Extracting Dataflow Objects and other Flow Objects

By Radu Vanciu and Marwan Abi-Antoun
Department of Computer Science
Wayne State University
Detroit, Michigan, USA
Architectural Risk Analysis (ARA) finds architectural flaws that lead security vulnerabilities [Howard and Lipner’06]

• Architects use forest-level view (not reading code)
  • Runtime architecture – not code architecture
  • Architects assign security properties to component instances and write constraints

• Limitations of ARA approaches
  • Limited support for reverse engineering forest-level view from code
  • Runtime architecture is missing or inconsistent with code
  • Lack of traceability to code from runtime architecture
Scoria approach to support ARA

- Extract forest-level view
  - Extract object graph that shows instances – not classes
  - Object graph has traceability to code
  - Use object hierarchy to achieve architectural abstraction

- Abstraction by object hierarchy
  - Architecturally significant objects near top of hierarchy
  - Implementation details (data structures) further down

- Use static analysis to extract object graph
  - Static analysis to achieve soundness
  - Security requires worst case analysis

- Soundness: represent all objects and relations that may exist at runtime, in any possible execution
At runtime, object-oriented program appears as runtime object graph.
Abstract multiple runtime objects into an abstract object

- Each runtime object has exactly one representative in extracted object graph
Abstract multiple runtime objects into an abstract object

- Each runtime object has **exactly one** representative in extracted object graph
Abstract edge is between abstract objects

- Runtime edge between two objects maps to edge between representative of two objects
Dataflow edge between abstract objects refers to abstract object
Unique edge representative: Distinct runtime edges refer to distinct runtime object
Abstract edge refers to representative of the runtime object that runtime edge refers to
Extracting dataflow edges

- Nodes represent abstract objects
- Edges represent usage of objects:
  - method invocation
  - field read
  - field write

---

```java
class A{
    B b; C c; D d; E e;
    void m1(){
        d.setB(b);
    }
    B m2(){
        return d.getB();
    }
    C m3(){
        return b.c;
    }
    void m4(){
        b.c = c;
    }
    D m5(){
        return e.m(b,c);
    }
}
```
Hierarchy: organize object hierarchically

Flat object graph
[Spiegel, Thesis ’02]

Ownership Object Graph with points-to edges
[Abi-Antoun and Aldrich, OOPSLA’09]
Use ownership types to guide extraction of object hierarchy [Aldrich and Chambers, ECOOP’04]

- Hierarchical organization of objects
  - Not available in plain Java code
  - Use annotations
  - Annotations implement type system

- Assign each object to one **ownership domain**
  - **Domain:** defn. a **named, conceptual group of objects**
  - Domain is similar to architectural runtime tier

- Typechecker ensures annotations and code are consistent
- Annotations are local/modular (checked one class at a time)
- Use language support for annotations to handle legacy code
Static analysis abstractly interprets program with annotations

- Static analysis: ScoriaX
  - Whole program analysis
  - Construct a global, hierarchical Ownership Object Graph
  - Object/domain hierarchy
  - Dataflow edges
ScoriaX extracts dataflow edges and flow objects

- Extracting dataflow edges require interpreting method invocations, field reads and field writes
- Local variables, method parameters could be `lent`
  - In Ownership Domains, `lent` = borrowed object
- Method parameters, return value could be `unique`
  - In Ownership Domains, `unique` = objects passed linearly

- ScoriaX attempts to resolve `lent` and `unique`
  - It may resolve to domain declared in code
  - If not, create special domain (child of creator object) *with an automatically generated name and without a domain declaration*
  - **Flow Object**: *defn. an object that resides in a special domain*
**ScoriaX is a kind of points-to analysis**

- Approximates a set of runtime objects which a reference in the program may alias
- Object-sensitive parameterized by constant $k$ [Milanova et al. TSE’05]
- $\text{new } C_1() \rightarrow \ldots \rightarrow \text{new } C_k()$  
  \[ \text{class } C_k\{ \text{new}^{h} C(); \} \]
- For $k=1$: merges all objects created at object allocation site into same equivalence class

- ScoriaX distinguishes between allocations in different domains and that have different domain parameters

\[ \text{new } C<\text{p}_{owner} \text{ p}_{params}^...>() \]

- $\text{p}_{owner}$: owner domain
  - locally declared domain $d$
  - formal domain parameter
  - shared
  - unique

- $\text{p}_{params}$: additional domain parameters
  - locally declared domain $d$
  - formal domain parameter
  - shared
Analysis does multiple passes over code

Pass1: Extracts objects and domains
Pass2: Extracts dataflow edges
  - Also extracts value flow graph (FG)
Pass3: Summarizes value flow graph and compute transitive flow
  - $FG^* = \text{summarize}(FG)$
  - $FG_p = \text{propagateAll}(FG)$
  - Only for references declared as lent or unique (improves scalability)
Pass4: Attempts to resolve lent and unique using propagated flow graph
  - extracts more dataflow edges
  - extracts flow objects

At every pass:
  - start from the root class
  - stop at a fixed-point
Pass1: Extract objects and domains

- Track **this** $\rightarrow$ abstract object **Othis**
- Track binding of formal domain parameter $\rightarrow$ actual domain

```c
[y = new C<T>::DOM>(); x = y.m(a);
```

```c
[C::owner $\rightarrow$ DOM]
class C<owner>{
  X<Y> m(A<Bf> fa) { ... return ret; }
}
```
Pass1: Extract objects and domains

```java
m = new Main<shared>();

class Main<owner>{
    domain DATA, LOGIC1, LOGIC2;
    EActivity<LOGIC1,LOGIC2,DATA> e;
    e = new EActivity<LOGIC1,LOGIC2,DATA>();
}

class EActivity<owner,L,D>
    extends Activity{
}

class Activity<owner,L,D>{
    domain MSG;
}
```
Pass 1: Extract objects and domains

class EActivity<owner,L,D> extends Activity<owner,L,D> {
}

SHARED
m:
Main

DATA LOGIC1 LOGIC2

e:
EActivity

MSG
Pass 1: Extract objects and domains

```java
class EActivity<owner,L,D> extends Activity<owner,L,D> {
    Activity<L,L,D> v;
    v = new ViewActivity<L,L,D>();
}
```

```java
class Activity<owner,L,D>{
    domain MSG;
}
```
Pass1: Extract objects and domains

```java
[this \rightarrow O_e]
[Activity::owner \rightarrow LOGIC1,
 EActivity::L \rightarrow LOGIC2,
 EActivity::D \rightarrow DATA]

class EActivity<owner,L,D>
    extends Activity<owner,L,D> {
    Activity<L,L,D> v;
    v = new ViewActivity<L,L,D>();
}

[this \rightarrow O_v]
[Activity::owner \rightarrow LOGIC2,
 Activity::L \rightarrow LOGIC2,
 Activity::D \rightarrow DATA]

class Activity<owner,L,D>{
    domain MSG;
}
```
Pass 2: extract value Flow Graph (FG)

- Nodes ($O, x, B$) where
  - Object $O$ for domain-sensitivity
  - Reference $x$ of type $C<B...>$
  - $B$ can be:
    - locally declared domain $d$
    - formal domain parameter
    - shared
    - unique
    - lent

- Edges ($O_1, x, B_1$) $\rightarrow$ ($O_2, y, B_2$)

- Track assignments $\rightarrow$ value flow edge
  - $x = y$ ($O_{this}, y, By$) $\rightarrow$ ($O_{this}, x, Bx$)
  - $x = y.m(a)$
    ($O_{this}, y, By$) $\rightarrow$ ($O_y$, this, owner)
    ($O_{this}, a, B_a$) $\rightarrow$ ($O_y$, fa, $B_f$)
    ($O_y$, ret, $B_r$) $\rightarrow$ ($O_{this}, x, Bx$)
  - $x = y.f$ ($O_y$, f, $B_y$) $\rightarrow$ ($O_{this}, x, B_f$)
  - $x.f = y$ ($O_{this}, y, By$) $\rightarrow$ ($O_x$, f, $B_f$)

- Attempt to resolve lent and unique
  - Resolve unique: forward in value flow
  - Resolve lent: backward in value flow
Pass2: Extract objects and domains

```java
class EActivity<owner, L, D> extends Activity<owner, L, D> {
    Activity<L, L, D> v = ...;
    Intent<unique> i = new Intent<unique>();
    v.start(i);
}
```

```java
class Activity<owner, L, D> {
    domain MSG;
    void start(Intent<MSG> intnt);
}
```

```java
class Intent<owner> {
}
```

Legend
- `O`, `x`, `B`: value flow node
- `value flow edge`
Pass 2: Extract dataflow edges

```java
// this → O_e

class EActivity<owner,L,D>
    extends Activity<owner,L,D> {
    Activity<L,L,D> v = ...;
    Intent<unique> i = new Intent<unique>();
    v.start(i);
}

// this → O_e
[EActivity::owner→LOGIC1,
 EActivity::L→LOGIC2,
 EActivity::D→DATA ]

lookup(Activity<L,L,D>) = {O_v}
solveUnique(O_e, Intent) = {v.MSG}
lookup(Intent<unique>) = {O_i}
```

O_e, i, unique → O_v, intnt, v.MSG
Pass2: Extract objects and domains

class EActivity<owner,L,D> extends Activity<owner,L,D> {
    LctMngr<L> lm = this.getMngr("LOC");
    FileMngr<D> fm = this.getMngr("FILE");
}

class Activity<owner,L,D> {
    Mngr<unique> getMngr(String<SHARED> s){
        if (s.equals("FILE"))
            return new FileMngr<unique>();
        else
            return new LctMngr<unique>();
    }
}
Pass2: Extract dataflow edges

\[
\text{class EActivity<owner,L,D>}
\]
\[
\text{extends Activity<owner,L,D> } \{ ...
\]
FileMngr<D> fm = \text{this}.getMngr("FILE")
File<lent> tFile = fm.read("history")
\}

\[
\text{class FileMngr<owner>}
\]
\[
\text{extends Mmgr<owner> } \{
\text{domain DOM;}
\text{File<DOM> read(String<SHARED> s) }\{
\text{return new File<DOM>();}
\}
\}

\[
O_{\text{fm}}, \text{ret, DOM} \rightarrow O_{\text{e}}, tFile, \text{lent}
\]

\[
\text{SHARE}\]

\[
\text{DATA} \rightarrow \text{LOGIC1} \rightarrow \text{LOGIC2} \rightarrow \text{lm: LctMngr}
\]

\[
e: \text{EActivity}
\]
\[
f: \text{FileMngr}
\]
\[
\text{f: File}
\]
\[
\text{DOM}
\]
\[
i: \text{Intent}
\]
\[
v: \text{ViewActivity}
\]
\[
\text{MSG}
\]
\[
\text{MSG}
\]
Pass2: Extract dataflow edges

```java
class EActivity<owner,L,D> extends Activity<owner,L,D> {
    ... File<lent> tFile = fm.read("history");
    Service<D> service = new Service<D>();
    File<lent> eFile = service.run(tFile);

    File<unique> run(File<lent> f){
        return service.encrypt(f);
    }
}
class Service<owner> {
    domain OWNED;
    Cipher<OWNED> cipher;
    File<unique> encrypt(File<lent> x){
        return cipher.doFinal(x);
    }
}
class Cipher<owner> {
    File<unique> encrypt(File<lent> x){
        return new File<unique>();
    }
}
```
Pass2: Extract value flow graph

class EActivity<owner,L,D> ... {
    File<lent> tFile = fm.read("history");
    File<lent> eFile = service.run(tFile);
    File<unique> run(File<lent> f){
        return service.encrypt(f);
    }
}
class Service<owner> {
    domain OWNED;
    Cipher<OWNED> cipher;
    File<unique> encrypt(File<lent> x){
        return cipher.doFinal(x);
    }
}
class Cipher<owner> {
    File<unique> doFinal(File<lent> x){
        return new File<unique>();
    }
}
Pass2: Extract value flow graph

class EActivity<owner,L,D> ... {
    File<lent> tFile = fm.read("history");
    File<lent> eFile = service.run(tFile);
    File<unique> run(File<lent> f){
        return service.encrypt(f);
    }
}
class Service<owner> {
    domain OWNED;
    Cipher<OWNED> cipher;
    File<unique> encrypt(File<lent> x){
        return cipher.doFinal(x);
    }
}
class Cipher<owner> {
    File<unique> encrypt(File<lent> x){
        return new File<unique>();
    }
}
Pass3: Compute propagated flow graph

```java
class EActivity<owner,L,D> {... {
  File<lent> tFile = fm.read("history");
  File<lent> eFile = service.run(tFile);
  [solveLent(Oe,File) = {fm.DOM}]
  File<unique> run(File<lent> f) {
    return service.encrypt(f);
  }
}

class Service<owner> {
  domain OWNED;
  Cipher<OWNED> cipher;
  [solveLent(Oservice,File) = {fm.DOM}]
  File<unique> encrypt(File<lent> x) {
    return cipher.doFinal(x);
  }
}

class Cipher<owner> {
  [solveLent(Ocipher,File) = {fm.DOM}]
  File<unique> doFinal(File<lent> x) {
    return new File<unique>();
  }
}
```
Pass3: Compute propagated flow graph

class EActivity<owner,L,D> ... {
    File<lent> tFile = fm.read("history");
    File<lent> eFile = service.run(tFile);
    File<unique> run(File<lent> f){
        return service.encrypt(f);
    }
}
class Service<owner> {
    domain OWNED;
    Cipher<OWNED> cipher;
    File<unique> encrypt(File<lent> x){
        return cipher.doFinal(x);
    }
}
class Cipher<owner> {
    File<unique> encrypt(File<lent> x){
        return new File<unique>();
    }
}
Pass4: Extract dataflow edges

- Resolve **lent** to `fm.DOM`, the parent of `f:File`

- Extract another dataflow edge that refers to `f:File`
Pass 4: Extract dataflow edges

- Analysis cannot resolve unique do a domain
- Create a fresh ODomain **UNIQUE** as child of cipher:Cipher
- Create flow object **ret:File** as child of **UNIQUE**

- Extract 3 dataflow edges that refer to same flow object
- Same type different objects:
  - **ret:File** encrypted file
  - **f:File** unencrypted file
  - Encrypted file flows to **i:Intent**
Challenges of static analysis that extract object graph with dataflow edges for security

- Soundness
- Hierarchy
- Summarization
- Support for legacy code
- **Precision**
- Aliasing
Precision: extract multiple abstract objects for same object allocation expression

- Using a simple transitive closure of value flow graph can be imprecise

- Imprecision occurs when client code invokes same method multiple times or in different contexts

- Make the value flow domain-sensitive, call-site context sensitive, and flow insensitive

- Distinguish between receivers of method using domains

- Consider same invocation different contexts
Domain-insensitive value flow graph

class A<owner,F> {
    Integer<F> f;
    void set(Integer<F> num) {
        f = num;
    }
    Integer<F> get() { return f; }
}
class Main<owner>{
    void main() {
        domain DATA, DOM1, DOM2;
        Integer<DOM1> n1 = new Integer(-1);
        A<DATA, DOM1> a1 = new A();
        a1.set(n1);
        Integer<DOM2> n2 = new Integer(2);
        A<DATA, DOM2> a2 = new A();
        a2.set(n2);
        [resolveLent(m,Integer)={DOM1,DOM2}]
        Integer<lent> dest = a2.get();
        dest.compareTo(n1);
    }
}
Domain-insensitive value flow graph

class A<owner,F> {
    Integer<F> f;
    void set(Integer<F> num) {
        f = num;
    }
    Integer<F> get() { return f; }
}
class Main<owner>{
    void main() {
        domain DATA, DOM1, DOM2;
        Integer<DOM1> n1 = new Integer(-1);
        A<DATA, DOM1> a1 = new A();
        a1.set(n1);
        Integer<DOM2> n2 = new Integer(2);
        A<DATA, DOM2> a2 = new A();
        a2.set(n2);
        Integer[lent] dest = a2.get();
        dest.compareTo(n1);
    }
}
Domain-sensitive value flow graph

class A<owner,F> {  
    Integer<F> f;
    void set(Integer<F> num) {  
        f = num;
    }  
    Integer<F> get() { return f; }
}  

class Main<owner>{
    void main() {
        domain DATA,DOM1,DOM2;
        Integer<DOM1> n1= new Integer(-1);
        A<DATA,DOM1> a1 = new A();
        a1.set(n1);
        Integer<DOM2> n2 = new Integer(2);
        A<DATA,DOM2> a2 = new A();
        a2.set(n2);
        [resolveLent(m,Integer)={DOM2}]
        Integer<lent> dest = a2.get();
        dest.compareTo(n1);
    }
}
Domain-sensitive value flow graph

class A<owner,F> {  
    Integer<F> f;
    void set(Integer<F> num) {  
        f = num;
    }
    Integer<F> get() { return f; }  
}
class Main<owner>{
    void main() {
        domain DATA,DOM1,DOM2;
        Integer<DOM1> n1= new Integer(-1);
        A<DATA,DOM1> a1 = new A();
        a1.set(n1);
        Integer<DOM2> n2 = new Integer(2);
        A<DATA,DOM2> a2 = new A();
        a2.set(n2);
        [resolveLent(m,Integer)={DOM2}]  
        Integer<lent> dest = a2.get();
        dest.compareTo(n1);
    }
}
Extract multiple dataflow edges for same invocation

class A<owner,F> {
    Integer<F> f;
    void set(Integer<F> num){f = num;}
    Integer<F> get() { return f; }
}
class B<owner,F> {
    domain OWNED;
    A<OWNED,F> A a = new A();
    void assign(Integer<F> n) {
        a.set(n);
    }
}
class Main<owner>{
    domain DATA, DOM1, DOM2;
    void main() {
        Integer<DOM1> n1= new Integer(-1);
        B<DATA,DOM1> b1 = new B();
        b1.assign(n1);
        Integer<DOM2> n2 = new Integer(2);
        B<DATA,DOM2> b2 = new B();
        b2.assign(n2);
        Integer<lent> dest = b2.a.get();
        dest.compareTo(n1);
    }
}
Aliasing: No one runtime object has two representatives in OGraph

class EActivity<owner,L,D> ... {
    File<lent> tFile = fm.read("history");
    File<lent> eFile = service.run(tFile);
    File<unique> run(File<lent> f){
        return service.encrypt(f);
    }
}
class Service<owner> {
    domain OWNED;
    Cipher<OWNED> cipher;
    File<unique> encrypt(File<lent> x){
        return cipher.doFinal(x);
    }
}
class Cipher<owner> {
    File<unique> encrypt(File<lent> x){
        return new File<unique>();
    }
}
Extended example

• Universal Password Manager Application (UPMA) (4KLOC)
  • Encrypts and stores passwords in file
  • Android application
  • Annotate only code of UPMA, not Android framework
  • Follows State-Logic-Display architectural pattern
Distinguish between database file and encryption key file. Dataflow edge refers to database file.
Flow object: UPMA transforms password to array of byte before encryption
Discussion: More precise ownership types produce more precise dataflow edges

class Main{
    domain LOGIC2, UI, LOGIC, DATA;
    new UPMAApplication<LOGIC, UI, LOGIC, DATA>();
    new CreateNewDatabase<LOGIC2, UI, LOGIC, DATA>();
}

class UPMAApplication<owner, U, L, D> extends Activity<...> {} // [owner→LOGIC

class CreateNewDatabase<owner, U, L, D> extends Activity<...> {} // [owner→LOGIC2
    void onClick(){
        new SaveDatabaseAsyncTask<owner, U, L, D>(this);
    }
}

class SaveDatabaseAsyncTask<owner, U, L, D> ... {} // [owner→LOGIC2
    Activity<owner, U, L, D> activity;
    SaveDatabaseAsyncTask (Activity<owner, U, L, D> a){
        this.activity = a; // avoid creating false positive dataflow edges
    }
}
Discussion: Imprecision may be due to overuse of lent and unique

class Main<owner>{
    domain DATA, DOM1, DOM2;
    void main() {
        Integer<unique> n1 = new Integer(-1);
        B<DATA, DOM1> b1 = new B();
        b1.assign(n1);
        Integer<unique> n2 = new Integer(2);
        B<DATA, DOM2> b2 = new B();
        b2.assign(n2);
        Integer<lent> dest = b2.a.get();
        dest.compareTo(n1);
    }
}
Conclusion and future work

- ScoriaX extracts sound approximation of runtime architecture as abstract object graph
- Dataflow edges refer to objects
- Resolves lent and unique
- Evaluated analysis on Android application: can distinguish between database file and file that stores encryption keys

- Evaluate ScoriaX on more systems
- Find architectural flaws using object graphs [Vanciu and Abi-Antoun, ASE’13]
  - Finding Architectural Flaws in Android Apps Is Easy
  - [Vanciu and Abi-Antoun, SPLASH’13, Tool Demo] **Wed 3:30-4:15 pm**
- Use abstract object graphs for program comprehension
  - Finding the Missing Eclipse Perspective: the Runtime Perspective
  - [Giang and Abi-Antoun, SPLASH’13, Tool Demo] **Thu 10:30-11:15 am**
Extra slides
More details on Flow Graph – call-site context sensitive

• Edges \((O1, x, B1) \xrightarrow{a} (O2, y, B2)\)
  • Edge label \(a\) tracks call-site context sensitivity:
    • direct assignment, field read
    * field write

\(i\) invocation of method \(m\) in the context of \(O\)
\(i\) return from method \(m\) in the context of \(O\)

• Properties of FG
  • call-site context sensitive
  • domain-sensitive
  • flow-insensitive
Related Work

- Value flow and points-to analyses
  - Object sensitivity [Milanova et al. TSE’05] [Liu, Thesis’10]
  - Type Sensitivity [Smaragdakis et al. PLDI’11]

- Extraction of object graph
  - Flat object graph [Jackson and Waingold, TSE’01] [Spiegel, Thesis’02]
  - Dataflow edges (between classes) that refer to objects [Lienhard et al. COMLAN’09]
  - Ownership object graph with points-to edges [Abi-Antoun and Aldrich, OOPSLA’09]
  - Abstract Object Heaps [Marron et al. TSE’13]