). The object `o:O` is reachable from `c:C` if a path exists from `c:C` to `o:O`. Object reachability is important for security because the object that represents confidential information can be reachable from a dataflow object. Then, the architects can consider such an edge to be suspicious. There are different types of relations between objects. In addition to dataflow, we also consider points-to and creation communication.

A points-to communication *exists from the source object `a:A` to the destination object `b:B` if one of `a`'s field `f` is a reference to `b:B`.* The label of the points-to edge is the field `f`. A points-to edge represents a persistent relation between objects and is relevant for security because a dataflow object can be used to reach an object that represents confidential information, although the dataflow object itself is not confidential.

SecGraph
- *root: SecObject*
- *edges(): EdgeSet*
- *objects(): ObjSet*
- *objectProvenance(...): EdgeSet*
- *objProvenanceIndirect(...): EdgeSet*
- *connectedByDF(...): EdgeSet*
- *getFlowToSink(...): EdgeSet*
- *reachables(o:SecObject, eTypes: Set⟨EdgeType⟩): ObjSet*

ObjSet `extends` Set⟨SecObject⟩
- *name:String*
- *prop: Property*

EdgeSet `extends` Set⟨SecEdge⟩
- *name:String*
- *prop: Property*

SecObject `extends` SecElement
- *C: Type*
- *parentObj:SecObject*
- *parentDom:SecDomain*
- *descendants() : ObjSet*
- *ancestors() : ObjSet*

SecDomain `extends` SecElement
- *d: String*

SecEdge `extends` SecElement
- *src: SecObject*
- *dst: SecObject*
- *edgeType: EdgeType*

*EdgeType:**DataFlow**|**PointsTo**|**Creation***

DataFlowEdge `extends` SecEdge
- *flow: SecObject*
- *direction: **Import**|**Export***

PointsToEdge `extends` SecEdge
- *fieldName: String*

CreationEdge `extends` SecEdge
- *flow: SecObject*

SecElement
- *name: String*
- *traceability(): Set⟨Traceability⟩*
- *trustLevel: **Trusted**|**Untrusted**|**Unknown***
- *isConfidential: **True**|**False**|**Unknown***
- *isEncrypted: **True**|**False**|**Unknown***
- *isSerialized: **True**|**False**|**Unknown***
- *isSanitized: **True**|**False**|**Unknown***
- *isTransient: **True**|**False**|**Unknown***

Traceability
- *expression: AstNode*

ClassInstanceCreation extends AstNode

Fig. 2: Partial representation of the *SecGraph*. Continued in Fig. 3,4

A creation communication *exists from the source object `a:A` to the destination object `b:B` if `a` creates `b` from an object `o:O`.* Similar to a dataflow edge, a creation edge refers to the object `o:O` and is relevant for security it may lead to vulnerabilities such as information disclosure or tampering.

Architects can obtain additional information about objects that are indirectly connected by a dataflow communication through descendants or reachable objects.

Indirect communication *exists from a source object `a:A` to a destination object `b:B` if a dataflow or a creation edge exists from a descendant of `a` or an object reachable from `a` to a descendant of `b` or an object reachable from `b`.* Indirect communication is relevant for security because a confidential object may be passed to a descendant of an untrusted object and the communication is not explicit in the abstract graph.

B. Security Graph

The *SecGraph* has *SecObject*s and *SecDomain*s as nodes, and *DataFlowEdge*s, *PointsToEdge*s and *CreationEdge*s as edges (Fig. 2). The *SecGraph* provides queries such as *descendants()* for finding the set of descendants of a given *SecObject*, and *reachables()* for getting the set of reachable *SecObject* from a given *SecObject*. In the *SecGraph*, we omit the details of the static analysis that extracts the OOG [7], and focus on the security properties, queries, and constraints. All these data types extend the *SecElement*, which has traceability information consisting of a set of traceability entries that correspond to expressions in the code. A traceability entry is a reference to a node in the Abstract Syntax Tree (AST) such as a *ClassInstanceCreation*. In the *SecGraph*, the *ObjSet* and *EdgeSet* have a name and property values for their elements. An *ObjSet* is different from a *SecDomain* such that the same *SecObject* can be part of two *ObjSet*s, but the same *SecObject* cannot be part of two *SecDomain*s. The *SecDomain* allows

Query based on object provenance

objectProvenance(*fSrc* : SecObject, *fDst* : SecObject, *sSrc* : SecObject, *sDst* : SecObject) : $EdgeSet$

objectProvenance := $\bigcup\{edge\}$ s.t. $edge$: DataFlowEdge $\in edges() \wedge$
$\exists o \in objects()$ s.t. $o \in$ *flow*(*connectedByDF*(*fSrc*, *fDst*)) $\wedge$
$edge \in$ *connectedByDF*(*sSrc*, *sDst*) $\wedge\ edge.flow = o$

flow(*dfEdges* : $Set\langle DataFlowEdge\rangle$) : $ObjSet$

flow := $\bigcup\{obj\}$ s.t. obj : SecObject $\in objects() \wedge$
$\exists edge \in dfEdges$ s.t. $edge.flow = obj$

connectedByDF(src : SecObject, dst : SecObject) : $EdgeSet$

connectedByDF := $\bigcup\{edge\}$ s.t. $edge$: DataFlowEdge $\in edges() \wedge$
$(edge.src = src \ \wedge\ edge.dst = dst)$

objProvenanceIndirect(*fSrc* : SecObject, *fDst* : SecObject, *sSrc* : SecObject, *sDst* : SecObject) : $EdgeSet$

objProvenanceIndirect := $\bigcup\{edge\}$ s.t. *edge:*DataFlowEdge $\in edges() \wedge$
$\exists o \in objects()$ s.t. $o \in$ *flow*(*connectedIndirect*(*fSrc*, *fDst*)) $\wedge$
$edge \in$ *connectedByDescendants*(*sSrc*, *sDst*) $\vee$
$edge \in$ *connectedByReachablility*(*sSrc*, *sDst*) $\wedge\ edge.flow = o$

Fig. 3: Object provenance queries on a *SecGraph*

then the architects to place *SecObject*s of the same type in different *ObjSet*s with different security properties.

Queries on the *SecGraph*. During ARA, the architects have some initial knowledge about the system. The queries allow the architects to increase their knowledge as needed by finding details in the extracted architecture that they are unaware of.

The simplest queries use methods specified on an instance of the *SecGraph*, G. For example, *G.edges()* returns all the edges in the graph. An instance *edge* of type *DataFlowEdge*, *edge.flow* returns the dataflow object that *edge* refers to. A query can also invoke methods in complex expressions such as *edge.flow.descendants()*, which returns the set of all descendants of the dataflow object.

The query *objectProvenance* returns a set of dataflow edges such that the same object that flows from a first source to a first destination also flows from a second source to a second destination (Fig. 3). The query uses *connectedByDF()* that returns the set of dataflow edges between two objects, and *flow()* that returns the set of dataflow objects that the dataflow edges in a given set refer to.

Reasoning about object provenance may also involve indirect communication through descendants and reachable objects. The query *objProvenanceIndirect* uses *connectedByDescendants()* that returns the set of dataflow edges from a descendant of the source to a descendant of the destination, and *connectedByReachability()* that return the set of dataflow edges from an object reachable from the source to an object reachable from the destination.

Enhancing OOG with security properties. To allow architects to specify design information unavailable in the code, a *SecElement* has a extensible set of security properties. Each property includes predefined values and the value ***Unknown*** used for default initialization. For example, the architects can specify objects that are trusted, or confidential. The property

Assign property values and select objects

setObjectsProperty(*props* : $Set\langle$Property$\rangle$, *cond* : Condition)

$\forall obj \in objects()$, s.t. $cond.satisfiedBy(obj) \implies$
$\exists p : IsConfidential \in props \ \wedge\ obj.isConfidential := p \vee$
$\exists p : TrustLevel \in props \ \wedge\ obj.TrustLevel := p \ \vee \ldots$

getObjectsByCond(*cond* : Condition) : $ObjSet$

getObjectsByCond := $\bigcup\{obj\}$ s.t. obj : $SecObject \in objects() \ \wedge$
$cond.satisfiedBy(obj)$

Select edges based on security properties

getFlowToSink(*fProps* : $Set\langle$Property$\rangle$, *sProps* : $Set\langle$Property$\rangle$) : $EdgeSet$

flows := *getObjectsByCond(fProps)* *sinks* := *getObjectsByCond(sProps)*

getFlowToSink := *getFlowToSink(flows,sinks)*

getFlowToSink(*flows* : ObjSet, *sinks* : $ObjSet$) : $EdgeSet$

getFlowToSink := $\bigcup\{e\}$ s.t. $e \in edges() \wedge$
e : DataFlowEdge $\vee$ e : CreationEdge $\wedge$
$\exists sink : SecObject \in sinks \ \wedge\ \exists o : SecObject \in objects()$ s.t.
$e \in$ *connectedByDescendants*$(o, sink) \vee$
$e \in$ *connectedByReachability*$(o, sink, \{$***PointsTo***$\}) \wedge$
flw $\in$ *flows* $\cap$ (*e.flow.descendants*() $\cup$ *reachables*(*e.flow*, {***PointsTo***}))

Conditions based on type+object hierarchy

InstanceOf($objT$: $Type$)
$satisfiedBy(obj : SecObject) :=$ *obj.C.isSubTypeOf*($objT$)

IsInDomain($prntT : Type, prntD : String, objT : Type$)
$satisfiedBy(obj : SecObject) :=$ *obj.C.isSubTypeOf*($objT$) $\wedge$
$obj.parentDom.d = prntD \ \wedge\ obj.parentObj.C.isSubTypeOf(prntT)$

IsChildOf($prntO : SecObject, prntD : String, objT : Type$)
$satisfiedBy(obj : SecObject) :=$ *obj.C.isSubTypeOf*($objT$) $\wedge$
$obj.parentDom.d = prntD \ \wedge\ obj.parentObj = prntO$

Condition based on type+object reachability

IsInstOfRchblFromInstOf(st : *Type*, dt : *Type*, *eType* : $Set\langle$*EdgeType*$\rangle$)
$isSrc :=$ *InstanceOf*(st) $isDst :=$ *InstanceOf*(dt)
satisfiedBy(obj : SecObject) := $\exists\ obj \in$ *getObjectsByCond*($isSrc$) $\wedge$
$\exists\ dobj \in$ *getObjectsByCond*($isDst$) $\wedge\ obj \in$ *reachables*($dobj, eType$)

Condition based on type+object traceability

IsCreateadInMD(*methDecls* : $Set\langle$`MethodDeclaration`$\rangle$)
satisfiedBy(obj : SecObject) :=
$\exists cic$: `ClassInstanceCreation` $\in obj.traceability()$ s.t.
md : `MethodDeclaration` $\in cic.declarations() \ \wedge\ md \in$ *methDecls*

Fig. 4: Selection and property queries on a *SecGraph*

values are stored at the level of the *SecGraph*. In the rest of this paper, when we say the architects assign to an object *TrustLevel.**True***, we mean the architects assign to the object the security property *TrustLevel* with the value ***True***.

Two types of queries traverse the *SecGraph* for a given *Condition*: a selection query, and a property query. A selection query such as *getObjectsByCond()* returns the set of objects or the set of edges that satisfy the *Condition*. A property query such as *setObjectsProperty()* assigns security properties to a set of objects or a set of edges (Fig. 4). The *SecGraph* has several *Condition*s that can be further extended. A *Condition* is a predicate that takes as arguments elements of the object structure, code structure, or property values. The predicate is implemented by the *satisfiedBy* method that is invoked for every element as the query traverses the *SecGraph*.

For example, a *Condition* such as *IsCreatedInMD* uses information from the code structure or the traceability information. A selection query using *IsCreatedInMD* returns the set of *SecObject*s instantiated in the body of one of the methods provided as argument. A query based on type only is less precise than a query that combines information from the code structure and the *SecGraph*. For example, a query using the condition *InstanceOf* returns the set of all *SecObject*s of a type specified as argument. For additional precision, using the condition *IsInDomain* , a selection query returns a subset of these *SecObject*s by specifying the type of the parent object, and name of a parent domain. In addition, a *Condition* can use reachability information. For example a selection query based on the condition *IsInstOfRchblFromInstOf* returns the set of *SecObject*s of a type *dt* that are reachable from *SecObject*s of type *st* through various type of edges.

The *SecGraph* also provides the selection query *getFlowToSink* that takes as arguments two sets of property values *fProps* and *sProps* and returns those dataflow edges or creation edges that refer to a *SecObject* with the property values *fProps* and has as destination a *SecObject* with the property values *sProps*. In the implementation, *getFlowToSink* also considers descendant and reachable *SecObject*s, as well as indirect communication, such that the architects can write complex queries in terms of property values only.

IV. Evaluation

To evaluate the expressiveness of the approach, we formalize as machine-checkable constraints several informal rules in the CERT Oracle Secure Coding Standard for Java [9] and one Android security policy. The study materials are available in an online appendix [15]. We evaluate the following hypothesis:

Machine-checkable constraints in terms of object provenance and indirect communication that analyze the hierarchy and the reachability of abstract objects on a SecGraph can find architectural flaws such as information disclosure and tampering.

A. Constraints that Implement Rules in the CERT Standard

We provide evidence to support the hypothesis using five extended examples based on the CERT rules. Each rule has an informal description, examples of non-compliant code, and available solutions for automated detection. One of the authors of this paper created complete examples from the non-compliant code fragments, then, guided by the informal description, wrote constraints to trigger the automated detection of the architectural flaws. To select the five CERT rules, we used two criteria: the rule is related to architectural flaws, and communication of objects, and the automated detection of the rule was previously unavailable.

MSC00-J. Use `SSLSocket` rather than `Socket` for secure data exchange. Echo uses the `Socket` class to implement the client-server communication over a network. However, such a communication channel does not provide encryption and the communication is vulnerable to eavesdropping.

To find a possible information disclosure, the constraint *insecureDataflows* (line 1) uses object provenance and checks

$$insecureDataflows(g : SecGraph, isSrc : Condition, \quad (1)$$
$$isDst : Condition, isUntrusted : Condition) \quad (2)$$
$$insecureDataflows := \bigcup\{e\} \text{ s.t. } e : DataFlowEdge \in g.edges() \wedge \quad (3)$$
$$e \in g.objProvenanceIndirect(inObj, tObj, tObj, uObj) \text{ where} \quad (4)$$
$$inObj \in g.getObjectsByCondition(isSrc) \quad (5)$$
$$tObj \in g.getObjectsByCondition(isDst) \quad (6)$$
$$uObj \in g.getObjectsByCondition(isUntrusted) \quad (7)$$
$$isSrc := IsInDomain(\texttt{Main}, \texttt{DATA}, \texttt{InputStream}) \quad (8)$$
$$isDst := InstanceOf(\texttt{EchoClient}) \cup InstanceOf(\texttt{EchoServer}) \quad (9)$$
$$isUntrusted := InstanceOf(\texttt{Socket}) \quad (10)$$
$$\{e1, e2\} \subseteq insecureDataflows(G, isSrc, isDst, isUntrusted) \quad (11)$$
$$e1 := \langle \texttt{client:EchoClient}, \texttt{out:PrintWriter}, \texttt{ss:String} \rangle \quad (12)$$
$$e2 := \langle \texttt{server:EchoServer}, \texttt{wrtr:OutputStream}, \texttt{ss:String} \rangle \quad (13)$$

```
1 class EchoClient {
2   void run(){
3     Socket sckt = new Socket("localhost", 9999);
4     OutputStream out = sckt.getOutputStream();
5     PrintWriter out = new PrintWriter(out, true);
6     while ((userInput = System.in.read()) != null) {
7       out.write(userInput); //inf. disclosure
8       System.out.println(userInput) //false positive
9     }
10 }
```

if the same object that flows from an object of type `InputStream` in the domain `DATA` of the class `Main` (line 8) to an object of type `EchoClient` also flows to an object `sckt:Socket`. The constraint also uses indirect object provenance (line 4) and checks that the object flows to the descendants or reachable objects of `sckt:Socket`. Indeed, the same object `ss:String` flows from `client:EchoClient` to the object `out:Writer` from which a descendant of `sckt:Socket` is reachable (line 12). The object `ss` also flows from `server:EchoServer` to a descendant of `sckt:Socket` (line 13). The constraint highlights these two dataflow edges as suspicious. From a suspicious dataflow edge, the architects can trace to the method invocation `out.write(stdInput)` (line 7 in the code fragments). There are no coding defects, however an information disclosure does exist.

If the constraint were to use the type information only and consider all objects of type `OutputStream` as untrusted, it would report four suspicious dataflow edges including two false positives: the dataflow edge from `client` to the standard output stream, and the dataflow edge from `server` to a child of `sslSocket:SSLSocket`.

FIO05-J. Do not expose buffers created using the wrap() or duplicate() methods to untrusted code. The class `CharBuffer` has methods that wrap an array of primitive types into an object of type `CharBuffer`. According to the Java documentation, modifications of an object of type `CharBuffer` cause the array to be modified and vice-versa [9], [16]. Therefore, passing `cb:CharBuffer` to an untrusted object exposes the array that is reachable from `cb`. Instead, an object with a reference to a copy of the array should be passed.

The constraint *bufferExposures* (line 14) selects the objects of type `CharBuffer` created in the body of the method

bufferExposures($g : SecGraph$) (14)
bufferExposures := g.*getFlowToSink*({*IsConfidential*.***True***}, (15)
{*TrustLevel*.***Untrusted***}) (16)
mdWrap = G.*getMethodDecl*(`CharBuffer.wrap`) (17)
mdDuplicate = G.*getMethodDecl*(`CharBuffer.duplicate`) (18)
isInMD := *IsCreatedInMD*({*mdWrap*, *mdDuplicate*}) (19)
$\forall o \in$ *getObjectsByCond*(*isInMD*), (20)
isChild := *IsChildOf*(o, `OWNED`, `char[]`) (21)
G.*setObjectsProperty*(*IsConfidential*.***True***, *isChild*) (22)
G.*setObjectsProperty*(*TrustLevel*.***Untrusted***, *InstanceOf*(`Client`)) (23)
$\{e\} \subseteq$ *bufferExposures*(G) (24)
$e := \langle$`m:Main`, `c:Client`, `cb:CharBuffer`$\rangle$ (25)

`wrap` or duplicate (line 19). Next, the constraint selects the children of these objects that are of type `char[]` (line 19) and assigns to them *IsConfidential*.***True*** (line 22). The constraint also assigns *TrustLevel*.***Untrusted*** to `c:Client`, which represents untrusted code (line 23). The query *getFlowToSink* returns a suspicious edge (line 25) because the dataflow object `cb:CharBuffer` has a points-to edge to the confidential object of type `char[]` and the destination object is `c`. This constraint may also consider objects of types that are similar to `CharBuffer` such as `IntBuffer`.

SER03-J. Do not serialize unencrypted, sensitive data. Serialization saves an object persistently. All the fields reachable from the object are also saved unless the developers mark them as transient. An information disclosure exists if a confidential, unencrypted object is serialized.

The CERT example [9, SER03-J] uses the `Serializable` interface. Since other libraries can also be used for serialization, the *SecGraph* has the property *IsTransient* to keep the constraint independent from a specific library. The constraint *insecureSerializations* (line 26) assigns *TrustLevel*.***Untrusted*** to objects of type `ObjectOutputStream` (line 32), then assigns *IsConfidential*.***True*** and *IsEncrypted*.***False*** to objects of type `Double` in the domain `OWNED` of objects of type `Point` (line 34). Next, it assigns *IsTransient*.***False*** to points-to edges from objects of type `Point` to objects of type `Double` (line 36). Finally, *insecureSerializations* checks if the *EdgeSet* returned by *getFlowToSink* is not empty such that the outgoing points-to edge from the dataflow object has *IsTransient*.***False***.

The constraint finds an information disclosure as a dataflow from `coord` to `oout` (line 38), where the dataflow object is `p:Point` that has references to `value:Double`. The object `value:Double` has the property *IsConfidential*.***True*** and should not be serialized. However, the points-to edge from `p` to `value` has *IsTransient*.***False***, and serialization occurs. Next, the architects can trace to code and find the method invocation `oout.writeObject(p)` in the body of `Coordinates.run`.

By using the condition *IsInDomain*, the constraint assigns property values only to a subset of all the objects of type `Double`, i.e., it does not include children of objects of type `Coordinate`. If the constraint were to use the *InstanceOf* condition instead of *IsInDomain*, all the objects of type `Double` would be considered confidential, and *insecureSerializations* would return one false positive.

insecureSerializations($g : SecGraph$) (26)
insecureSerializations := $\bigcup\{e\}$ e : *DataflowEdge* $\wedge$ (27)
$e \in g$.*getFlowToSink*({*IsConfidential*.***True***, *IsEncrypted*.***False***}, (28)
{*TrustLevel*.***Untrusted***}) s.t. (29)
$\exists pte \in g$.*getOutEdges*(e.*flow*(), ***PointsTo***) $\wedge$ *pte*.*isTransient* = ***False*** (30)
$isostream$:= *InstanceOf*(`ObjectOutputStream`) (31)
G.*setObjectsProperty*(*TrustLevel*.***Untrusted***, $isostream$) (32)
$points$:= *IsInDomain*(`Coordinate`, `OWNED`, `Point`) (33)
isXY := *IsInDomain*(`Point`, `OWNED`, `Double`) (34)
G.*setObjectsProperty*({*IsConfidential*.***True***, *IsEncrypted*.***False***}, *isXY*) (35)
G.*setEdgesProperty*(*IsTransient*.***False***, $IsPtEdge(points, isXY)$) (36)
$\{e\} \subseteq$ *insecureSerializations*(G) (37)
$e = \langle$`coord:Coordinates`, `oout:ObjectOutputStream`, `p:Point`$\rangle$ (38)

insecureLogging($g : SecGraph$) (39)
insecureLogging := g.*getFlowToSink*(*IsConfidential*.***True***, (40)
TrustLevel.***Untrusted***) (41)
rch := *IsInstOfRchblFromInstOf*(`InetAddress`, `String`, {***PointsTo***}) (42)
G.*setObjectsProperty*(*IsConfidential*.***True***, *rch*) (43)
$isLog$:= *InstanceOf*(`Logger`) (44)
G.*setObjectsProperty*(*TrustLevel*.***Untrusted***, $isLog$) (45)
$\{e\} \subseteq$ *insecureLogging*(G) (46)
$e := \langle$`srvr:LogServer`, `log:Logger`, `ip:String`$\rangle$ (47)

FIO13-J. Do not log sensitive information outside a trust boundary. Logging enables a program to collect essential information for debugging. However, logging confidential information such as IP addresses may be prohibited by law in many countries [9, FIO13-J]. The class `Logger` in Java provides basic functionality for logging.

The constraint *insecureLogging* (line 39) checks if dataflow edges exist such that a dataflow object has *IsConfidential*.***True*** and the destination has *TrustLevel*.***Untrusted***. The constraint uses reachability information to assign *IsConfidential*.***True*** only to the objects of type `String` reachable from an object of type `InetAddress` (line 43). The constraint highlights a dataflow edge from `srvr` to `log` that refers to the object `ip:String`. The architects can trace to the line of code `logger.severe(machine.getHostAddress())`, which indeed logs the IP address (line 47).

To assign the *IsConfidential* property, the architects use a condition that considers type and reachability information. If the architects were to use a condition that considers only the type information, *insecureLogging* would return a false positive for the expression `logger.getLogger(name)`, where the `name` argument does not refer to a confidential object.

IDS07-J. Do not pass untrusted, unsanitized data to the Runtime.exec() method. A Java program can invoke external commands provided by the operating system, such as a command that lists a folder's content. An architectural flaw exists if an attacker can tamper with the input and launch arbitrary injected commands with the privileges of the target user. As a solution, the program must sanitize untrusted objects flowing along untrusted communication [9, IDS07-J].

```
commandInjections(g : SecGraph, isUntrusted : Condition,                    (48)
                  isTrusted : Condition, mdExec : MethodDeclaration)        (49)
commandInjections := ⋃{e} s.t. e : DataFlowEdge ∧                           (50)
∃e ∈ g.objectProvenance(main, o, o, rt) where                               (51)
 (∃ main : SecObject ∈ g.getObjectsByCond(isUntrusted) ∧                    (52)
  ∃ rt : SecObject ∈ g.getObjectsByCond(isTrusted)∧                         (53)
  ∃ o : SecObject ∈ g.objects()) ∧                                          (54)
   mInvk : MethodInvocation ∈ edge.traceability() ∧                         (55)
   mInvk.methodDecl = mdExec ∧ edge.isExport()                              (56)
mdExec := G.getMethodDecl(Runtime.exec)                                     (57)
isUntrusted := InstanceOf(Main)                                             (58)
isTrusted := InstanceOf(Runtime)                                            (59)
{e} ⊆ commandInjections(G, isUntrusted, isTrusted, mdExec)                  (60)
e := ⟨dirList:DirList, rt:Runtime, command:String⟩                          (61)
```

We adapted the example from CERT where the attacker can invoke arbitrary commands on Windows by tampering with the arguments of the method `Runtime.exec`. The constraint *commandInjections* (line 48) finds tampering by using object provenance, and traceability information. It uses object provenance to check if the same object that is passed as an argument to `m:Main` is also passed to an object of type `Runtime` (line 51). It also uses traceability (line 55) and highlights only dataflow edges due to the method invocation `Runtime.exec` (line 61). The architects trace to code and find the `rt.exec("cmd.exe /C dir " + dir)`. If the constraint were to use only information about the type of the arguments, it would also return a false positive for `rt.load(filename)`.

B. Finding Architectural Flaws in an Android application

Previous examples presented constraints that found injected architectural flaws. We use Scoria to find an architectural flaw in one Android application that the authors did not write.

An Android application is written in Java and consists of several Activity components that interact with the user using a View. Two objects of type `Activity` communicate with each other using an Intent that represents a message used for inter- and intra-application communication. The following security policy is documented both by researchers [17] and in the Android documentation [18]: *"Do not put sensitive data into Intents used to start Activities [...]. For example, applications with the GET_TASKS permission are able to see ActivityManager.RecentTaskInformation which includes the base Intent used to start Activities"*.

In this case study, we evaluate if Scoria can find an architectural flaw that violates this security policy. As our subject system, we selected Universal Password Manager Android app (UPMA) version 1.13, a 4KLOC open-source Android app that encrypts and stores usernames, passwords and confidential notes in a database protected by a master password. We selected UPMA because a security vulnerability would impact a large number of users interested in protecting their confidential information. Also, UPMA is actively maintained and has a repository that we can use to confirm if developers changed the code to fix the suspected architectural flaws.

```
insecureIntents(g : SecGraph)                                               (62)
insecureIntents := g.getFlowToSink({IsConfidential.True,                    (63)
                                    IsEncrypted.False}, {TrustLevel.Untrusted})  (64)
isIntent := InstanceOf(Intent)                                              (65)
setObjectsProperty(TrustLevel.Untrusted, isIntent)                          (66)
isPwd := IsInDomain(AccountInformation, PWD, String)                        (67)
setObjectsProperty({IsConfidential.True, IsEncrypted.False}, isPwd)         (68)
{e1, e2, e3, e4} ⊆ insecureIntents(G)                                       (69)
e1 := ⟨vad:VADetails, intnt:Intent, vad:VADetails⟩                          (70)
e2 := ⟨vad:VADetails, s:String, pwd:String⟩                                 (71)
e3 := ⟨al:AccountsList, s:String, pwd:String⟩                               (72)
e4 := ⟨aea:AddEditAccount, s:String, pwd:String⟩                            (73)
```

We implement the above security policy using the constraint *insecureIntents*. We then execute the constraint on the *SecGraph* extracted from annotated code of UPMA.

Information Disclosure in UPMA. A brief inspection of the code revealed that the class `AccountInformation` has the fields `username` and `password` to store confidential information in plain text. The constraint *insecureIntents* (line 62) uses object hierarchy to distinguish between different objects of type `String`, where only the password and account represent confidential information (line 67). Thus, a potential information disclosure exists for those objects of type `Intent` that receive such information. The constraint uses the *getFlowToSink* query (lines 63), which in turn uses indirect communication through object hierarchy and object reachability to check that a dataflow object contains or can reach into confidential information.

Indeed, in version 1.14, the developer changes the class `VADetails` and set a security flag in the method `onCreate`[1]. The value of the security flag is checked at runtime by the Android platform to ensure that task managers and screenshots cannot disclose information if the View of this Activity is left open. The following commit is associated with this change and shows a plus sign at the beginning of the lines added in `VADetails`. Scoria would still find the architectural flaw in UPMA 1.14 because it relies only on code structure information, and the architectural flaw would be a false positive. To prevent the false positive, the architects can change the property value for the object `intnt:Intent`.

```
GIT commit @bd53100
Hide accnt details in screenshots and task manager
Security precaution to ensure a third party can't view
account details left open when UPM was last used.

src/com/u17od/upm/ViewAccountDetails.java
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  +if (Utilities.VERSION.SDK_INT >=
        Utilities.VERSION_CODES.HONEYCOMB) {
  +  getWindow().setFlags(LayoutParams.FLAG_SECURE,
                          LayoutParams.FLAG_SECURE);
  +}
  setContentView(R.layout.view_account_details);
```

[1]https://github.com/adrian/upm-android/commit/bd53100. `VADetails` is a shorthand for `ViewAccountDetails`.

TABLE I: Summary of constraints that implement rules in the CERT Oracle Secure Coding Standard for Java

CERT-ID	vulnerability	object provenance	object transitivity	object hierarchy	object reachability	object traceability	indirect communication	suspicious edges	avoided false positives	object security properties	edge security properties
MSC00-J	inf. disclosure	✓	✓	✓	✓		✓	2	2	0	0
FIO05-J	inf. disclosure			✓	✓	✓		1	1	2	0
SER03-J	inf. disclosure			✓				1	1	2	2
FIO13-J	inf. disclosure				✓			1	1	2	0
IDS07-J	tampering	✓	✓			✓		1	1	0	0

C. Discussion

The results show that the *SecGraph* is expressive and can detect security vulnerabilities that are architectural flaws. Two of the constraints use object provenance and do not require security properties. For the others, the architects need to assign security properties to only a few of the *SecElement*s and use the defaults elsewhere (Table I). The constraints use the information provided by both types of edges in the *SecGraph*. While all the constraints highlight dataflow edges, three of them use reachability information through points-to edges.

Both evaluations highlight the fact that ARA requires a runtime architecture—not a code architecture. For example, *bufferExposures* distinguishes between different objects of the same type with different property values such as the two *SecObject*s of type `char[]` and the two *SecObject*s of type `CharBuffer`. By using the runtime architecture, the constraints return fewer false positives. Some constraints also use information from the code architecture, such as subtyping information or method names.

In recent work [4], OOGs were abstracted into a runtime architecture represented in an architectural description language. ARA checks such as information disclosure were expressed as predicates that check only the presence or the absence of communication, which is not enough. In Scoria, not only can architects check the presence or absence of communication, they can also express constraints about the information content of the communication between objects because a dataflow edge refers to an abstract object. For example, in the constraint *insecureLogging*, the *SecGraph* has two dataflow edges from `srvr:LogServer` to `log:Logger`. If the constraint were to check only for the absence of communication, it would be too restrictive. The communication from `srvr` to `log` is required, otherwise the system becomes vulnerable to other security vulnerabilities such as repudiation. Only one dataflow edge leads to information disclosure, and the constraint can tell the two edges apart by tracking the object flowing along the edge and using object provenance.

D. Limitations

Our results may not generalize to other programs and architectural flaws. The CERT examples cover both tampering and information disclosure to evaluate the expressiveness of constraints. UPMA has the size of a typical Android application and is smaller than a typical Java application [19]. Scoria is focused on finding architectural flaws related to connectors in a runtime architecture. The constraints are based on the runtime and code architecture and could also consider information from other architectures such as deployment architecture. Other architectural flaws are related to the state of the object, and the *SecGraph*, which focuses only on runtime structure, does not capture such information.

An *SecGraph* merges multiple objects of the same type in one domain, which may lead to false positive edges due to a method invocation in a super class [20]. None of the dataflow edges highlighted by the constraints in the CERT examples is a false positive. However, one of four dataflow edges highlighted by *insecureIntents* in UPMA is a false positive due to excessive merging of objects of type `String` in the domain `SHARED`. A points-to edge exists from `intnt:Intent` to `s:String` in `SHARED`. A creation edge from `aea:AddEditAccount` to `s:String` refers to the confidential object `pwd:String` (line 73). The object `s:String` excessively merges objects of type `String` including the object of type `String` created when the UPMA user edits an account. Sending the password in plain text to be changed by the user is part of the intended UPMA functionality and does not constitute an architectural flaw. The architects can avoid this false positive by placing these objects of type `String` in the `UI` domain to avoid excessive merging.

V. Related Work

Static analysis of legacy object-oriented code. Many static analyses automatically extracted dataflow edges between objects in non-hierarchical object graphs [14], [20], [21]. For example, Spiegel [14] builds a type graph, which he then unfolds into an object graph. These analyses do not track the content of dataflow edges as references to objects. Static analysis frameworks such as SOOT [22] and WALA [23] have initially been used to compare different variants of alias analyses. Recently, they have been used to find security vulnerabilities in Android applications [24], [25], [26].

Analyses that find local coding defects. Analyses that are fully automatic usually avoid a manual setup such as annotations [27]. They traverse the representation of the program such as the Abstract Syntax Tree (AST) looking for vulnerabilities localized in a class or a method that are consequences of coding defects. To find a local defect, a shallow analysis does not have to consider the global information about the context in which the method or class is used. Vulnerabilities are described as patterns or AST visitors [2]. Although these pattern-based approaches provide automated detection, the output of a static analysis tool still requires human evaluation [28]. While an AST-based analysis supports legacy, the AST representation is often too detailed to find architectural flaws.

Annotation-based approaches. To enhance the code with an authorization mechanism such as role-based access control many approaches use Java annotations. However, these annotations are only checked at runtime. In contrast, a typechecker ensures that annotations and code are consistent at compile-time. For example, Object-sensitive Role Based Access Control (ORBAC) [29] uses a typechecker to enforce the security policy that a user cannot read confidential data of other users unless their roles allow him to do so. However, ORBAC does not handle object hierarchy and the role-based protection is not propagated to children objects, although the children may represent confidential data.

Dynamic analysis of legacy code. A dynamic analysis can extract a hierarchical object graph using as input heap snapshots [30], [31], [32]. However, dynamic analysis can infer a hierarchy based on only strict encapsulation where a child object at the lower levels is accessible only through its parent. In Scoria, however, a parent object can have multiple child objects that are conceptually part of it without being dominated by the parent. Although a dynamic analysis has fewer false positives than a static analysis, a dynamic analysis works only if the architects are able to write unit tests to expose the vulnerability. The goal of ARA is to find vulnerabilities that may not be already known.

Operating system-level approaches. Many researchers focus on preventing vulnerabilities at the level of the operating system. However, developers may still introduce vulnerabilities at the application level. For example, Mai et al. [33] propose a lightweight operating system (ExpressOS) and prove that it is free of vulnerabilities. In the evaluation, the authors consider about 300 vulnerabilities listed by the Common Weakness Enumeration Initiative (CWE) [34]. Although ExpressOS prevented all OS-level vulnerabilities, it cannot prevent some application-level architectural flaws.

Architectural-level approaches. Based on ARA, Sohr et al. propose an architectural centric approach to find architectural flaws [35]. We share the same motivation in our work, but the solutions proposed differ. Their approach uses Bauhaus [36] that extracts a code architecture rather than a runtime architecture. In a recent work [5], Bauhaus approximates the runtime architecture for two web applications, which are then used to find architectural flaws. However, their approximated architecture does not distinguish between components of the same type used in different contexts, such as configuration files. In Berger et al. [5], the architects can write constraints in the Object Constraint Language [37]. However, the edges on the runtime architecture have no labels, which makes it difficult for the architects to write constraints about the information content of a dataflow. Indeed, the evaluation is conducted in a style similar to Reflexion models [38], focusing only on the presence or absence of communication.

Other researchers also explored the idea formally defining a vulnerability as a set of constraints [39]. Almorsy et al. [39], [40] used OCL to write constraints on various UML models such as sequence and class diagram extracted from program AST. However, the static analysis that extracts the sequence diagram is an AST visitor that does not consider aliasing. Therefore, in the extracted sequence diagram, the object type is the same as the declared type of the reference to the object, which can be an interface. Moreover, an interface can have multiple implementations, and each object of these classes can have distinct property values.

He et al. [41] formalize the software architectural model (SAM) using Petri nets. Similar to SAM, a *SecGraph* is hierarchical, but is less restrictive than a Petri net that requires two disjoint sets of nodes: transitions and places. Dataflow objects are similar to transitions, but with fewer restrictions, i.e., dataflow objects can also be nodes in the *SecGraph*.

Taint Analyses Similar to Scoria, a static taint analysis [26], [42], [43], [44] finds information disclosure and tampering. By tracking how the values flow from a tainted method, such as the one that reads from user input, into a sink method without being sanitized. A static taint analysis uses an inter-procedural control flow graph that has nodes that represent variables and edges that account for conditional and assignment statements and method invocations. A sound static taint analysis considers aliasing using a separate alias analysis provided by a static analysis framework such as SOOT [22] or WALA [23]. However, an alias analysis may not scale if it considers all the variables. To improve scalability, FlowDroid [44] uses an on-demand may-alias analysis [45] that considers only the variables with assigned security properties. A taint analysis is thus less flexible than Scoria because it hard-codes the constraints and the security properties.

Querying object graphs. Researchers proposed approaches that support querying object graphs to find software defects and architectural flaws. For example, Object Query Language (OQL) [46] queries are on a snapshot of a heap. Program Query Language (PQL) queries allow developers and architects to reason about the results of an aliasing analysis [47]. Both OQL and PQL allow reasoning about reachability. However, since the object graphs are flat, they do not support queries in terms of object hierarchy or object provenance.

VI. Conclusion

To support ARA, we propose Scoria, a semi-automatic approach that finds architectural flaws using machine-checkable constraints. We define a sound, hierarchical security graph with dataflow edges that refer to abstract objects. By automatically tracking object hierarchy and object reachability, Scoria enables reasoning about object provenance and indirect communication that lead to architectural flaws such as information disclosure and tampering. To evaluate Scoria, we enforce a subset of rules in the CERT standard for Java and an Android security policy using machine-checkable constraints that find architectural flaws in several extended examples and in one Android application.

Acknowledgement

This work was supported in part by the Army Research Office under Award No. W911NF-09-1-0273.

References

[1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.

[2] University of Maryland, "FindBugs™ – Find Bugs in Java Programs," http://findbugs.sourceforge.net/, 2007.

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

[4] M. Abi-Antoun and J. M. Barnes, "Analyzing Security Architectures," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010, pp. 3–12.

[5] B. Berger, K. Sohr, and R. Koschke, "Extracting and Analyzing the Implemented Security Architecture of Business Applications," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 285–294.

[6] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," in *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2009, pp. 321–340.

[7] R. Vanciu and M. Abi-Antoun, "Ownership Object Graphs with Dataflow Edges," in *Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 267–276.

[8] P. Torr, "Demystifying the Threat-Modeling Process," *IEEE Security & Privacy*, vol. 3, no. 5, pp. 66–70, 2005.

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

[10] M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed. Microsoft Press, 2003.

[11] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," in *International Conference on Software Engineering (ICSE)*, 2002, pp. 187–197.

[12] J. Jürjens and P. Shabalin, "Automated verification of UMLsec models for security requirements," in *UML: Modelling Languages and Applications*, 2004, pp. 365–379.

[13] T. Lodderstedt, D. A. Basin, and J. Doser, "SecureUML: a UML-Based Modeling Language for Model-Driven Security," in *UML: Modelling Languages and Applications*, 2002, pp. 426–441.

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

[15] R. Vanciu and M. Abi-Antoun, "Scoria Evaluation. Online Appendix," http://www.cs.wayne.edu/~mabianto/scoria/, 2013.

[16] Oracle, "Java Platform Standard Edition 6 API Specification," 2012. [Online]. Available: http://docs.oracle.com/javase/6/docs/api/

[17] J. Burns, "Mobile Application Security on Android," Black Hat, 2009. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-09/BURNS/BHUSA09-Burns-AndroidSurgery-PAPER.pdf

[18] Google, "Android API Guides," http://developer.android.com/training/articles/security-tips.html, 2012.

[19] R. Minelli and M. Lanza, "Software Analytics for Mobile Applications – Insights & Lessons Learned," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 144–153.

[20] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized Object Sensitivity for Points-To Analysis for Java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[21] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," *Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 156–169, 2001.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot Framework for Java Program Analysis: A Retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS)*, 2011.

[23] IBM, "T. J. Watson Libraries for Analysis (WALA)," 2012. [Online]. Available: http://wala.sourceforge.net

[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 274–277.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *USENIX Conference on Security*, 2011, pp. 21–21.

[26] A. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid: Automated Security Certification of Android Applications," Univ. of Maryland, Tech. Rep., 2009, http://babu.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf.

[27] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM (CACM)*, vol. 53, no. 2, pp. 66–75, 2010.

[28] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[29] J. Fischer, D. Marino, R. Majumdar, and T. Millstein, "Fine-Grained Access Control with Object-Sensitive Roles," in *European Conference on Object-Oriented Programming (ECOOP)*, 2009, pp. 173–194.

[30] A. Lienhard, S. Ducasse, and T. Gîrba, "Taking an Object-Centric View on Dynamic Information with Object Flow Analysis," *Computer Languages, Systems and Structures (COMLAN)*, vol. 35, no. 1, pp. 63–79, 2009.

[31] N. Mitchell, E. Schonberg, and G. Sevitsky, "Making Sense of Large Heaps," in *European Conference on Object-Oriented Programming (ECOOP)*, 2009, pp. 77–97.

[32] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich, "Abstracting Runtime Heaps for Program Understanding," *Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 774–786, 2013.

[33] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying Security Invariants in ExpressOS," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 293–304.

[34] MITRE Corporation, "The Common Weakness Enumeration (CWE) Initiative," 2012. [Online]. Available: http://cwe.mitre.org

[35] K. Sohr and B. Berger, "Idea: Towards Architecture-Centric Security Analysis of Software," in *ESSOS*, 2010, pp. 70–78.

[36] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering," in *International Conference on Reliable Software Technologies (Ada-Europe)*, 2006, pp. 71–82.

[37] OMG, "Object Constraint Language," 2013. [Online]. Available: http://www.omg.org/spec/OCL/

[38] G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation" *Transactions on Software Engineering (TSE)*, vol. 27, no. 4, pp. 364–380, 2001.

[39] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 100–109.

[40] ——, "Automated Software Architecture Security Risk Analysis using Formalized Signatures," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 662–671.

[41] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, "Formally Analyzing Software Architectural Specifications Using SAM," *J. Systems & Software*, vol. 71, no. 1, pp. 11–29, 2004.

[42] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 228–241.

[43] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective Taint Analysis of Web Applications," in *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 87–97.

[44] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Highly Precise Taint Analysis for Android Applications," EC SPRIDE, TU Darmstadt, Tech. Rep., 2013, https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CASED/Publikationen/TUD-CS-2013-0113.pdf.

[45] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications," in *Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013, pp. 210–225.

[46] VisualVM, "Analyzing a Heap Dump Using Object Query Language (OQL)," 2013. [Online]. Available: http://visualvm.java.net/oqlhelp.html

[47] M. Martin, B. Livshits, and M. S. Lam, "Finding Application Errors and Security Flaws Using PQL: a Program Query Language," in *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2005, pp. 365–383.