2015 IEEE 39th Annual International Computers, Software & Applications Conference

Buffer Overflow Vulnerability Prediction from x86 executables using Static Analysis and Machine Learning

Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University
50 Nanyang Avenue, Singapore 639798
padm0010@e.ntu.edu.sg, ibktan@ntu.edu.sg

***Abstract*— Mining static code attributes for predicting software vulnerabilities has received some attention recently. There are a number of approaches for detecting vulnerabilities from source code, but commercial off the shelf components are, in general, distributed in binary form. Before using such third-party components it is imperative to check for presence of vulnerabilities. We investigate the use of static analysis and machine learning for predicting buffer overflow vulnerabilities from binaries in this study. To mitigate buffer overflows, developers typically perform size checks and input validation. We propose static code attributes characterizing buffer usage and defense mechanisms implemented in the code for preventing buffer overflows. The proposed approach starts by identifying potential vulnerable statement constructs during binary program analysis and extracts static code attributes for each of them as per proposed characterization scheme to capture buffer usage patterns and defensive mechanisms employed in the code. Data mining methods are then used on these collected code attributes for predicting buffer overflows. Our experimental evaluation on standard buffer overflow benchmark binaries shows that the proposed static code attributes are effective in predicting buffer overflow vulnerabilities.**

Keywords- binary static analysis; static code attributes; disassembly; vulnerability prediction; buffer overflow; control and data dependency; buffer usage pattern

I. INTRODUCTION

Buffer overflow (BO) happens during program execution when more data is written into the buffer than its pre-allocated size. BO vulnerabilities can be exploited to carry out privilege escalation, launch denial-of-service attacks, or execute arbitrary code supplied by attacker resulting in dangerous security breaches. The grave nature of the vulnerability led to many tools for mitigating BO vulnerabilities. In spite of presence of such tools BO vulnerabilities continue to surface abounds in wide-range of applications from file decompression utilities to image manipulation and multimedia viewing software [1, 2]. Vulnerability trends too report that BO vulnerabilities continue to be highly prevalent [3]. Programs written in C/C++ are mostly prone to buffer overflows. C/C++ languages offer great degree of flexibility and high run-time performance. Hence many commercial applications are written in C/C++, but it leaves the onus of incorporating security in the hands of the developers. This is because C/C++ does not have any built-in security mechanisms for buffer bounds checking.

Existing BO solutions in literature can be broadly classified as detection tools before software deployment (e.g., test generation, static analysis) [4-6], monitoring tools during run-time or network packet delivery [7-9], patching or vulnerability signature generation tools after known attacks [10]. Most of the afore-mentioned detection tools work with source code, whereas commercial-off-the-shelf (COTS) applications are typically distributed in the form of binaries. Before deploying COTS applications or incorporating third party component binaries into a system it is important to check for presence of vulnerabilities in the binaries. Complier optimization too can sometimes render changes to executables from what is intended in source code [11]. Analyzing executables is highly desirable in such scenarios as it can help uncover potential vulnerabilities. Very few tools are available for analyzing binaries as it is deemed challenging since the source code level semantics information is not available in the executable. Hence much of vulnerability detection research is limited to source code analysis.

It is extremely difficult to use static analysis alone to infer BO vulnerabilities precisely from binaries. Combining prediction method and static analysis could provide important avenue for addressing this problem. Hence, in this paper, we study the use of static analysis and machine learning for BO vulnerability prediction in executables. To the best of our knowledge, no approach has been proposed to predict BO from executables using machine learning.

Buffer overflows can be prevented by carrying out bounds checks before filling the buffers. BO exploits can be thwarted by validating the input thoroughly so that it meets the requirements before propagating it in the program. Examining the characteristics of buffer usage patterns and BO defensive measures implemented in the code could be useful in predicting vulnerabilities. These observations motivated us to develop a binary vulnerability analysis framework by combining static analysis and machine learning. Our attributes for BO characterization use only control and data dependencies and buffer usage information obtained from static analysis. If effective, our approach can be considered as a practical alternative to binary BO detection tools. To investigate these claims we implemented

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

a tool called *"VulMiner"* for extracting the proposed static code attributes. BO prediction models were then built using the collected attribute data by making use of classifier algorithms in WEKA [12]. In our evaluation on 6 well-known BO benchmark programs our best classifier achieved a recall of 75% with a precision of 84% and accuracy over 94% suggesting that the binary BO prediction models are practical and viable.

We make the following contributions in this paper:

- ➢ Characterizing BO defenses implemented in the code from binary disassembly.
- ➢ Enable the prediction of BO vulnerability from x86 executables using static analysis possible.

The paper is organized as follows. Section II describes binary analysis framework. We present our BO characterization scheme for predicting vulnerable statements in Section III. Section IV presents details of our experimental set- up. The evaluation results are discussed in Section V. Section VI presents the works that are closely related to ours and we provide our conclusions in Section VII.

II. BINARY ANALYSIS FRAMEWORK

We used binary static analysis tool built on top of Rose open-source compiler binary analysis framework (developed by Lawrence Livermore National Laboratories) [13] to gather code attribute information for predicting vulnerable BO statements in binary disassembly. Since the precision and assumptions of analysis may affect vulnerability prediction, in this section, we briefly describe the binary analysis tool and outline how the binary analysis challenges were addressed.

Code and data are sometimes interspersed in binary and distinguishing code from data is tough due to presence of variable length instructions. Variables in high-level language are accessed using their addresses (e.g., global variables) or stack frame offsets (e.g., local variables) in binaries. We used IDA Pro commercial disassembler [14] to identify all statically known addresses and stack offsets from the disassembly since Rose disassembler did not have such features. Since there might be holes in these addresses we treat a variable as starting from the statically known start address (or offset) to next known address (or offset).

In order to perform program analysis, the semantics of each instruction has to be accurately represented to model the instruction behavior. Rose compiler infrastructure framework provides x86 assembly instruction semantics that caters to this requirement. This enables us to capture its effect on machine state which helps in propagating constants and identifying those indirect branches or calls that can be determined statically through constant propagation. It is important to note that we do not evaluate path feasibility criteria or keep track of buffer usage in the program by representing them in form of constraints as in some source code based BO detection solutions. Constraint solver is only used in the analysis to identify memory variables being accessed by given variable address to determine data dependencies.

Assembly instructions use different addressing modes to access memory based on variable type in high level-language construct. The address is specified in the form of register expression *(base + index * scale +displacement)*. Since we use symbolic execution in our static analysis, if the values contained by base and displacement are statically known, the variable being accessed can be identified. To identify memory accesses to dynamically allocated variables, we keep track of the symbolic values returned by modeled library function call prototypes of dynamic memory allocation functions (further details on library function modeling are provided later in the discussion). Fig. 1 shows an overview of binary static analysis frame work. We briefly outline the framework in the following discussion.

A. Disassembly and Variable Information

The first stage of binary analysis involves function boundary identification and variable information gathering. Executables do not have information about function boundaries. We therefore used IDA Pro to identify functions along with variable information. These are given as inputs to the binary static program analysis tool, *"BinAnalysis"*, built on top of Rose binary analysis framework. Rose interprets these disassembled functions and builds AST for each of the functions and subsequently constructs their Control Flow Graph *(CFG).*

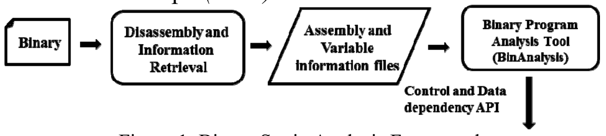
Binary
Disassembly and Information Retrieval
Assembly and Variable information files
Binary Program Analysis Tool (BinAnalysis)
Control and Data dependency API

Figure 1. Binary Static Analysis Framework

B. Control Dependency

Second stage of binary static analysis starts by calculating intra-procedural control dependency of the function *CFG*. Rose provides API for building post dominator relationships from *CFG*. This is used to find out immediate post dominators. From this control dependency successors were calculated using standard approach. The call graph generated by IDA Pro may be incomplete due to indirect calls (for example, calls made using function pointers). We, therefore augment inter control-dependency as and when a call to known function is made.

C. Data Dependency Computation using Function Summaries

In the third stage of our approach container level inter-procedural data dependency analysis is performed using *CFGs* and variable information retrieved from the first stage of analysis. By container level we mean that for container data accesses (e.g., arrays) our tool computes data dependency in terms of the whole container basis. This is in line with precision commonly provided by source-code data analysis tools. For *struct*, when the *struct* information is available from IDA Pro, we apply the *struct* offsets onto the variable information to identify individual element boundaries of *struct* variable.

Rose provides policy for symbolic semantics which can emulate execution of a single basic block of x86 instructions. Each basic block is associated with a symbolic state

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

representing the state of registers, memory and flags (except instruction pointer, which is maintained separately by the policy). Yices SMT solver [15] is used to answer various questions such as when two memory addresses can alias one another. Each x86 instruction is associated with semantics to emulate how it manipulates the symbolic state. We augmented the emulation process for *CFG*. Additionally, we also modeled calls to library functions using library function prototypes. For example when a call to *strcpy* is made, the first argument is identified as being defined and the second argument as being referenced. We now present other extensions done by us to features provided by Rose infrastructure.

We handled merging of states (when a basic block has more than one predecessor) by assigning a new symbolic value if the variable value in the two predecessor states is different. A mapping is maintained from the new symbolic value to the merged values it represents. If the function has more than one return block, states of all return blocks are merged to get a final return state. We used standard work list algorithm for intra-procedural data dependency analysis.

In order to perform an inter-procedural analysis, the callee function state should be set to that of the caller at the point of call and proceed as usual with intra-procedural analysis for callee. This approach though precise, is not efficient as real life applications contain many procedures and procedure calls. Efficiency and scalability assume more significance in inter-procedural analysis than intra-procedural analysis. We therefore used function summaries for handling calls, since this makes the approach to be scalable, while compromising little on precision.

Function summary is context-independent summary generated to represent the call effect (that is, there is only single summary for the function regardless of the calling context). When the summarized function is called from a caller, based on calling context and synthesized function summary, control and data dependency information is updated and propagated back to the caller. Caller resumes analysis after propagation at instruction subsequent to call.

To generate function summary, the function to be summarized is symbolically executed to record known dependencies and gather unknown references. During analysis the definitions of registers, flags, global variables, argument values and dynamically allocated variables are collected and an end state of function is generated by gathering the values of global variables, function argument values, dynamically allocated variables and eax, esp from the function's return block state.

To apply function summary, at the call instruction in the caller function, the summary of the callee is retrieved first. If definitions of unknown references in the callee summary exist in the caller, dependencies from callee unknown reference instructions to instructions defining those in that of caller are added. If no instruction exists that define these variables/registers in the caller, they are added to unknown references of the caller summary to be propagated up in the call graph and updated when the definition is found. We traverse through the definitions generated in the callee and update their corresponding entries in caller, if they exist, or create new ones if there is no entry. State information is also updated similarly. The tool then resumes with analysis of next instruction in caller. To facilitate function summaries usage, the leaf nodes in the call graph are summarized first before going on to next level and so on. To summarize, given the lack of high-level information and intrinsic complexity and impreciseness associated with binary analysis, it is not impossible to use static analysis alone for inferring BO vulnerabilities precisely from executables. Therefore, using machine learning and static analysis together provides an important means of handling this issue.

III. STATIC CODE ATTRIBUTES

In this paper, we characterize possible defenses implemented in the code for preventing buffer overflows and propose static code attributes for representing them. BO prediction models are then built using this collected attribute data. We use the control and data dependencies of potential vulnerable statement computed by the binary analysis tool to collect the data. Static analysis information is computed using *CFG* of the disassembled binary. Each node in a *CFG* represents a program statement in disassembled binary. Throughout the following discussion we use the terms node and statement interchangeably. Before further discussion we introduce few terms used in the paper.

We define sink *k* as a potential vulnerable program statement in binary disassembly such that the execution of *k* may lead to unsafe operation if values of variables referenced at k are not limited properly. We treat calls to C library functions and statements that write to buffers/containers as sinks in this paper as we predict buffer overflow vulnerabilities.

A predicate node *d* in the *CFG* is called an input validation node of sink *k* if:

i. Both *k* and *d* transitively reference to a common input variable defined at the same node

ii. *k* is transitively control dependent on *d*

A predicate node *p* in the *CFG* is called an input dependent predicate node of sink *k* if:

i. *p* refers to an input variable that *k* is not directly or transitively data dependent upon

ii. *k* is transitively control dependent on *p*

Input dependent predicate nodes are important in depicting that some constraints on non-sink inputs are to be met before reaching the sink.

For ease of understanding we present a source code example in Fig. 2 with BO sinks at nodes 6, 12 and 17 to explain the characterization scheme. *Source* refers to variables referenced at sink and *destination* refers to variable (i.e., buffer) defined at sink. It should be noted that the *source* and *destination* can represent more than one variable (e.g., function arguments passed by reference).

A. Input and Source Characterization

To counteract input-related exploits, it is important to enumerate external inputs that the sink is data dependent upon and study their characteristics. We treat data submitted using command line, read from external files or values from environment variables as input.

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

1) Input Count: This attribute counts the number of sink inputs. This attribute value is set to 1 for sink at node 12 in Fig. 2, as it is data dependent on input node at 17.

2) Inputs with Limiting: This attribute counts the number of sink inputs which have been received by imposing length restrictions. It is relevant in flagging that buffer sizes were considered when receiving external inputs. This attribute value is set to 0 for sink at 12 in Fig. 2 as input at 17 does not impose any length restrictions.

3) Environment Dependent Input: This attribute counts the number of sink inputs that are dependent upon environment variables and indicates that sink is dependent upon them apart from external inputs.

4) Is Source a Buffer: This attribute is used to identify if *source* is a buffer or not. It can have one of the two values, namely, *TRUE* (=1) or *FALSE* (=0). This is set to '1' for sink at 12 in Fig. 2, whereas sink at 6 has this attribute value set to '0'.

5) Is Source Null Terminated: Non-null terminated buffers may cause BO in some program constructs. For example, consider vulnerable C library function like *strcpy* which looks for terminating null character in a *source* string and copies it to the *destination* including the terminating null character. Hence, it is important that strings in the program be null-terminated before further access. We therefore use this attribute to reflect this defensive measure. This attribute can have one of the three values. It is *FALSE* (=0) if none of the variables represented by *source* buffer is null-terminated at least once in the program, *TRUE* (=1) when the *source* buffer (or all the variables represented by it) are null-terminated at least once in the program and *SOMETIMES* (=2) when only some of the variables represented by *source* buffer are null-terminated. This value is set to 2 for sink at 12 because one of the buffers that the *source* (*srcStr*) points to is null terminated (since *gets* null-terminates *localBuf* at 17) whereas the other (i.e., *globalBuf*) is not null-terminated.

B. *Sink Charactersitics*

Different sinks perform different operations. Depending on type of operation they may need diverse forms of bounds checking mechanisms and defensive measures. For example, a *strcpy* call needs to be preceded by bounds check to see if the *destination* can accommodate *source* string, whereas lengths of both *source* buffer as well as *destination* should be compared with *destination* size before a string concatenation operation. Some sinks like *strncpy* have in-built BO defense mechanisms to help limit buffer filling operations. We use attributes capturing these defense measures in our sink characteristics classification.

1) Operation Type: We classify the sinks based on the operation being performed into one of the following five types:

1. *Copy*: Calls to C library functions which perform copying operations (e.g., *strcpy*, *memcpy*)
2. *Concatenation*: Calls to C library functions which perform concatenation operations (e.g., *strcat*, *strncat*)
3. *Formatted Write*: Calls to C library functions which perform formatted write operations (e.g., *sprintf*, *snprintf*)
4. *UnFormatted Write*: Calls to C library functions which perform unformatted write operations (e.g., *gets*, *fgets*)
5. *Array Write*: All other statements that write to containers (e.g., sink at node 6 in Fig. 2).

2) Has Defensive Limiting: This attribute is used to flag usage of secure versions of C library functions where number of elements for filling destination is specified in the sink statement (e.g., *strncpy*, *snprintf*). It can have one of three values namely, *FALSE* (=0), *TRUE* (=1), and *NOT-APPLICABLE* (=2). It is set to *NOT-APPLICABLE* for *Array Write* sinks. This attribute has value of '0' for sinks at 12 and 17 in Fig. 2.

```
1.  # define BUFSZ 32
2.  char globalBuf[BUFSZ];
3.
4.  void   initializeBuf(char   *bufToFill,
    size_t sz) {
5.  for(size_t  count  =0;  count  <  sz;
    count++)
6.  bufToFill[count] = 'a';//OK
7.  }
8.
9.  void copy(char *srcStr) {
10. char dstStr[BUFSZ];
11. if(strlen(srcStr) < BUFSZ)
12. strcpy(dstStr, srcStr); //OK
13. }
14.
15. int main(int argc, char **argv) {
16. char localBuf[BUFSZ];
17. gets(localBuf);//BAD
18. initializeBuf(globalBuf, BUFSZ);
19. if(argc > 1) {
20. if(strcmp(argv[1],"Global") == 0)
21. copy(globalBuf);
22. else copy(localBuf);
23. }
24. … }
```

Figure 2. Sample code snippet

C. *Destination Buffer Characterization*

In this paper we study the *destination* usage patterns in predicting BO vulnerabilities. We therefore propose the following attributes to capture *destination* characteristics.

1) Is Destination Null Terminated: The purpose of this attribute is similar to *Is Source Null Terminated* attribute in *Input and Source Characterization* and likewise can have one of the three values. It is *FALSE* (=0) if none of the variables represented by *destination* is null-terminated at least once in the program, *TRUE* (=1) when the *destination* (or all the variables represented by it) are null-terminated at least once in the program and *SOMETIMES* (=2) when only some of the variables represented by *destination* are null-

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

terminated. This attribute value is set to 1 for sinks at 12 and 17 in Fig. 2.

2) Multiple Writes to Destination: This attribute is used to gather information pertaining existence of multiple writes to *destination*. When multiple writes exist, checks should be made on available space and care should be taken that further filling operations do not overflow buffer. It can have one of the three values. It is *FALSE* (=0) if none of the variables represented by the *destination* was written more than once in the program, *TRUE* (=1) when the *destination* (or all the variables represented by it) is written more than once in the program and *SOMETIMES* (=2) when only some of the variables represented by *destination* are written more than once in the program.

3) Location: This attribute depicts the buffer location and can have one of the following four values namely:

1. *Global*: *Destination* is a global variable.
2. *Local*: *Destination* is a local variable. Sink node at 12 in Fig. 2 is classified as this type.
3. *Mixed*: *Destination* is represented by global, local or heap variables.
4. *Heap*: *Destination* is a heap variable.

4) Same Source And Destination Size: In some coding constructs both *source* buffer and *destination* have same size and *source* buffer size is taken into consideration while filling it before it is copied to the *destination*. Hence we use this attribute to capture such defensive measures. It can have one of the three values namely *TRUE* (=1) when the variables represented by *destination* and *source* have same size, *FALSE* (=0) when the *destination* and *source* do not have same size even once and *SOMETIMES* (=2) when otherwise. This attribute value was set to '2' for sink at 12, since the IDA Pro plugin identifies the size to be 40 instead of 32 for global variable due to holes in compiled code.

5) Declaration: Buffer declaration is used as defensive measure in some coding constructs where *destination* is allocated based on *source* buffer before filling operation. We therefore include this attribute to gather such information. It can have one of the following 5 values:

1. *Static*: *Destination* is declared statically. All the sinks in Fig. 2 are classified as this type.
2. *Dynamic Source Dependent: Destination* is declared dynamically and the declaration statement is data dependent upon sink *source*.
3. *Dynamic Source Independent: Destination* is declared dynamically and the declaration statement is not data dependent upon sink *source*.
4. *Dynamic Constant Size: Destination* is declared dynamically using constant size.
5. *Mixed:* If *destination* is declared through more than one of the four above mentioned types, then it is set to this value.

D. Control Dependency Characteristics

It is important to carry out bounds checking operations before filling buffers to prevent overflows. Therefore predicates checking for buffer size and length are important in predicting BO vulnerabilities. In addition to this, all inputs should be thoroughly validated before propagation to safeguard the system from input-manipulation attacks. We therefore include attributes depicting control dependency characteristics of sink node to capture this information.

1) String Length of Source: This attribute counts the number of sink predicates that refer to string length of buffer referenced at sink, *k* via call to *strlen*. Node 11 in Fig. 2 is performing this check for sink at node 12.

2) Size of Source: This attribute counts the number of sink predicates that refer to size of buffer referenced at *k*. Node 11 in Fig. 2 is classified as performing this check for sink at 12 as both *source* buffer and *destination* have same size.

3) String Length of Destination: This attribute counts the number of sink predicates referring to string length of buffer defined at *k* via call to *strlen*.

4) Size of Destination: This attribute counts the number of sink predicates that refer to size of buffer defined at *k*. Node 11 in Fig. 2 is performing this check for sink at 12.

5) Source Dependent Loop Termination: Buffers are sometimes filled in loops. If the loops are terminated by taking into account only *source* without buffer size the application could be susceptible to overflows. To examine such constructs we include this attribute. It is *TRUE* (=1) when sink statement is in a loop and loop termination directly or transitively refers to sink *source*, *FALSE* (=0) when otherwise and *NOT-APPLICABLE* (=2) when sink statement is not in loop. This is set to '1' for sink at 6 in Fig. 2 due to *count*, which is a sink *source*.

6) Validation: This attribute counts the number of sink validation nodes. Node 11 is the validation node for sink at 12 in Fig. 2 and hence this attribute value is set to '1' for this sink.

7) Input Dependent Predicates: This attribute counts the number of sink input dependent predicates. It is set to '2' for sink at 12 due to input predicates at 19 and 20 in Fig. 2.

8) Source Buffer Predicates: This attribute counts the number of sink predicates which are directly or transitively dependent on buffer referenced at *k* and helps in inferring on number of checks on sink *source* buffer.

9) Destination Buffer Predicates: This attribute counts the number of sink predicates which are directly or transitively dependent on buffer defined at *k* and helps to infer on checks performed on *destination*.

E. Data Dependency Precision Characterization

It can be seen from Section II that container-level precision and function summaries introduce some imprecision in data dependency algorithm as trade-off for scalability. We therefore include attributes that capture this imprecision and observe their effect in predicting

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

vulnerabilities. We introduce few terms here pertaining data dependency ambiguity.

If statement *'s'* writes to container *'c'*, and if *'c'* has writes to it such that at least one write is data dependent on an input variable and there exists another that is not data dependent on any input variable, then container *'c'* is termed as ambiguous. Conversely, if writes to *'c'* are not ambiguous with respect to input then *'c'* is termed as non-ambiguous. If either all the writes to *'c'* are data dependent on some input, or if all of them are independent of input or if *'c'* is only written once in the program then it is considered non-ambiguous. The following attributes are used to capture data dependency precision ambiguities:

1) Inter Procedural Destination Buffer Access: This attribute is used to capture effect of function summaries on *destination* dependencies. It is *TRUE* (=1) if *destination* is accessed and written inter-procedurally in the program and *FALSE* (=0) if vice-versa. This is set to '1' for sink node at 6 in Fig. 2.

2) Number of Destination Writes: This attribute counts the total number of writes to variables represented by the *destination* other than the write at sink node.

3) Is Destination Buffer Ambiguous: This attribute is set to *TRUE* (=1) if *destination* is ambiguous container and *FALSE* (=0) if vice-versa. This attribute value for sink at 12 in Fig. 2 is set to '0' since there is only one write to *destination*.

4) Is Source Ambiguous: This attribute is set to *TRUE* (=1) if *source* buffer is ambiguous container and *FALSE* (=0) if vice-versa. It is set to *NOT-APPLICABLE* (=2) when there are no writes to *source* (e.g., global variables initialized at declaration, constant sources). This attribute takes on value of '2' for sinks at 6 and 17 in Fig. 2.

5) Sink Dependent Ambiguous Containers: This attribute counts the number of ambiguous containers that the sink is directly or transitively dependent upon.

6) Sink Dependent Non-Ambiguous Containers: This attribute counts the number of non-ambiguous containers that the sink is directly or transitively dependent upon.

7) Predicates Referring To Ambiguous Containers: This attribute counts the number sink predicates that refer to ambiguous containers.

8) Predicates Referring To Non-Ambiguous Containers: This attribute counts the number sink predicates that refer to non-ambiguous containers. This is set to '1' for sink at 12 due to the predicate at 11.

All non-numeric attributes like those representing *TRUE*, *FALSE*, *NOT-APPLICABLE* are declared as nominal in our data set and are treated as such by WEKA algorithms despite their numeric values.

IV. EXPERIMENTS

In this paper, we build classifier models for predicting BO vulnerabilities from binaries. Since we predict buffer overflows, calls to string or memory block manipulation functions like *strcpy*, *memcpy* are regarded as potential vulnerable statements. All container writes are also treated as potential vulnerable statements since we do not have type information for container to determine if they are *"char"* arrays or not. For each of the sinks identified, our tool collects attributes as per the BO characterization scheme proposed in Section III using the information from static analysis tool. In this experiment we evaluated four well-known classifiers on data set with known vulnerability information. By training the prediction models on application binaries with known vulnerability information we can predict vulnerabilities in other applications whose binaries are available.

We used off-the-shelf tools like WEKA [12] and IDA Pro [14] to facilitate adoption and built-upon existing tools like ROSE [13] open-source binary analysis infrastructure to help replication. Fig. 3 shows the work flow of the proposed approach. We used IDA Pro for disassembling binaries. IDA Pro provides access to its internal data structure through API enabling users to write plugins which can be executed by IDA Pro. We wrote an IDA Pro plugin called *"Extractor"* which writes the disassembly of all the functions identified by IDA Pro in the executable to a file. *"Extractor"* also gathers variable address/stack offset and size information and data segment boundary addresses and writes them to Xml files. These files are inputs to our *"BinAnalysis"* tool built on top of ROSE binary analysis framework.

"BinAnalysis" is a static program analysis tool for computing control and data dependencies. While performing static analysis, *"BinAnalysis"* simultaneously identifies sinks and collects data about buffers (i.e., containers) defined at sink. To identify buffers, the tools uses library function call semantics, and variable access information (e.g., index based addressing) collected during static analysis. *"VulMiner"* uses the API of *"BinAnalysis"* to extract code attributes. This data set is then modified to add sink vulnerability information and is given as input to WEKA data mining tool for training and testing BO prediction models.

A. Benchmark

For our experimental evaluation we used applications from MIT Lincoln Laboratories buffer overflow benchmark [16]. This benchmark was used in previous buffer overflow studies [10, 17, 18]. It contains model programs with and without buffer overflow bugs developed from reported vulnerabilities in three real-world open source network servers, namely Bind, Wu-ftpd and Sendmail. Each of the three application programs has several bugs reported in CVE and CERT data bases and accordingly captured in the model programs. The programs have bugs due to unsafe C library function calls and also due to array accesses. Vulnerable buffers location varies among stack, heap and bss segment (used to allocate memory for global variables)

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

and buffers are mostly accessed using indirection caused by aliases.

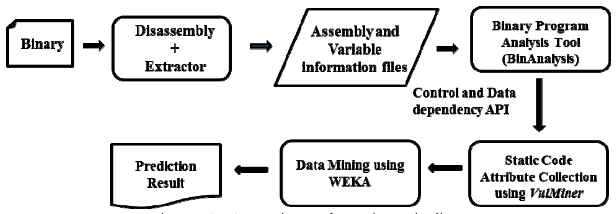
Binary
Disassembly + Extractor
Assembly and Variable information files
Binary Program Analysis Tool (BinAnalysis)
Control and Data dependency API
Static Code Attribute Collection using VulMiner
Data Mining using WEKA
Prediction Result

Figure 3. Overview of Tool work-flow

Table I gives the statistics of 6 programs used in the evaluation. As the programs are designed to be self-contained, b1 and b2 programs include code to generate input. We modified it in line with b3 to receive input and also removed a user-defined function (*newstr*) which is not called by any other user-defined function in the program. "LOC" column in Table I represents the program size in C source code and "Sinks" represents the number of potential vulnerable statements identified by binary static analysis tool. The number of sinks in binary data set is higher than actual buffer writes in the source code since IDA Pro could not identify few *struct* types in some binaries and hence all writes to *struct* variables were taken to be sinks. Given that we deal with binaries which lack such information, this is to be expected.

TABLE I. DATASET STATISTICS

Name	LOC	Sinks	Bugs
b1(NXT-Bug)	1.15K	32	1
b2(SIG-Bug)	1.39K	40	1
b3(Iquery)	0.28K	10	1
f1(mapped chdir)	0.42K	21	4
f2(off-by-one)	0.94K	24	3
f3(realpath)	1.01K	57	19

B. *Data Collection*

"BinAnalysis" identifies sinks while performing data dependency analysis. *"VulMiner"* queries transitive control and data dependencies along with information collected during static analysis for each of the identified sinks to extract attribute values based on BO defense characterization scheme described in Section III. This is then modified to add known sink vulnerability information retrieved manually to prepare the final data set. The data set thus prepared is given as input to data mining tool for building vulnerability prediction models.

For each sink in the program, *"VulMiner"* extracts the attribute values as per characterization scheme described in Section III. There are, in total, 30 attributes consisting of *"Input Count"* and 28 other attributes along with the target attribute -*"Vulnerable?"*. Hence, the attribute vector for sink node at 12 in Fig. 2 is (1,0,0,1,2,1,0,1,0,2,2,1,1,1,0,1,2,1,2,1,0,0,0,0,0,0,2,0,1,FALSE) corresponding to *(Input count, Inputs With Limiting, .., Operation Type,.., Declaration, String Length Of Source Buffer, .., Predicates Referring To Non-Ambiguous Containers and Vulnerable).*

C. *Experimental Design*

There are in total 184 sinks, out of which 29 are vulnerable. We used standard 10-fold cross validation option in WEKA on the data set prepared as outlined in Section IV-B.

D. *Classifiers*

To assess our approach we used four well-known machine learning algorithms, each with different principles and yielding different prediction models in the WEKA data mining tool kit: naïve Bayes (NB), Multi-Layer Perceptron (MLP), Simple Logistic (SL) and Sequential Minimum Optimization (SMO). These algorithms use features or attributes to predict a class (in our context BO vulnerabilities).

V. EVALUATION

A. *Objective*

The objective of evaluation is to answer the following research questions:

1. *"Is proposed approach combining machine learning and static analysis effective in predicting vulnerable sinks?"*
2. *"How do binary BO prediction models compare with source code static analysis tools?"*
3. *"Which static code attributes are best indicators of BO vulnerabilities?"*

B. *Performance Measures*

We used four criteria commonly used in defect prediction model studies to assess the classifier's performance. Based on the confusion matrix depicted in Fig. 4, they are defined as follows:

- Probability of detection or recall *(pd) = tp / (tp + fn)*
- Probability of false alarm *(pf) = fp / (fp + tn)*
- Precision *(pr) = tp / (tp + fp)*
- Accuracy *(acc) = ((tp + tn) / (tp + fp + fn + tn))*

	Classified Vulnerable	Classified Non-Vulnerable
Actual Vulnerable	True Positive (tp)	False Negative (fn)
Actual Non-Vulnerable	False Positive (fp)	True Negative (tn)

Figure 4. Confusion matrix generated by WEKA

To detect all bugs *pd* should be close to 1 whereas *pf* should be close to 0 to minimize false alarms. *pr* measures how many of those predicted vulnerable are bugs, in actual. *acc* measures the overall correctly classified instances.

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

C. Results

The focus of our experiment is to evaluate the accuracy of proposed approach. Different prediction models were used as it cannot be known prior which model will yield better performance. The result of all classifiers using 10-fold cross validation on data set prepared as described in Section IV-B is depicted in Table II.

TABLE II. PERFORMANCE MEASURES FOR CLASSIFIERS

Model	***pd***	***pf***	***pr***	***acc***
NB	89.7	15.5	52	85.32
MLP	75.9	5.2	73.3	91.85
SL	62.1	2.6	81.8	91.85
SMO	75.9	2.6	84.6	94.02

It can be seen from Table II that tested classifiers had a recall over 60%, which means that 6 out of 10 actual vulnerabilities were predicted correctly. Except NB which had an accuracy of 85.32% all the classifiers tested had accuracy over 90%. While accuracy of NB is lower than rest, it has the highest recall rate of 89.7% along with highest false alarm probability of 15.5%.

SMO reported the best performance with 75.9% recall and 84.6% precision suggesting that more than 7 out of 10 bugs were predicted correctly and more than 8 out of 10 statements predicted to be vulnerable are indeed vulnerable. This could be due to the fact that SMO builds binary support vector machine, which constructs a hyper-plane to maximally separate instances belonging to different classes. MLP also had recall of 75.9% but had lesser precision than SMO on the data set. The results suggest that proposed static analysis cum machine learning approach is effective in predicting vulnerable sinks.

D. Comparison with Static Source Code Analysis Tools

In this section we compare performance of our prediction models with that obtained by static source code analysis technologies using the results reported in [19]. However, it should be noted that the number of sinks is different between those reported in [19] and that in this study but comparison of recall is relevant as it considers only truly vulnerable sinks. Nevertheless, we give all the performance measures of the tools with positive recall in Table III to provide an insight of their functioning on the benchmark programs.

A brief detail of each of the tested tools is provided here for understanding. Splint monitors buffer creation and accesses to detect BO vulnerabilities [5]. It uses annotated constraints for ensuring safety and propagates these constraints across statements and raises warnings when it cannot determine them to be safe. PolySpace [20] is a commercial tool which uses advanced formal methods like abstract interpretation to detect buffer overflows, division by zero and other source code errors. BOON [6] models buffers based on its size and usage. It converts buffer overflow to an integer constraint solving problem by generating constraints for calls to vulnerable C string manipulation functions and checking if the inferred allocated size is at least as large as the maximum used size.

UNO [21] uses a public-domain compiler extension to generate parse tree for each procedure in a program, which are later turned into *CFGs*. The resulting *CFGs* are analyzed using a model checker to find array indexing errors. Archer (Array Checker) [4] uses bottom-up inter-procedural analysis to detect memory access violations in source code. Starting at bottom of call graph Archer uses symbolic constraints to determine ranges for function parameters that result in memory access violations. The computed constraints are passed to the callers of these functions to deduce new constraints and so on. When the top-most caller is reached, the constraints are solved. If any of the constraints is satisfied, it flags the statement for memory access violation.

It can be seen from Table III that Splint has highest recall of all the tools followed by PolySpace. Boon could detect only one bug in all the 6 tested programs. Uno and Archer could not detect any bugs but they also did not raise any false warnings in patched versions of buggy programs. Hence their performance measures were not included in Table III as recall is zero. Splint and PolySpace also have high false alarm probabilities and none of the tested tools had a precision over 60%.

These results answer our second research question by showing that our binary BO classifiers are effective in predicting BO vulnerabilities with a recall higher than 62%, which is better than the best performing static source code analysis tool, Splint, with recall of 61.11%. Our best classifier, SMO, has a notable recall of 75.9%. It can be inferred from the above discussion that using static analysis alone to detect BO bugs from binaries could be difficult and that in the absence of source code our approach using prediction method and static analysis can be considered as an effective and practical alternative to BO detection.

TABLE III. PEROFRMANCE MEASURES FOR STATIC SOURCE CODE ANALYZERS

Tool	***pd***	***pf***	***pr***	***acc***
Splint [5]	61.11	40.54	59.46	60.27
PolySpace [20]	55.56	48.65	52.63	53.42
Boon [6]	2.78	2.70	50.00	50.68

E. Attribute Ranking using InfoGainAttributeEval in WEKA

To answer our third research question we ranked the attributes using InfoGainAttributeEval attribute selection algorithm in WEKA. This algorithm evaluates the worth of algorithm by measuring the information gain achieved by its selection with respect to class. Fig. 5 depicts the outcome of attribute information gain ratio for 15 best ranked attributes. It can be seen from the figure that as expected, attributes representing defense practices like input validation, usage of

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

safe C library functions, sink operation type and source and destination buffer predicates along with destination size checks are important in BO prediction. Presence of data dependency ambiguity attributes indicates that container-level precision also effects prediction.

```
0.27786        Validation
0.19146        Predicates  Referring  To  Non-
Ambiguous Containers
0.18436        Has Defensive Limiting
0.17212        Predicates    Referring    To
Ambiguous Containers
0.14389        Operation Type
0.14296        Size of Destination
0.12873        Sink  Dependent  Non-Ambiguous
Containers
0.12656        Source Buffer Predicates
0.12286        Location
0.11937        Declaration
0.10353        Is Source Null Terminated
0.09236        Same  Source  and  Destination
Size
0.09044        Is Destination Null Terminated
0.08503        Destination Buffer Predicates
0.04561        Is     Destination     Buffer
Ambiguous
```

Figure 5. InfoGainAttributeEval Outcome for 15best ranked attributes

F. *Limitations*

While the proposed approach is effective in predicting BO vulnerabilities, it has certain limitations. Our approach uses machine learning algorithm for training prediction models using static code attributes extracted from binary disassembly. Although the static code attributes can be easily collected, it needs a binary analysis tool to do so and requires sufficient known vulnerability data to train the prediction models. We rely on IDA Pro to identify function boundaries and variable offsets which, though capable, is not fool-proof. Our data dependency analysis introduces container-level precision (which is generally in-line with that offered by commercial source code analysis tools) and function summaries, which may cost us in precision. IDA Pro variable identification and data dependency computation may effect sink identification. In spite of these expected binary analysis impediments we believe that our approach provides an effective and practical alternative of addressing binary vulnerability detection.

G. *Threats To Validity*

For every empirical study it is important to be aware of potential threats to the validity of obtained results. In this paper, we use binary static analysis and machine learning for predicting buffer overflows. Training and testing procedures may influence the outcome of data mining experiments resulting in threats to internal validity. We used 10-fold cross validation option provided in WEKA to cancel out any factors that may influence the results due to training and testing data set preparation. External validity threats refer to factors preventing the generalization of results. It is hard to get many publicly available real-world programs with known vulnerability information for training the models. We therefore used benchmarked applications for evaluating generalization abilities of our prediction models.

VI. RELATED WORK

In this section we present works that are related to ours in binary BO detection and vulnerability prediction.

A. *Binary Buffer Overflow Detection*

LibVerify [22], implemented as a DLL, works with binaries to intercept calls to vulnerable string manipulation functions like *strcpy* and limits any stack buffer overflows to the current stack frame, thus preventing function frame activation record based exploits. It also implements return address verification in addition to limiting overflows to current stack frame. This approach can therefore only handle frame-activation record exploits but not others like those involving pointer manipulation, virtual table and heap-based exploits.

Dytan [23] is a customizable dynamic taint analysis framework that allows users to specify taint sources and sinks, and, taint propagation rules. Based on user specifications, Dytan produces an instrumented executable and flags when tainted data reaches the sink. Since it stores all *CFGs*, post-dom trees etc. in the memory and implements taint markings for each byte, the approach results is very high performance overheads, of the order or 240 times approximately for GZIP. In [24] a path-feasibility based symbolic execution approach was employed to statically analyze binaries and raise warnings when tainted data from external input reaches sensitive sinks. This could be computationally expensive in binaries with complex call graph and it does not specifically deal with proving BO bugs.

B. *Binary Buffer Overflow Test Input Generation*

Symbolic execution techniques examine one execution path at time to examine the effect of data dependency on symbolic variables. LESE [17] generalizes this concept to loops and introduces symbolic variables for representing number of loop executions and links its dependency to input variables. These constraints are solved to identify the input length for triggering overflows. This approach is suitable in triggering bugs which are dependent upon input dependent loop execution and may not be suitable for others.

In [18], a three stage-process was employed for test generation. In the first stage it performs dynamic analysis to resolve indirect jumps and build the program's control flow. Second stage involves static analysis to identify potential vulnerable statements for generating test inputs. In the third stage it uses dynamic symbolic execution to generate inputs for proving vulnerabilities identified by static analysis in second stage. The solution was effective in generating bug revealing inputs in all but one of tested programs and it was reported that static analysis stage generated warnings on all

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.

known bugs but it also had high number of false alarms resulting in low precision of 28%. Our approach with good recall and precision could also serve as complimentary aid in such tools to predict likely vulnerabilities after static analysis stage and devote the guided test generation to confirm predicted vulnerabilities.

C. Vulnerability Prediction

Vulnerability prediction approaches can be broadly classified into statistical prediction and data mining based approaches. Tools like Vulture [25] mine existing vulnerability databases and version archives and map vulnerabilities to software components using attributes representing imports and function-calls made by these components. This may be effective only for predicting applications which have such existing database while our approach can be used on any x86 executable. Data mining based approaches use attributes representing LOC, execution complexity etc. [26] or domain specific bug characterization attributes [27] for predicting vulnerabilities. These solutions need presence of source code which may not always be available.

VII. Conclusion

In this paper, we proposed a novel methodology using static analysis and machine learning for predicting buffer overflow vulnerabilities from x86 executables by characterizing buffer usage patterns and defensive mechanisms implemented in the code. The proposed characterization attributes are extracted by statically analyzing disassembled binaries. Our approach can be used to automate bug detection in third-party components or binaries whose source code is not readily available. Performance measures of prediction models from our experimental evaluation using standard BO benchmark were encouraging. Our best classifier model had a recall of 75% with 84% precision and 94% accuracy suggesting that our proposed attributes are effective indicators in predicting buffer overflow vulnerabilities. In future, we intend to conduct more experiments to further validate these claims.

References

[1] http://osvdb.org/show/osvdb/116167

[2] http://osvdb.org/show/osvdb/116359

[3] http://cwe.mitre.org/documents/vuln-trends/index.html

[4] Y. Xie, A. Chou, and D. Engler, “ARCHER: using symbolic, pathsensitive analysis to detect memory access errors,” Proc. of Foundations of software engineering (FSE 03), Sep. 2003, pp. 327-336, doi:10.1145/940071.940115.

[5] D. Evans and D. Larochelle, “Improving Security Using Extensible Lightweight Static Analysis,” IEEE Softw., vol. 19, no. 1, Jan. 2002, pp. 42-51, doi:10.1109/52.976940.

[6] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken., "A first step towards automated detection of buffer overrun vulnerabilities," Proc. Network and Distributed System Security Symposium, 2000, pp. 3–17.

[7] C. Cowan, C. Pu, D. Maier et al., “StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks,” Proc. USENIX Security Symposium, Volume 7, 1998.

[8] F. Gadaleta et al., “Instruction-level countermeasures against stack-based buffer overflow attacks,” Proc. EuroSys Workshop on Virtualization Technology for Dependable Systems (VDTS 09), pp. 7-12, doi:10.1145/1518684.1518686.

[9] X. Wang, C. Pan, P. Liu, and S. Zhu, “SigFree: A Signature-Free Buffer Overflow Attack Blocker,” Dependable and Secure Computing, IEEE Transactions on, vol. 7, no.1, Jan. 2010, pp. 65-79, doi:10.1109/TDSC.2008.30.

[10] H. Shahriar, H.M. Haddad, and I. Vaidya, “Buffer overflow patching for C and C++ programs: rule-based approach,” SIGAPP Appl. Comput. Rev., vol. 13 no. 2, June 2013, pp. 8-19, doi:10.1145/2505420.2505421.

[11] G. Balakrishnan, and T. Reps, “WYSINWYX: What you see is not what you eXecute,” ACM Trans. Program. Lang. Syst., vol. 32, no. 6, Aug. 2010, pp. 1-84, doi:10.1145/1749608.1749612.

[12] http://www.cs.waikato.ac.nz/ml/weka/

[13] http://rosecompiler.org/

[14] https://www.hex-rays.com/products/ida/

[15] http://yices.csl.sri.com/

[16] M. Zitser, R. Lippmann, and T. Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” Proc. Foundations of software engineering (FSE 04), Nov. 2004, pp. 97-106, doi:10.1145/1041685.1029911.

[17] P. Saxena, P. Poosankam, S. McCamant, and D. Song, “Loop-extended symbolic execution on binary programs,” Proc. International Symposium on Software testing and analysis (ISSTA 09), 2009, pp. 225-325, doi:10.1145/1572272.1572299.

[18] D. Babic, L. Martignoni, S. McCamat and D. Song, “Statically-directed dynamic automated test generation,” Proc. International Symposium on Software Testing and Analysis (ISSTA 11), 2011, pp. 12-22, doi:10.1145/2001420.2001423.

[19] M. Zitser, “Securing software: An evaluation of static source code analyzers,” Master’s thesis, Massachusetts Institute of Technology, 2003.

[20] http://www.polyspace.com/downloads.html

[21] G. J. Holzmann., "UNO: Static source code checking for user-defined properties," Proc. World Conference on Integrated Design and Process Technology, 2002.

[22] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” Proc. USENIX Annual Technical Conference, 2000.

[23] J. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” Proc. International Symposium on Software testing and analysis (ISSTA 07), 2007, pp. 196-206, doi:10.1145/1273463.1273490.

[24] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, “Static Detection of Vulnerabilities in x86 Executables,” in Computer Security Applications Conference (ACSAC 06), Dec. 2006, pp. 269-278, doi:10.1109/ACSAC.2006.50.

[25] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” Proc. ACM conference on Computer and communications security (CCS 07), 2007, pp. 529-540, doi:10.1145/1315245.1315311.

[26] Y. Shin, and L. Williams, “An initial study on the use of execution complexity metrics as indicators of software vulnerabilities,” Proc. International Workshop on Software Engineering for Secure Systems (SESS 11), 2011, pp. 1-7, doi:10.1145/1988630.1988632.

[27] L. K. Shar, H.B.K. Tan, and L.C. Briand, “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” Proc. International Conference on Software Engineering (ICSE 13), May 2013, pp. 642-651.

Authorized licensed use limited to: SLIIT - Sri Lanka Institute of Information Technology. Downloaded on February 18,2021 at 04:02:23 UTC from IEEE Xplore. Restrictions apply.