2017 IEEE/ACM 39th International Conference on Software Engineering

Machine-Learning-Guided Selectively Unsound Static Analysis

Kihong Heo
Seoul National University
Seoul, Korea

Hakjoo Oh*
Korea University
Seoul, Korea

Kwangkeun Yi*
Seoul National University
Seoul, Korea

***Abstract*—We present a machine-learning-based technique for selectively applying unsoundness in static analysis. Existing bug-finding static analyzers are unsound in order to be precise and scalable in practice. However, they are uniformly unsound and hence at the risk of missing a large amount of real bugs. By being sound, we can improve the detectability of the analyzer but it often suffers from a large number of false alarms. Our approach aims to strike a balance between these two approaches by selectively allowing unsoundness only when it is likely to reduce false alarms, while retaining true alarms. We use an anomaly-detection technique to learn such harmless unsoundness. We implemented our technique in two static analyzers for full C. One is for a taint analysis for detecting format-string vulnerabilities, and the other is for an interval analysis for buffer-overflow detection. The experimental results show that our approach significantly improves the recall of the original unsound analysis without sacrificing the precision.**

***Keywords*-Static Analysis, Machine Learning, Bug-finding**

I. Introduction

Any realistic bug-finding static analyzers are designed to be unsound. Ideally, a static analyzer is expected to be sound, precise, and scalable; that is, it should be able to consider all program executions and hence do not miss any intended bug while avoiding false positives and scaling to large programs. In reality, however, achieving the three at the same time is extremely challenging, and therefore existing commercial static analysis tools (e.g., [1]) and published static bug-finders (e.g., [2], [3], [4], [5], [6]) trade soundness in order to obtain acceptable performance in precision and scalability.

To our knowledge, all of the existing unsound analysis tools are *uniformly* unsound. For instance, since loops and unknown library calls are major sources of imprecision in static analysis, most static bug-finding tools compromise soundness in analyzing them (e.g., [2], [3], [4], [5], [6]); loops are unrolled for a fixed number of times and subsequent loop iterations are ignored entirely, and unknown library calls are considered as pre-defined behaviors such as skip. All of these approaches are uniformly unsound in that they ignore *every* loop and library call in a given program regardless of their different conditions.

However, this uniform approach to unsoundness has a considerable shortcoming; it causes the analysis to miss a significant amount of real bugs. For instance, our taint analysis for detecting format-string vulnerabilities ignores the possible data flows of all unknown library calls in the program and therefore only report 5 false alarms in the 13 benchmark C programs (Section V). However, it only managed to detect 16 bugs among the 106 potentially detectable format-string bugs. In other words, this unsound analysis has low false positive rate (FPR = $\frac{\#\text{False Alarms}}{\#\text{All Alarms}}$) but it has high false negative rate (FNR = $\frac{\#\text{Missing Bugs}}{\#\text{All Bugs}}$).

On the other hand, a simple-minded, uniformly sound analysis poses the opposite problem; it has low FNR at the cost of high FPR. For example, a simple solution to decrease the FNR of the unsound taint analysis is to modify the analysis to consider the potential data flows of *every* unknown library call in the program. This uniformly sound analysis is able to find all 106 bugs in the benchmark programs. However, it reports 276 false alarms too.

Our work is to reduce the FNR of an unsound bug-finder while maintaining the original (low) FPR by being *selectively* unsound only when it is likely to be harmless. For example, we unsoundly analyze library calls only when it is likely to reduce FPR while maintaining low FNR. With our approach, the selectively unsound taint analysis reports 92 real bugs (among 106) with 27 false alarms only.

We achieve this by using a machine learning technique that is specialized for anomaly detection [7]. Our key insight is that the program components (e.g., loops and library calls) that produce false alarms are alike, predictable, and sharing some common properties. Meanwhile, the real bugs are often caused by different reasons that are atypical and unpredictable in their own ways (Section III-B2) [8]. Based on this observation, we aim to capture the common characteristics of the harmless and precision-decreasing program components by using one-class support vector machines. The entire learning process in our approach (i.e. generating labelled data and learning a classifier) is fully automatic once a codebase with known bugs is given.

The experimental results show that our method effectively reduces false negatives of the baseline analyzer without sacrificing its precision. We evaluated our method with two realistic static analyzers for C and open-source benchmarks. The first experiment is done with a taint analysis for finding out format-string bugs. In our benchmarks with 106 bugs, the baseline, uniformly unsound analysis detects 16 bugs with 5 false alarms (FPR: 24%, FNR: 85%). Uniformly improving the soundness impairs the precision too much: it reports 106 real bugs with

[*] Corresponding authors

1558-1225/17 $31.00 © 2017 IEEE

DOI 10.1109/ICSE.2017.54

```
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
```

Fig. 1. Example program

276 false alarms (FPR: 72%). Our selectively unsound analysis maintains the original precision while greatly decreasing the number of false negatives: it reports 92 bugs with 27 false alarms (FPR: 23%, FNR: 13%). The second experiment is done with an interval analysis for buffer-overflow detection, where we control the soundness for both loops and library calls. In the benchmarks with 138 bugs, the uniformly unsound analysis detects 33 bugs with 104 false alarms (FPR: 76%, FNR: 76%). The uniformly sound analysis detects 118 bugs with 677 false alarms (FPR: 85%). Our selectively unsound analysis detects 96 bugs with 266 false alarms (FPR: 73%, FNR: 30%).

To summarize, our contributions are as follows:

- We present a new approach of selectively employing unsoundness in static analysis. All of the existing bug-finding static analyzers are uniformly unsound.
- We present a machine-learning technique that can automatically tune a static analysis to be selectively unsound. Our technique is based on anomaly detection with automatic generation of labelled data.
- We demonstrate the effectiveness of the technique by experiments with two bug-finding static analyzers for C.

II. Overview

We illustrate our approach using a static analysis with the interval domain. The goal of the analysis is to detect buffer overflow bugs in a program. For simplicity, we only concern with loops in this section, which could be a potential cause of the buffer overflow bugs.

Consider a simple program in Figure 1. In the program, there are two loops and two buffer-access expressions. The first loop iterates over a constant string until the null value in the string is found. In the loop, buffer access 1 is always safe, since `i` is guaranteed to be smaller than the length of `str` inside the loop. On the other hand, buffer access 2 is not always safe, because the index `i` has the value of `size` after the second loop, which can be an arbitrary value due to the external input and may cause a buffer overflow.

A. Uniformly Unsound Analysis

Consider an analysis that is uniformly unsound for every loop. That is, all the loops in the given program are unrolled for a fixed number of times, and subsequent loop iterations are ignored during the analysis. From the perspective of such an unsound analysis, the example program is treated as follows.

```
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
```

Note that each loop is unrolled once and replaced with an if-statement. The analysis does not report a false alarm for buffer access 1, since the value of `i` remains as $[0, 0]$. However, it also fails to report a true alarm for buffer access 2; the value of `i` is approximated to $[0, 0]$, hence the analysis considers the buffer access to be safe.

B. Uniformly Sound Analysis

On the other hand, a sound interval analysis can detect the bug at buffer access 2 with a false alarm at buffer access 1. Inside the first loop, the analysis conservatively approximates the value of `i` to $[0, +\infty]$, since this value is not refined by the loop condition `!str[i]`. It is because the interval domain cannot capture non-convex properties (e.g. `i` $\neq 11$, where 11 is the null index of `str`). Thus, the analysis reports an alarm for buffer access 1 as a potential buffer overflow error, which is a false alarm that we want to avoid. Meanwhile, the variable `i` in the second loop is upper bounded by `size` whose range is approximated as $[0, +\infty]$ due to the unknown input value. Therefore the analyzer reports an alarm for buffer access 2, which is a true alarm in this case.

C. Selectively Unsound Analysis

Our selectively unsound analyzer applies unsoundness only to the loops that are likely to remove false alarms only. In the example program in Figure 1, we ignore the first loop since analyzing it soundly results in reporting a false alarm at buffer access 1. The second loop, on the other hand, needs to be analyzed soundly, since it has the possibility of causing an actual buffer overflow. The selectively unsound analysis on the given program corresponds to analyzing the following program.

```
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
```

Note that we only unroll the first loop, not the second loop. By being unsound for the first loop and sound for the second loop, the analysis is able to report the true alarm for buffer access 2 while avoiding the false alarm for buffer access 1.

D. Our Learning Approach

We achieve the selectively unsound analysis via machine learning-based anomaly detection. Assume that we have a codebase and a set of features. The codebase is a set of programs in which all the bugs are found and their locations are annotated so that we can classify alarms into true or false alarms. Then, we need to decide which set of program components to apply unsoundness selectively. In our example, it is the set of loops in the program we want to analyze. The features in this case describe general characteristics of the loops.

The learning phase consists of three steps.

1) We collect harmless loops from the codebase. A loop is harmless if unsoundly analyzing the loop does not cause to miss real bugs but reduces false alarms. For simplicity, we assume there is only one program in the codebase, and the program contains n loops. When analyzed soundly, it reports certain number of true alarms and false alarms. Then, we examine each loop by replacing it with an if-statement (i.e., unrolling) one by one and compare the result to that of the original program. If the replacement of a loop makes the number of true alarms remain same, but makes the number of false alarms decrease, we consider the loop to be harmless. We collect all the loops satisfying the condition.
2) Next, we represent the loops as feature vectors. Once all the harmless loops in the codebase are collected, we create a feature vector for each loop using the set $f = \{f_1, f_2, \ldots, f_k\}$ where f_i is a predicate over loops. For example, f_1 may indicate whether a loop has a conditional statement containing nulls.
3) Finally, having the generated feature vectors as training data, we learn a classifier that can distinguish such harmless loops. We use one-class classification algorithm [7] for learning the classifier that requires only positive examples (i.e., harmless loops). We use the anomaly detection algorithm to learn the common characteristics and regularities of the harmless loops.

In the testing phase, the classifier takes the feature vectors of all the loops in a new program as an input. If the classifier considers a loop to be harmless, then the loop is analyzed unsoundly, meaning that it is unrolled once and replaced with an if-statement. Otherwise, if the classifier considers a loop to be harmful (i.e., anomaly), then the loop is analyzed soundly.

III. Our Technique

Our goal is to find *harmless* components and selectively employ unsoundness only to them. In this section, we describe how to build a selectively unsound static analyzer in detail. First, we introduce a parameterized static analysis that applies unsoundness only to certain program components. Then, we explain how to learn a statistical model from an existing codebase, which is used to derive a soundness parameter.

A. Parameterized Static Analysis

Our analysis employs a parameterized strategy for selecting the set of program components that will be analyzed soundly. This is a variant of the well-known setting for the parameterized static analysis [9], [10], except the parameter controls the soundness of the analysis, not the precision.

Let $P \in Pgm$ be a program that we want to analyze. $\mathbb{C}_P$ is the set of program points in P. $\mathbb{J}_P$ is the set of program components such as the set of loops, the set of library calls, or the set of other operations in P. In the rest of this section, we omit the subscript P from $\mathbb{C}_P$ and $\mathbb{J}_P$ when there is no confusion.

The selectively unsound static analyzer is a function

$$F : Pgm \times \wp(\mathbb{J}) \rightarrow \wp(\mathbb{C})$$

which is parameterized by the soundness parameter $\pi \in \wp(\mathbb{J})$ (i.e. a set of program components). Given a program P and its parameter π, the analyzer outputs alarms (i.e. a set of program points).

A soundness parameter $\pi \in \wp(\mathbb{J})$ is a set of program components which need to be analyzed soundly. In other words, it selects the program components that are likely to produce true alarms as a result of detecting real bugs in the program. For instance, when $\mathbb{J} = \{j_1, \cdots, j_n\}$ is the set of loops in the program P, $j_i \in \pi$ means that the ith loop in the program is not considered to be harmless; we analyze the loop as it is rather than unrolling the loop once and ignoring all the subsequent loop iterations.

We want to find a good soundness parameter which allows the analyzer to apply costly soundness only to the necessary components which are not harmless. Let $\mathbf{1}$ be the parameter where every component is selected and $\mathbf{0}$ be the parameter where no component is selected. Then, $F(P, \mathbf{1})$ denotes the analysis that is fully sound, which can detect the maximum number of the real bugs along with lots of false alarms. $F(P, \mathbf{0})$ means the fully unsound analysis, reporting the minimum number of false alarms with risk of missing many real bugs. For our analysis, it is important to find a proper parameter which strikes the balance between $\mathbf{1}$ and $\mathbf{0}$, reporting false alarms as few as possible while detecting most of the real bugs.

B. Learning a Classifier

We want to build a classifier which can predict whether a given program component is harmless or not. The classifier in our approach exploits general properties of harmless components and uses the information for new, unseen programs.

1) Features: We define features to capture common properties of program components. Features are either syntactic or semantic properties of program components, which have either binary or numeric values. For simplicity, we assume them to be binary properties: $f_i : \mathbb{J} \rightarrow \{0, 1\}$. Given a set of features, we can derive a feature vector for each program component. Suppose that we have n features: $f = \{f_1, \ldots, f_n\}$. With the set of features, each program component $j \in \mathbb{J}$ can be represented as a feature vector $f(j) = \langle f_1(j), \ldots, f_n(j) \rangle$.

Our approach requires analysis designers to come up with a set of features for each parameterized static analysis F. In Section IV, we discuss how to construct program features with two case studies for loops and library calls.

2) Learning Process: A classifier is defined as a function $\mathcal{C} : \{0,1\}^n \rightarrow \{0,1\}$ which takes a feature vector of a program component as an input. It returns 1 if it considers the component to be harmless or 0, otherwise.

We define a model $\mathcal{M} : Pgm \rightarrow \wp(\mathbb{J})$ that is used to derive a soundness parameter for a given program as follows:

$$\mathcal{M}(P) = \{j \in \mathbb{J} \mid \mathcal{C}(f(j)) = 0\}.$$

The model collects the program components that are potentially harmful, which may cause real bugs. With the model, we run the static analysis for a new, unseen program P: $F(P, \mathcal{M}(P))$. That is, we first obtain the soundness parameter $\mathcal{M}(P)$ from the model and instantiate the static analysis with the parameter. As a result, the analysis becomes sound for the program components that are selected by the parameter from the model and unsound for the others.

We learn the model with One-Class Support Vector Machine (OC-SVM) [7]. OC-SVM is an unsupervised algorithm that learns a model for anomaly detection: classifying new data as similar or different to its training data. Our intuition is that harmless program components tend to be typical, sharing common properties, whereas harmful components are atypical, therefore difficult to be characterized. It is because bugs in the real world are introduced unexpectedly by nature. In addition, collecting examples for all kinds of bugs is infeasible, whereas collecting and generalizing the characteristics of harmless components is relatively easy to achieve. Therefore, we use this one-class classification method; it only requires positive examples (e.g., harmless loops) that are expected to share some regularities, learns such regularities, and classifies new data as similar or different to its training data.

Note that the characteristics of harmless components are largely determined by the design choices of a given static analysis (e.g., abstract domain), whereas that of harmful components are not affected by the analysis design. For example, for an interval analysis of C programs, the following loops are typically harmless:

- Loops iterating over constant strings:

  ```
  str=``hello world'';
  for(i=0; !str[i]; i++) // false alarm
    ...
  ```

  As explained before, analyzing such loops soundly is likely to cause false alarms, rather than detecting true bugs, because of the non-disjunctive limitation of the interval domain.
- Loops involving variable relationships:

  ```
  p=malloc(len);
  for (i = 0; i < len; i++)
    p[i] = ... // false alarm
  ```

  Sound analysis of this kind of loops is likely to produce false alarms because of the non-relational limitation of the interval domain. The analysis cannot track the relationship between the value of `len`, the value of `i`, and the size of buffer `p`.

Algorithm 1 Training data generation

```
1: $\mathbf{T} := \emptyset$
2: for all $(P_i, B_i) \in \mathbf{P}$ do
3:     $A_i = F(P_i, \mathbf{1})$
4:     $(A_t, A_f) := (A_i \cap B_i, A_i \setminus B_i)$
5:     for all $j \in \mathbb{J}_{P_i}$ do
6:         $A'_i = F(P_i, \mathbf{1} \setminus \{j\})$
7:         $(A'_t, A'_f) = (A'_i \cap B_i, A'_i \setminus B_i)$
8:         if $|A'_t| = |A_t| \ \wedge \ |A'_f| < |A_f|$ then
9:             $\mathbf{T} := \mathbf{T} \cup \{f(j)\}$
10:        end if
11:    end for
12: end for
```

3) Generating Training Data: From an existing codebase, we generate training data for learning the classifier. The training dataset is composed of a set of feature vectors. Note that we only collect the feature vectors of harmless components, because OC-SVM is designed to learn the regularities of positive examples. The positive examples in our case are the harmless components.

The codebase of the system is a set of annotated programs $\mathbf{P} = \{(P_1, B_1), \ldots, (P_n, B_n)\}$, in which each program P_i is associated with a set of buggy program points $B_i \subseteq \mathbb{C}_{P_i}$. Once all the programs in the codebase is annotated accordingly, we can automatically generate training data for the classifier. We first applies unsoundness to each component one by one, runs the analysis, and collects the feature vectors from all the harmless components in the given codebase. We consider a program component to be harmless if the number of true alarms remains same and the number of false alarms is decreased, when analyzed unsoundly.

The algorithm for generating training data is shown in Algorithm 1. For each program P_i in the codebase, we run the fully sound static analysis and classify the output alarms A_i into true alarms A_t and false alarms A_f (line 3 and 4). Then, for each program component $j \in \mathbb{J}_{P_i}$, we run the static analysis without the jth component (i.e. $\mathbf{1} \setminus \{j\}$) (line 6). The component j is considered to be harmless when the analysis which is unsound for j still captures all the real bugs (i.e. $|A'_t| = |A_t|$) but reports fewer false alarms (i.e. $|A'_f| < |A_f|$) compared to the fully sound analysis (line 8). We collect feature vectors from all the harmless program components into the training set $\mathbf{T} \subseteq \{0,1\}^n$.

IV. Instance Analyses

In this section, we present a generic static analysis that is selectively unsound for loops and library calls as well as a set of features for them. We have chosen loops and library calls because they are the main sources of false alarms from real-world static analyzers and thus often made unsound in practice (e.g. [2], [3], [4], [5], [6]). In the analysis, loops are unrolled for a fixed number of times and library calls are simply ignored

$$
\begin{array}{rcl}
F_\pi(L := E, s) & = & s[\mathcal{L}(L,s) \mapsto \mathcal{V}(E,s)] \\
F_\pi(C_1; C_2, s) & = & F_\pi(C_2, F_\pi(C_1, s)) \\
F_\pi(\texttt{if}\ E\ C_1\ C_2, s) & = & F_\pi(C_1) \sqcup F_\pi(C_2) \\
F_\pi(\texttt{while}_l\ E\ C, s) & = & \begin{cases} \mathsf{fix}(\lambda X.s \sqcup F_\pi(C, X)) & \text{if } l \in \pi \\ F_\pi(C, s) & \text{otherwise} \end{cases} \\
F_\pi(L := \mathit{lib}_l(), s) & = & \begin{cases} s[\mathcal{L}(L,s) \mapsto \top] & \text{if } l \in \pi \\ s & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 2. Static analysis selectively unsound for loops and library calls

as skips. Our aim is to selectively unroll and ignore loops and library calls, respectively, only when doing so is harmless.

We present two instances of the analysis, one for an interval analysis and the other for a taint analysis. The interval analysis is used to find out possible buffer-overflow errors, and the taint analysis is for detecting format string vulnerabilities (i.e. uses of unchecked user input as format string parameters of certain C functions such as `printf`). The soundness of these instance analyses is tuned by our technique (Section III), where we used the same set of features designed for the generic analysis.

In Section IV-A, we define the generic analysis with features. Section IV-B and Section IV-C present two instances, namely the interval analysis and the taint analysis.

A. A Generic, Selectively Unsound Static Analysis

1) Abstract Semantics: We consider a family of static analyses whose soundness is parametric for loops and library calls. Consider the following simple imperative language:

$$
\begin{array}{rcl}
C & \rightarrow & L := E \mid C_1; C_2 \mid \texttt{if}\ E\ C_1\ C_2 \\
 & \mid & \texttt{while}_l\ E\ C \mid L := \mathit{lib}_l() \\
E & \rightarrow & n \mid L \mid \texttt{alloc}_l(E) \mid \&L \mid E_1 + E_2 \\
L & \rightarrow & x \mid *E \mid E_1[E_2]
\end{array}
$$

A command is an assignment, sequence, if statement, while statement, or a call to an unknown library function. In the program, loops and library calls are labelled and the set of labels forms $\mathbb{J}$ in Section III-A. The parameter space is the set of all subsets of program labels, i.e., $\wp(\mathbb{J})$. We assume that labels in the program are all distinct. An expression is an integer (n), l-value expression (L), array allocation ($\texttt{alloc}_l(E)$) where E is the size of the array to be allocated and l is the label for the allocation site, address-of expression ($\&L$), or compound expression ($E{+}E$). An l-value expression is a variable (x) or array access expression ($E_1[E_2]$).

The abstract semantics of the analysis is defined in Figure 2. The analysis is parameterized by $\pi \in \wp(\mathbb{J})$, a set of labels, and is unsound for loops and library calls not included in π. The abstract semantics is defined by the semantic function $F_\pi : C \times \mathbb{S} \rightarrow \mathbb{S}$, where $\mathbb{S}$ is the domain of abstract states mapping abstract locations to abstract values, i.e., $\mathbb{S} = \mathbb{L} \rightarrow \mathbb{V}$. The analysis is generic in that abstract locations ($\mathbb{L}$) and values ($\mathbb{V}$) are unspecified. They will be given for each analysis instance in subsequent subsections. We assume that the abstract domain is accompanied by two functions $\mathcal{L} : L \times \mathbb{S} \rightarrow \wp(\mathbb{L})$ and $\mathcal{V} : E \times \mathbb{S} \rightarrow \mathbb{V}$, which compute abstract locations and values of given l-value and r-value expressions, respectively.

The abstract semantics is standard except for the selective treatment of soundness. For a loop statement (while$_l$ E C), the analysis applies the usual (sound) fixed point computation (fix is a pre-fixpoint operator) when the label l is included in the parameter π. When a loop is not included in π, the analysis ignores the loop and execute the body C only once (i.e. unrolling the loop once). For unknown library calls, the analysis conservatively updates the return location L when l is chosen, i.e., $l \in \pi$. Otherwise, we completely ignore the effect of the library call. Thus, π determines how soundly we analyze the program with respect to loops and unknown library calls. For instance, when $\pi = \mathbb{J}$, the analysis is maximally conservative for loops and library calls, and when $\pi = \emptyset$, the analysis is completely unsound and ignores all of the loops and library calls in the program.

$$
\begin{array}{rcl}
\mathbb{L} & = & \mathit{Var} + \mathit{AllocSite} \\
\mathbb{V} & = & \mathbb{I} \times \wp(\mathbb{L}) \times \wp(\mathbb{A}) \\
\mathbb{I} & = & \{\bot\} \cup \{[l, u] \mid l, u \in \mathbb{Z} \cup \{\pm\infty\}\} \\
\mathbb{A} & = & \mathbb{L} \times \mathbb{I} \times \mathbb{I} \\
\mathcal{L}(x, s) & = & \{x\} \\
\mathcal{L}(*E, s) & = & \mathcal{V}(E, s).2 \\
\mathcal{L}(E_1[E_2], s) & = & \{a \mid \langle a, \_, \_\rangle \in \mathcal{V}(E_1, s).3\} \\
\mathcal{V}(n, s) & = & \langle [n, n], \emptyset, \emptyset \rangle \\
\mathcal{V}(L, s) & = & \bigsqcup \{s(l) \mid l \in \mathcal{L}(L, s)\} \\
\mathcal{V}(\texttt{alloc}_l(E), s) & = & \langle \bot, \{l\}, \{\langle l, [0,0], \mathcal{V}(E, s).1 \rangle\} \rangle \\
\mathcal{V}(\&L, s) & = & \langle \bot, \mathcal{L}(L, s), \emptyset \rangle \\
\mathcal{V}(E_1 + E_2, s) & = & \mathcal{V}(E_1, s) \hat{+} \mathcal{V}(E_2, s)
\end{array}
$$

Fig. 3. Abstract domain and semantics for interval analysis

$$
\begin{array}{rcl}
\mathbb{L} & = & \mathit{Var} + \mathit{AllocSite} \\
\mathbb{V} & = & \{\bot, \top\} \times \wp(\mathbb{L}) \\
\mathcal{L}(x, s) & = & \{x\} \\
\mathcal{L}(E_1[E_2], s) & = & \mathcal{V}(E_1, s).2 \\
\mathcal{V}(n, s) & = & \begin{cases} \langle \top, \emptyset \rangle & \text{if } n \in \mathbb{T} \\ \langle \bot, \emptyset \rangle & \text{otherwise} \end{cases} \\
\mathcal{V}(L, s) & = & \bigsqcup \{s(l) \mid l \in \mathcal{L}(L, s)\} \\
\mathcal{V}(\texttt{alloc}_l(E), s) & = & \langle \bot, \{l\} \rangle \\
\mathcal{V}(\&L, s) & = & \langle \bot, \mathcal{L}(L, s) \rangle \\
\mathcal{V}(E_1 + E_2, s) & = & \mathcal{V}(E_1, s) \sqcup \mathcal{V}(E_2, s)
\end{array}
$$

Fig. 4. Abstract domain and semantics for taint analysis

2) Features: We have designed a set of features for loops and library calls, which can be used for instantiating the generic analysis above. We examined open-source C programs and identified 37 features (Figure 5) that describe common characteristics of loops and library calls in typical C programs.

The features are classified into syntactic and semantic features. A syntactic feature describes a property that can be checked by a simple syntax analysis. For example, a syntactic feature characterizes loops whose conditional expressions involve constant values, or library calls whose return type is an integer. A semantic feature describes a property that requires a (yet simple) data-flow analysis. For instance, a semantic feature for loops describes that the loop condition involves an expression whose value depends on some external input of the program:

```
c = input();  // external input
```

```
b = c;
while (a < b) { ... }
```

To figure out that the value of b comes from the external input, we need to track the data-flow of the external value. Each feature is either binary or numeric, where all the numeric features are normalized to a real number between 0 and 1 based on relative quantities within a single program.

We designed those features with generality in mind so that the features can be reused for different analyses as much as possible. Note that the features in Figure 5 are not dependent on a particular static analysis, but describe rather general, syntactic and semantic program properties. We use the same set of features for the interval and taint analyses and show that we can effectively tune the soundness of both analyses with the single set of features as shown in Section V.

B. Instantiation 1: Interval Analysis

We first instantiate the generic analysis with the interval domain and use it to find out potential buffer-overflow errors in the program.

The generic analysis left out the definitions of abstract locations ($\mathbb{L}$), abstract values ($\mathbb{V}$), and the evaluation functions for them ($\mathcal{L}$ and $\mathcal{V}$). These definitions for the interval analysis are given in Figure 3. An abstract location is either a variable or an allocation-site. An abstract value is a tuple of an interval ($\mathbb{I}$), which is an abstraction of set of numeric values, a points-to set ($\wp(\mathbb{L})$) and a set of abstract arrays ($\wp(\mathbb{A})$). Abstract array $\langle a, o, s\rangle$ has the abstract location($a \in \mathbb{L}$), offset ($o \in \mathbb{I}$), and size ($s \in \mathbb{I}$). The evaluation function $\mathcal{L}$ takes an l-value expression and an abstract state, and computes the set of abstract locations that the l-value denotes. The function $\mathcal{V}(E, s)$ evaluates to the abstract value of E under s. In the definition, we write $\mathcal{V}(E, s).n$ for the nth component of the abstract value of $\mathcal{V}(E)$.

The analysis reports a buffer-overflow alarm when the index of an array can be greater than its size according to the analysis results. For example, consider an expression arr[idx]. Suppose the analysis concludes that arr has an array of $\langle l, [0, 0], [5, 10]\rangle$ (i.e. an array of size $[5, 10]$) and the interval value of idx is $[3, 7]$. The analysis raises an alarm at the array expression because the index value may exceed the size of the array (e.g. when the size is 5 and the index is 7).

C. Instantiation 2: Taint Analysis

The second instance is a taint analysis for detecting format string vulnerabilities in C programs. The abstract domain and semantics are given in Figure 4. The analysis combines a taint analysis and a pointer analysis, and therefore an abstract location is still either a variable or an allocation-site. An abstract value is a tuple of a taint value and a points-to set. The taint domain consists of two abstract values: $\top$ is used to indicate that the value is tainted and $\bot$ represents untainted values. For simplicity, we model taint sources by a particular set $\mathbb{T} \subseteq \mathbb{Z}$ of integers; constant integer n generates a taint value $\top$ if $n \in \mathbb{T}$. In actual implementation, $\top$ is produced by function calls that receives user input such as fgets. The analysis reports an alarm whenever a taint value is involved in a format string parameter of functions.

V. Experiments

We empirically show the effectiveness of our approach on selectively applying unsoundness only to harmless program components. We design the experiments to address the following questions:

- **Effectiveness of Our Approach**: How much is the selectively unsound analysis better than the fully sound or fully unsound analyses?
- **Efficacy of OC-SVM**: Does the one-class classification algorithm outperform two-class classification algorithms?
- **Feature Design**: How should we choose a set of features to effectively predict harmless program components?
- **Time Cost**: How does our technique affect cost of analysis?

A. Setting

1) Implementation: We have implemented our method on top of a static analyzer for full C. It is a generic analyzer that tracks all of numeric, pointer, array, and string values with flow-, field-, and context-sensitivity. The baseline analyzer is unsound by design to achieve a precise bug-finder; it ignores complex operations (e.g., bitwise operations and weak updates) and filters out reported alarms that are unlikely to be true.

We modified the baseline analyzer and created two instance analyzers, an interval analysis and a taint analysis, as described in Section IV. For each analysis, we built a fully sound version (BASELINE), a uniformly unsound version (UNIFORM), and a selectively unsound version (SELECTIVE) with respect to the soundness parameter in Section IV. In the interval analysis for buffer-overflow errors, UNIFORM is set to be uniformly unsound for every loop and library call, and SELECTIVE is selectively unsound for them. In the taint analysis for format string vulnerabilities, UNIFORM is uniformly unsound for all the library calls (but not for loops), and SELECTIVE is selectively unsound for them.

To implement the OC-SVM classifier, we used scikit-learn machine-learning package [11] with the default setting of the algorithm (specifically, we used the radial basis function (RBF) kernel with $\gamma = 0.1$ and $\nu = 0.1$).

2) Benchmark: Our experiments were performed on 36 programs whose buggy program points are known. They are the programs from open source software packages or previous work on static analysis evaluations [12], [13]. Table I and II contain the list of the benchmark programs for the interval and the taint analysis, respectively. SM-X, BIND-X, and FTP-X are model programs from [12], which contain buffer overflow vulnerabilities. Most of the bugs in the benchmarks are reported as critical vulnerabilities by authorities such as CVE [14]. In total, our benchmark programs have 138 real buffer-overflow bugs and 106 real format string bugs.

Target	Feature	Property	Type	Description
Loop	Null	Syntactic	Binary	Whether the loop condition contains nulls or not
	Const	Syntactic	Binary	Whether the loop condition contains constants or not
	Array	Syntactic	Binary	Whether the loop condition contains array accesses or not
	Conjunction	Syntactic	Binary	Whether the loop condition contains `&&` or not
	IdxSingle	Syntactic	Binary	Whether the loop condition contains an index for a single array in the loop
	IdxMulti	Syntactic	Binary	Whether the loop condition contains an index for multiple arrays in the loop
	IdxOutside	Syntactic	Binary	Whether the loop condition contains an index for an array outside of the loop
	InitIdx	Syntactic	Binary	Whether an index is initialized before the loop
	Exit	Syntactic	Numeric	The (normalized) number of exits in the loop
	Size	Syntactic	Numeric	The (normalized) size of the loop
	ArrayAccess	Syntactic	Numeric	The (normalized) number of array accesses in the loop
	ArithInc	Syntactic	Numeric	The (normalized) number of arithmetic increments in the loop
	PointerInc	Syntactic	Numeric	The (normalized) number of pointer increments in the loop
	Prune	Semantic	Binary	Whether the loop condition prunes the abstract state or not
	Input	Semantic	Binary	Whether the loop condition is determined by external inputs
	GVar	Semantic	Binary	Whether global variables are accessed in the loop condition
	FinInterval	Semantic	Binary	Whether a variable has a finite interval value in the loop condition
	FinArray	Semantic	Binary	Whether a variable has a finite size of array in the loop condition
	FinString	Semantic	Binary	Whether a variable has a finite string in the loop condition
	LCSize	Semantic	Binary	Whether a variable has an array of which the size is a left-closed interval
	LCOffset	Semantic	Binary	Whether a variable has an array of which the offset is a left-closed interval
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the loop
Library	Const	Syntactic	Binary	Whether the parameters contain constants or not
	Void	Syntactic	Binary	Whether the return type is void or not
	Int	Syntactic	Binary	Whether the return type is int or not
	CString	Syntactic	Binary	Whether the function is declared in `string.h` or not
	InsideLoop	Syntactic	Binary	Whether the function is called in a loop or not
	#Args	Syntactic	Numeric	The (normalized) number of arguments
	DefParam	Semantic	Binary	Whether a parameter are defined in a loop or not
	UseRet	Semantic	Binary	Whether the return value is used in a loop or not
	UptParam	Semantic	Binary	Whether a parameter is update via the library call
	Escape	Semantic	Binary	Whether the return value escapes the caller
	GVar	Semantic	Binary	Whether a parameters points to a global variable
	Input	Semantic	Binary	Whether a parameters are determined by external inputs
	FinInterval	Semantic	Binary	Whether a parameter have a finite interval value
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the arguments
	#ArgString	Semantic	Numeric	The (normalized) number of string arguments

Fig. 5. Features for typical loops and library calls in C programs

B. Effectiveness of Our Approach

We evaluate the effectiveness of our approach by comparing precision of SELECTIVE to that of the other analyzers, BASELINE and UNIFORM. We use cross-validation, a model validation technique for assessing how the results of a statistical analysis will generalize to new data. We show the results from three types of cross-validation: leave-one-out, 2-fold, and 3-fold cross-validation.

1) Leave-one-out Cross-validation: This is one of the most common types of cross-validation, which uses one observation as the validation set and the remaining observations as the training set. In case of the interval analysis, for example, among the 23 benchmark programs, one program is used for validating and measuring the effectiveness of the learned model, and the other remaining 22 programs are used for training.

Table I shows the results of the leave-one-out cross-validation for the interval analysis. We measured the number of true (**T**) and false (**F**) alarms from BASELINE, UNIFORM, and SELECTIVE. In terms of true alarms, BASELINE detects 118 real bugs (FNR: 14.5%) in the programs. While UNIFORM detects only 33 bugs (FNR: 76.1%), SELECTIVE effectively de-

			BASELINE		SELECTIVE		UNIFORM	
Program	**LOC**	**Bug**	**T**	**F**	**T**	**F**	**T**	**F**
SM-1	0.5K	28	28	18	28	15	13	5
SM-2	0.8K	2	2	16	1	4	0	0
SM-3	0.7K	3	3	3	3	3	0	0
SM-4	0.7K	10	10	6	10	6	6	0
SM-5	1.7K	3	3	6	3	6	0	0
SM-6	0.4K	1	0	0	0	0	0	0
SM-7	1.1K	2	2	32	0	2	0	0
BIND-1	1.2K	1	1	35	1	33	0	0
BIND-2	1.7K	1	1	45	0	41	0	0
BIND-3	0.5K	1	1	4	0	1	0	0
BIND-4	1.1K	2	2	0	0	0	0	0
FTP-1	0.8K	4	4	13	4	3	0	0
FTP-2	1.5K	1	1	7	1	6	0	3
FTP-3	1.5K	24	24	25	23	17	7	12
polymorph-0.4.0	0.7K	10	10	6	3	6	0	6
ncompress-4.2.4	1.9K	12	0	10	4	0	0	0
129.compress	2.0K	7	7	34	7	14	4	7
spell-1.0	2.2K	1	0	0	0	0	0	0
man-1.5h1	4.7K	6	5	60	1	28	0	13
256.bzip2	4.7K	3	3	149	3	21	3	21
gzip-1.2.4a	8.2K	13	11	87	8	34	0	24
bc-1.06	17.0K	2	0	57	0	10	0	9
sed-4.0.8	25.9K	1	0	64	0	14	0	4
Total		138	118	677	100	264	33	104

TABLE I
THE NUMBER OF ALARMS IN INTERVAL ANALYSIS

			BASELINE		SELECTIVE		UNIFORM	
Program	**LOC**	**Bug**	**T**	**F**	**T**	**F**	**T**	**F**
mp3rename-0.6	0.6K	1	1	0	1	0	1	0
ghostscript-8.71	1.5K	2	2	0	2	0	2	0
uni2ascii-4.14	5.7K	7	7	0	7	0	7	0
pal-0.4.3	7.4K	3	3	0	0	0	0	0
shntool-3.0.1	16.3K	1	1	10	1	1	1	0
sdop-0.61	23.9K	65	65	78	65	0	0	0
latex2rtf-2.3.8	28.7K	2	2	9	2	8	0	1
rrdtool-1.4.8	34.8K	1	1	12	1	1	1	0
daemon-0.6.4	58.4K	1	1	7	1	1	1	0
rplay-3.3.2	61.0K	3	3	7	2	4	1	2
urjtag-0.10	64.2K	12	12	78	6	0	0	0
a2ps-4.14	64.6K	6	6	26	3	12	1	0
dico-2.0	84.3K	2	2	46	1	1	1	2
Total		106	106	273	92	28	16	5

TABLE II
THE NUMBER OF ALARMS IN TAINT ANALYSIS

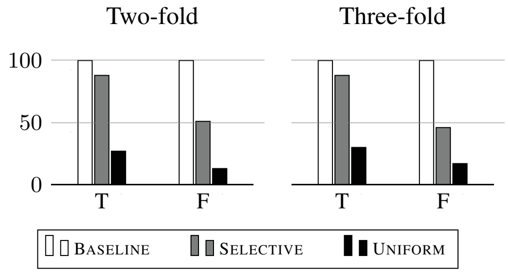

Fig. 6. Performance with different training and test data

tects 100 bugs (FNR: 27.5%). Meanwhile, BASELINE reports 677 false alarms (FPR: 85.2%).[1] UNIFORM, on the other hand, reports 104 false alarms (FPR : 75.9%), which indicates 573 alarms can be potentially removable by being unsound for loops and library calls. Among the 573 alarms, SELECTIVE can remove 72.1% (413/573) of them (FPR:72.5%).

Table II shows the results for the taint analysis. In total, BASELINE detects all of the 106 real format-string bugs in the programs, while UNIFORM detects only 16 bugs (FNR: 84.9%). On the contrary, SELECTIVE effectively detects 92 bugs (FNR: 13.2%). Meanwhile, BASELINE, UNIFORM, and SELECTIVE report 273, 5, and 28 false alarms, respectively. That is, among 273 false alarms, which can be potentially removable by being unsound for library calls, SELECTIVE can remove 89.7% (245/273) of them.

The result implies that selectively applying unsoundness is also crucial for reducing FPR of the analysis. For the interval analysis, the FPR is 85.2% for BASELINE and 75.9% for UNIFORM, whereas 72.5% for SELECTIVE on average. For the taint analysis, the FPR is 72.0% for BASELINE, 23.3% for SELECTIVE, 23.8% for UNIFORM on average.

2) Two- and Three-fold Cross-validation: Next, we evaluate the performance of the interval analysis with 2-fold and 3-fold cross-validation. The benchmark is randomly divided into 2 or 3 subsets that are equal size. Then, one of them is used as the validation set and the others as the training sets. We repeated this process ten times and reported the number of alarms for each trial.

Figure 6 shows the number of true and false alarms for each trial of 2-fold and 3-fold cross-validation. The numbers are normalized with respect to the number of alarms produced by BASELINE. In total, BASELINE reported 486 true alarms and 3,696 false alarms. SELECTIVE detected 427 (87.9%) true alarms, whereas UNIFORM detected only 129 (26.5%) true alarms in the 2-fold cross-validation. Compared to BASELINE, SELECTIVE reduced 1,812 (49.0%) false alarms, while UNIFORM reduced 3,216 (87.0%). During the 3-fold cross-validation, BASELINE reported 399 true alarms and 2,119 false alarms. In terms of true alarms, SELECTIVE detected 352 (88.2%) true alarms, whereas UNIFORM managed to detect only 119 (29.8%) true alarms. As for false alarms, among 1,769 (83.5%) false alarms that are reduced by UNIFORM, SELECTIVE was able to reduce 1,150 (54.3%).

C. *Efficacy of OC-SVM*

In this section, we justify the use of OC-SVM for learning common properties of harmless program components. We compare the performance of SELECTIVE whose classifier is learned by OC-SVM to that of three other analyzers with a binary classifier and two random classifiers, respectively.

Let BINARY be the analyzer with a binary classifier. We use C-SVM for the binary classifier, which is a support vector machine-based binary classification algorithm [15]. It learns two classes of training data (i.e. a set of harmless components and the complement set), and then decides whether a new input is harmless or not. In these experiments, we used the interval analyzer with leave-one-out cross validation.

RANDA and RANDB are the analyzers with random classifiers that are built and used for the comparison. RANDA randomly classifies components as harmless with the probability of 0.5. Stochastically, a half of loops and library calls are selected as harmless. RANDB randomly classifies components as harmless with the same probability of the OC-SVM. We ran each analyzer 10 times and measured the number of alarms for each trial.

Figure 7 compares the number of true and false alarms produced by SELECTIVE, BINARY, RANDA, and RANDB for 10 trials. BINARY reports more true alarms than SELECTIVE; BINARY reports 103 true alarms, whereas SELECTIVE reports 96 true alarms. However, using BINARY considerably sacrifices the precision; it reports 573 false alarms, whereas SELECTIVE reports only 266. The results from RANDA and RANDB are also inferior to SELECTIVE; RANDA reports 387.5 false alarms and 70.5 true alarms, and RANDB reports 267.2 false alarms with 79.4 true alarms on average.

The result shows SELECTIVE clearly outperforms the other classifiers. SELECTIVE is more precise than BINARY, indicating that the anomaly detection by OC-SVM is more suitable to find harmless components than the binary classification. Also, SELECTIVE always detects more bugs and reports less false alarms than other analyzers with the random classifiers. Despite the fact that RANDB detects more bugs than RANDA,

[1] In practice, eliminating these false alarms is extremely challenging in domain-unaware static analysis, because they arise from a variety of reasons: e.g., large recursive call cycles, unknown library calls, complex loops, heap abstractions, etc.

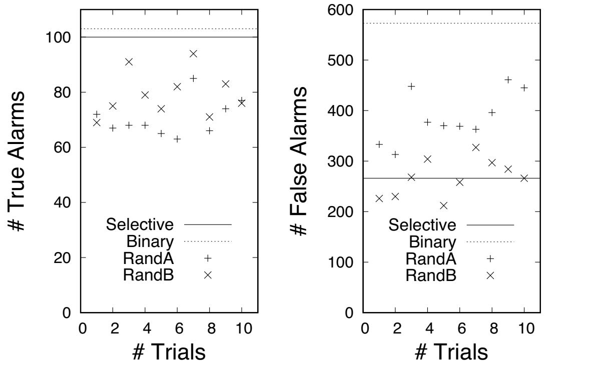

Fig. 7. Comparison between Selective, Binary, RandA, and RandB

it is still insignificant since both of them are much more imprecise than our system.

D. Feature Design

1) Wining Features: The learned classifier tells us which feature is most useful for learning harmless unsoundness. The features we used capture general characteristics of harmless program components. In order to determine the ordering of features, we used information gain which is the expected reduction in entropy when a feature is used to classify training examples (in classification, low entropy, i.e., impure data, is preferred) [16].

The results show that harmless loops tend to have pointers iteratively accessing (PtrInc) arrays (Array) or strings (FinString) with loop conditions that compare array contents with null (Null) or constant values (Const). These features collectively describe loops like the first example loop in Section II. The result also shows that most harmless library calls for the interval analysis return integer values (Int) and manipulate strings (CString). This is because our interval analyzer aggressively abstracts string values, so unsound treatment of string libraries (e.g., `strlen`, `strcat`) is likely to improve the analysis precision. For the taint analysis, the results show that library calls with less arguments (#Args) and abstract locations (#AbsLoc) (e.g., `random`, `strlen`) are likely to be irrelevant to propagation of user inputs compared to ones with more arguments (e.g., `fread`, `recv`).

2) Different Feature Sets: We measured the performance of the classifier with less features in three ways: 1) with syntactic features only; 2) with semantic features only; and 3) with randomly-chosen half of the features. For the interval analyzer, the classifier learned with only syntactic features reported 1% more bugs but 26% more false alarms than the classifier with all features, the classifier with only semantic features reported 1% more false alarms and missed 41% more bugs, and the classifier with half of the features reported 17% more false alarms and missed 1% more bugs on average.

E. Time Cost

We measured how long it takes to run each analysis on our benchmark programs and compare it with the time that our selective unsound analysis takes. For the benchmark programs in Table I, the sound interval analysis Baseline took 42.1 seconds for analyzing all the listed programs, Uniform only took 27.7 seconds, reducing the total time by 14.4 seconds (34.2%). Selective took 33.8 seconds, reducing the total time by 8.3 seconds (19.7%). RandA and RandB took longer than Selective: 35.4 and 37.5 seconds, respectively. In summary, Selective takes less time than Baseline, RandA, and RandB.

F. Discussion

As addressed in the experiments, our technique may miss some true alarms which can be detected by the fully sound analysis or fail to avoid some false alarms which are not reported by the fully unsound analysis. In this section, we discuss why these limitations occur and how to overcome.

1) Remaining False Alarms: Compared to the fully unsound analysis, our technique reports more false alarms. It is mainly because reporting the false alarms is inevitable in order to detect true alarms. Consider the following example program excerpted from SM-5:

```
1 size = 10 + positive_input();
2 arr = malloc(size);
3
4 for(i = 0; i < size; i++){
5   arr[i] = ...           // buffer access 1
6   arr[i+1] = ...         // buffer access 2
7 }
```

By soundly analyzing the loop, the analysis reports an alarm for the buffer-overflow bug at line 6 at the cost of a false alarm at line 5. The unsound analysis removes the false alarm, but it also fails to report the true alarm. Our selective method may decide to analyze such a loop soundly in order to detect the bug, even though reporting the false alarm is inevitable.

We found that these inevitable false alarms are the primary reason for Selective to report more false alarms compared to Uniform. For example, when analyzing SM-5 in our benchmark programs, five among six false alarms are inevitable. In order to remove such false alarms in a harmless way, we need a more fine-grained parameter space for soundness so that we can apply different degrees of soundness to different statements in a single loop, which would be an interesting future direction to investigate.

2) Missing True Alarms: Compared to the fully sound analysis, our technique reports less true alarms. It is mainly because the bugs are involved in typically-harmless loops. Consider the following code snippet from man-1.5h1:

```
char arr[10] = ``string'';
size = positive_input();
for (i = 0; i < size; i ++)
  skip;
arr[i] = 0;              // buffer access 1

for(i = 0; !arr[i]; i++)// buffer access 2
  skip;
```

The two buffer access expressions both contain buffer overflow bugs. However, our technique detects the first bug, but not the second. It is because it classifies the second loop as harmless– it learns that loops that iterate constant strings are likely to be harmless.

However, we found that most of the missing bugs share the root causes with other bugs detected by our technique. For instance, in the above example, fixing the first bug at line 5 automatically fixes the second one. In our case, therefore, missing true alarms is in fact not a huge drawback in terms of practicability.

VI. Related Work

A. Unsoundness in Static Analysis

Existing unsound static analyses are all uniformly unsound (e.g., [2], [3], [4], [5], [6]). In addition to their unsound handling of every loop and library call in a given program, they consider only a specific branch of all conditional statements in a program [2], deactivate all recursive calls [5], [3], or ignore all the possible inter-procedural aliasing [2], [5], [3]. As shown in this paper, these uniform approaches have a considerable drawback; it significantly impairs the capability of detecting real bugs. This paper is the first to tackle this problem and presents a novel approach of selectively employing unsoundness only when it is likely to be harmless.

Mangal et al. proposed an interactive system to control the unsoundness of static analysis online based on the user feedback [17]. They define a probabilistic Datalog analysis with "hard" and (unsound) "soft" rules, where the goal of the analysis is to find a solution that satisfies all of the hard rules while maximizing the weight of the satisfied soft rules. The feedback from analysis users is encoded as soft rules, and based on the feedback, the analysis is re-run and produces a report that optimizes the updated constraints. In our setting (non-Datalog), however, it is not straightforward to tune the unsoundness from user feedbacks. Instead, our approach automatically learns harmless unsoundness and selectively applies unsound strategies depending on the different circumstances.

Our goal is different from the existing work on unsoundness by Christakis et al. [18], which empirically evaluated the impact of unsoundness in a static analyzer using runtime checking. They instrumented programs with the unsound assumptions of the analyzer and check whether the assumptions are violated at runtime. On the contrary, we introduce a new notion of selective unsoundness and evaluate its impact in terms of the number of true alarms and false alarms reported.

B. Parametric Static Analysis

Our work uses a parametric static analysis in a novel way, where the parameters specify the degree of soundness, not the precision setting of the analysis. The existing parametric static analyses have been focused on balancing the precision and the cost of static analysis [19], [20], [21], [10]. They infer a cost-effective abstraction for a newly given program by iterative refinements [19], [20], impact pre-analyses [21], or learning from a codebase [10]. On the other hand, our goal is to find a soundness parameter striking the right balance between existing fully sound and unsound approaches. Furthermore, the existing techniques for deriving static analysis parameters (e.g., [19], [20], [21]) cannot be used for our purpose since it is simply impossible to automatically judge truth and falsehood of alarms. We address this problem by designing a supervised learning method that learns a strategy from a given codebase with known bugs. Because we have labelled data, using the learning algorithm via black-box optimization [10] is inappropriate. Instead, we use an off-the-shelf learning method, which uses the gradient-based optimization algorithm and works much faster than the black-box optimization approach.

C. Statistical Alarm Filtering

Our approach is orthogonal to statistical post-processing of alarms [22], [23], [24]. The post-processing (e.g. ranking) approach aims to remove false positives (reported false alarms). Instead, our approach aims to remove false negatives (unreported true alarms). From the undiscerning, uniformly unsound analysis that will have too many unreported true alarms, we tune it to be selectively unsound.

These post-processing systems are also complementary to our approach. Because in practice any realistic bug-finding static analyzer cannot but be unsound (for the analysis precision and scalability), our technique provides a guide on how to design an unsound one. The existing post-processing techniques (e.g. ranking) can be anyway applied to the results from such selectively unsound static analyzers.

VII. Conclusion

In this paper, we presented a novel approach for selectively employing unsoundness in static analysis. Unlike existing uniformly unsound analyses, our technique applies unsoundness only when it is likely to be harmless (i.e., in a way to reduce the number of false alarms while retaining true alarms). We proposed a learning-based method for automatically tuning the soundness of static analysis in such a harmless way. The experimental results showed that the technique is effectively applicable to two bug-finding static analyzers and reduces their false negative rates while retaining their original precision.

Acknowledgment

We thank Jonggwon Kim and Woosuk Lee for their implementation of the taint analysis, and Mina Lee for comments on an early version of the paper. This work was partly supported by Samsung Electronics, Samsung Research Funding Center of Samsung Electronics (No.SRFC-IT1502-07), and Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIP) (No.B0717-16-0098, Development of homomorphic encryption for DNA analysis and biometry authentication and No.R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security). This work was also supported by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea (NRF) (21A20151113068) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062).

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[2] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *ISMM*, 2008, pp. 131–140.

[3] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2005, pp. 115–125.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, 2002, pp. 234–245.

[5] Y. Xie and A. Aiken, "Saturn: A sat-based tool for bug detection," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, 2005, pp. 139–143.

[6] Y. Xie, A. Chou, and D. Engler, "Archer: Using symbolic, path-sensitive analysis to detect memory access errors," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[7] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[8] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[9] P. Liang, O. Tripp, and M. Naik, "Learning minimal abstractions," in *POPL*, 2011.

[10] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[11] Scikit-learn, http://scikit-learn.org.

[12] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004.

[13] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005, pp. 1–5.

[14] "Common vulnerabilities and exposures," https://cve.mitre.org.

[15] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92. New York, NY, USA: ACM, 1992, pp. 144–152. [Online]. Available: http://doi.acm.org/10.1145/130385.130401

[16] T. M. Mitchell, *Machine learning*. The Mc-Graw-Hill Companies, Inc., 1997.

[17] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 462–473.

[18] M. Christakis, P. Müller, and V. Wüstholz, "An experimental evaluation of deliberate unsoundness in a static program analyzer," in *VMCAI*, ser. LNCS, D. D'Souza, A. Lal, and K. G. Larsen, Eds., vol. 8931. Springer, 2015, pp. 333–351.

[19] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv, "Abstractions from tests," in *POPL*, 2012.

[20] X. Zhang, M. Naik, and H. Yang, "Finding optimum abstractions in parametric dataflow analysis," in *PLDI*, 2013.

[21] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective context-sensitivity guided by impact pre-analysis," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[22] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," in *SAS 2005: 12th Annual International Static Analysis Symposium*, ser. Lecture Notes in Computer Science, vol. 3672. Springer, 2005, pp. 203–217.

[23] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, 2003, pp. 295–315.

[24] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '04/FSE-12, 2004, pp. 83–93.