FlowMiner: Automatic Summarization of Library Data-Flow for Malware Analysis

December 19, 2015

Tom Deering, Ganesh Ram Santhanam, Suresh Kothari

Knowledge Centric Software Laboratory
Iowa State University, Ames, Iowa 50014 USA

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
Need for Summarizing Libraries
Need for Summarizing Libraries

Modern software uses large, **reusable library** components
Static Analysis **including entire library does not scale**
Analysis of an application **without library is inaccurate**
**Summaries** - Scalably include **relevant** parts of library in analysis
Need for Summarizing Libraries

Partial Program Analysis:
Analyzing a proper subset of a software

Often, A is available in source or binary, but not L

Or, L is too large to be analyzed with A
Need for Summarizing Libraries

**Solution:** Analyze $A+L^S$ instead of $A+L$

$L^S$: Summary of $L$ must be

- Expressive (fine-grained) for accurate subsequent analysis
- More compact than library for scalability
- Sound
- Independent of specific analysis tools
- Independent of app that uses $L$
Role of Summaries in Malware Detection
Detecting **consumer malware** is a well-studied problem.

Detecting **novel, sophisticated, domain-specific** malware is not.

- Crafted specifically to disrupt one aspect of one organization
- Payload is customized for target
- Domain knowledge is used to camouflage malicious behavior within benign mechanisms
- Responds to a specific trigger from adversary or environment

*All very different characteristics from consumer malware!*
STUXNET: Example of Real-World Targeted Malware

Targets Seimens uranium centrifuge PLCs in Iran

Undiscovered for years
Centrifuges began breaking in 2008
Authors gradually made it more conventional (wanting to get caught)
Discovered in 2010 by conventional means

http://limn.it/the-morris-worm/
A Malware Example without Summaries
Human-in-the-loop Malware Detection

Human analyst is indispensable in detecting targeted malware.

Automated tools must aid human analyst to devise, test and validate hypotheses about the existence of malware.

Summaries are especially critical for detecting targeted malware:

+ Aids quick what-if experiments
+ Reuse of summaries
+ Enables scalability without entire library
+ Allows accurate detection of malware
A Malware Example with Summaries
FlowMiner: Summarization using Graph-based Program Analysis
Polynomial-time analyzers pre-process the AST
Optimized in-memory graph database is populated
Powerful query API (select, traverse, combine)
Multiple ways to interact with graph artifacts
Nodes and edges of program graph have properties
ID, Name, Kind, Keywords, etc.
Binary properties are expressed as “tags”

The eXtensible Common Software Graph provides:
A hierarchical structure of node and edges kinds
Proper abstraction of common semantic meaning (even across languages)
Well-defined semantics for each node or edge kind
FlowMiner: Research Question

How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?
FlowMiner: Goals

One-time, automatic static analysis of $L$ to produce summaries $L^S$ that:

- Are **expressive** enough to be used with context, field, type, flow, and object sensitivity
- Are **compacted** to elide uninteresting details of flows
- Are **sound** (indicated flows actually occur)
- Are **portable** for use by existing tools
- Are **independent of $A_i$**
- Capture **callbacks** from the library back into the application
Preserve key pieces of information, discard the rest

Control-flow details are discarded
- Cannot use summaries for path-sensitivity
- Retaining control flow does not scale anyway in practice

Statically-resolvable call sites are pre-resolved
- No need to retain signature of the call site

Unimportant data-flow details *elided* with respect to *key nodes*
Preserve “Key” Nodes

Key nodes (for Java) includes

Field Definitions
Method Signature Elements
  Parameters, return values
Definitions read/written to fields
Call Sites
  Parameters, return, invoked signature, invoked type
Literal Values
Array components, accesses
Computing Summaries with Fine Granularity

FlowMiner summaries support

Context Sensitivity
   Individual methods, call sites from the original library are preserved

Flow Sensitivity
   Preserved from Atlas data flow graph by eliding algorithm

Field Sensitivity
   Individual field definitions are tracked

Object Sensitivity
   Field access paths preserved for use in points-to analyses

Type Sensitivity
   Call sites that cannot be statically-resolved under open-world assumption are left to be resolved in the context of a client application

Array / Array Index Sensitivity
   Array components, access operands are preserved
Summarizing Intra-Procedural Data Flow

```java
package com.example;

class ProblemStatement{
    static int average(List<Integer> l){
        int lSum = sum(l);
        int lLength = l.size();
        return lSum / lLength;
    }

    static int sum(List<Integer> l){
        int s = 0;
        for(Integer i : l) s += i;
        return s;
    }
}
```
Intra-procedural Flow - Elided

Local Flow Algorithm

Elided in the summary

Preserved in the summary

Algorithm 1 Mining summary data flows

1: procedure MINEFLOW($K, G(\Psi))$
2:   for all $k \in K$ do
3:     reachable $\leftarrow$ ElidedFlow($k, K, G(\Psi)$)
4:   for all $k' \in$ reachable do
5:     Add summary flow edge from $k$ to $k'$
6:   end for
7: end for
8: end procedure

9: procedure ELIDEDFLOW($k, K, G(\Psi)$)
10: frontier $\leftarrow \{k\}$
11: result $\leftarrow \{\emptyset\}$
12: for all $f \in$ frontier do
13:   frontier $\leftarrow$ frontier - $f$
14:   for all $f'$ s.t. $(f, f')$ is a data flow edge in $G(\Psi)$ do
15:     if $f' \in K$ then
16:       result $\leftarrow$ result $\cup f'$
17:     else if $f' \notin$ frontier then
18:       frontier $\leftarrow$ frontier $\cup f'$
19:     end if
20: end for
21: end for
22: return result
Open World Assumption: Client applications may introduce new virtual dispatch targets when the library is used. Should not pre-resolve open-world virtual call sites in the summary!

All possibilities may not be captured
Must be able to capture callbacks into the target application
**FlowMiner: Compaction**

**Argument:** FlowMiner summaries cannot be further compacted without information loss

Removing *any* summary node removes a *key* program artifact
- Parameter, Identity, Return, Field, Array Component, Literal Value, Call Site

Removing *any* summary edge (A, B) disconnects *at least one* possible flow between key artifacts
- Can construct a client application such that this leads to a false negative
FlowMiner Implementation

Architecture

- Targets arbitrary Java library bytecode (JAR)
- One-time static analysis
- Expressed as extension to XCSG graph schema (Atlas)
- Portable XML packaging of summaries
- Existing analyzers can leverage summary file
FlowMiner: Evaluation on Android
## FlowMiner on Android: Evaluation Results

| Library    | $|V|$ | $|E|$ | $|V^S|/|V|$ (%) | $|E^S|/|E|$ (%) | Field Flows | Object Flows | % False Positives avoided |
|------------|-----|-----|-------------|-------------|-------------|-------------|--------------------------|
| Android 4.2.2 | 6651277 | 33964070 | 37.11% | 22.57% | 1129523 | 16053060 | 92.96% |
| Android 4.3.1 | 6867245 | 35165616 | 37.10% | 22.51% | 1206542 | 16816490 | 92.83% |
| Android 4.4.4 | 7707688 | 44150241 | 36.98% | 20.06% | 1216178 | 17069468 | 92.88% |
| Android 5.0.2 | 8684208 | 45649066 | 37.05% | 21.93% | 1556027 | 21874691 | 92.89% |

$|V|$, $|E|$ - # Nodes, # Edges in the original program graph

$|V^S|$, $|E^S|$ - # Nodes, # Edges in the summary program graph

Field Flows - Data-flow edges in FlowMiner’s summary that tracks flows at field level granularity

Object Flows - Data-flow edges if object level flows are tracked
## FlowMiner on Android: Correctness

| Library   | $|V|$     | $|E|$     | $|V^S|/|V|$ (%) | $|E^S|/|E|$ (%) | Field Flows | Object Flows | % False Positives avoided |
|-----------|---------|---------|---------------|---------------|--------------|--------------|--------------------------|
| Android 4.2.2 | 6651277 | 33964070 | 37.11%        | 22.57%        | 1129523     | 16053060     | 92.96%                   |
| Android 4.3.1 | 6867245 | 35165616 | 37.10%        | 22.51%        | 1206542     | 16816490     | 92.83%                   |
| Android 4.4.4 | 7707688 | 44150241 | 36.98%        | 20.06%        | 1216178     | 17069468     | 92.88%                   |
| Android 5.0.2 | 8684208 | 45649066 | 37.05%        | 21.93%        | 1556027     | 21874691     | 92.89%                   |

4 Recent versions of Android

**Sound:** No spurious flows added (no false positives)

**Complete:** All flows covered (no false negatives)
| Library | $|V|$ | $|E|$ | $|V^s|/|V|$(%) | $|E^s|/|E|$(%) | Field Flows | Object Flows | % False Positives* avoided |
|---------|-----|-----|-----------|-----------|----------|-------------|--------------------------|
| Android 4.2.2 | 6651277 | 33964070 | 37.11% | 22.57% | 1129523 | 16053060 | 92.96% |
| Android 4.3.1 | 6867245 | 35165616 | 37.10% | 22.51% | 1206542 | 16816490 | 92.83% |
| Android 4.4.4 | 7707688 | 44150241 | 36.98% | 20.06% | 1216178 | 17069468 | 92.88% |
| Android 5.0.2 | 8684208 | 45649066 | 37.05% | 21.93% | 1556027 | 21874691 | 92.89% |

Summary Graph $G^s=(V^s,E^s)$ retained from the original graph only $\sim$37% Nodes  
20% - 23% Edges  
Considerably smaller than original program graphs
FlowMiner on Android: Expressiveness

| Library    | $|V|$     | $|E|$     | $|V^S|/|V|$(%) | $|E^S|/|E|$(%) | Field Flows | Object Flows | % False Positives avoided |
|------------|---------|---------|------------|------------|-------------|--------------|--------------------------|
| Android 4.2.2 | 6651277 | 33964070 | 37.11%     | 22.57%     | 1129523     | 16053060     | 92.96%                   |
| Android 4.3.1 | 6867245 | 35165616 | 37.10%     | 22.51%     | 1206542     | 16816490     | 92.83%                   |
| Android 4.4.4 | 7707688 | 44150241 | 36.98%     | 20.06%     | 1216178     | 17069468     | 92.88%                   |
| Android 5.0.2 | 8684208 | 45649066 | 37.05%     | 21.93%     | 1556027     | 21874691     | 92.89%                   |

*False Positives comparison
Field-sensitive vs Object-sensitive flow tracking
Comparison to Clapp et al.

~93% fewer false positive flows
Malware Example with Summaries

Deering, Santhanam & Kothari · kothari@iastate.edu

FlowMiner: Automatic Summarization of Library Data-Flow for Malware Analysis
FlowMiner: Related Work

Component-Level Data-flow Analysis (Rountev et al.)
Theoretical framework for summarizing an Interprocedural Control Flow Graph (ICFG)
Captures virtual calls (callbacks), elides uninteresting details
*Incomplete handling of fields*
*Lacking concrete implementation*

Mining Information Flow Specifications From Concrete Executions (Clapp et al.)
Instrument Android and create a special emulation environment
Dynamically exercise Android APIs to produce execution traces
Post-process traces to infer *coarse* information flow summaries.
*Coarse object tainting is inaccurate, misses callbacks*
*Incomplete path coverage*
Summary

FlowMiner

One-time, automatic static extraction of data flow summaries

Expressive & fine-grained
  Can be used with context, field, type, flow, and object sensitivity

Compact
  Elides uninteresting details of flows

Sound
  Indicated flows actually occur

Portable for use by existing tools

Captures callbacks from the library back into the application

Practically Efficient open source tool
  Validated on recent versions of Android
Related Publications


Thank You!

EnSoft Team

Jon Mathews, Jeremias Saucedas, Nikhil Ranade, Kevin Korslund,

Theodore Murdock

DARPA APAC & STAC programs
XCSG/Atlas Additional Slides
private void method1(){ method2(); }
public static void method2{}
## Atlas: XCSG Directed Property Multigraph

<table>
<thead>
<tr>
<th>Edge Kind</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains</td>
<td>Destination is nested within origin.</td>
</tr>
<tr>
<td>Element Type</td>
<td>Origin array contains destination element kind.</td>
</tr>
<tr>
<td>Overrides</td>
<td>Origin method overrides the destination method.</td>
</tr>
<tr>
<td>Supertype</td>
<td>Destination is a supertype of the origin type.</td>
</tr>
<tr>
<td>Type Of</td>
<td>Destination type is static type of origin.</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Dest block follows origin block.</td>
</tr>
<tr>
<td>Call</td>
<td>Origin calls destination method.</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Origin def flows to destination use.</td>
</tr>
</tbody>
</table>
Atlas: API for Automated Analyzers

Analysis results can be built using low-level *graph* or convenience *select, traverse*, and *combine* operations on the XCSG-compliant graph.

Q someType = types("AnInterestingType");
Q supertypeHierarchy = edges(XCSG.Supertype).forward(someType);

Q someMethod = methods("anInterestingMethod");
Q reverseCallGraph = edges(XCSG.Call).reverse(someMethod);
Q combinedResult = supertypeHierarchy.union(reverseCallGraph);