Assisting in Auditing of Buffer Overflow Vulnerabilities via Machine Learning

Qingkun Meng, Chao Feng, Bin Zhang, and Chaojing Tang

School of Electronic Science and Engineering, National University of Defense Technology (NUDT), Changsha, Hunan, China

Correspondence should be addressed to Chao Feng; chaofeng@nudt.edu.cn

Received 1 July 2017; Revised 10 October 2017; Accepted 27 November 2017; Published 21 December 2017

Academic Editor: Nazrul Islam

Copyright © 2017 Qingkun Meng et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Buffer overflow vulnerability is a kind of consequence in which programmers’ intentions are not implemented correctly. In this paper, a static analysis method based on machine learning is proposed to assist in auditing buffer overflow vulnerabilities. First, an extended code property graph is constructed from the source code to extract seven kinds of static attributes, which are used to describe buffer properties. After embedding these attributes into a vector space, five frequently used machine learning algorithms are employed to classify the functions into suspicious vulnerable functions and secure ones. The five classifiers reached an average recall of 83.5%, average true negative rate of 85.9%, a best recall of 96.6%, and a best true negative rate of 91.4%. Due to the imbalance of the training samples, the average precision of the classifiers is 68.9% and the average $F_1$ score is 75.2%. When the classifiers were applied to a new program, our method could reduce the false positive to 1/12 compared to Flawfinder.

1. Introduction

Buffer overflow occurs when the bytes of data used exceed the prepared allocated boundary on either the stack or the heap. It has been one of the most popular exploitable vulnerabilities since the 1980s. Its hazard to the target victim system ranges from denial of service to executing arbitrary code in administrator permission. Even though this type of vulnerability is not fresh anymore, hundreds of buffer overflow vulnerabilities are still reported every year.

From the perspective of source code, buffer write operations such as array write and memory manipulation provided by programming language like C/C++ are the main causes of buffer overflow. If not handled properly, even bounded functions like `strncpy` lead to overflow. Generally, the occurrence of buffer overflow relies on three main characteristics: user-input data controlling the buffer, no validation statement that enforces the use of data inside a safe scope, and the complexity of buffer operations causing the programmer to fail to add proper validation.

Primary methods of automatically detecting buffer overflow fall into two types: static analysis and dynamic test-case generation. Static analysis leverages a pattern to find vulnerabilities, whereas dynamic test-case generation tries to uncover the unexpected behaviors by executing the program with generated test cases. Some open-source static analysis tools can generate too many false positives, which cannot be completely reviewed [1]. Dynamic test generation generally involves fuzzing and symbolic execution. To detect vulnerabilities, fuzzing randomly generates a test case to trigger program faults, while symbolic execution collects constraints when walking through program paths and employs a constraint solver to generate related test cases. Fuzzing cannot understand the program thoroughly and comprehensively, while symbolic execution has problems in terms of path explosion, constraint solving, and memory modeling [2]. As a result, the main work of discovering vulnerabilities still relies on code auditors, which requires an enormous amount of manpower.

In this paper, a static method based on machine learning is proposed to narrow down the search scope of auditing buffer overflow vulnerabilities in source code. There are three contributions in this paper. First, we design seven kinds of static code attributes to represent buffer overflow according to the 22 taxonomies of buffer overflow [3]. Second, we append interprocedural sanitization graph (IPSG) and declaration-spread-sink graph (DSSG) to code property graph (CPG) [4]...
to form extended property graph (ECPG), which is described in Section 4 in detail, to extract static code attributes in source code. We use ECPG to extract the static code attributes of existing buffer overflow vulnerabilities obtained from common vulnerabilities and exposures (CVE) and map them to vectors. Third, we apply several supervised machine learning algorithms to train classifiers and we apply the classifiers to a new source code base and review only the positive outcomes to reduce manpower in code auditing.

2. Overview

The objective of our method is to utilize buffer overflow vulnerabilities that have already been found to assist in auditing vulnerabilities in new software efficiently. The overview of our method is depicted in Figure 1. In this paper, the term "buffer" means the variable that represents a memory region or just a memory region such as `swapbuff` at line (15) in Algorithm 1, which is also the subject investigated in Section 3. If not otherwise specified, vulnerability means buffer overflow vulnerability.

The purpose of the method illustrated in Figure 1 is to generate suspicious vulnerable functions (SVFs) and related suspicious areas (SAs). SA, in the form of a line number and file name, means the specific area in which buffer overflow may take place, such as line (15) in Algorithm 1. To build a classifier with good performance to distinguish vulnerable buffers from others, the choice of code metrics as the features affects the output significantly. To build a classifier for source code, we extended the static code attributes in [3] by summing up six observable crucial attributes and introducing the sanitization attribute. The details of the static code attribute extraction are described in Section 3.

A robust parser [5, 6] is employed to parse source code to Abstract Syntax Tree (AST), which is directly or indirectly used to generate multiple representations. The robust parser allows analysis of code even when a build environment is not configured, which saves a lot of work of compiling source code to machine code. The parser takes advantage of the ANTLR parser generator and C/C++ grammar definition to extract an AST from individual source files. The AST is used to generate multiple representations consisting of the Control Flow Graph (CFG), Program Dependence Graph (PDG), interprocedural sanitization graph (IPSG), and declaration-spread-sink graph (DSSG). Gathering all the representations, we form an extended code property graph (ECPG). The detailed definitions of IPSG, DSSG, and ECPG are described in Section 4. From ECPG, the code attributes are extracted and then embedded into a vector space. The vectors obtained from labeled code are utilized to train classifiers using some frequently used classifier algorithms. Finally, the vectors obtained from the test code are fed to the classifiers to get SVFs and SAs.

3. Static Code Attributes and Mapping

Buffer overflow can be classified into 22 taxonomies [3]. Based on these 22 taxonomies and investigation of many real-world buffer overflow functions from the CVE database, we summarized seven kinds of attributes to represent buffer overflow. All the attributes together are mapped to a multidimensional vector to be fed to a classification algorithm.

3.1. Sink Type. Three sink types are discussed in this subsection, namely, pointer dereference, array write, and dangerous function, as listed in Table 1. If a statement falls into one of the three sink types without proper buffer bound check, a buffer overflow may occur. In C/C++ language, an array element can be accessed by either pointer dereference or array subscript, but the operations of these two types are different, so we consider them separately. A pointer dereference appears in

<table>
<thead>
<tr>
<th>Sink type</th>
<th>Example</th>
<th>Mapping value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer dereference</td>
<td><code>*p++ = 1</code></td>
<td>1</td>
</tr>
<tr>
<td>Array write</td>
<td><code>p[i] = 1</code></td>
<td>2</td>
</tr>
<tr>
<td>Dangerous function</td>
<td><code>memcpy(dst, src, n)</code></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><code>strcpy(dst, src, n)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>strcat(dst, src)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>strncat(dst, src, n)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>memmove(dst, src, n)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>gets(str)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fgets(str, n, fp)</code></td>
<td></td>
</tr>
</tbody>
</table>
the left part or right part of an assignment statement, like in lines (24) and (25) in Algorithm 1, which are classified as pointer dereference sinks. Dangerous functions are the standard-library or user-defined calls that copy or pad the buffer. Formatted string output also could lead to potential buffer overflow; however, it will not be discussed here. Besides all the standard-library function calls, some user-defined functions have the same effect. For example, CVE-2016-9537 results from a user-defined function ("TIFFmemcpy"), shown in lines (15), (16), and (17) in Algorithm 1, which has the same effect as "memcpy" in the standard C library. Thus, user-defined functions that have a similar function name and exactly the same number of parameters are classified into the dangerous function case.

Algorithm 1: Sample code from CVE-2016-9537.

3.2. Memory Location. The memory location attribute represents the location where the sink buffer resides. Five kinds of memory areas that accommodate sink buffer are stack, heap, data segment, BSS segment, and shared memory. These are different from the programming perspective. A stack buffer describes arrays that are defined locally and nonstatically; a heap buffer describes the dynamic allocated memory used to satisfy a large portion of the memory application; a data segment describes static variables or global variables; a BSS segment describes uninitialized global or static variables; and shared memory describes a method of interprocess communication (IPC). In this paper, only the stack, heap, and data segment are considered. We map the memory location to a three-dimensional vector as (stack, heap, and data segment); if the memory location appears, the related entry is set to 1; otherwise it is 0. For pointer swapbuff in Algorithm 1, the vector representation is (1, 0, 0) because it is declared locally.

3.3. Container. Container describes the structure in which the sink buffers are wrapped. Generally, the more complex the container structure is, the more vulnerability-prone the buffer will be. According to the investigation of Zitser et al. [9], 7% of vulnerable buffers are contained in the Union structure, and according to our research on recent CVE buffer overflow vulnerabilities, nearly 30% of vulnerabilities have containers, so we consider the container of the buffer as another static code attribute. The container attribute is shown in Table 2. The Union and Struct structures are similar in programming; therefore, we map them to the same value. The Others row represents the more complex structures such as the double Union and Struct structures.

3.4. Index/Address/Length Type. Index type records various accumulated operations of array index. The index type attribute comes from the assumption that the more complex
array index operations are, the more error-prone buffer access will be. Array index operations can be divided into six categories, namely, constant, addition, multiplication, nonlinear, function call, and array access. Each category will be mapped to one dimension to construct a six-dimensional vector.

Addition contains addition and subtraction operations. Multiplication contains multiplication, division, bitwise left shift, and bitwise right shift operations. Nonlinear contains modulo and other functions such as pow() and sqrt() operations. Function call describes the situation that a function (except the function call contained in nonlinear) returns a value involved in a buffer index. Array access indicates whether an array content read operation is involved in a buffer index. Index type is described in Table 3. The six operation types contribute, to different extents, to buffer overflow. Despite the difficulty of exploiting vulnerabilities, we find some buffer overflows that are caused by buffer access with a constant index and we denote this type as the constant type. Besides the constant type, each of the others has two different forms, such as $p[i-8]$ and $(p-8)[i]$.

Figure 2 is the schematic depiction of the accumulating index operation. In the case of $p[i]$, we accumulate the operation of $i$ along with the data flow. Finally, the index type of $i$ is converted to a 6-dimensional vector ($0, 1, 0, 1, 1, 1$).

Address and length type attributes, which are described in Tables 4 and 5, respectively, are akin to the index type except for the difference in their sink types. We map index/address/length type into a six-dimensional vector and the corresponding value is increased by one when the related operation is encountered. For pointer dereference sink buffers src and dst at line (24) and line (25) in Algorithm 1, the values of address type are $(0, 1, 0, 0, 0, 0)$ and $(0, 3, 0, 0, 0, 0)$, respectively. For dangerous function sink buffers at lines (15), (16), and (17), the values of length type are the same, namely, $(0, 1, 2, 0, 0, 0)$. Besides, index/address/length type takes buffer aliasing into consideration and the operations of the alias buffer should be accumulated, too.

### 3.5. Sanitization
Sanitization is the bound check operation of buffers. Even though the existence and correctness of the sanitization cannot be precisely obtained from static analysis, the pattern of bound check can be summarized to estimate them. If statements fall into several modes, we argue that the programmer might have considered sanitization and the more the times the modes appear, the higher the possibility that the programmer has added the sanitization. Sanitizations are classified into three types:

1. **Direct sanitization**: provided variable $b$ is a sink buffer or sink buffer index; $c$ is a condition statement that has a control flow to $b$ and $b$ is a subexpression of $c$; we will increase the sanitization value by one. The sink buffer swapbuff in Algorithm 2, line (1), and the bound checks on array indexes fall into this scope

2. **Indirect sanitization**: given a data flow from variable $a$ to variable $b$ ($b$ is a sink buffer index or involved in a buffer index expression), if $a$ is involved in a condition statement $c$, we will increase the value of indirect sanitization by one. Figure 3(a) is a code example,
where the expression $i + j$ is the buffer index of $buf$ and $i$ is involved in a condition statement.

(3) **Interprocedural sanitization**: we found that many buffer overflow vulnerabilities are patched by sink function argument sanitization. Therefore, if there is a data flow between arguments and sink buffer index and also the arguments are involved in a condition statement in the superior function, we will increase sanitization value by one. Figure 3(b) is a code example where $param1$ is a buffer index and parameter of $foo$, and $arg1$ is involved in a condition statement in the superior function in Algorithm 1.

Additionally, we add some exceptions, where a condition statement falls into the description of three kinds of sanitizations but does not count as a sanitization. For example, the condition statement at line (7) in Algorithm 1 cannot be regarded as a sanitization, because comparing $src$ or $dst$ against $NULL$ is not for bound checking of buffer $src$ and $dst$. We map the sanitization attribute into a three-dimensional vector. When a type appears, the corresponding value of the sanitization attribute will be increased by one.

3.6. **Loop/Condition/Call Depth.** Loop/condition/call depth reflects the complexity of the program which leads to program faults. Loop/condition depth describes the maximum hierarchy of the loop/condition statement that wraps the sink statement. Call depth describes the maximum number of function calls from $main$ to the current function, which can be obtained by declaration-spread-sink graph, described in Section 4 in detail. We map these three attributes into a three-dimensional vector and the corresponding value is increased by one when a type appears, the corresponding value of the sanitization attribute will be increased by one.

3.7. **Taint.** Taint indicates whether there exists a data flow from source input to sink statement. Taint can be classified into five types, namely, command line, environment variable, file input, network transmission, and argument inflow. The former four types are usually characterized by standard-library function calls such as `scanf`, `getwd`, `fscanf`, and `recvfrom`. Argument inflow describes whether there is a data flow between arguments and the sink statements. If a sink buffer falls into any of the taint types, it is attacker-controlled and we assign 1 to the attribute; otherwise, 0 is assigned.

Based on these seven kinds of static code attributes, we can construct an 18-dimensional vector, which is then used to train the classifiers.

### 4. Extended Code Property Graph

We extract static code attributes based on an extended code property graph. The following content introduces the basic theory of property graphs and extended property graphs and then explains the fundamental traversals to get specific static code attributes.

#### 4.1. Property Graph Theory

A property graph [10] can be formally defined as $G = (V, E, \lambda, \mu)$, where $V$ is a set of nodes, $E \subseteq \{V \times V\}$ is a multiset of directed edges, $\lambda : E \rightarrow \Sigma$ is a label function, $\Sigma$ is a set of edge labels, and $\mu : (V \cup E) \times K \rightarrow S$ is a key-value mapping function that maps property keys of nodes and edges to their values, where $K$ is a key set and $S$ is the set of property values. What makes a property graph more expressive is that it contains multiple key-value maps on every node and edge.

Graph traversal is a procedure to search for a proper node or edge with certain preconditions. The fundamental traversals are to search for the value of a node or edge given the key and the in and out edges of nodes, which are described in the equations below. Through traversal composition, complex traversal can be performed to explore an arbitrary node or edge.

$$
\epsilon : P (V \cup E) \times K \rightarrow P (S),
$$

$$
e_{in} : P (V) \rightarrow P (E),
$$

$$
e_{out} : P (V) \rightarrow P (E),
$$
4 Mathematical Problems in Engineering

```c
void main(int argc, char **argv){
    int b = atoi(argv[1]);
    int buf[10];
    if(b > 0){
        buf[5*b] = 1;
    }
}
```

(a)

![CPG Diagram](image)

(b)

**Figure 4:** Schematic representation of CPG.

\[ V_{in}: P(E) \rightarrow P(V), \]
\[ V_{out}: P(E) \rightarrow P(V). \]  \hspace{1cm} (1)

### 4.2. Extended Code Property Graph

CPG (code property graph) [4] is a joint representation of AST (Abstract Syntax Tree), CFG (Control Flow Graph), and PDG (Program Dependence Graph). Figure 4 is a schematic representation of CPG, where Figure 4(a) is the related exemplary code. All nodes are linked by different edges and we can search any edges and nodes through the composition of traversals on edges and nodes. There are three kinds of edges in Figure 4, namely, the AST edge, CFG edge, and PDG edge. Every operator and operand from the source code can be accessed through traversal by AST edge, so the sink type and container attribute can be easily obtained. Memory location and taint require AST and PDG. Condition/loop depth requires both AST and CFG. Direct and indirect sanitization and index/address/length type need all three types of graphs.

However, code property graphs cannot handle interprocedural sanitization and call depth, because they need interprocedural data flow and control flow. To solve this problem, we append the interprocedural sanitization graph (IPSG) and declaration-spread-sink graph (DSSG) to CPG to form the extended code property graph (ECPG).

IPSG is a property graph: \( G = (V_A, E_{IP}, \lambda_{IP}, \mu_{IP}) \). In the formula, \( V_A \) is the AST node set; \( E_{IP} \) links the predicate node to the sink statement of the invocation function, where the predicate actually sanitizes the invocation and the parameters of the invocation function data control the sink statement; \( \lambda_{IP} \) is the labeling function, \( \lambda_{IP} : E_{IP} \rightarrow \Sigma_{IP} \), where \( \Sigma_{IP} = \{ IC, ID \} \) and IC corresponds to interprocedural control dependency. We assign a property symbol to indicate ID and a property condition to indicate IC. Figure 6(a) is the representation of IPSG of code from Algorithm 3 and Figure 5 is the CPG of the code from Algorithm 3, which delete the detailed AST nodes and edges. In order to generate IPSG edge, the predicate if \( (size < 100) \) must control and sanitize \( foo \). Also, the arguments of \( memcpy src \) and \( count \) must be data-dependent on the parameters of \( foo \), src, and \( n \).

DSSG is a property graph: \( G = (V_A, E_{DS}, \lambda_{DS}, \mu_{DS}) \). In the formula, \( V_A \) is the AST node set; \( E_{DS} \) links the functions \( f_1 \) and \( f_2 \), if \( f_1 \) invokes \( f_2 \); \( E_{DS} \) also links sink function node to sink statement; \( \lambda_{DS} \) is the labeling function, \( \lambda_{DS} : E_{DS} \rightarrow \Sigma_{DS} \), where \( \Sigma_{DS} = \{ DS \} \). We also assign a property symbol to indicate the symbols transmitted from function node to function node and from function node to sink statement. Figure 6(b) is the representation of DSSG of code from Algorithm 3. DSSG not only helps collect call depth...
int main(int argc, char *argv)
int i = atoi(argv[1])
char p = argv[2]
void foo(char *src, int n)
char dst[200]
int count = n + 100
memcpy(dst, src, count)

Figure 5: Simplified CPG of code for Algorithm 3.

int woo(char *src, int size)
if (size < 100)
exit foo(src, size)

Figure 6: Representations of (a) IPSG and (b) DSSG of code from Algorithm 3.

IN\textsubscript{\textit{k,s}}(X) finds the nodes that are reachable through ingoing edge.

\textbf{OUT}\textsubscript{\textit{k,s}}(X)
= \bigcup_{v \in X} \{u : (v, u) \in E, \lambda ((v, u)) = l, u ((v, u), k) = s\},

(3)

IN\textsubscript{\textit{k,s}}(X)
= \bigcup_{u \in X} \{v : (v, u) \in E, \lambda ((v, u)) = l, u ((v, u), k) = s\}.

Furthermore, we also introduce TNodes(X) to search the set of AST nodes reachable from the AST root, which can be formulated as follows. In the formula, \( v \) is a node from \( X \), and Out\(_A\) is the representation of the outgoing edge of \( v \) used to find its child node \( v_{c} \). TNodes(X) indicates that all its AST child nodes can be traversed given any node in \( X \).

\textbf{TNodes(X)}
= \bigcup_{v \in X} \bigcup_{v_{c} \in \text{Out}(v)} TNodes(\{v_{c}\}).

(4)

Upon the three traversals, all the nodes and edges can be explored in ECPG, so the compositional traversals can be designed to extract static code attributes.
5. Experiment

The experiment was conducted on a PC with Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40 GHz CPU and 16.0 GB memory, using Ubuntu 14.04.4 LTS. We use scikit-learn to implement the machine learning algorithms [11], which are built on python libraries such as Numpy, SciPy, and matplotlib.

5.1. Evaluation Metrics. In the field of machine learning, the confusion matrix [12] is a general applied metric to assist in understanding the performance of a classifier, as described in Table 6. The matrix describes the mixtures between actual classes and predicted classes, namely, true positive (TP), false negative (FN), false positive (FP), and true negative (TN). TP represents the case when a vulnerable function is classified as positive. FN represents the case when a vulnerable function is classified as negative. FP represents the case when a nonvulnerable function is classified as positive. TN represents the case when a nonvulnerable function is classified as negative.

In a normal program, the number of vulnerable functions is much less than that for nonvulnerable functions, which would make the training data and test data unbalanced, so we additionally leverage four other assessment metrics: recall, true negative rate (TNR), precision, and \( F_1 \) score. Recall is also known as true positive rate (TPR) and precision is also known as positive predictive value (PPV). The formula is described as follows. Recall describes the proportion of TPs to all buffer overflows; TNR represents the proportion of TNs to all nonvulnerable functions. Precision represents the proportion of TPs to all predicted positives. \( F_1 \) is a measure of test’s accuracy which considers both the precision and recall.

\[
\text{recall} = \frac{TP}{TP + FN}, \\
\text{TNR} = \frac{TN}{FP + TN}, \\
\text{precision} = \frac{TP}{TP + FP}, \\
F_1 = 2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}}.
\]

5.2. Experiment and Comparison. This section contains evaluations of the five classifiers. As shown in Table 7, we investigated 58 vulnerable functions manually from the official CVE database together with 174 functions that are not vulnerable in recent years. For all the functions, we select a buffer to represent each one to extract attributes. Columns Vul-Num and Not-Vul-Num display vulnerable and nonvulnerable function numbers. Vulnerable functions are labeled 1, while nonvulnerable functions are labeled 0. The samples originate from eight open-source programs of various versions, which are ffmpeg, HDF5, libtff, mupdf, openssl, qemu, zziplib, and blueZ.

Different classifier algorithms are employed to train classifiers because which algorithm is better is not predictable. To evaluate our static analysis, five well-known supervised machine learning algorithms, K-Nearest Neighbors (KNN), Decision Tree (DT), Naive Bayes (NB), AdaBoost, and Support Vector Machines (SVM), are used. 10-Fold cross-validations are performed on the 232 labeled pieces of data employing the five classifier algorithms. The parameter \( k \) used in KNN is 7. The specific DT algorithm we use is C4.5. The number of weak classifiers used in AdaBoost is 20. In SVM, we use the RBF kernel; the parameter \( C \) is 10 and \( \gamma \) is 0.01.

5.2.1. Evaluation on Test Suite. The performances of the five classifier algorithms are listed in Table 8. The average recall is 83.5%, which means that 48 out of 58 vulnerabilities are detected through our method. The average TNR is 87.3%, which means that nearly 152 out of 174 nonvulnerable functions are classified correctly. The average precision is 68.9% and the average \( F_1 \) is 75.2%, which are not very high because the number of nonvulnerable functions is three times as many as vulnerable functions. In the field of vulnerability detection, finding the largest possible number of vulnerabilities is more important. Therefore, we deem NB the best classifier, since it outperforms the other algorithms to detect 56 vulnerabilities with the highest recall of 96.6%.

5.2.2. Comparison to BOMiner. In [13], a tool, BOMiner, is implemented to predict buffer overflows using machine learning. However, it focuses too much on pointer reference sinks and overfits the result of the classifiers. We try to compare with BOMiner on features selection using the five classifier algorithms. Table 9 shows the performance of BOMiner on our test suite. The maximum number of vulnerabilities correctly classified by BOMiner is 38, which is 4 less than that by our worst classifier KNN and 18 less than that by our best classifier, NB. The highest recall of BOMiner is 65.5%, which is also less than that for all our classifiers. Figure 7(a) shows the comparison of average confusion matrix. On average, our method classifies 14.8 more TPs and 10.4 more TNs than BOMiner. Figure 7(b) shows the comparison of recall, TNR, precision, and \( F_1 \). The average recall is increased by 25.6%, average TNR is increased by 6%, average precision is increased by 17.5%, and the average \( F_1 \) is increased by 21.1%. The main reason why our method outperforms BOMiner is that, in [13], the array write sink is not considered in detail. When it encounters an array write sink, BOMiner has too little information to classify it correctly.
Table 7: CVE list for attribute extraction.

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE-ID</th>
<th>Vul-Num</th>
<th>Not-Vul-Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF5</td>
<td>CVE-2016-4333, CVE-2016-4330</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>openssl</td>
<td>CVE-2016-2182, CVE-2015-0235, CVE-2014-3512</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>qemu</td>
<td>CVE-2016-7170, CVE-2016-5238, CVE-2016-4439, CVE-2013-4151, CVE-2013-4150</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>BlueZ</td>
<td>CVE-2016-9917, CVE-2016-9804, CVE-2016-9803, CVE-2016-9800</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 8: Performances of our five classifier algorithms.

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>Recall (%)</th>
<th>TNR (%)</th>
<th>Precision (%)</th>
<th>$F_1$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN</td>
<td>42</td>
<td>16</td>
<td>20</td>
<td>154</td>
<td>72.4</td>
<td>88.5</td>
<td>67.7</td>
<td>70</td>
</tr>
<tr>
<td>DT</td>
<td>44</td>
<td>14</td>
<td>31</td>
<td>143</td>
<td>75.9</td>
<td>82.2</td>
<td>58.7</td>
<td>66.2</td>
</tr>
<tr>
<td>NB</td>
<td>56</td>
<td>2</td>
<td>27</td>
<td>147</td>
<td>96.6</td>
<td>84.5</td>
<td>67.5</td>
<td>79.5</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>46</td>
<td>12</td>
<td>15</td>
<td>159</td>
<td>79.3</td>
<td>91.4</td>
<td>75.4</td>
<td>77.3</td>
</tr>
<tr>
<td>SVM</td>
<td>54</td>
<td>4</td>
<td>18</td>
<td>156</td>
<td>93.1</td>
<td>89.7</td>
<td>75</td>
<td>83.1</td>
</tr>
</tbody>
</table>

Table 9: Performance of BOMiner for our test suite.

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>Recall (%)</th>
<th>TNR (%)</th>
<th>Precision (%)</th>
<th>$F_1$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN</td>
<td>29</td>
<td>29</td>
<td>21</td>
<td>153</td>
<td>50</td>
<td>87.9</td>
<td>58</td>
<td>53.7</td>
</tr>
<tr>
<td>DT</td>
<td>31</td>
<td>27</td>
<td>35</td>
<td>139</td>
<td>53.4</td>
<td>79.9</td>
<td>47</td>
<td>50</td>
</tr>
<tr>
<td>NB</td>
<td>38</td>
<td>20</td>
<td>46</td>
<td>128</td>
<td>65.5</td>
<td>73.6</td>
<td>45.2</td>
<td>53.5</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>33</td>
<td>25</td>
<td>28</td>
<td>146</td>
<td>56.9</td>
<td>83.9</td>
<td>54.1</td>
<td>55.5</td>
</tr>
<tr>
<td>SVM</td>
<td>37</td>
<td>21</td>
<td>33</td>
<td>141</td>
<td>63.8</td>
<td>81</td>
<td>52.9</td>
<td>57.8</td>
</tr>
</tbody>
</table>

5.2.3. Comparison to Joern. Yamaguchi et al. [4] developed a platform, Joern, which can detect buffer overflows with very few FPs, based on code property graph. As reported in [4], six buffer overflows in drivers directory of linux kernel3.11 source code are detected through specific pattern. However, as we investigated from CVE database, there are 18 buffer overflows and 179 nonvulnerable sink functions in drivers directory of linux kernel3.11. We apply the proposed method to the code base and the result is compared to Joern in Table 10. Using our proposed method, the classifiers can detect at least 12 buffer
overflows and the least recall of the five classifiers is 66.7%, which are all better than Joern. The highest TNR is 93.9%, 6% lower than Joern's 98.9%. Figure 8(a) shows the comparison of confusion matrix between our method and Joern. On average, our method detects 8.2 more TPs than Joern but outputs 11.8 more FPs. Figure 8(b) shows the comparison of average recall, TNR, precision, and $F_1$. Our average recall is 78.9%, 45.6% higher than Joern and only 6.6% lower at TNR. Our precision is 24.1% lower than Joern, while our $F_1$ is 15.5% higher than Joern. Two reasons can explain the low TP of Joern: (1) Joern only handles buffer overflows caused by dangerous function memcpy; (2) Joern employs two sanitization rules, dynamic allocation of the destination and relational expressions [4]; the rules fall into the region of our direct sanitization; the two rules would reduce false positives, while contributing to low TP.

5.3. Comparison to Flawfinder on Poppler 0.10.6. Poppler is a widely used open-source PDF library from which many buffer overflow vulnerabilities are detected. In this subsection, Poppler 0.10.6 is experimented on to evaluate our classifiers. As far as we are concerned, Poppler 0.10.6 has ten proven CVEs: CVE-2015-8868, CVE-2013-1788, CVE-2010-3704, CVE-2009-3938, CVE-2009-3608, CVE-2009-3607, CVE-2009-3606, CVE-2009-3604, and CVE-2009-3603. We use these CVEs and our trained classifier to describe how our method assists in auditing buffer overflow vulnerabilities.

Poppler 0.10.6 source code is inputted and the output is described in Table 11. The TP, FN, FP, TN, recall, TNR, precision, and $F_1$ columns describe the performances of the five classifiers. We evaluate to what extent our method helps in code auditing based on the number of functions needed to be audited. The SVFs column contains the number of suspect vulnerable functions that need to be audited and the value of SVFs is the sum of TP and FP. Sink functions displays the number of functions that have buffers that satisfy one of the three sink types. All functions describes the total number of functions from the Poppler 0.10.6 source code. The average TP is 8.8, which means that we can find nearly 9 of 11 vulnerabilities. The average recall is 80% and the average TNR is 94%. Because there are far more nonvulnerable functions than vulnerable functions, the precisions and $F_1$ are low where the average $F_1$ is 28.9% and the average precision is 17.7%. Taking SVM as an example, code auditors only need to review 45 functions to find 10 of 11 vulnerabilities using our method. However, without our method, all 685 sink functions should be reviewed, which is a very heavy workload.
Mathematical Problems in Engineering

Our method

Joern

(a)

Figure 8: Comparison between our method and Joern. (a) Comparison of average confusion matrix. (b) Comparison of average recall, TNR, precision, and $F_1$.

Table 1: Performances of our five classifiers on Poppler 0.10.6.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>Recall (%)</th>
<th>TNR (%)</th>
<th>Precision (%)</th>
<th>$F_1$ (%)</th>
<th>SVFs</th>
<th>Sink functions</th>
<th>All functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN</td>
<td>7</td>
<td>4</td>
<td>37</td>
<td>648</td>
<td>63.6</td>
<td>94.6</td>
<td>15.9</td>
<td>25.4</td>
<td>44</td>
<td>685</td>
<td>4876</td>
</tr>
<tr>
<td>DT</td>
<td>8</td>
<td>3</td>
<td>44</td>
<td>641</td>
<td>72.7</td>
<td>93.6</td>
<td>15.4</td>
<td>25.4</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NB</td>
<td>10</td>
<td>1</td>
<td>52</td>
<td>633</td>
<td>90.9</td>
<td>92.4</td>
<td>16.1</td>
<td>27.4</td>
<td>62</td>
<td></td>
<td>685</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>9</td>
<td>2</td>
<td>39</td>
<td>646</td>
<td>81.8</td>
<td>94.3</td>
<td>18.8</td>
<td>30.6</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVM</td>
<td>10</td>
<td>1</td>
<td>35</td>
<td>650</td>
<td>90.9</td>
<td>94.9</td>
<td>22.2</td>
<td>35.7</td>
<td>45</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also run Flawfinder 1.31 [14] on Poppler 0.10.6. Flawfinder is a static analyzer that scans for various kinds of vulnerabilities from source code. We identify all buffer overflow vulnerabilities from the output of Flawfinder. Flawfinder detected 8 of 11 vulnerabilities, which is slightly lower than the average performance of our method; however, it also generated 500 false positives, which is 12 times as many as ours. Thus, our method outperforms Flawfinder significantly in reducing the number of false positives of buffer overflow vulnerabilities, which definitely saves a lot of manual code auditing work.

6. Related Work

Static analysis tools fall into two categories: lightweight rough approaches and more thorough ones. Lightweight tools like Flawfinder [14] and Rats [15] are based on lexical analysis. Both translate source files to tokens and match them with certain vulnerable constructs in a library. Splint [16] can find abstract violations, unannounced modifications of global variables, and so forth, with manual annotations. For more thorough tools, Archer [17] symbolically computes buffer usage and employs a constraint solver to evaluate illegal memory accesses. Model checker [18] converts a buffer violation to a path to an error statement and then, using a constraint solver, verifies whether the path is feasible. Coventry [19], Fortify [20], and CodeSonar [21] are commercial tools that require manual configuration work.

Dynamic test-case generation mainly involves two techniques, namely, fuzzing and symbolic execution. Fuzzing tools generate test cases to trigger program faults by mutating input bytes randomly. Recently, many methods [22–24] have been proposed to augment the fuzzing effect and [22] proved that a good seed test case contributed greatly to the fuzzing effect. Symbolic execution was first put forward by Clarke [25] and underwent great development because of the improvement of the constraint solver. Many tools were developed by various academic and research labs such as DART [26], CUTE [27], CREST [28], KLEE [29], SAGE [30], and S2E [31]. All of these tools fork a state once a branch instruction is encountered, which could lead to path explosion.

There are also many spot-on methods or tools that target buffer overflow vulnerability exclusively. Rawat and Mounier [32] implement an evolutionary computing approach to find buffer overflow, but it can only detect superficial faults. Another work from Rawat and Mounier [33] hunts buffer overflow in binary executables through a pattern obtained from “strcpy.” Li et al. [34] utilized symbolic analysis representation to filter out irrelevant dependencies to scale to a large-scale code base for buffer overflow. Haller et al. [35] provided a guided fuzzing tool aimed only at array boundary violations.
Recently, machine learning algorithms have been applied in the vulnerability detection field. Yamaguchi et al. [6] proposed a vulnerability extrapolation method to assist code auditors, using the similarity in AST structure of similar functions. The effectiveness of this method depends on the existence of a similar function of a known vulnerability. Yamaguchi et al. also leveraged anomaly detection to identify missing checks of buffers [36] and applied a clustering algorithm to taint-style vulnerabilities [37]. Padmanabhi and Tan’s work [38] is the closest to our work, but it did not provide the concept of complexity, which is very important in modern software.

7. Conclusion

In this paper, a method that assists in auditing buffer overflow vulnerabilities using machine learning is proposed. We define seven kinds of static code attributes according to the 22 taxonomies of buffer overflow vulnerabilities and also design the extended code property graph to extract these attributes. Then the digitalized attributes are used to train five classifiers. In our experiment, the classifiers reached an average recall of 83.5%, average true negative rate of 85.9%, a best recall of 96.6%, and a best true negative rate of 91.4%. Due to the imbalance of the training samples, the average precision of 96.6%, and a best true negative rate of 91.4%. Due to the imbalance of the training samples, the average precision of 96.6%, and a best true negative rate of 91.4%.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References


