Specification Inference for Explicit Information Flow Problems

Merlin

Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani
Microsoft Research

Anindya Banerjee
IMDEA Software
Problem: Can we automatically infer which routines in a program are sources, sinks and sanitizers?

Technology: Static analysis + Probabilistic inference

Applications:
- Lowers false errors from tools
- Enables more complete flow checking

Results:
- Over 300 new vulnerabilities discovered in 10 deployed ASP.NET applications
Motivation
Static Analysis Tools for Security

- Web application vulnerabilities are a serious threat!
$username = $_REQUEST['username'];
$sql = "SELECT * FROM Students WHERE username = '$username';"
void ProcessRequest()
{
    string s1 = ReadData1("name");
    string s2 = ReadData2("encoding");

    string s11 = Prop1(s1);
    string s22 = Prop2(s2);

    string s111 = Cleanse(s11);
    string s222 = Cleanse(s22);

    WriteData("Parameter " + s111);
    WriteData("Header " + s222);
}
**Specification**

- **Source**
  - returns tainted data
- **Sink**
  - error to pass tainted data
- **Sanitizer**
  - cleanse or untaint the input
- **Regular nodes**
  - propagate input to output

**Vulnerability**

- Every path from a source to a sink should go through a sanitizer
- Any source to sink path without a sanitizer is an information flow vulnerability
void ProcessRequest()
{
    string s1 = ReadData1("name");
    string s2 = ReadData2("encoding");

    string s11 = Prop1(s1);
    string s22 = Prop2(s2);

    string s111 = Cleanse(s11);
    string s222 = Cleanse(s22);

    WriteData("Parameter " + s111);
    WriteData("Header " + s222);
}
```csharp
void ProcessRequest()
{
    string s1 = ReadData1("name");
    string s2 = ReadData2("encoding");

    string s11 = Prop1(s1);
    string s22 = Prop2(s2);

    string s111 = Cleanse(s11);
    string s222 = Cleanse(s22);

    WriteData("Parameter " + s111);
    WriteData("Header " + s222);
}
```
void ProcessRequest()
{
    string s1 = ReadData1("name");
    string s2 = ReadData2("encoding");

    string s11 = Prop1(s1);
    string s22 = Prop2(s2);

    string s111 = Cleanse(s11);
    string s222 = Cleanse(s22);

    WriteData("Parameter " + s111);
    WriteData("Header " + s222);
}
Given a propagation graph, can we infer a specification or ‘complete’ a partial specification?

Assumption

Most flow paths in the propagation graph are secure
Algorithms
Merlin Architecture

Initial specification

Program

Merlin

Prop. graph construction
Factor graph construction
Probabilistic inference

Final specification

Vulnerabilities

Microsoft Visual Studio CAT.NET
void ProcessRequest()
{
    string s1 = ReadData1("name");
    string s2 = ReadData2("encoding");

    string s11 = Prop1(s1);
    string s22 = Prop2(s2);

    string s111 = Cleanse(s11);
    string s222 = Cleanse(s22);

    WriteData("Parameter " + s111);
    WriteData("Header " + s222);
}
For every acyclic path $m_1 \ m_2 \ ... \ m_n$ the probability that $m_1$ is a source, $m_n$ is a sink, and $m_2, \ ..., \ m_{n-1}$ are not sanitizers is very low.

Exponential number of path constraints: $O(2^{|V|})$!
For every triple $<m_1, m_i, m_n>$ such that $m_i$ is on a path from $m_1$ to $m_n$, the probability that $m_1$ is a source, $m_n$ is a sink, and $m_i$ is not a sanitizer is very low.

Cubic number of triple constraints: $O(|V|^3)!$
Minimizing Sanitizers

Diagram:
- Source
- ReadData1
- Prop1
- Cleanse
- WriteData
- Sink
- ReadData2
- Prop2
For every pair of nodes $m_1, m_2$ such that $m_1$ and $m_2$ lie on the same path from a potential source to a potential sink, the probability that both $m_1$ and $m_2$ are sanitizers is low.
Need for probabilistic constraints

Triple constraints
- \( \neg (a \land \neg b \land d) \)
- \( \neg (a \land \neg c \land d) \)

Avoid double sanitizers
- \( \neg (b \land c) \)
- \( a \land d \Rightarrow b \)
- \( a \land d \Rightarrow c \)
- \( \neg (b \land c) \)
Boolean formulas as probabilistic constraints

\[(x_1 \lor x_2) \land (x_1 \lor \neg x_3)\]

\[C_1 \quad C_2\]

\[f(x_1, x_2, x_3) = f_{C_1}(x_1, x_2) \land f_{C_2}(x_1, x_3)\]

\[f_{C_1}(x_1, x_2) = \begin{cases} 1 \text{ if } x_1 \lor x_2 = \text{true} \\ 0 \text{ otherwise} \end{cases}\]

\[f_{C_2}(x_1, x_3) = \begin{cases} 1 \text{ if } x_1 \lor \neg x_3 = \text{true} \\ 0 \text{ otherwise} \end{cases}\]
Boolean formulas as probabilistic constraints

\[ (x_1 \lor x_2) \land (x_1 \lor \neg x_3) \]
\[ C_1 \quad C_2 \]

\[ f(x_1, x_2, x_3) = f_{C_1}(x_1, x_2) \land f_{C_2}(x_1, x_3) \]

\[ f_{C_1}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 \lor x_2 = \text{true} \\ 0 & \text{otherwise} \end{cases} \]

\[ f_{C_2}(x_1, x_3) = \begin{cases} 1 & \text{if } x_1 \lor \neg x_3 = \text{true} \\ 0 & \text{otherwise} \end{cases} \]

\[ p(x_1, x_2, x_3) = f_{C_1}(x_1, x_2) \times f_{C_2}(x_1, x_3)/Z \]

\[ Z = \sum_{x_1, x_2, x_3} (f_{C_1}(x_1, x_2) \times f_{C_2}(x_1, x_3)) \]
Solution = Marginalization

\[ p_i(x_i) = \sum_{x_1} \ldots \sum_{x(i-1)} \sum_{x(i+1)} \ldots \sum_{x_N} p(x_1, \ldots, x_N) \]

• **Step 1:** choose \( x_i \) with highest \( p_i(x_i) \) and set \( x_i = \text{true} \) if \( p_i(x_i) \) is greater than a threshold, \( \text{false} \) otherwise
• **Step 2:** recompute marginals and repeat **Step 1** until all variables have been assigned
Factor graphs: efficient computation of marginals

\[ f_{C1}(x_1, x_2) = \begin{cases} 
1 & \text{if } x_1 \lor x_2 = \text{true} \\
0 & \text{otherwise}
\end{cases} \]

\[ f_{C2}(x_1, x_3) = \begin{cases} 
1 & \text{if } x_1 \lor \neg x_3 = \text{true} \\
0 & \text{otherwise}
\end{cases} \]
Factor Graphs
### Probabilistic Inference

<table>
<thead>
<tr>
<th></th>
<th>Source</th>
<th>Sanitizer</th>
<th>Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadData1</td>
<td>.95</td>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>ReadData2</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
</tr>
<tr>
<td>Cleanse</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
</tr>
<tr>
<td>WriteData</td>
<td>.5</td>
<td>.5</td>
<td>.85</td>
</tr>
</tbody>
</table>

...
**Paths vs. Triples**

Path\(G = (V, E)\)

Returns:
Mapping \(m\) from \(V\) to the set \(\{0, 1\}\)

1: for all paths \(p = s, \ldots, n\) from potential sources to sinks in \(G\) do
2: \hspace{1em} assume\((m(p) \not\in 10^*1) \oplus_{c_p} \text{assume}(m(p) \in 10^*1)\)
3: end for

Post expectation: \(\forall\) paths \(p\) in \(G\), \(m(p) \not\in 10^*1\).

Triple\(G = (V, E)\)

Returns:
Mapping \(m\) from \(V\) to the set \(\{0, 1\}\)

1: for all triples \(t = (s, w, n)\) such that \(s\) is a potential source, \(n\) is a potential sink and \(w\) lies on some path from \(s\) to \(n\) in \(G\) do
2: \hspace{1em} assume\((m((s, w, n)) \neq 101) \oplus_{c_t} \text{assume}(m((s, w, n)) = 101)\)
3: end for

Post expectation: \(\forall\) paths \(p\) in \(G\), \(m(p) \not\in 10^*1\).
Experiments
Merlin is implemented in C#

- Uses CAT.NET for building the propagation graph
- Uses Infer.NET for probabilistic inference
  - http://research.microsoft.com/infernet
10 line-of-business applications written in C# using ASP.NET

- Customer Support Portal: 67 KLOC
- Relationship Management: 1811 KLOC
- Expense Report Approval Tool: 79 KLOC
- New Hire Tool: 6 KLOC
- Commitment Management Tool: 26 KLOC
- Software Field Management Tool: 16 KLOC
- Software Club App: 12 KLOC
- Bicycle Club App: 15 KLOC
- Chat Application: 7 KLOC
- Alias Management Tool: 11 KLOC

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>27</td>
<td>16</td>
</tr>
<tr>
<td>Sinks</td>
<td>77</td>
<td>8</td>
</tr>
<tr>
<td>Sanitizers</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
Summary of Discovered Specifications

- **Sources**: 27 (Original), 94 (With Merlin)
- **Sanitizers**: 7 (Original), 32 (With Merlin)
- **Sinks**: 77 (Original), 152 (With Merlin)
Summary of Discovered Vulnerabilities

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>With Merlin</th>
</tr>
</thead>
<tbody>
<tr>
<td>False positives</td>
<td>89</td>
<td>13</td>
</tr>
<tr>
<td>eliminated</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experiments - summary

- 10 large Web apps in .NET
- Time taken per app < 4 minutes
- New specs: 167
- New vulnerabilities: 322
- False positives removed: 13
- Final false positive rate for CAT.NET after Merlin < 1%
Summary

- **Merlin** is the first practical approach to infer explicit information flow specifications.
- Design based on a formal characterization of an approximate probabilistic constraint system.
- Able to successfully and efficiently infer explicit information flow specifications in large applications which result in detection of new vulnerabilities.
http://research.microsoft.com/merlin