Towards Fully Automatic Placement of Security Sanitizers and Declassifiers

Ben Livshits, Microsoft Research
Stephen Chong, Harvard University
Why Track Explicit Information Flow?

(1) preventing injection

LinkedIn’s Leaky Mobile App Has Access to Your Meeting Notes

By NICOLE PERLROTH

Researchers analyzed the traffic generated by LinkedIn’s mobile app and found details in the software code of a private calendar appointment was leaking back to LinkedIn.

LinkedIn mobile app subscribers may be surprised to learn that the calendar entries on their iPhones or iPads—which may include details about meeting locations, participants, dial-in information, passwords and sensitive meeting notes—are transmitted back to LinkedIn’s servers without their knowledge.

(2) preventing private data leaks in mobile

Top 10 – 2010 (New)

Injection and Session Management

Project References

Get Forgery (CSRF)
A Decade of Research & Development

- Type systems [8, 32]
- Static analysis [17, 18, 22, 42, 43, 47]
- Runtime monitoring and enforcement [6, 7, 11, 25]
Developers Are Not Able to Manage

- Places burden for sanitizer placement on developer
- Uses analysis (static/runtime/hybrid) to find placement violations
- Large modern applications exceed developer’s ability to reason about inter-procedural and inter-modular data flow
- We argue that data sanitization should not be developer’s responsibility
- Runtime tracking is a possibility, but overhead too high!
- Here we provide a hybrid static/runtime technique for taint tracking with low overhead
I. Motivation

Three motivating applications from security and privacy:

- Web
- Mobile
- Cloud
**Data Flow Graph & Sanitization Policy**

- Examined large-scale applications
- Data processing done via a fixed set of sanitizers (for integrity) or declassifiers (for confidentiality)
- Reasoning done via reachability on data flow graph
- Developer’s mental model involves deciding which sanitizer to use at which point
- Sanitizer used depends on where data comes from (source) and where it is going (sink)

<table>
<thead>
<tr>
<th></th>
<th>Sink 1</th>
<th>Sink 2</th>
<th>Sink 3</th>
<th>Sink 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sanitizing Inputs in Web Applications

- The OWASP Enterprise Security API (ESAPI) is an open-source web-application security library.

- Usage guidelines of ESAPI reveal that the correct sanitization to apply to data depends on how the data will be used: sink context.

- To sanitize user-provided URL, use `encodeForURL(input)`. To sanitize input that will be used to construct a CSS attribute use `encodeForCSS(input)`.

<table>
<thead>
<tr>
<th>URL</th>
<th>CSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>encodeForURL</td>
</tr>
<tr>
<td></td>
<td>encodeForCSS</td>
</tr>
</tbody>
</table>
Declassifying Sensitive Data in Android Mobile Apps

- Consider a Gmail app that needs to communicate with its parent site `mail.google.com`. It is necessary to send information to that hosting URL, including keystrokes, files to be attached to email.
- No compelling need to send user data to `AdMob.com`, a third-party mobile ad provider, whose library is embedded in the app.
- So data sent to a third-party should be cleansed, i.e., should have sensitive information removed, a form of declassification.

<table>
<thead>
<tr>
<th></th>
<th>screen output</th>
<th>isolated app storage</th>
<th>to host</th>
<th>to 3rd-party site</th>
</tr>
</thead>
<tbody>
<tr>
<td>user input</td>
<td></td>
<td></td>
<td></td>
<td>cleanse</td>
</tr>
<tr>
<td>host</td>
<td></td>
<td></td>
<td></td>
<td>cleanse</td>
</tr>
<tr>
<td>3rd-party site</td>
<td>ascii-sanitizer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encrypted Cloud

- Consider a web application using a public cloud provider for storage.

- Web application does not trust the cloud to protect the **confidentiality** of its data.

- The application therefore will use encryption when serializing data to the database, and decryption when **deserializing**.
Contributions

1) **Fully automatic sanitizer placement.** Argue that sanitizer placement should be automatic, given a policy and an application.

2) Propose **node-based placement**, a simple node-based strategy for static sanitizer placement.
   - simple to implement
   - incurs no run-time overhead

3) Propose **edge-based placement**
   - attempts to place sanitizers statically
   - “spills over” into run time whenever necessary.
   - appropriate when simple node-based strategy fails

4) **Evaluation** on programs 1.8M LOC
   - edge-based approach provides full sanitization,
   - reduces number of instrumentation points by 6.19x on average
   - 27x for sparse graphs
Graph & Valid Placement

- Our algorithms work on the inter-procedural data flow graph.

- The graph can be obtained through different means:
  - Sound pointer analysis
  - Unsound dataflow extraction tool such as used by static analysis companies Fortify and Coverity

- Our graph came from CAT.NET, a static analysis tool for security for .NET applications

Definition: Valid placement

- Given a data flow graph $G = \langle N, E \rangle$ and a source-sink pair $<I, O>$
  - if $P(I, O) = S$, on every path from $I \rightarrow O$, sanitizer $S$ is applied exactly once and no other sanitizer is applied;
  - if $P(I, O) = \bot$, no sanitizer is applied on the path.
Motivating Example
I. Algorithms

Here: intuition
Glimpses of algorithm specification

Paper: full algorithms
Theorem showing correctness of placement
Two Placement Strategies

Given inter-procedural flow graph $G=\langle N, E \rangle$ and Policy table $P$

Node-based placement

- Intuitive, easy to understand
- Fast to compute

- Fails to find a valid placement all too frequently

Edge-based placement

- Finds valid placement and escapes into runtime tracking whenever necessary

- Quite a bit more involved to compute and understand
Node-Based Placement

- We say that a node $n$ is $S_i$-possible if it is on a path from a source node $I$ to a sink node $O$ that requires sanitizer $S_i$
- These are merely possible points for $S_i$ sanitizer placement
- $S_i$-possible are pretty plentiful in most graphs
- We say a node $n$ is $S_i$-exclusive if it is $S_i$-possible, and it is not $S_j$-possible for any $j \neq I$
- Placing at $S_i$-exclusive nodes can be done without fear of trampling over another path
- Unfortunately, exclusive nodes are pretty rare

Given inter-procedural flow graph $G=\langle N, E \rangle$ and Policy table $P$
How Do We Compute It?

- Data flow formulation
- Inspired by research on Common Subexpression Elimination (CSE) from the 1990s
- Scales linearly
- See Dragon Book for details

<table>
<thead>
<tr>
<th>Semi-lattice</th>
<th>$L$</th>
<th>bit vector of length $k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>$\top$</td>
<td>0</td>
</tr>
<tr>
<td>Initial value</td>
<td>$\text{init}(n)$</td>
<td>0</td>
</tr>
</tbody>
</table>
| Transfer function  | $TF(n)$         | \[\begin{align*}
                   \text{bit } i &= 1 & \text{ if } n \text{ is } S_i\text{-exclusive} \\
                   \text{identity} & & \text{otherwise}
                 \end{align*}\]  |
| Meet operator      | $\sqcap(x, y)$  | bitwise or $x|y$         |
| Direction          |                 | backward                 |
Node-based Placement Example

Exclusive nodes are not necessarily unique: multiple $S_i$-exclusive nodes on a single path!

<table>
<thead>
<tr>
<th>Possible</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1, 2, 3, 6, 7, 10, 18, 19</td>
</tr>
<tr>
<td>$S_2$</td>
<td>2, 3, 6, 7, 8, 10, 11, 12, 14, 15, 17, 19, 20</td>
</tr>
<tr>
<td>$S_3$</td>
<td>3, 4, 7, 8, 11, 13, 16, 21</td>
</tr>
<tr>
<td>$S_4$</td>
<td>5, 9, 16, 21</td>
</tr>
<tr>
<td>$\perp$</td>
<td>4, 8, 11, 12, 14, 15, 17, 20</td>
</tr>
</tbody>
</table>
Latest-Exclusive Nodes

- Late sanitization:
  - prefer to perform sanitization as late as possible
  - saving work in case program is terminated
  - sanitized values are larger than original ones

- We say that node n is $S_i$-latest-exclusive if
  - it is $S_i$-exclusive, and
  - for every path through n, it is the last $S_i$-exclusive on that path.
Latest-Exclusive Nodes Example

- Does not protect
  - 2 → 18
  - 2 → 19
  - 3 → 20
  - 4 → 20

<table>
<thead>
<tr>
<th>Possible</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1, 2, 3, 6, 7, 10, 18, 19</td>
</tr>
<tr>
<td>$S_2$</td>
<td>2, 3, 6, 7, 8, 10, 11, 12, 14, 15, 17, 19, 20</td>
</tr>
<tr>
<td>$S_3$</td>
<td>3, 4, 7, 8, 11, 13, 16, 21</td>
</tr>
<tr>
<td>$S_4$</td>
<td>5, 9, 16, 21</td>
</tr>
<tr>
<td>$\bot$</td>
<td>4, 8, 11, 12, 14, 15, 17, 20</td>
</tr>
</tbody>
</table>

Diagram showing nodes and connections with exclusivity conditions.
Edge-based Placement

- Computation pipeline
- Classify edges into groups
- Compute groups using multiple (pipelined) data flow analysis stages
- Try fully static placement
- Minimize the amount of runtime tracking
Source- and Sink-Dependent

- An edge $e$ is source-dependent if there exist sources $I_0$ and $I_1$ and sink $O$ such that
  - $e$ is on a path from $I_0$ to $O$ and
  - on a path from $I_1$ to $O$ and
  - $P(I_0, O) \neq P(I_1, O)$

- In other words, sanitizer to use depends on the source.
In- and Out-Trigger

- Edge $e$ is an in-trigger edge if it is a source-independent edge but has a successor edge that is source-dependent.

- At in-trigger edges we have sufficient information to know where a value came from.

- Need to start run-time tracking because origin affects sanitizer choice.

- Edge $e$ is an out-trigger edge if it is a sink-independent edge but has a predecessor edge that is source-dependent.

- At out-trigger edges we have sufficient information to know where a value came from.

- We can cease tracking at out-trigger edges.
Sanitization Edges

- Sanitization edges are sink-independent edges that are the earliest sink-independent edge on some path from a source to a sink.

- We prefer early sanitization to avoid extra tracking.

\[
\begin{align*}
\text{Sanitization Edges} & \\
\quad & \\
\text{dom_sani}(e): \text{whether edge } e \text{ is dominated by sanitization edges}
\end{align*}
\]
Tagging, Untagging, and Carry

- **Edge** $e$ is a tag edge if $e$ is an in-trigger edge that is not dominated by sanitization edges.

- **An untag edge** is either
  - (a) an out-trigger edge that is not dominated by a sanitization edge; or
  - (b) a sanitization edge.
Putting it All Together...

- Source-dependent edge
- Sink-dependent edge
- In-trigger edge
- Out-trigger edge
- Sanitization edge
- Tag edge
- Untag edge
- Carry edge edge
III. Experiments

Evaluation on real and synthetic benchmarks
Benchmark Applications

- 10 large, long-running server-side applications written in C#, up to 1.8 M LOC [PLDI’09]
- Runtime overhead of taint tracking is typically workload-and path-specific
- Goal: Reducing # instrumentation points

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DLLs</th>
<th>DLL (KB)</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias Management</td>
<td>3</td>
<td>65</td>
<td>10,812</td>
</tr>
<tr>
<td>Chat Application</td>
<td>3</td>
<td>543</td>
<td>6,783</td>
</tr>
<tr>
<td>Bicycle Club App</td>
<td>3</td>
<td>62</td>
<td>14,529</td>
</tr>
<tr>
<td>Software</td>
<td>15</td>
<td>118</td>
<td>11,941</td>
</tr>
<tr>
<td>Sporting Field Management</td>
<td>3</td>
<td>290</td>
<td>15,803</td>
</tr>
<tr>
<td>Commitment Management</td>
<td>7</td>
<td>369</td>
<td>25,602</td>
</tr>
<tr>
<td>New Hire</td>
<td>11</td>
<td>565</td>
<td>5,595</td>
</tr>
<tr>
<td>Expense Report Approval</td>
<td>4</td>
<td>421</td>
<td>78,914</td>
</tr>
<tr>
<td>Customer Support Portal</td>
<td>14</td>
<td>2,447</td>
<td>66,385</td>
</tr>
<tr>
<td>Relationship Management</td>
<td>5</td>
<td>3,345</td>
<td>1,810,585</td>
</tr>
</tbody>
</table>
What is The Baseline?

- What is the naïve approach?

- Traditional explicit tainting:
  - Find nodes or edges that are both reached from a source and go to a sink (FW and BW data slices)

- Instrument these to carry taint information

Unfortunately, in practice, there are too many such places
Node-based Placement

Many nodes tainted: tracking taint involves instrumenting half the graph

Last-exclusive node placement: considerably less instrumentation

Most of the time, does not achieve complete sanitization!

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph nodes</th>
<th>Taint sources</th>
<th>Taint sinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expense Report Approval</td>
<td>805</td>
<td>214</td>
<td>322</td>
</tr>
<tr>
<td>Customer Support Portal</td>
<td>3,881</td>
<td>967</td>
<td>1,219</td>
</tr>
<tr>
<td>Relationship Management</td>
<td>3,639</td>
<td>1,054</td>
<td>982</td>
</tr>
</tbody>
</table>
Edge-based Placement Results

100% of source-sink paths are protected. Some runtime tracking is needed

<table>
<thead>
<tr>
<th>Application</th>
<th>Edge count</th>
<th>Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Untag</td>
<td>Carry</td>
</tr>
<tr>
<td>Terral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias Management</td>
<td>22.66</td>
<td>6.5</td>
</tr>
<tr>
<td>Client Management</td>
<td>10.93</td>
<td>2.96</td>
</tr>
<tr>
<td>Relationship Management</td>
<td>2.29</td>
<td>2.68</td>
</tr>
<tr>
<td>Total</td>
<td>2.68</td>
<td>4.54</td>
</tr>
<tr>
<td>Ratio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total number of instrumented edges</td>
<td>126</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>266</td>
<td>4,690</td>
</tr>
<tr>
<td></td>
<td>11,686</td>
<td>1,72</td>
</tr>
</tbody>
</table>

Our analysis is most effective at reducing the # of instrumentation points in sparsely connected graphs.
Savings for Sparse Graphs

Significant savings compared to the naïve approach
Conclusions

1) The algorithms developed in our paper pave the way for completely automatic placement of sanitizers and declassifiers.

2) Theory of placement provides a way to reason about static and runtime taint tracking in the same framework.

3) Showed that it works well in practice and produces significantly lower overhead (as high as 27x) compared to other strategies.