Machine-Learning-Guided Selectively Unsound Static Analysis

Kihong Heo
Seoul National University

Hakjoo Oh
Korea University

Kwangkeun Yi
Seoul National University

26 May 2017
ICSE'17 @ Buenos Aires
Goal

False Positive

Uniformly Sound

Uniformly Unsound

False Negative
Goal

False Positive

Selectively Unsound

Uniformly Sound

False Negative

Uniformly Unsound
Selectively Unsound Analysis

- Selectively apply unsound strategies
  - e.g.) unrolling loops, skipping lib calls

\[
\begin{align*}
&\text{while}(e)\{C\} \quad \text{if}(e)\{C\} \\
&A;\text{lib}();B; \quad A;B;
\end{align*}
\]

Uniformly Sound  Selectively Unsound  Uniformly Unsound
Example

- Sound buffer-overrun analyzer with interval domain
- soundly analyze all the loops

```c
str = "hello world";
for (i=0; str[i]; i++) // buffer access 1
    skip;

size = positive_input();
for (i=0; i<size; i++)
    skip;

... = str[i]; // buffer access 2
```
Example

• Sound buffer-overflow analyzer with interval domain

• soundly analyze all the loops

```c
str = "hello world";
for(i=0; str[i]; i++) // buffer access 1
    skip;

size = positive_input();
for(i=0; i<size; i++)
    skip;

... = str[i]; // buffer access 2
```

str.size: [12, 12]
i: [0, +\infty]
size: [0, +\infty]
```
Example

- Uniformly unsound buffer-overrun analyzer
- unsoundly unroll all the loops

```c
str = "hello world";
i = 0;
if ( str[i] ) // buffer access 1
    skip;

size = positive_input();
i = 0;
if ( i < size )
    skip;

... = str[i]; // buffer access 2
```
Example

- Uniformly unsound buffer-overrun analyzer
- unsoundly unroll all the loops

```c
str = "hello world";
i = 0;
if (str[i]) // buffer access 1
    skip;

size = positive_input();
i = 0;
if (i < size)
    skip;

... = str[i]; // buffer access 2
```

Example

- Selectively unsound buffer-overflow analyzer
- unsoundly unroll only harmless loops

```c
str = "hello world";
i = 0;
if( str[i]) // buffer access 1
    skip;

size = positive_input();
for(i = 0; i < size; i++)
    skip;

... = str[i]; // buffer access 2
```
Example

- Selectively unsound buffer-overrun analyzer
- unsoundly unroll only harmless loops

```c
str = "hello world";
i = 0;
if( str[i] )  // buffer access 1
  skip;
size = positive_input();
for(i = 0; i < size; i++)
  skip;
...

... = str[i];  // buffer access 2
```

i: [0, 0]

i: [0, +oo]
Performance

• Experiments with 2 analyzers & open source SW

• Taint: 106 format string bugs / 13 programs

• Interval: 138 buffer overrun bugs / 23 programs
Setting

\[
F \in Pgm \times \Pi \to A
\]

- Find a set of targets \( \pi \in \Pi \) for unsound strategies
- loops to analyze unsoundly (\( \Pi = 2^{\text{Loop}} \))
- library calls to analyze unsoundly (\( \Pi = 2^{\text{Lib}} \))
- Selectively apply unsound strategies to \( p \in \pi \)
In this section, we explain the details of our technique. Our system automatically infers the new harmless unsoundness for the test program.

We use a variant of the well-known setting for the parameterized static analysis [8], where the parameter dictates the analysis's soundness, not the analysis's precision as typical parameterized static analysis [3], and machine learning-based parameter inference [1]

The codebase is a set of annotated programs that specifies which program components to soundly analyze. For instance, when the parameter where every component is selected and ignore all the subsequent loop iterations.

Let comes unsound for the loop: we unroll the loop only once to be analyzed soundly, otherwise (the parameter space with respect to the soundness parameter where no component is selected).

Ideally, the output soundness parameter that predicts a soundness parameter for a given program. That is, we first determine the soundness parameter (resp., the parameter).

For a given program to analyze, the existing search algorithms infer a precision setting by analyzing the program a priori (either by iterative refinements [12, 26] or a quick pre-analysis [16]). This approach, however, is feasible only when the evaluation criterion (i.e., precision) can be determined automatically. In our case, the evaluation involves

judging truth and falsehood of alarms from static analyzers, which is undecidable in general. This explains why we take learning, where a classifier is learnt from a set of training data. Using the training data, a machine learning algorithm trains a classifier that effectively learnt by an anomaly detection algorithm. It is believed.

3.1 Static Analysis Parameterized by Soundness

3.2) and machine learning-based parameter inference

We use a variant of the well-known setting for the parameterized static analysis [8], where the parameter dictates the analysis's soundness, not the analysis's precision as typical parameterized static analysis [3], and machine learning-based parameter inference [1].

We model a static analyzer as a function model that predicts a soundness parameter for a given program. That is, the unsound treatment of program components in the codebase is associated with a set of program points that the analyzer returns alarms, a set of program points that the analyzer concludes as dangerous.

For instance, when there is no confusion.

between them, where the analysis reports the fewest possible false alarms yet still detects most of the real bugs.

Ideally, the output soundness parameter that predicts a soundness parameter for a given program. That is, we first determine the soundness parameter (resp., the parameter).

For a given program to analyze, the existing search algorithms infer a precision setting by analyzing the program a priori (either by iterative refinements [12, 26] or a quick pre-analysis [16]). This approach, however, is feasible only when the evaluation criterion (i.e., precision) can be determined automatically. In our case, the evaluation involves

judging truth and falsehood of alarms from static analyzers, which is undecidable in general. This explains why we take learning, where a classifier is learnt from a set of training data. Using the training data, a machine learning algorithm trains a classifier that effectively learnt by an anomaly detection algorithm. It is believed.

3.3 Learning Harmless Unsoundness

The codebase is a set of annotated programs that specifies which program components to soundly analyze. For instance, when the parameter where every component is selected and ignore all the subsequent loop iterations.

Let comes unsound for the loop: we unroll the loop only once to be analyzed soundly, otherwise (the parameter space with respect to the soundness parameter where no component is selected).

Ideally, the output soundness parameter that predicts a soundness parameter for a given program. That is, we first determine the soundness parameter (resp., the parameter).

For a given program to analyze, the existing search algorithms infer a precision setting by analyzing the program a priori (either by iterative refinements [12, 26] or a quick pre-analysis [16]). This approach, however, is feasible only when the evaluation criterion (i.e., precision) can be determined automatically. In our case, the evaluation involves

judging truth and falsehood of alarms from static analyzers, which is undecidable in general. This explains why we take learning, where a classifier is learnt from a set of training data. Using the training data, a machine learning algorithm trains a classifier that effectively learnt by an anomaly detection algorithm. It is believed.

3.3 Learning Harmless Unsoundness

The codebase is a set of annotated programs that specifies which program components to soundly analyze. For instance, when the parameter where every component is selected and ignore all the subsequent loop iterations.

Let comes unsound for the loop: we unroll the loop only once to be analyzed soundly, otherwise (the parameter space with respect to the soundness parameter where no component is selected).

Ideally, the output soundness parameter that predicts a soundness parameter for a given program. That is, we first determine the soundness parameter (resp., the parameter).

For a given program to analyze, the existing search algorithms infer a precision setting by analyzing the program a priori (either by iterative refinements [12, 26] or a quick pre-analysis [16]). This approach, however, is feasible only when the evaluation criterion (i.e., precision) can be determined automatically. In our case, the evaluation involves

judging truth and falsehood of alarms from static analyzers, which is undecidable in general. This explains why we take learning, where a classifier is learnt from a set of training data. Using the training data, a machine learning algorithm trains a classifier that effectively learnt by an anomaly detection algorithm. It is believed.
Training Data Generation

- Given a codebase w/ known bugs + a sound static analyzer
- Collect precision-decreasing yet harmless pgm components
  - e.g.) unrolling a loop reduces only FP but retains all TP

```
<table>
<thead>
<tr>
<th>Loop 1</th>
<th>Loop 2</th>
<th>Loop 3</th>
<th>...</th>
<th>Loop n</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Features & Learning

• Encode each program component as a feature vector

\[ f(x) = <f_1(x), f_2(x), \ldots, f_n(x)> \]

\[ f(\text{loop}_1) = <1, 0, \ldots, 1> \]
\[ f(\text{loop}_2) = <0, 1, \ldots, 1> \]
\[ f(\text{lib}_1) = <0, 1, \ldots, 0> \]
\[ f(\text{lib}_2) = <1, 1, \ldots, 1> \]

• Derive a classifier using an off-the-shelf algorithm

• e.g.) SVM
Features

- 22 features for loops

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains nulls or not</td>
</tr>
<tr>
<td>Const</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains constants or not</td>
</tr>
<tr>
<td>Array</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains array accesses or not</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains &amp;&amp; or not</td>
</tr>
<tr>
<td>IdxSingle</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains an index for a single array in the loop</td>
</tr>
<tr>
<td>IdxMulti</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains an index for multiple arrays in the loop</td>
</tr>
<tr>
<td>IdxOutside</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the loop condition contains an index for an array outside of the loop</td>
</tr>
<tr>
<td>InitIdx</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether an index is initialized before the loop</td>
</tr>
<tr>
<td>Exit</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) number of exits in the loop</td>
</tr>
<tr>
<td>Size</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) size of the loop</td>
</tr>
<tr>
<td>ArrayAccess</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) number of array accesses in the loop</td>
</tr>
<tr>
<td>ArithInc</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) number of arithmetic increments in the loop</td>
</tr>
<tr>
<td>PointerInc</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) number of pointer increments in the loop</td>
</tr>
<tr>
<td>Prune</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether the loop condition prunes the abstract state or not</td>
</tr>
<tr>
<td>Input</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether the loop condition is determined by external inputs</td>
</tr>
<tr>
<td>GVar</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether global variables are accessed in the loop condition</td>
</tr>
<tr>
<td>FinInterval</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a variable has a finite interval value in the loop condition</td>
</tr>
<tr>
<td>FinArray</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a variable has a finite size of array in the loop condition</td>
</tr>
<tr>
<td>FinString</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a variable has a finite string in the loop condition</td>
</tr>
<tr>
<td>LCSIZE</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a variable has an array of which the size is a left-closed interval</td>
</tr>
<tr>
<td>LCOFFset</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a variable has an array of which the offset is a left-closed interval</td>
</tr>
<tr>
<td>#AbsLoc</td>
<td>Semantic</td>
<td>Numeric</td>
<td>The (normalized) number of abstract locations accessed in the loop</td>
</tr>
</tbody>
</table>
Features

- 15 features for library calls

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the parameters contain constants or not</td>
</tr>
<tr>
<td>Void</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the return type is void or not</td>
</tr>
<tr>
<td>Int</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the return type is int or not</td>
</tr>
<tr>
<td>CString</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the function is declared in string.h or not</td>
</tr>
<tr>
<td>InsideLoop</td>
<td>Syntactic</td>
<td>Binary</td>
<td>Whether the function is called in a loop or not</td>
</tr>
<tr>
<td>#Args</td>
<td>Syntactic</td>
<td>Numeric</td>
<td>The (normalized) number of arguments</td>
</tr>
<tr>
<td>DefParam</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a parameter are defined in a loop or not</td>
</tr>
<tr>
<td>UseRet</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether the return value is used in a loop or not</td>
</tr>
<tr>
<td>UptParam</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a parameter is update via the library call</td>
</tr>
<tr>
<td>Escape</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether the return value escapes the caller</td>
</tr>
<tr>
<td>GVar</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a parameters points to a global variable</td>
</tr>
<tr>
<td>Input</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a parameters are determined by external inputs</td>
</tr>
<tr>
<td>FinInterval</td>
<td>Semantic</td>
<td>Binary</td>
<td>Whether a parameter have a finite interval value</td>
</tr>
<tr>
<td>#AbsLoc</td>
<td>Semantic</td>
<td>Numeric</td>
<td>The (normalized) number of abstract locations accessed in the arguments</td>
</tr>
<tr>
<td>#ArgString</td>
<td>Semantic</td>
<td>Numeric</td>
<td>The (normalized) number of string arguments</td>
</tr>
</tbody>
</table>
Winning Features

- Interval analysis
- loops iterating on finite strings
- library calls that return integers or manipulate strings

```c
str = "hello world";
for (p = str; *p; p++)
...
```

```c
int r = lib1();
lib2(str1, str2);
```
Winning Features

• Interval analysis

• loops iterating on finite strings

• library calls that return integers or manipulate strings

```
str = "hello world";
for (p = str; *p; p++)
...
```

```
int r = lib1();
lib2(str1, str2);
```

- finite string
- array access
- ptr increment
- return integer
- str manipulation
Winning Features

- Taint analysis
- Library calls not propagating user inputs

\[
\begin{align*}
    r1 &= \text{random}() \\
    r2 &= \text{strlen}(s) \\
    r3 &= \text{fread}(fd, buf, len) \\
    r4 &= \text{recv}(s, len, flags)
\end{align*}
\]
Winning Features

- Taint analysis
- Library calls not propagating user inputs

\[
\begin{align*}
r1 &= \text{random}() \\
r2 &= \text{strlen}(s)
\end{align*}
\]

\[
\begin{align*}
r3 &= \text{fread}(fd, buf, len) \\
r4 &= \text{recv}(s, len, \text{flags})
\end{align*}
\]
Summary

• First *selectively unsound* static analysis

• more effective than uniformly sound / unsound ones

• systematic way to tune unsoundness by ML
Summary

• First **selectively unsound** static analysis
  
  • more effective than uniformly sound / unsound ones
  
  • systematic way to tune unsoundness by ML