IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 22, 2019, accepted May 12, 2019, date of publication May 20, 2019, date of current version June 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2917668

Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study

HONGFA XUE, (Student Member, IEEE), SHAOWEN SUN, GURU VENKATARAMANI, (Senior Member, IEEE), AND TIAN LAN, (Member, IEEE)
The George Washington University, Washington, DC 20052, USA
Corresponding author: Hongfa Xue (hongfaxue@gwu.edu)

This work was supported by the U.S. Office of Naval Research (ONR) under Award N00014-15-1-2210 and Award N00014-17-2786.

ABSTRACT Binary code analysis is crucial in various software engineering tasks, such as malware detection, code refactoring, and plagiarism detection. With the rapid growth of software complexity and the increasing number of heterogeneous computing platforms, binary analysis is particularly critical and more important than ever. Traditionally adopted techniques for binary code analysis are facing multiple challenges, such as the need for cross-platform analysis, high scalability and speed, and improved fidelity, to name a few. To meet these challenges, machine learning-based binary code analysis frameworks attract substantial attention due to their automated feature extraction and drastically reduced efforts needed on large-scale programs. In this paper, we provide the taxonomy of machine learning-based binary code analysis, describe the recent advances and key findings on the topic, and discuss the key challenges and opportunities. Finally, we present our thoughts for future directions on this topic.

INDEX TERMS Machine learning, program binary analysis, taxonomy.

I. INTRODUCTION

Binary code analysis (BCA) allows software engineers to directly analyze binary executables without access to source code. It is widely used in various domains where there is limited availability of source code, e.g., due to proprietary issues or simply impossible to trace any source code. Today, BCA has become more important than ever due to legacy programs that have been installed in a variety of environments, including the Internet of Things (IoT). It is currently estimated that there are more than 20 billion IoT devices worldwide by 2020 [67]. BCA can be useful for IoT and other mission-critical environments (e.g., defense, hospitals) and provide key tools for improving software security in such places [56], [115], [129].

Note that it is difficult to directly analyze binary executables when compared to program source code. First, it is challenging, if not impossible at all, to recover the original source code or semantic information from the representation of binary code. Second, commercial software and operating systems are usually slightly obfuscated to deter reverse engineering and unlicensed use. On the other hand, system and kernel libraries are often optimized to reduce disk space requirements. For instance, it may be difficult to locate function entry points (FEPs) since the full symbol or debug information is usually not available in optimized binaries.

Recently, machine learning techniques have been employed to automatically extract features through large amounts of data and have achieved significant success in the field of source code analysis [5], [117], [142], [144]. Inspired by those prior works on the source code, machine learning-based BCA has also been widely studied as well. Tesauro *et al.* [123] first introduced a neural network-based method for recognizing virus in application binaries. Since then, machine learning-based BCA has become a significant research topic in vulnerability detection, function recognition, and other areas.

In this article, we discuss several aspects of binary code analysis and outline the machine learning algorithms used in such analyses. We provide a taxonomy of machine learning-based binary code analysis techniques and discuss the key challenges and opportunities. Finally, we present our thoughts for future directions on this topic.

II. BACKGROUND

BCA is a key requirement for many software engineering tasks that include:

The associate editor coordinating the review of this manuscript and approving it for publication was Hongbin Chen.

- vulnerability discovery (*including integer, heap or stack buffer overflow, denial-of-service attack, Input manipulation, authentication bypass vulnerabilities*)
- refactoring (*a restructuring process of changing a program binary to creates new versions that implement or propose change to the program binary while preserving the program's external behavior (functionality and semantics)*)
- plagiarism detection (*detecting a plagiarized program, which is a program that has been generated from another program with trivial text edit operations*).

Especially with the rapid growth of IoT devices and the complexity of software applications, program binaries are often shared among multiple platforms. Imagine a single bug is injected at source code level, it may spread across thousands or more devices that have diverse hardware architectures and software platforms. Thus, binary analysis is particularly critical and more crucial than ever.

BCA is used to provide information about a program's content (instructions, basic blocks, functions, and modules), structure (control and data flow), and data structures (global and stack variables), and is, therefore, a foundation of many security applications. Code Clone Detection in binaries seeks to find code sequence used more than once, copy/paste or reused code, or same source code but compiled under different Instruction Set Architectures [19], [66], [110], [150]. Malware detection detects malicious programs which have vulnerabilities inside and will cause damage to systems or programs crashes [10], [12], [18], [32], [36], [41], [58], [82], [126], [127], [146]. Code obfuscation translates the original program into another one preserving its function but making it hard for analysis [74], [86], [114], [119], [135]. Binary reverse engineering translates the binary program into high-level readable language, such as converting binaries back to source code [29], [79], [85], [118]. Binary customization directly changing the binary program through binary rewriting, removing unused or vulnerable codes/functions in program binaries [25], [26]. Recent studies have shown that software can manipulate hardware features as well to do malicious activities like information leakage [2], [145], [147] and memory corruption [23], [24], [119]. Such studies call for a more thorough understanding of software binaries [95], [128] and assessing their potential to be exploited in security attacks.

A. KEY CHALLENGES

Legacy program binaries exist in many production systems, e.g., airspace, military, and banking. To detect bugs or analyze software system safety, executable binary codes are the only source of information about program content and behavior. The compile, link, and optimize steps can cause a program's detailed execution behavior to differ substantially from its source code. Although it is easy to compile source codes into binary executables, it is hard to reverse the binary codes back to source codes for further analysis. Often, it is relatively better to have binary executables be analyzed into intermediate representations (e.g., binary assembly code), which consist of limited code syntax and semantic features comparing to the source code.

We note that the source code can be compiled in different platforms (e.g. x86, MIPS). This leads to the syntactic binary representations to be very different for the same program compiled on two platforms, bearing very different structures. Such cross-platform binary code analysis problems have been tackled, only recently [43], [47]. These efforts use frameworks to extract various robust platform-independent features directly from binary code. We will discuss more details in Section VI.

Example: A vulnerable function in the source code may propagate into hundreds or more IoT devices that have diverse hardware architectures and software platforms after being compiled using the same source code. For instance, a real-world denial-of-service attack vulnerability CVE-2013-6449 has completely different control flows when compiled under x86 and MIPS architecture as shown in Figure 1, even though they both are compiled from the same source code from OpenSSL library (version 1.0.1a).

To deal with these challenges, the traditional approaches for BCA are mainly through pure statistical methods or pure formal analysis (e.g., binary symbolic execution). Statistical methods depend on logging of program states for analysis. In reality, the program execution often yields partial/incomplete logs, it becomes extremely hard for statistical methods alone to accurately achieve the goal through binary code analysis. For example, in vulnerability discovery, it could easily miss vulnerable paths (false negatives) due to inadequate statistical profile data and low code analysis coverage. On the other hand, pure formal analysis can guarantee no false positives with a better analysis accuracy and significantly improve code coverage. However, it is always a time-consuming approach and cannot be deployed in large scale programs due to exponential state increase (path explosion) problem.

B. GENERAL FRAMEWORK

To overcome the difficulty of processing binary codes and perform the code analysis in an automated fashion, Machine Learning techniques have been adopted. Several machine learning-based code analysis frameworks have been deployed at the source code level that adapt natural language processing (NLP) techniques at source code for different purposes. For example, White *et al.* [133] introduce a learning-based framework for code similarity detection in the source code, where Recursive Neural Network (RNN) is deployed to profile code sequences.

A machine learning-based BCA framework has generally two stages: the training stage and the analysis stage. The general framework of machine learning based binary code analysis is shown in Figure 2. In the training stage, the target

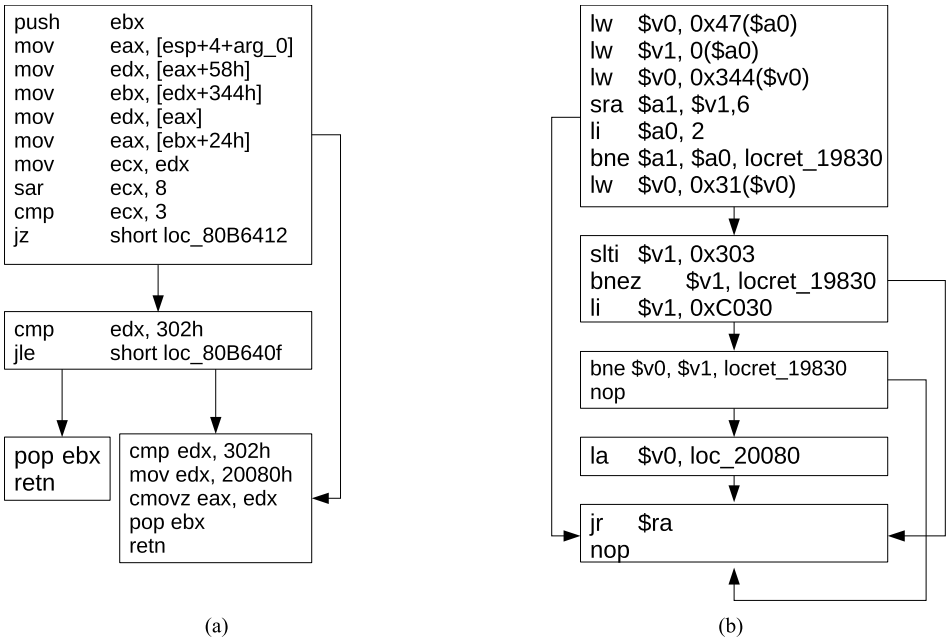

FIGURE 1. The control flow graph comparison for the vulnerable function CVE-2013-6449 under different architectures [46]. (a) x86 assembly. (b) MIPS assembly.

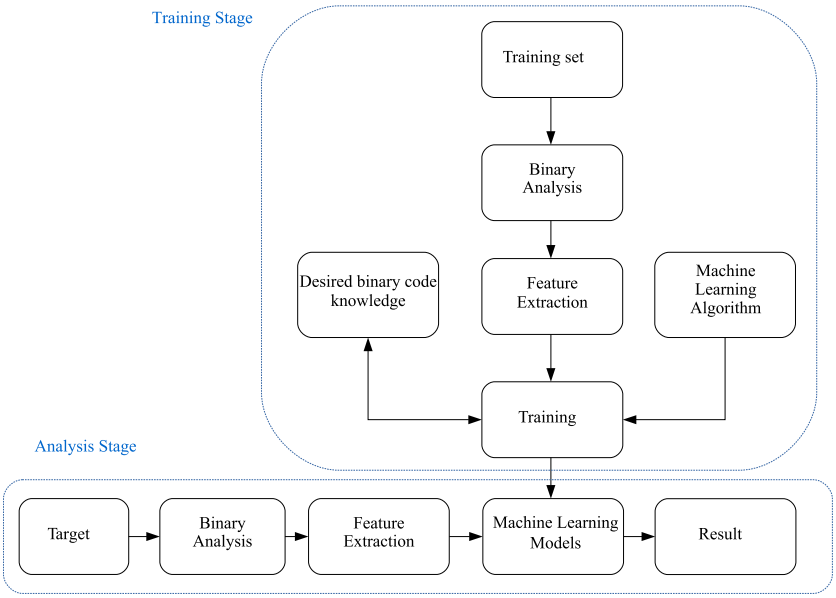

FIGURE 2. General framework of machine learning based binary code analysis.

binary code will be analyzed first and unique features will be extracted from the analysis results. Then, the Machine learning algorithm will be trained with input features and desired binary code knowledge. In the analysis stage, given a target binary code, features will be extracted after program analysis (e.g. lexical analysis and syntax analysis). According to the training result, machine learning models can be used to identify such features to achieve certain analyzed goals such as discovering vulnerable paths, finding performance bugs and imbalance and so on.

C. THE TAXONOMY

According to the general framework shown in Section II-B, existing machine learning-based BCA systems can be broadly divided into four major components. In Figure 3, we present the taxonomy of machine learning-based BCA framework in detail. The remaining sections of this paper discuss the various existing methods from the perspective of feature extraction, feature embedding, analysis techniques used in BCA and corresponding applications in the real-world.

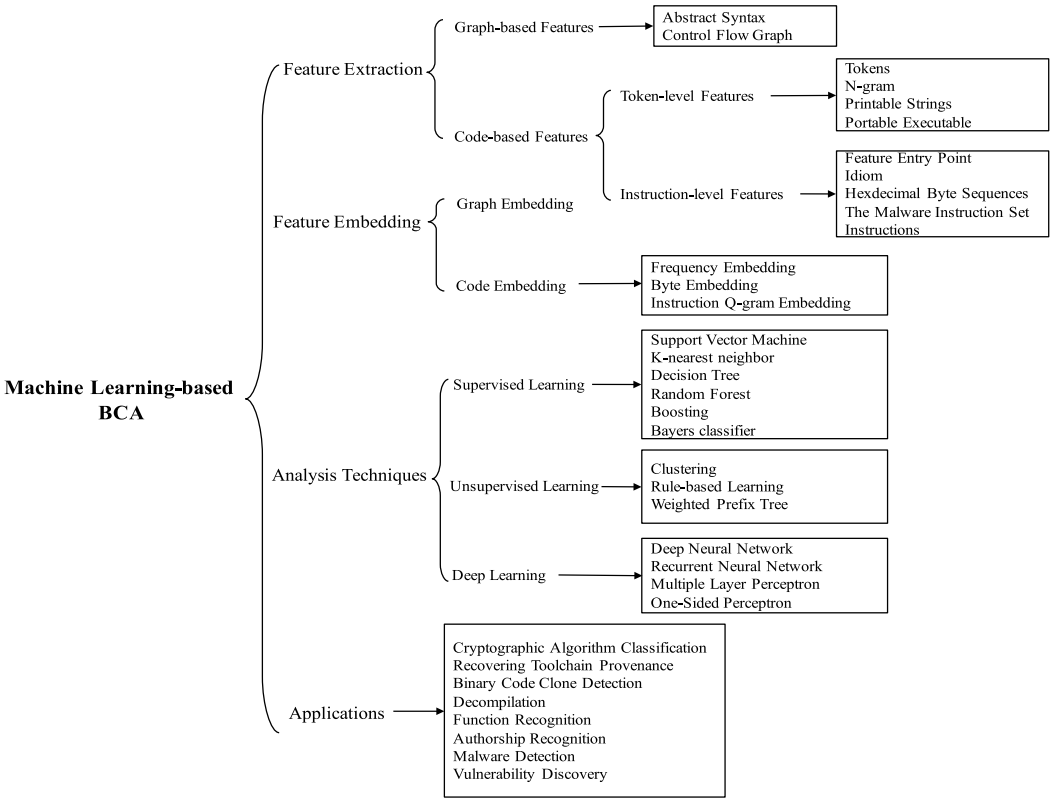

FIGURE 3. The taxonomy of machine learning-based BCA framework.

1) FEATURE EXTRACTION

Different from applying machine learning in NLP tasks, optimized program binary codes contain a huge vocabulary block of codes (e.g. basic blocks) that have complex relationships and less redundancy in terms of syntax and semantic information compared to human languages or even the original program source codes. The first requirement of machine learning-based BCA is to extract features that represent binary code. Here, we further divide the features extracted from program binaries into different categories.

1. *Graph-Based Feature:* Program Binaries can be represented as a program flow graph, such as control flow graph (CFG). We can use such graph to extract graph-based features [43], [47], [65], [136].
2. *Code-Based Feature:* Here, we extract the features directly from raw binary code. We summarize code-based feature into two different levels: token-level and instruction-level.

2.1 *Token-Level Features:* Tokens (e.g., words, characters, or symbols) from binary code to extract tokens-based feature using decompiler or binary disassembly tools, such as IDA Pro [102], OllyDbg [148] and capstone [90].

2.2 *Instruction-Level Features:* Every machine-level instruction in binary executables is a combination of tokens, such as memory references, registers, and immediate values. Such instructions sequence can also be used as program features for analyzing program bugs, information flow and so on [95], [126].

2) FEATURE EMBEDDING

The extracted raw features from binaries cannot be fed into machine learning modules directly since machine learning needs numerical data as inputs (e.g. vectors). Thus, a general phase is to transfer features to feature vectors or some formal rules, this phase is commonly named as feature embedding.

1. *Graph Embedding:* Graph embedding networks have been proposed for classification domains such as molecule classification [35], which we can covert graph-based features into graphs.
2. *Code Embedding:* Based on existing sequence-to-sequence models (e.g., Recursive Neural Network), These and related models are well-suited to tasks that have tokens or instructions as inputs and embed them into vector space [72], [103], [120].

3) ANALYSIS TECHNIQUES

After feature extraction and feature embedding, we need to choose a suitable machine learning algorithm for further program analysis. Existing analysis techniques can be classified into three categories: supervised learning, unsupervised learning, and deep learning. We will discuss each of them in detail in Section V.

4) APPLICATIONS

BCA for malware detection is a widely developed area of research. Rieck *et al.* [103] introduced Malware behavior analysis based on machine learning. Liangboonprakong and Sornil [83] create a method to classify Malware families using Machine learning. Rosenblum *et al.* [105] recover toolchain provenance to record the compilation-related information. discovRE [43] and Genius [47] develop tools that can identify bugs automatically via clustering the already known vulnerabilities.

The remainder of this article is organized as follows. In Section III and Section IV, we discuss the overall principles and strategies of feature extraction and feature embeddings. Sections V address the commonly used machine learning models that are deployed in binary code analysis and they will be further compared, while Section VI discusses the application of machine learning based code analysis for real-world problems. Section VII discusses how recent advances in other areas could be applied to enhance binary code analysis. Concluding remarks are presented in Section VIII.

III. FEATURE EXTRACTION

In this section, we list the various types of features that can be extracted from binaries. In general, the goal of feature extraction is to automatically link binary code patterns mined at the lexical level with patterns mined at the syntactic level.

A. GRAPH-BASED FEATURES

The Program flow graph (e.g., control flow graph, data dependency graph) is the common feature used in various approaches of BCA. Especially for the cross-platform bug search problem, the program flow graph and the basic block margins typically remain equivalent (or at least similar) in cross-compiled code. Thus, such graph-based features are adaptive by design and with high efficiency for large-scale BCA applications.

1) ABSTRACT SYNTAX TREE

Abstract Syntax Tree (AST) is typically used by compilers to represent the structure of program code and to analyze the dependencies between variables and statements. Many works adopt AST in source code level syntax extraction analysis [20], [34], [72], [76], [93].

AST can also be used in machine learning-based BCA. For instances, a function recognition tool, FID [130], first translates each instruction within a basic block into assignment formulas which can represent the data flow exchanging and the semantics of each basic block. Then, each data movement between registers and/or memory (assignment) will be translated into a syntax tree, which can be further converted into a numerical vector by calculating the maximum levels of nested parentheses and the maximum depth of an AST as two syntactic features.

Example: As shown in Figure 4, (A) shows a function entry point basic block and its corresponding data *movement* operations (register/memory copy, assignments and computations), while (B) shows memory access dependency behavior between caller's and callee's basic blocks.

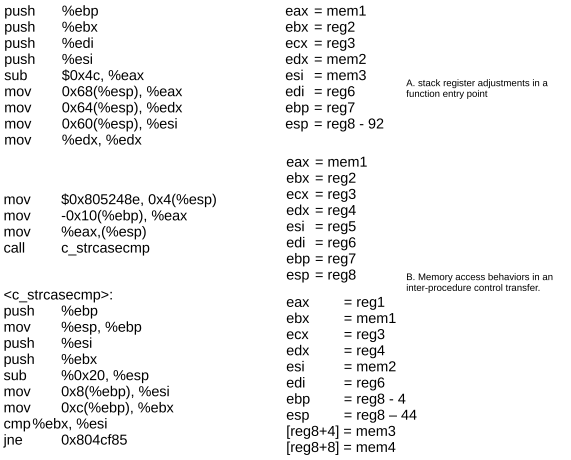

FIGURE 4. Instruction and its corresponding assignment formulas [130].

2) CONTROL FLOW GRAPH

The concept of Control Flow Graph (CFG) is first introduced by Allen [4]. CFG represents a graph of all possible execution paths that might be traversed through a program during its execution. To construct a basic static binary CFG, function entry and exit point should be found first. In a function, a sequence of consecutively executing instructions (defined as nodes) and control flow transfers between such sequences via jumps, function calls and returns (defined as edges). A binary CFG is constructed to capture the relationship between nodes and edges in the underlying binary program. It is a useful method to reflect internal relations between basic blocks. For this purpose, plenty of works have discussed CFG extraction from binary codes as program features, such as Theiling [124], Kinder *et al.* [75], Kruege *et al.* [77], Yadegar *et al.* [141].

In machine learning-based BCA, many works have adopted CFG features as inputs to machine learning models [3], [21], [53]. For instance, discovRE [43] uses CFG to reflect the structural similarity between two static functions in program binaries which will be further discussed in Section IV; Caliskan-Islam *et al.* [20] disassemble the executable binary and recover the CFG as features to extract the abstract syntax trees of decompiled source code for code authorship recognition. We give a simple example of how to extract CFG features utilizing simple NLP model (N-gram model).

Example: Assuming a static binary function CFG shown in Figure 5. The corresponding control-flow features are generated using the N-gram model. For instance, CFG bigrams (while $N = 2$) extract 2 adjacent basic blocks as control-flow features.

To reflect the binary code behaviors at the instruction-level, graphlets [99] (small, non-isomorphic subgraphs of the CFG) is introduced to analyze instruction patterns. It was first introduced for BCA domain by Rosenblum *et al.* [105], [107].

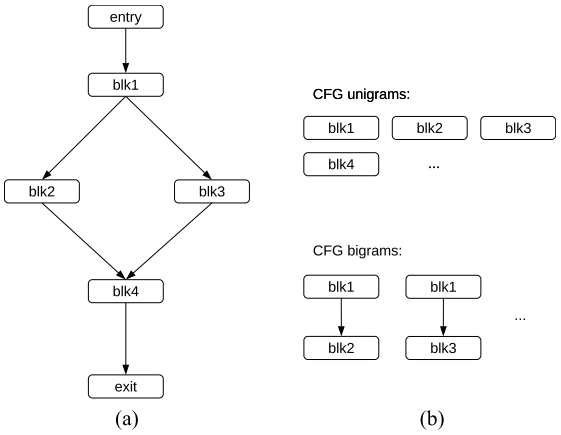

FIGURE 5. Control flow features [20].

TABLE 1. List of attributes used in ACFG.

Type	Attribute name
Block-level attributes	String Constants Numeric Constants # of Arithmetic Instructions # of Calls # of Transfer Instructions # of Instructions
Inter-block attributes	# of offsprings Betweenness

Rosenblum *et al.* [107] use graphlet features to represent details of the program structure. In this work, graphlets are three-node subgraphs of CFG. Such graphlet features can be used to capture the layout of particular classes of instructions. Simultaneously, it can also be used to represent the program structure (the local control flow). We illustrate graphlet features using an example as follows.

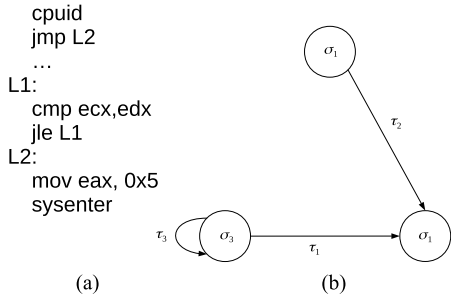

FIGURE 6. Example code and its corresponding grapglet [107].

Example: As shown in Figure 6, A is a binary code snippet with three basic blocks and B is the corresponding graphlet. $\sigma_{i=\{1,2,3\}}$ are the three different types of nodes in graphlet respectively. For example, σ_1 represents a block which contains privileged system instructions. Since there are two basic blocks with privileged instructions (*cpuid* and *systenter* in the code snippet), there are two σ_1 nodes in this case. $\tau_{i=\{1,2,3\}}$ indicates the control flow edge type between basic blocks. For instance, τ_1 is its fall through edge and τ_3 is corresponding to the *jle* conditional branch.

Besides of traditional CFG, there is a similar graph named as Attributed Control Flow Graph (ACFG) to represent a static binary function (mostly used for the tasks of vulnerability discovery and cross-platform code similarity detection). In an ACFG, each vertex is a basic block labeled with a set of attributes. Table 1 lists the commonly used attributes in ACFG. As we can see, the number of string, function calls, control transfer instructions, and arithmetic instructions are extracted as features for further analysis. Those ACFG generated from binaries will be further embedded into a hash table and similarities are compared. Genius [47] proposes ACFG as raw features to compare the code similarity between static binary functions. Xu *et al.* [136] also adopt this ACFG construction technique to extract raw features but with a different feature embedding method. We discuss more details about how to convert ACFG to embeddings in Section IV-A.

Example: Figure 7 illustrates an ACFG for a function in OpenSSL containing the Heartbleed vulnerability. To generate the ACFG for a binary function, we first need to extract its control flow graph (as showing on the left hand of the figure), along with occurrence of each attribute for each basic block in the graph, and store them as the features associated with the basic block (the generated ACFG is showing on the right hand of the figure). Each ACFG node can be converted into an N-dimension vector through counting the occurrence of N attributes (In this example, $N = 8$). According to the types of instructions, the first block can be represented as [0, 1, 10, 1, 11, 0, 11, 0.296] (e.g., There is only one function call instruction, thus, the fourth dimension of the vector equals to one. Also, the last dimension represents the Betweenness, which is a centrality measure of a vertex within a graph.)

In summary, CFG-based features mainly are used to show structural characteristics of binary codes with high-level analysis granularity (e.g. control flow dependency or data dependency).

B. CODE-BASED FEATURES

1) TOKENS

Different from program flow graph, which needs to be recovered and constructed from binaries through control or dependency analysis, a sequence of tokens (e.g., words, characters, or symbols) can be easily extracted and contain enough information to represent a code's syntax and structure. It is also easy to transform token sequences to other types of features and it is efficient for large-scale programs analysis and scalable to large software systems [20], [81], [98].

Based on the appropriate level of granularity, the binary code sequence can be divided into several pieces with the same length token sequences. Katz *et al.* [72] propose token-based binary to source code translation framework by tokenizing binary code byte by byte to identify identifier, a keyword or a constant corresponding to the source program in the high-level language such as C. Sæbjørnsen *et al.* [110], Xue *et al.* [139], [140] and Byteweight [9] adopt token-based methods for analysis as well. In the following sub-sections,

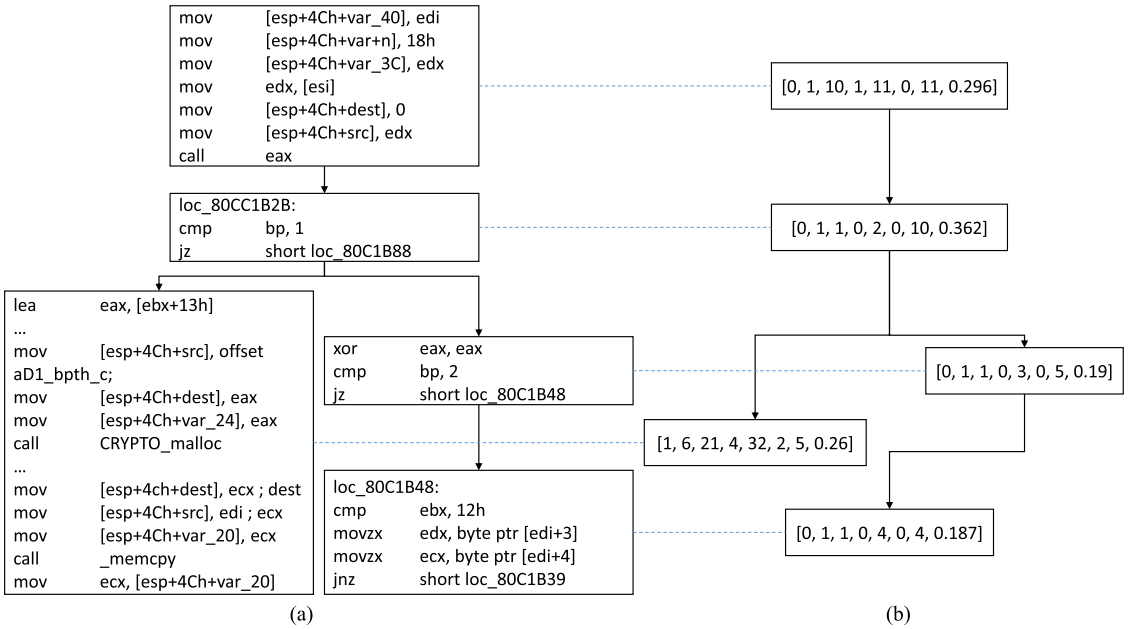

FIGURE 7. Partial CFG of function *dtls_process_heartbeat* and its corresponding ACFG [136].

TABLE 2. N-gram-malware matrix [83].

malware file	Terms(N-Gram)					
mw1	t1	t2	t3	t9	...	...
mw2	t3	t4	t6	t8	...	...
mw3	t3	t4	t6	t7	...	...
mw4	t3	t4	t6	t8	...	...
mw5	t1	t2	t4	t6	...	...
mw6	t1	t2	t4	t9	...	...
mw7	t2	t5	t6	t7	...	...
mw8	t2	t5	t6	t7	...	...
mw9	t3	t4	t5	t8	...	...
...	...	...	...	...	...	...
mwn	ti	...	...	...	...	tk

we present various token-based features that have been used in BCA domain.

a: N-GRAM

N-gram is a contiguous sequence of N items extracted from the given samples. For program binary analysis, N-gram based features are defined as a sequential pattern where an individual sample can be identified as binary instructions, file names, function arguments and so on.

Liangboonprakong and Sornil [83] proposes an N-gram based feature extracted from the binary form of Malware.

Example: Table 2 uses n=1,2,3,4 to generate n-gram slices and *t* is a set of n-gram terms.

Shijo and Salim [120] propose an integrated static and dynamic analysis framework for vulnerable binary code detection. It first discovers call sequences in the application programming interface (API) via the cuckoo malware analyzer [96], which is a sandbox used to outline the malware behavior. It then uses N-gram based method to analyze API call sequences called API-call-grams. Finally, it generates feature vectors by counting the frequency of printable strings and using N-grams with the top N of the highest frequency of printable strings. Rosenblum *et al.* [107] introduce the byte N-gram method to extract short strings from instruction, such as a specific type of instruction and memory access pattern behavior.

b: PORTABLE EXECUTABLE

The Portable Executable (PE) is a file format for binary files that encapsulates important information for program loader to manage executable binary code, such as the dynamic libraries, API exported and so on. This executable-specific information in PE format's headers can be extracted as features to represent a binary file.

Schultz *et al.* [116] extracted file size, the name of dynamic link libraries (DLLs) which are shared libraries in Windows OS, the names of functions within a DLL and relocation table from PE file's header using libBFD, a library of GUN's Bin-Utils. Saxe and Berlin [112] extracted the import address table into a 256-integrate-array and convert text information in numerical PE fields into 256-length-array from malware. Based on these malware features and group information, malware can be classified into different families.

We note that the Portable executable feature is a unique feature for certain executable files, especially for files in Windows platforms.

c: PRINTABLE STRINGS

Printable strings are un-encoded strings preserved after compiling the source code into binaries. Many works [68], [69] have proven that the printable strings are one of the most

useful features that can be obtained from binary executables. It can be relatively easily extracted from binary files, such as "*Openfile*", "*GetError*" or "*CopyMemory*". The printable strings extracted from binary files are sorted according to the frequency of occurrence within a file. We further eliminate the printable strings whose frequency below a particular threshold. Here we give an example.

Example: Consider we have three binary files (one benign and two malware) correspondingly. After feature extraction and processing, we assume the printable strings are included in each binary file as follows:

File1 (benign): {GetProcessWindowStation, FindFirstFile, GetLongPathName, HeapReAlloc}

File2 (malware): {FindFirstFile, GetLongPathName, GetProcessHeap, GetLastError}

File3 (malware): {GetLastError, FindFirstFile, GetProcAddress}

Then we count the frequency of these printable strings. Based on the analysis results from Islam *et al.* [69]. they found 10% feature reduction rate gives the optimum result, we say it threshold value (integer value). In this example, we have total of 11 printable strings, thus the threshold can be set up as 2 if we calculate the upper limit of $\lceil 11 \times 10\% \rceil$ (note that this threshold value can be an arbitrary value depending on users as long as it is larger than 20% feature reduction rate), then the features selected will be *{FindFirstFile, GetLongPathName and, GetLastError}*:

Printable strings	Frequency
FindFirstFile	3
GetLongPathName	3
GetLastError	2
GetProcessWindowStation	1
HeapReAlloc	1
...	...

2) INSTRUCTION-LEVEL FEATURES

To further lift tokens-based features to a higher level representation, we can also use instruction-based features that are a combination of tokens, such as memory references, registers, and immediate values. We then list several instruction-based features in this section.

a: FUNCTION ENTRY POINTS

The function or procedure, identified in the static binary code, is a collection of basic blocks with one entry point (i.e., the next instruction after a call instruction) and possibly multiple exit points (i.e., a return or interrupt instruction). The entry point of a function is named as the Function Entry Point (FEP). Several tools are able to manually identify the FEP, such as Dyninst [63] and IDA Pro [102].

b: IDIOM

To represent the order of a sequence of binary instructions, the idiom feature is introduced into the features extraction process. An idiom is a short instruction sequence template including undecided instructions inside, which is similar to N-grams based feature with optional single-instruction wildcards. Any short instruction sequences satisfying the idiom template are considered as idiom features. For instance, Rosenblum *et al.* [108] use idiom feature to extract compiler provenance without the assistance of machine learning techniques. There are two types of idioms which are prefix idiom and entry idiom. On the one hand, an entry idiom reflects FEP's instructions and its offset immediately.

Example: For instance, an idiom

$$u_1 = (\ push\ ebp \mid * \mid mov\ esp, ebp\ ) \tag{1}$$

where $*$ can be any type of instructions. An instruction sequence matches this idiom will be extracted to capture patterns indicative of compiler provenance, such as a sequence of instruction (*push ebp* | *push eax* | *mov esp*, *ebp*) is a match and will be identified as idiom features.

On the other hand, a prefix idiom places the offset at the beginning. For example, an instruction sequence which will immediately precede $(ret|int3)$ will be:

$$u_2 = (PRE : ret|int3) \tag{2}$$

c: THE MALWARE INSTRUCTION SET

The Malware Instruction Set (MIST) [125] is a special representation of vulnerable program behavior. The MIST instruction can be translated from arbitrary binary instructions. In contrast to traditional textual or XML-based instruction formats, the MIST of a malware binary code is described as a sequence of instructions, where encode individual system calls obtained from the execution trace of a malware program. Figure 8 shows the structure of a MIST instruction. In the MIST instruction, *CATEGORY OPERATION* reflects the system call, while *ARGBLOCK* represents arguments of system calls. The system call arguments are arranged in blocks in different levels and these levels are the MIST levels. The MIST level divides a MIST into several parts which higher levels contain attributes with higher variability and lower levels are more constant. Rieck *et al.* [103] use MIST to search numeric identifiers representing system calls and arguments using the program monitoring tool CWSandbox [134]. Similarly, MIST is also used in Firdausi *et al.* [49] as features for analysis.

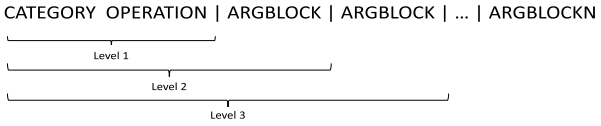

FIGURE 8. Schematic depiction of a MIST instruction [125].

Example: Given a CWsandbox original XML move file operation of system call as:

<move_file srcfile = "c:\foo.exe" dstfile="c:\windows\system32\kernel32.dll" filetype="file" creationdistribution = "CREATE_NEW">

TABLE 3. MIST categories and encoding as well as the number of contained unique operations within each category [125].

Category		# syscalls	Category		# syscalls
01	Windows COM	4	0B	Windows Services	11
02	DLL Handling	3	0C	System	2
03	Filesystem	14	0D	Systeminfo	7
04	ICMP	1	0E	Thread	3
05	Inifile	5	0F	User	8
06	Internet Helper	5	10	Virtual Memory	5
07	Mutex	2	11	Window	5
08	Network	6	12	Winsock	13
09	Registry	9	13	Protected Storage	9
0A	Process	7	14	Windows Hooks	1

According to table 3, the corresponding MIST will have the category filesystem (03) and the system call move file (05). The other part of MIST instruction will be arranged as path names and file extensions in level 2 and the base names of the files in level 3:

$$\underbrace{03\ 05}_{move\_file}\ |\underbrace{01\ 000000\ 01}_{createflages}\ \underbrace{00006ce5}_{"exe"}\ \underbrace{000066fc}_{"c:\"}\ \underbrace{00006b2c}_{"dll"}$$
$$\underbrace{002e6d6c}_{"c:\w..."}\ |\underbrace{00006d5f}_{"foo"}\ \underbrace{071c94bc}_{"kernel"}$$

d: HEXADECIMAL BYTE SEQUENCE

To deal with the binary files that are not in PE format, Schultz *et al.* [116] propose Hexadecimal features that convert original binary files into hexadecimal files via a raw byte translation tool Hexdump [92]. The result will be hexadecimal byte sequences that can represent machine code instructions.

Example: Given a sequence of machine code instructions, Hexdump can translate them into hexadecimal files. As shown in the following example, each line of hexadecimal numbers corresponds to a short sequence of machine code instructions.

```
1f0e 0eba b400 cd09 b821 4c01 21cd 6854
7369 7020 6f72 7267 6d61 7220 7165 6975
6572 2073 694d 7263 736f 666f 2074 6957
646e 776f 2e73 0a0d 0024 0000 0000 0000
454e 3c05 026c 0009 0000 0000 0302 0004
0400 2800 3924 0001 0000 0004 0004 0006
000c 0040 0060 021e 0238 0244 02f5 0000
0001 0004 0000 0802 0032 1304 0000 030a
```

e: OTHERS

We have listed several well-used features in previous sections, there are still some works that are using other types of special features that cannot be classified into the same category.

Hosfelt [65] use PIN tool, a dynamic binary instrumentation framework [89], to trace each instruction in binaries and counts the number of execution for each instruction. Then, the PIN tool will classify each instruction into different categories, such as *Nop*, *Syscall* and Binary. Hosfelt [65] also use PIN to detect and count the number of loops in programs.

As introduced in section III-A.1, FID [130] translates binary instructions within a basic block into assignment formulas. On the one hand, operations, constants, and token related features will be extracted from these assignment formulas as lexical features shown in table 4. On the other hand, stack registers (formulas of register *esp* and *ebp*) are extracted as stack features.

TABLE 4. Lexical Features extracted from assignment formulas in FID [130].

Feature	Definition
numOperator/length	the number of occurrences of operators divided by the formula length of characters
numToken/length	the number of tokens divided by the formula length of characters
numConstant/length	the number of constants divided by the formula length of characters
decOperator/length	the number of subtraction operators divided by the formula length of characters
decNum/length	the number of "small operands" in subtraction operations divided by the formula length of characters

IV. FEATURE EMBEDDING

As extracted raw features from program binaries cannot be fed into machine learning modules directly since machine learning needs numerical data as inputs (e.g. vectors), we would like to learn an indexable feature representation from the feature extraction that we need to encode (i.e., embed) a feature representation into an embedding (e.g., numeric vectors). In this section, we present the common feature embedding approaches corresponding to different feature representations.

A. GRAPH EMBEDDING

A control flow graph describes the flow of basic blocks in the instruction level. However, such basic blocks or instructions cannot be directly used as input for a machine learning model. Thus, some numeric and non-numeric features have to be normalized into vectors of some length. discovRE [43] introduces a concept called Basic Block Distance d_{BB} to detect the similarity of functions in binaries. Assuming we have a vulnerable function code sample, we can use Basic Block Distance to identify similar functions in other binaries across platforms, compilers, and optimizations.

It first represents a static function f's CFG, identified from program binary, with each node labeled with features F, that can be its topological order, string references, numeric constants, and robust features (e.g., No.of function calls, No.of store/load instructions and so on)in the function. Based on these representing features, the basic block distance d_{BB} is defined as:

$$d_{BB}(c_{if}, c_{ig}) = \frac{\sum \alpha_i |c_{if} - c_{ig}|}{\sum \alpha_i max(c_{if}, c_{ig})} \tag{3}$$

with α_i is the weight metric with a range as [0...100] to achive $max(d_{BB}(f_i, g_j) - d_{BB}(f_i, f_j))$, and c_{if} and c_{ig} represent numeric feature $i \in F$ of function f and g where $f \neq g$. In another way, we want to maximize the difference between the same and different functions.

Besides, Genius [47] calculates the similarity of two static binary functions using ACFGs. It first defines a ACFG as:

$$G = \langle V, E, \phi \rangle \tag{4}$$

where V is a set of basic blocks, E is a set of edges and $\phi : V \rightarrow \sum$ is a labeling function that maps a basic block in V to a set of attributes in $\sum$. Similar definitions such as $g = \langle v, \varepsilon \rangle$ are proposed in Xu *et al.* [136]. Based on G_1, G_2, the bipartite graph which combines two graphs can be represented as $G_{bp} = (\hat{V}, \hat{E})$, where

$$\hat{V} = V(G_1 \cup G_2) \tag{5}$$

and

$$\hat{E} = \{\hat{e_k} = (v_i, v_j | v_j \in V(G_1) \wedge v_j \in V(G_2))\}. \tag{6}$$

where $\hat{e_k} = (v_i, v_j)$ is an edge from v_i to v_j. So, based on the distance (equation 7), the similarity of two ACFGs (g_1, g_2), denoted as $k(g_1, g_2)$, can be described as:

$$k(g_1, g_2) = 1 - \frac{d_{BB}(g_1, g_2)}{max(d_{BB}(g_1, \Phi), d_{BB}(\Phi, g_2))} \tag{7}$$

where Φ is an empty ACFG whose nodes has an empty feature vector, and the size is set to that of the corresponding compared graph (g_1 and g_2).

B. CODE EMBEDDING

1) FREQUENCY EMBEDDING

Katz *et al.* [72] introduce a decompilation framework to reverse binaries to the source code. It uses frequency embedding as the translation from short snippets of higher-level code (C source code) to corresponding bytes of binary code in a compiled version of the program. A popularity ranking (the top N most frequent tokens) is generated to represent a token as the input for the Recurrent Neural Network.

Additionally, Liangboonprakong and Sornil [83] also propose sequential pattern extraction [1] and pattern statistic to statistics the frequency of pattern extracted from raw features. Given the result of n-gram extraction $T = \{t_1, t_2, \ldots, t_m\}$, a sequential pattern can be represented as an ordered list of terms as $S = \langle t_1, \ldots, t_r \rangle : (t_i \in T)$. So, n-gram patterns will be a vector d_i and the embedding result will be Y.

$$\vec{d_i} = \langle (s_{i_1}, f_{j_1}), (s_{i_2}, f_{j_2}), \ldots, (s_{i_m}, f_{j_m}) \rangle \tag{8}$$

$$Y = \{\vec{d_1}, \vec{d_2}, .., \vec{d_n}\} \tag{9}$$

where s_i is a pattern (tokens), f_j is the frequency in d_i. For each vector, frequency-inverse document frequency (TF-IDF) weighting filtrates out the common n-gram sequential. Finally, it minimizes valued feature sets with sequential floating forward selection (SFFS) procedure [100]. Similarly, [120] lists API-call-grams and corresponding classes sorted by frequency and a vector for each feature is created.

2) BYTE EMBEDDING

Some machine learning-based BCA frameworks directly use raw bytes extracted from binary programs as features. However, raw bytes are not acceptable for some machine learning models. In Shin *et al.* [121], the recurrent neural network takes bytes as inputs. A byte cannot directly be input to RNN so that each byte has to be translated into a form that the RNN accepts. Shin *et al.* [121] introduce a method called the one-hot encoding. In this method, each byte will be encoded as a R^{256} vector and there is only a '1' as the identification of a byte and the other are 0s in each vector.

Example: NUL (0) and NOP in x86 (144) can be represented as equations 10 and 11.

$$[1 \underbrace{0 \ldots 0}_{255\ elements}] \tag{10}$$

$$[\underbrace{0 \ldots 0}_{144\ elements} 1 \underbrace{0 \ldots 0}_{111\ elements}] \tag{11}$$

3) Q-GRAMS EMBEDDING

Inspired by NLP techniques and host-based intrusion detection [37], [50], [80], [111], Rieck *et al.* [103] have developed a unique feature embedding approach of instruction Q-grams which is similar to N-grams. A window will slide over a MIST instructions sequence x and a sequence with the length of Q called instruction Q-gram will be extracted. The set of Q-grams S can be represented as:

$$S = \{(a_1, \ldots, a_q | a_i \in A \ with \ 1 \leq i \leq Q)\} \tag{12}$$

where A is the set of all possible instructions. Within the set S, the extraction result is translated into $|S|$-dimensional vector. Then, the embedding function can be represented as:

$$\phi = (\phi(x))_{s \in S} \tag{13}$$

where

$$\phi_s(x) = \begin{cases} 1 & if \ report \ x \ contains \ q-grams \ s, \\ 0 & otherwise \end{cases} \tag{14}$$

Example: If there is a report x of malware behavior corresponding to a simple sequence of instructions $x = \{1|A\ 2|A\ 1|A\ 2|A\}$, where $N|A$ is a MIST instruction A with MIST level N. It can be formed by two simplified instructions: $A = \{1|A,\ 2|A\}$. Consider the sliding window threshold is set up as Q=2, there are only two possible instruction sequences as $'1|A\ 2|A'$ or $'2|A\ 1|A'$. Then the embedding function is:

$$\phi('1|A\ 2|A\ 1|A\ 2|A') \rightarrow \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{matrix} '1|A\ 1|A' \\ '1|A\ 2|A' \\ '2|A\ 1|A' \\ '2|A\ 2|A' \end{matrix} \tag{15}$$

To reduce the bias, the embedding result will be further normalized as:

$$\hat{\phi}(x) = \frac{\phi(x)}{||\phi(x)||} \tag{16}$$

TABLE 5. Summary of supervised learning in machine learning-based BCA.

Supervised Learning	Title	Purpose
Support Vector Machine	Liangboonprakong et al. [83]; Shijo et al. [120]; Hosfelt et al. [65]; Rosenblum et al. [105]	Classify String Tokens; Classify Malware and Benign; Classify Cryptographic Algorithms; Create Model for Toolchain Provenance
K-Nearest Neighbors	discovRE [43]	Detect Code clone
Bayes Classifier	Hosfelt et al. [65]; Schultz et al. [116]	Classify Cryptographic Algorithms; Detect Malware
Decision Tree	Hosfelt et al. [65]; liangboonprakong et al. [83]	Classify Cryptographic Algorithms; Classify Malware
Random Forest	Caliskan et al. [20]; Shijo et al. [120]	Learn De-anonymizing Pattern; Classify Malware
Boosting	FID [130]	Recognize Function in Binary Code

After normalization, the similarity between two embeddings (x and z) can be calculated as the distance $d(x, z)$:

$$d(x, z) = ||\hat{\phi}(x) - \hat{\phi}(z)|| \tag{17}$$

V. ANALYSIS TECHNIQUES

The early stage of BCA based on machine learning usually employs basic machine learning analysis techniques, such as logistics classification, linear regression and so on. With the rapid development of machine learning analysis (e.g., deep neural networks) techniques, more advanced machine learning algorithms have been used lately.

In section III and section IV, we discuss how to extract a feature and embed them into an appropriate format (e.g., numerical feature vectors) for machine learning. Here, we divide the machine learning analysis techniques used in BCA into three main types: supervised learning, unsupervised learning, and deep learning. The main difference between supervised learning and unsupervised learning is that the ground truth and prior domain knowledge are given for supervised learning. Unsupervised learning, on the other hand, does not have those conditions before its training process.

A. SUPERVISED LEARNING

In this section, we present several BCA systems using supervised learning. Table 5 shows the overview of existing works.

a: SUPPORT VECTOR MACHINE

Support Vector Machine (SVM) is a supervised leaning used as a classification generation. It was first invented as a practical method by Boser *et al.* [16], then it was further developed in Cortes and Vapnik [33]. The key ideal of SVM is trying to fairly separate a linear space into different classes. Given the input of a set of vector x_i, there will be a weight vector λ generated during training. The weight vector decides the boundary of different classes in the form of margin also defined as a kernel function $K(x, y)$ where (x, y) is a point in a feature space mapped from a hyperplane in the input space. As a significant segment in SVM, there are many kernel functions introduced to this area, such as linear kernel function, polynomial kernel function, radial-basis kernel function [113], and sigmoid kernel function. For linear kernel function, linear SVM calculate faster due to the linear function and less argument. Thus, the linear kernel function is used when the sample set is huge. Comparing with the linear kernel, the polynomial kernel has more parameters (α, d, c) than the linear kernel (c), so it is usually used to handle the classification problem with the orthogonal normalization.

Linear kernel function:

$$K(x, y) = x^T y + c \tag{18}$$

Polynomial kernel function:

$$K(x, y) = (\alpha x^T y + c)^d \tag{19}$$

Radial-basis kernel function:

$$K(x, y) = exp(-\frac{||x - y||^2}{2\lambda_2}) \tag{20}$$

Sigmoid kernel function:

$$K(x, y) = tanh(\alpha x^T y) + c \tag{21}$$

In the field of machine learning-based BCA, SVM has been applied in several works. Liangboonprakong and Sornil [83] adopt polynomial kernels SVM to classify string tokens of hexadecimals extracted from disassembling files and its frequency features with high dimensions in vector space via LIBSVM [22]. Shijo and Salim [120] use WEKA [60] machine learning tool to study its PSI features as static features, API call sequences as dynamic features to classify malware and benign. This paper adopts SVM, Random Forest as learning methods and SVM gets a better accuracy. Hosfelt [65] compare the training results of four different kernel functions: Linear, RBF, Polynomial and Sigmoid and finds that the linear kernel has the best performance to classify cryptographic algorithms in binary programs.

Besides, Rosenblum *et al.* [105] adopt linear SVM to scale all features via LIBLINEAR [45] due to its less cost and higher accuracy on recovering the details generated during transformation process through which the binary was produced.

b: K-NEAREST NEIGHBORS

K-Nearest Neighbors (K-NN) is a supervised learning classification approach. Given a sample set, K-NN identifies k samples whose distances will be shortest in the feature vector space. Basing on these k neighbors' information, the K-NN classifier can further identify input targets. discovRE [43] employs K-NN based on k-dimensional trees (k-d tree). This k-d tree is a binary search tree whose node is a k-dimensional vector and one dimension of each node is randomly chosen from the data set. When search for a point in the tree, the system will begin from the root and step down. In each step, the most match point will be chosen if it is better then the old one. This algorithm cannot handle high-dimensional data well.

c: BAYES CLASSIFIER

The Bayes model is a basic learning method to make decisions via probabilities. It classifies the groups by computing the probability of a certain feature belonging to a certain class. For instance, given features $\{x_1, \ldots, x_n\}$ of the program X, the probability of Program X belonging to class C_k for each of k possible outcomes will be:

$$P(C_k|X) = \frac{\prod_{i=1}^{n} P(x_i|C_k) * P(C_k)}{\prod_{j=1}^{n} P(x_j)} \tag{22}$$

In this case, classification model will be:

$$\hat{y} = argmax_{k\in\{1,\ldots,K\}}(P(C_k)\prod_{i=1}^{n} P(x_i|C_k)) \tag{23}$$

For example, Hosfelt [65] adopt Gaussian Bayes ($P(x|c)$ is Gaussian function) as one of the learning algorithms to represent the most likely function returning to the class C.

Besides, Schultz *et al.* [116] adopt both Bayes and Muti-naive Baye to detect malware. Similar to naive Bayes, the likelihood of Muti-Bayes of the class C given bytes X will be:

$$L_{NB}(C|X) = \prod_{i=1}^{|NB|} \frac{P_{NB_i}(C|X)}{P_{NB_i}(C)} \tag{24}$$

And classification model will be:

$$\hat{y} = max_C(P_{NB}(C) * L_{NB}(C|X)) \tag{25}$$

where NB stands for a set Naive Bayes of classifiers, $|NB|$ is the size of set, $NB_{i\in\{1,2,\ldots,|NB|\}}$ is an individual Naive Bayes classifier.

d: DECISION TREE

The Decision Tree is a tree-liked supervised learning model formed by decisions and corresponding consequences. Belson [13] is the earliest work that used the decision tree. After, some researchers have improved the decision tree algorithms, such as Freund and Mason [51] and Kearns and Mansour [73]. A decision tree is formed by nodes, arcs, and leaves which represent decisions, consequences, and results. Those three parts can reflect feature attributions, feature values and categories clearly.

In the BCA field, Hosfelt [65] use the decision tree as a supervised learning algorithm and compared it with other learning algorithms. The result shows that the decision tree has a weak performance in identifying the binary program of the cryptographic algorithm used to encode data. Besides, Liangboonprakong and Sornil [83] adopt the C4.5 decision tree [101] with tool KNIME [14] as one of learning methods to classify malware. Similar to the result in Hosfelt [65], the accuracy of malware classification of the decision tree is lower than SVM.

e: RANDOM FOREST

Random Forest is an Ensemble learning combining decision trees, so it has a better performance compared with a decision tree. It was first introduced in Ho [62] and was further developed in Dietterich [40]. In the area of machine learning based BCA, Caliskan-Islam *et al.* [20] randomly selected $(logM)+1$ features from M total features and each $(logM)+1$ features will be inputted to a tree. Shijo and Salim [120] also adopt random forest as their classifier for malware detection.

f: BOOSTING

In order to improve the performance of a certain machine learning algorithm, boosting is proposed. The boosting combine several weak learning algorithms together and create a strong learner. FID [130] proposes the majority voting on the top of multiple learning. It combines linear SVM and two Boosting which are AdaBoost and GradientBoosting.

Example: RIPPER sets five types of rules after learning thousands of malicious binaries and benign binaries. according to four principles:

1. *Does the malware have a GUI?*
2. *Does the executable perform malicious functions?*
3. *Does the executable compromise system security?*
4. *Does this executable delete a file?*

Then to check if a binary is malicious, we need to check the following classification table:

Has a GUI?	Malicious Function?	Compromise Security?	Deletes Files?	Malicious executable?
√	√	√	×	√
×	√	√	√	√
√	×	×	√	×
√	√	√	√	√

The heuristic for stop condition defined as a total description length will be computed with the rule set and examples. The total description length will stop adding rules when there is no positive example or it is more than 64 bits and larger than the smallest description length calculated from previous rules.

TABLE 6. Summary of unsupervised learning in machine learning-based BCA.

Unsupervised Learning		Title	Purpose
Clustering	K-means	Hosfelt et al. [65]; Rosenblum et al. [107]	Identify Instruction Features; Classify Authorship
	Affinity Propagation	Clone-Hunter [139]	Detect Code Clones in Binaries
	Hierarchical Clustering	Rieck et al. [103]	Classify Behavior
	Other	Genius [47]	Generate Cluster of ACFGs
Rule-based		Schultz et al. [116]	Identify Malicious Executables Features
Weighted Prefix Tree		Byteweight [9]	Identify Function Entry Points in Binary Program

B. UNSUPERVISED LEARNING

Different from supervised learning, unsupervised learning does not have the ground truth and prior domain knowledge before its training process. In this section, we then give the common unsupervised learnings that have been employed in BCA, and the overview is shown in Table 6.

1) CLUSTERING

Clustering is one of the most common unsupervised learning approaches used in general classification tasks. Given a considerable amount of unmarked feature embeddings, clustering can automatically group them into different homogeneous groups. Several existing works adopt clustering algorithms to gain homogeneous groups and prepare for further classification.

a: K-MEANS CLUSTERING

K-means Clustering is a prototype-based clustering that was first introduced by Lloyd [88]. Then, it was further developed and improved by many other researchers, such as Algorithm AS 136 [61], Alsabti *et al.* [6], filtering algorithm [71], etc.

Hosfelt [65] uses K-means clustering as an unsupervised learning module to identify instruction features, such as the number of times an individual instruction is executed, the type of instructions (e.g., NOP, SYSCALL) and the number of loops executed, from the implementation binaries of cryptographic algorithm for malware detection purpose. The goal of this work is to utilize K-means clustering to detect cryptoviruses in small (single purpose) programs.

Besides, Rosenblum [107] uses K-means clustering for authorship classification. Given program sets and label sets, we can cluster them with a distance metric between two feature vectors. During this process, Large Margin Nearest Neighbors (LMNN) [132] is proposed to learn the distance metric.

K-means clustering is a simple and efficient clustering algorithm. However, prior knowledge of the cluster numbers must be known before clustering and k-means clustering also cannot handle non-globular clusters well.

b: AFFINITY PROPAGATION CLUSTERING

Different from K-means clustering, Affinity Propagation clustering (AP clustering) is able to determine the number of clusters among the data points without any a prior knowledge [15]. AP clustering performs the ''message passing'' procedure to update the relationships between data points and candidate exemplars (a.k.a cluster centers). There are three matrices used in AP clusterings, where S is a similarity matrix, $R(i,k)$ and $A(i,k)$ are for *Responsibility matrix* and *Availability matrix* respectively. Assuming we have two distinct data points X_i and X_j, S is represented as the negated value of the squared Euclidean distance.

On the other hand, *Responsibility matrix* and *Availability matrix* are being updated in each iteration. In particular, $R(i,k)$ measures how well data point X_k is suited to serve as a candidate cluster exemplar for the point X_i, while $A(i,k)$ reflects how appropriate it is for X_i to choose X_k as its cluster exemplars. The more details of AP clustering can be found in reference [52].

In particular, Clone-Hunter [139] utilizes AP clustering to detect code clones in binaries. Given a binary code, Clone-Hunter first normalizes the assembly code into intermediate representations in order to remove instruction-specific details, such as register names and memory addresses. The feature vectors are then generated from each normalized instruction sequence, embed them into vector space and use AP clustering algorithms to find binary code clones.

c: HIERARCHICAL CLUSTERING

Hierarchical Clustering clusters data sets from different hierarchies, hence, it can handle non-globular clusters. In the machine learning based BCA, Rieck *et al.* [103] proposes a framework using the Hierarchical Clustering [42] for the behavior classification problem following by Bayer *et al.* [11]. It firstly uses a linear-time algorithm [55] to extract prototypes whose distance is smaller than a constant from feature vectors. Based on these prototypes, Clustering will then update the distance with the minimum Manhattan

TABLE 7. Summary of deep learning in machine learning-based BCA.

Deep Learning	Title	Purpose
Recurrent neural network	Clone-Slicer [140]; Shin et al. [121]	Detect Code Clone; Recognize Function in Binary Program
Deep Neural Network	Xu et al. [136]	Embed ACFGs
Multiple Layer Perceptron	Liangboonprakong et al. [83]	Classify Malware Families
One-Sided Perceptron	Gavriluţ et al. [54]	Detect Vulnerabilities

Distance between prototypes and finally decides the group according to the nearest prototype.

d: OTHER TYPES OF CLUSTERINGS

Genius [47] adopts a kernelized special clustering [94] where the input is a kernel matrix to generate a optimal cluster. Given a kernel matrix formed by a set of ACFG similarity score which has been discussed in IV-A, the output will be an optimal cluster of ACFGs.

2) RULE-BASED LEARNING

Schultz *et al.* [116] employ a special rule-based learning system [31] which is a set-valued extension of a rule inductive algorithm, the Repeated Incremental Pruning to Producing Error Reduction (RIPPER) [30]. For example, Schultz *et al.* [116] use this algorithm to learn malicious executables features (PE headers information) to construct a detection model to identify malicious executables.

3) WEIGHTED PREFIX TREE

Similar to AST, a prefix tree is a data structure that each non-root node is associated with bytes or instructions for a binary code to enable efficient information retrieval. Given a static execution path containing a n instructions, it can be represented in a prefix tree path from root to a child node with tree height of n. Byteweight [9] utilizes a weighted prefix tree approach to identify function entry point in a program. It proposes a program signature learning process using prefix tree and recognizes function entry pointer by simple matching binary code fragments with the corresponding learned signatures. First, the training data is constructed as a corpus of program binaries, which contain a set of functions with known function entry point and end point. Then, it builds weight prefix tree for each function from training data and the weights are learned by the likelihood that the instruction sequence corresponding to the path from the root node to this node is a function entry point in the training data set.

Example: In a weighted prefix tree, defined in Byteweight, each non-root node specifically represents an instruction. As shown in Figure 9, for example, the weight of {*push %ebp*} is 0.8459, which means there are 84.59% of all sequences in with prefix of {*push %ebp*} were function entry point, while the other 15.41% were not.

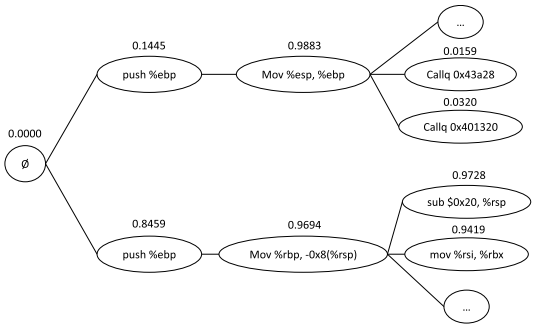

FIGURE 9. A sample of prefix tree [9].

C. DEEP LEARNING

In recent years, Artificial Neural Network (ANN), especially Deep Neural Network, has been applied to the BCA [121], and has shown better results than other methods. The advantage of Neural Network (NN) is that it can represent a binary analysis task, (e.g. using ACFG embedding to represent a binary function as we mentioned in Section III), as NN can train parameters in an end-to-end fashion so that it does not require too much prior domain knowledge. On the other hand, an NN-based BCA approach can be adaptive by design, as the input of NN can be arbitrary types of data into different application tasks. Table 7 gives the overall of current BCA frameworks based on deep learning

1) RECURRENT NEURAL NETWORK (RNN)

RNN is a type of ANN original proposed from Hopfield networks [64] in 1982. RNN has been proven as an effective approach for modeling a piece of sequential information, such as binary assembly code. Katz *et al.* [72] proposes the encoder-decoder RNN using the seq2seq model to generate a decompiler. From the token result discussed in the previous section, both source codes and binary tokens can be translated into integers according to the frequency of tokens. Then, these pairs of integers will be grouped into different buckets according to their length. RNN will take both source codes (C language) and corresponding binary code buckets as input and output a decompiler model.

Clone-Slicer [140] proposes a code clone detection framework based on RNN. It first parses assembly instructions into tokens and uses them as input to RNN to generate vector embeddings for each unique token in the lexical level. Then it

utilizes recursive auto-encoder to combine the embeddings to generate code signature in syntax level. Here, we give an example of how to use RNN to model binary assembly code.

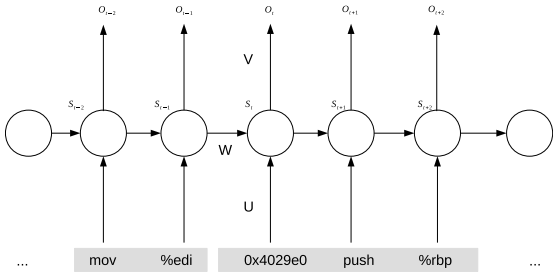

FIGURE 10. RNN learning process.

Example: As shown in Figure 10, there are 2 adjacent instructions which are [..., mov, %edi] and [0x4029e0, push, %rbp,...]. The input of each node is a one-hot vector representing the current term in the disassembly code corpus, the hidden layer stores the current state and previously calculated results. Specifically, it can be obtained using the following equation:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{26}$$

The output is probability vectors to predict the distribution of the next input. The parameters $\{U, V, W\}$ are trained using back propagation through the time (BPTT) method in an RNN network [44]. Once RNN training is complete, each term in the code corpus will have a unique embedding U for further analysis.

Similarly, Shin *et al.* [121] also adopts RNN as its learning method for function recognition and the learning process is just like Xue *et al.* [140].

2) DEEP NEURAL NETWORK (DNN)

DNN is a complex non-linear ANN with multiple hidden layers. Its first implementation is back to 1989 by LeCun *et al.* [78]. Recently, DNN has been substantially applied in different NLP tasks and aim to improve the representation power of the abstractions (e.g., language words embeddings).

As mentioned in Section III-A.2, Xu *et al.* [136] embed the whole ACFG through a DNN embedding network, then it implements a similarity detection based on the euclidean distance between the embeddings for two static functions that compiled in different architectures. We further present an example of such embedding process using DNN.

Example: As shown in Figure 11, given a set of ACFG basic blocks feature vector, the embedding network will generate a feature vector to represent a basic block and its adjacent basic block in each DNN layer. In particular, a code graph with a set of vertex v (In this figure, there are three vertex x_1, x_2 and x_3). Then embedding μ_v^{t+1} is updated at each iteration as:

$$\mu_v^{t+1} = f(x_v, \sum_{u \in v} \mu^t)$$

where f is a nonlinear function. In Figure 11, the function f is defined as:

$$tanh(W_1 x_v + \sigma \sum_{u \in v} \mu_u)$$

where x_v is a d-dimensional vector for graph node (or basic-block), $W1$ is a $d \times p$ matrix, and p is the embedding size as explained above. To make this transformation $\sigma(*)$ more refined, a n-layer neural network are used after as:

$$\sigma(l) = P_1 \times ReLU(P_2 \times ...ReLU(P_n l)))$$

where P_i is a $p \times p$ matrix, $ReLU$ is the activation function as $ReLU(x) = max\{0, x\}$.

For example, given the ACFG in Figure 7, the first layer's input will be the current vertex [0, 1, 10, 1, 11, 0, 11, 0.296] and the adjacent vertex is [0, 1, 1, 0, 2, 0, 10, 0.362]. In the second layer, the current vertex will be [0, 1, 1, 0, 2, 0, 10, 0.362] and adjacent vertex are [0, 1, 1, 0, 3, 0, 5, 0.19], [1, 6, 21, 4, 32, 2, 5, 0.26] and so forth. The output of embedding network will be a vector which represents the whole ACFG.

3) MULTIPLE LAYER PERCEPTRON (MLP)

Multi-layer perceptron (MLP), a feed-forward neural network that consists of at least three layers (an input and an output layer with one or more hidden layers). Different from conventional neural networks, each node is using a nonlinear activation function (e.g., sigmoid function) in MLP except for the input nodes. Thus, it can handle non-linear separable data due to its multi-layers and non-linear activation.

Back to Section III-B, we mentioned Liangboonprakong and Sornil [83] use N-gram-based features to classify malware families. It then adapts MLP as one of their classification models. However, the results have shown the classification accuracy is not as good as other classification models, such as SVM.

4) ONE-SIDED PERCEPTRON

One-sided perceptron is a modified version of Perceptron. This algorithm first trains data with a chosen label and data sets will be separated into two part through learned linear separator. On the one side of the separator, data will have that chosen label, while the others will have mixed labels. Gavriluţ *et al.* [54] propose one-sided perceptrons to detect vulnerabilities in binaries. One-sided perceptrons is modified from perceptrons algorithm in [104] for malware detection purpose. In Gavriluţ *et al.* [54], the Perceptron Training Subroutine, One-sided Perceptron, and Kernelized One-sided Perceptron Algorithms are integrated together by a Cascade Classifier. The Cascade Classifier is a case of ensemble learning which each classifier in Cascade Classifier will collect the previous classifiers' output and learning data itself.

VI. APPLICATIONS

In this section, some applications of machine learning based BCA will be classified. Furthermore, we elaborate and compare the techniques used in each application in detail.

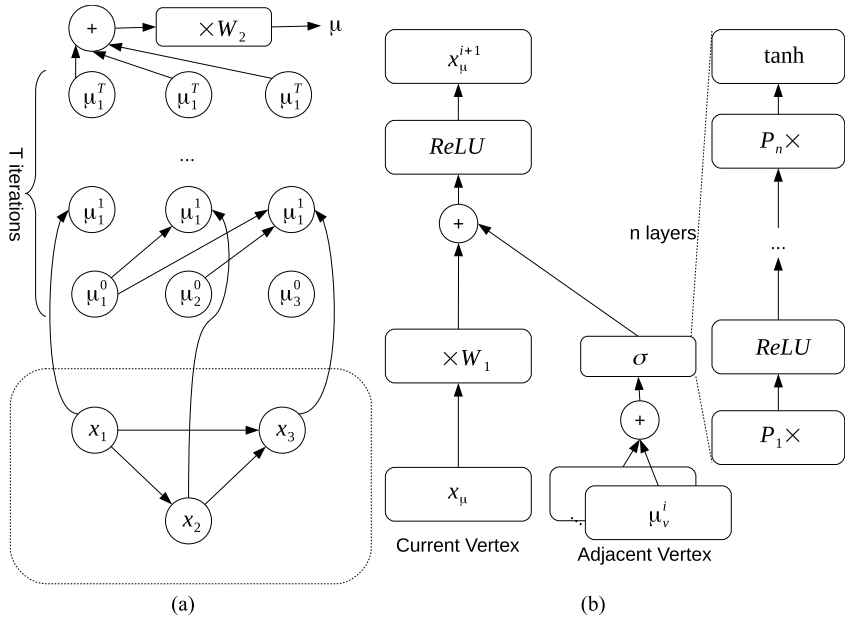

FIGURE 11. DNN embedding structure [136].

TABLE 8. Binary code clone detection.

Title	Year	Features	ML Model	Type I	Type II	Type III	Type IV	Type V
Saebjornsen et al. [110]	2009	Tokens	Clustering	√	√			
David et al. [39]	2014	CFG & Execution trace	Clustering	√	√			
Esh [38]	2016	Instructions	Logistic function	√	√	√		
discovRE. [43]	2016	CFG	K-NN	√	√	√		
Gemini [136]	2017	ACFG	Neural Network	√	√			√
Clone Hunter [139]	2018	Tokens	Clustering	√	√			
Clone-Slicer [140]	2018	Tokens	Recursive Neural Network (RNN)	√	√	√	√	

A. BINARY CODE CLONE DETECTION

Detecting similar code fragments, usually referred to as *code clones*, is an important task to understand software and detect duplicate code fragments. Many works have proposed various code clone detection frameworks for different purposes in the source code. For example, prior works make use of subsequence token matching, abstract syntax trees (ASTs) comparison or control flow graph analysis [8], [70]. Binary code clone detection is more difficult compared to detect code clones in source codes. As mentioned in the previous section, source codes leverage rich structural information such as syntax trees and variable names made available through source lines of program code comparing to binary codes. Due to this fundamental difficulty of binary code clone detection, machine learning has been adopted and proved as a sufficient approach.

Generally, there are five different clone types in terms of binaries. **Type I**: Identical code fragments that are copy/paste. **Type II**: Identical code fragments except for variations in identifiers, literals, and comments. **Type III**: Syntactically similar fragments with further modifications such as changed, added or removed statements with respect to each other. **Type IV**: Semantically equivalent code fragments that implement the same functionality. **Type V**: Cross-platform code fragments that are compiled from the same source code but in different platforms, e.g., x86, ARM, or MIPS.

Table 8 summaries the contrast of some binary code clone detection applications. We conducted experiments to analyze and compare the ability to detect different types of code clones. All experiments are performed on a 2.54 GHz Intel Xeon(R) CPU E5540 8-core server with 12 GByte of main memory. The operating system is Ubuntu 14.04 LTS. We measured the quality of code clones (in terms of clone types) that are detected from those clone detection tools. In the evaluation, we measure the types of code clones (five different types: Type I - Type V, defined in Section VI-A) that can be detected from those clone detection tools. For a fair comparison, we choose the same configuration to generate function-level code regions conducted on eight real-world software systems (in binary format) mentioned in Reference [133]. Finally, to mitigate the bias, two judges

(two authors from this paper) used a uniform set of guidelines to measure the similarity of code fragments.

As we can see, the majority of binary code clone detection frameworks are efficient to detect **Type I** and **Type II** clones. In particular, CFG based features can obviously show characteristics in the binary code and are able to detect **Type V** clones. For machine learning, discovRE [43] adopts K-NN due to faster set up time and less memory occupy compared with SVM. Xu *et al.* [136] introduce Neural Network because it can quickly retrain the data set. Mostly, deep learning techniques have been applied in the binary code clone detection and can be used to detect. **Type IV** clones. Clone-Slicer [140] first proposes a binary clone detector using RNN. The general idea is similar to use RNN for NLP tasks. Here, we give an example of how to apply deep learning to detect binary code clones.

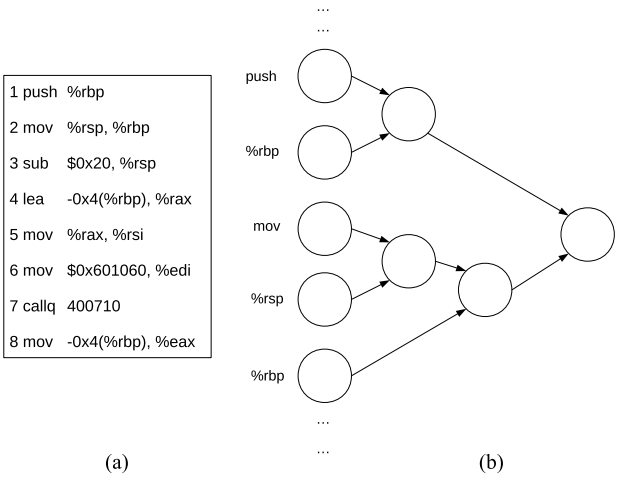

FIGURE 12. Code clone detection process by Xue *et al.* [140].

Example: As shown in Figure 12, given instructions in a basic block, the recursive auto-encoder will embed vectors into an embedding vector. It shows an execution path including a total of 8 instructions. Clone-Slicer makes use of a greedy method to combine the embeddings. For instance, the embedding for the first instruction (*push %rbp*) is encoded from terms embeddings [*push*; *%rbp*]. For the rest of the instructions, it repeats the same process until the end of the given execution path.

B. FUNCTION RECOGNITION

In binary code, only part of instructions can be extracted since different optimizations may have been made during program compiling time (e.g. inline functions). It is a difficult task to identify the boundary of each basic block and further find the entry point and the end point of functions. As mentioned in Section III, FEP features are used as inputs for the machine learning model to deal with this problem. Using machine learning, each boundary can be easily found. Table 9 lists several existing works for the function recognition in static binary codes.

Rosenblum *et al.* [106] and Rosenblum *et al.* [109] both set FEP based features and Idiom feature as the characteristic of binary code. Rosenblum *et al.* [106] use MRF model interface, while Rosenblum *et al.* [109] adopt CRF model interface as the learning model. The goal of these two papers is FEP identification. Besides, Byteweight [9] has the same goal which is to find the function entry points. It adopts a weight prefix tree to learning the CFG based features. Shin *et al.* [121] propose RNN as the ML model to learn tokens of byte sequences. FID [130] proposes a ensemble learning using LinearSVC, AdaBoost, and GradientBoosting to recognize function.

C. MALWARE DETECTION

The software security system has become more complex due to the growing software scale and complexity. Prior works have shown that there are about 5 to 20 bugs per 1,000 lines of software code [84]. To reduce the number of vulnerabilities in software systems, malware detection is becoming the focus of BCA research.

Identifying vulnerabilities in binaries have been studied for over 20 years, the main research consists of three major directions: static analysis, dynamic analysis, and hybrid method (the combination of static and dynamic). Tools for malware detection have been deployed to all stages of software development to reduce the damages caused by software security issues. To evaluate malware detection systems, we use two common measurements:

1. True Positives Rate (TPR), the number of malicious executable examples classified as malicious executables.

2. False Positives Rate (FPR), the number of benign programs classified as malicious executables.

We note that the reported TPR and FPR numbers vary in existing malware detection tools, that is because different tools may be used for different purposes (for example, some tools are proposed to detect memory-related vulnerable programs, some tools are to detect crytovirus, etc.). On the other hand, different tools utilize various types of features and machine learning analysis techniques. Thus, the performance of a malware detection tool can be affected by the different types of machines they used for evaluation, the hyperparameters in machine learning models (e.g., training iterations, vector embedding depth and size and so on), the type and the size of training and testing data. This results in that we cannot do a one-to-one simple comparison. Therefore, we report TPR and FPR into three different levels as $High/H$, $Medium/M$ and Low/L respectively. Table 12 shows the range of those levels.

Table 10 summarizes some widely used malware detection tools for binary executables. For instance, VDISCOVER has an FPR as H (31%), which is the highest FPR in this list. That is because it is developed to detect not only malware in large-scale programs, but to predict if the bugs are exploitable or probably exploitable. This can be limited by the amount of training data for machine learning model and require both static and dynamic (e.g., runtime

TABLE 9. Function recognition.

Title	Year	Features	ML Model
Rosenblum et al. [106]	2007	FEP based and Idiom features	MRF model interface
Rosenblum et al. [109]	2008	FEP based and Idiom features	CRF model interface
Byteweight [9]	2014	CFG	Weight Prefix Tree
Shin et al. [121]	2015	Byte Sequences	RNN
FID [130]	2017	Lexical, Syntactic and Stack features	Linear SVC, AdaBoost and GradientBoosting

TABLE 10. The comparison of classification performance in malware detection tools.

Title	Year	Features	ML Model	TPR	FPR
Schultz et al. [116]	2001	PE features and Byte sequence	RIPPER, Native Bayes model and Multi-Native Bayes model	High	Low
Gavriluct et al. [54]	2009	N/A	One-Sided Perceptron	High	Medium
Rieck et al. [103]	2011	MIST	Hierarchical Clustering and Nearest prototye classification	Not Applicable	Not Applicable
Liangboonprakong et al. [83]	2013	String Pattern	C4.5, Multilayer perceptron and SVM	High	Not Applicable
Saxe et al. [112]	2015	Byte Histogram Feature and PE Features	DNN	High	Low
Shijo et al. [120]	2015	PSI and API-calls	SVM and Random Forest	High	Low
Zak et al. [149]	2017	Sectional Byte N-grams and Assembly N-gram features	Elastic-Net Regularized Logistic Regression and Stacking	High	Not Applicable
RMVC [122]	2018	Opcodes	RNN and CNN	Medium	Low
Liu et al. [87]	2019	Image Features	MLP, KNN, and Random Forest	Medium	Not Applicable

TABLE 11. The comparison of experiment affecting factors in malware detection tools.

Title	Training data size		Testing data size		Evaluation Appraoch	Training Time
	#malicious binaries	#clean binaries	#malicious binaries	#clean binaries		
Schultz et al. [116]	3,265	1,001	206	38	cross-validation	Not Available
Gavriluct et al. [54]	7,822	415	2,581	137	cross-validation	22.75 mins
Rieck et al. [103]	33,698	Not Available	Not Available	Not Available	Metrics of precision and recall	Not Available
Liangboonprakong et al. [83]	9,759	2,440	1,950	488	Receiver operating characteristics (ROC)	Not Available
Saxe et al. [112]	350,016	81,910	Not Available	Not Available	cross-validation	40 mins
Shijo et al. [120]	997	490	Not Available	Not Available	Metrics of precision and recall	Not Available
Zak et al. [149]	200,000	200,000	40,000	37,349	Metrics of precision and recall	Not Available
RMVC [122]	4,133	Not Available	894	Not Available	Metrics of precision and recall	Not Available
Liu et al. [87]	36,736	Not Available	Not Available	Not Available	Metrics of precision and recall	Not Available

TABLE 12. Ranges of reported TPR and FPR for malware detection.

	High	Medium	Low
TPR	100% ∼95%	95% ∼75%	<75%
FPR	>15%	15% ∼5%	<5%

information) features. We further compare the experiment setup in those malware detection systems in detail, as shown in Table 11. As we can see, most of those works chose a larger amount of malicious binaries than clean/benign binaries for training their systems. On the other hand, the cross-validation approach is a common method to evaluate the performance. Finally, the training time is not always reported. That is because the training for machine learning is mainly an offline process, which does not consume online resources. Thus, it is not an important experiment affecting factor.

Schultz *et al.* [116] use RIPPER and different Bayes models learning PE features and token of byte sequence to detect vulnerabilities. Gavrilucţ *et al.* [54] direct learn the binary malware via the one-sided perceptron. Rieck *et al.* [103] generate a vulnerability detector through the Hierarchal Clustering and Nearest prototype classification. Liangboonprakong and Sornil [83] adopt both a decision tree C4.5, a multilayer perceptron and SVM to learn the string pattern. Saxe and Berlin [112] learn vulnerabilities through DNN. Shijo and Salim [120] adopt both SVM and Random Forest. Genius [47] searches bugs via spectral clustering. VDISCOVER [57] searches vulnerabilities through learning the static and dynamic calling sequence for c library of samples via the logistic regression model, MLP and Random Forest. Zak *et al.* [149] learn malware through two kinds of n-gram features. The first is sectional byte

TABLE 13. Vulnerabilities discovery.

Title	Year	Features	ML Model	Analysis Type	Type of vulnerabilities detected
Padmanabhuni et al. [97]	2015	CFG-based Features	Native Bayes, MLP, Simple Logistic, and Sequential Minimum Optimization	Syntax-level	Buffer overflow
Yamaguchi Fabian et al. [143]	2015	CFG-based Features	Clustering	Semantic-level	Memory corruption
Genius [47]	2016	ACFG	Spectral Clustering	Syntax level	Buffer/Heap/Stack overflow; Null pointer deference; Double free
VDISCOVER [57]	2016	Dynamic and static features	logistic regression, MLP of single hidden layer and random forest	Lexical-level	Buffer/Heap/Stack overflow; Null pointer deference; Double free; Use after free
NeuFuzz [131]	2019	Binary Programs	DNN	Semantic-level	Buffer/Heap/Stack overflow; Null pointer deference; Invalid memory access
Change et al. [27]	2019	Execution Paths	RNN	Semantic-level	Buffer/Heap/Stack/Integer overflow
PATCHDETECTOR [48]	2019	Tokens	Deep feed-forward Neural Network	Semantic-level	Buffer/Heap/Stack overflow

TABLE 14. Authorship recognition.

Title	Year	Features	ML Model
Rosenblum et al. [107]	2011	Idiom and graphlets features	SVM and LMNN
Caliskan et al. [20]	2015	Instruction feature	Random forest
Meng et al. [91]	2017	Instruction, data flow and context features	CRF and SVM

n-gram features which are byte sequences of PE features and the second is assembly n-gram features which are assembly code instructions. RMVC [122] first translates the assembly language into an image through RNN and hash algorithm Minhash [17], then uses CNN to learn image features and constructed a model for the malware identification.

D. VULNERABILITY DISCOVERY

Machine learning-based BCAs are also applied successfully in vulnerability discovery field. The existing vulnerability analysis methods based on machine-learning can be categorized into three different program analysis methods: (1) vulnerability analysis in the lexical level; (2) vulnerability analysis in the syntax level; (3) vulnerability analysis in the semantic level. Table 13 summarizes several distinguished works for vulnerability discovery, with respect of machine learning model used, analysis method, and the types of vulnerabilities that are detected. As we can see, semantic program analysis outperforms the other two analysis approaches in general, with the ability to detect diverse bugs and higher accuracy.

For example, Fabian *et al.* [143] propose a detection framework for automatically inferring search patterns for taint-style vulnerabilities. The inferred patterns are derived through tainting by identifying corresponding source-sink APIs and constructs patterns that model the control flow graph in binaries. The results show the inferred patterns reduce the amount of code to inspect for finding known vulnerabilities by over 95% and are able to identify multiple unknown types of vulnerabilities as well.

E. AUTHORSHIP RECOGNITION

Authorship means the creation of a piece of code and its attribution that will threaten the privacy and security community. To identify the author of a certain program by coding style without any source code or increase the accuracy, binary code is adopted for authorship recognition. Table 14 shows the contrast of Authorship recognition. Two papers all use features related to instructions. In Rosenblum *et al.* [107], Idioms and graphlets can reflect order and detail of instructions, while Caliskan-Islam *et al.* [20] directly use instructions as features. This means that the instruction is more likely to reflect the style of code by different authors. Meng *et al.* [91] construct a classification model via learning instructions, CFG, and data flow to identify basic block level authorship in a binary code.

F. OTHER TYPES OF APPLICATIONS

Besides the applications we have mentioned above, there are several other works using machine learning based BCA frameworks for other purposes. We list some distinguished other types of applications in Table 15. Hosfelt [65] test the ability of Cryptographic algorithm classification of learning algorithm SVM, Naive Bayes model, Decision Tree, and K-means Clustering. Katz *et al.* [72] create a tool to decompilation via RNN. Rosenblum *et al.* [105] use SVM to learn the compiler information and further recovers toolchain provenance.

VII. FUTURE RESEARCH DIRECTIONS

In this section, we discuss several possible directions for future work in BCA, including deep learning based BCA, and

TABLE 15. Other types of applications.

Title	Features	ML Model	Purpose
Hosfelt et al. [65]	Instruction, Category feature	SVM, Native Bayes model, Decision Tree and K-means Clustering	Cryptographic algorithm classification
Katz et al. [72]	String	RNN	Decompilation
Rosenblum et al. [105]	FEP based features	SVM	Recovering Toolchain Provenance

joint statistical and formal learning approaches to improve the robustness of BCA.

A. DEEP LEARNING

Deep learning has demonstrated great potential in various domains. Relating to software analysis, researchers have successfully applied deep learning for problems like malware detection [7], [112] and binary reverse [28]. More and more works are leveraging these advanced machine learning models for BCA.

LEMNA [59], a deep learning-based BCA framework, can take an input data sample and generates a small set of interpretable features to explain how the input sample is classified. The core idea is to approximate a local area of the complex deep learning decision boundary using a simple interpretable model. In a separate line of work, INNEREYE [151] borrows ideas from NLP to provide a solution for two important code similarity comparison problems. (I) Cross-platform binary code similarity detection; and (II) given a piece of binary code of interest, determining if it is contained in another piece of code compiled from a different instruction architecture set (ISA). Furthermore, MORPH [137] proposes an interactive program feature customization framework by leveraging deep learning technique to map dynamic execution trace to static binary functions and binary rewriting to remove unused program features.

These works are only the initial steps towards developing sophisticated and highly automated BCA models/tools through deep learning. By making the progress of developing machine learning models and address unique challenges arising from BCA, more efforts can make a positive contribution to building reliable deep learning systems for critical BCA applications.

B. JOINT STATISTICAL AND FORMAL LEARNING

As mentioned in Section II, The two lines of BCA techniques alone have certain fundamental limitations. 1. Pure statistical methods rely on probabilistic inference and often fail to guarantee complete accuracy. Any conclusions derived from sampling the runtime program states can offer only limited visibility and are prone to false alarms. As a result, considerable human effort is still required to verify the results from statistical analysis. 2. Formal methods require exhaustive analysis along with all paths in the application code, which can be prohibitively expensive in terms of time and resources. As such, strict symbolic execution methods can be less effective in analyzing software at-scale.

Recently, some researchers have proposed a new type of BCA framework, which integrates statistical and formal methods to harness the advantages of both techniques to perform rigorous code analysis in binary executables while maintaining scalability and swiftness. Statsym [146] explores the use of statistical data from faulty execution runs to swiftly identify vulnerable program paths in an application's source code. SIMBER [138] used statistical data from program execution runs to determine the safety of array accesses, and eliminate array bound checks in program locations where it is deemed redundant. We note that these joint learning techniques were performed at source code level.

Clone-Hunter [139] develops a joint learning framework to rapidly remove redundant array bound checks in binaries. It utilizes clustering algorithms from machine learning to detect binary code clones. Then, binary symbolic execution is used to verify if the samples within an identified cluster have redundant bound checks. If machine learning algorithms are able to cluster identical codes efficiently, this joint learning method enables a highly automated and scalable redundant bound check elimination process. Clone-Slicer [140] proposes a similar formal verification idea to verify if two binary clone clones detected from a machine learning module are true positives in terms of memory safety.

This hybrid (or joint) formal-statistical learning technique can add another important dimension to Machine-Learning-based BCA, potentially delivering both the effectiveness of statistical analysis and guarantees from symbolic execution.

VIII. CONCLUSION

Binary Code Analysis techniques have given us the opportunities to analyze binary executables without access to the source code and has notable applications in numerous domains like code clone detection, malware detection, software testing and so on. With the help of Machine learning techniques, BCA has been significantly sped up that helps better understand binary program behavior in a rapid manner, secure software systems, and troubleshoot errors during system runtime.

This article has discussed some key aspects and challenges of BCA based on machine learning, presenting for a broad audience the basic design principles of BCA frameworks. We hope this comparative study will help non-experts or other researchers to grasp the background knowledge and related

techniques, inspiring new novel ideas and further works in this important direction.

ACKNOWLEDGMENT

(Hongfa Xue and Shaowen Sun contributed equally to this work.)

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. Int. Conf. Very Large Data Bases*, vol. 1215. 1994, pp. 487–499.

[2] M. Alagappan, J. Rajendran, M. Doroslovački, and G. Venkataramani, "DFS covert channels on multi-core platforms," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2017, pp. 1–6.

[3] S. Alam, I. Traore, and I. Sogukpinar, "Annotated control flow graph for metamorphic malware detection," *Comput. J.*, vol. 58, no. 10, pp. 2608–2621, 2015.

[4] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[5] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2012, pp. 100–109.

[6] K. Alsabti, S. Ranka, and V. Singh, "An efficient k-means clustering algorithm," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 1997.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. NDSS Symp.*, vol. 14, 2014, pp. 23–26.

[8] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1343–1362, 1997.

[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proc. USENIX*, 2014, pp. 845–860.

[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, 2003, pp. 281–289.

[11] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, vol. 9, 2009, pp. 8–11.

[12] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," Ikarus Softw. Tech. Univ. Vienna, Vienna, Austria, Tech. Rep., 2006.

[13] W. A. Belson, "Matching and prediction on the principle of biological classification," *Appl. Statist.*, vol. 8, no. 2, pp. 65–75, 1959.

[14] M. R. Berthold *et al.*, "KNIME: The Konstanz information miner," in *Data Analysis, Machine Learning and Applications* (Studies in Classification, Data Analysis, and Knowledge Organization). New York, NY, USA: Springer, 2007.

[15] U. Bodenhofer, A. Kothmeier, and S. Hochreiter, "APCluster: An R package for affinity propagation clustering," *Bioinformatics*, vol. 27, no. 17, pp. 2463–2464, 2011.

[16] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proc. 5th Annu. Workshop Comput. Learn. Theory*, 1992, pp. 144–152.

[17] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. Compress. Complex. Sequences*, 1997, pp. 21–29.

[18] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[19] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," Dept. Elect. Eng. Comput. Sci., California Univ. Berkeley, Berkeley, CA, USA, 2009.

[20] A. Caliskan-Islam *et al.*, "When coding style survives compilation: De-anonymizing programmers from executable binaries," 2015, *arXiv:1512.08546*. [Online]. Available: https://arxiv.org/abs/1512.08546

[21] S. Cesare and X. Yang, "Classification of Malware using structured control flow," in *Proc. 8th Australas. Symp. Parallel Distrib. Comput.*, vol. 107, Jan. 2010, pp. 61–70.

[22] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[23] J. Chen, G. Venkataramani, and H. H. Huang, "RePRAM: Re-cycling PRAM faulty blocks for extended lifetime," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.

[24] J. Chen, G. Venkataramani, and H. H. Huang, "Exploring dynamic redundancy to resuscitate faulty PCM blocks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 10, no. 4, p. 31, 2014.

[25] Y. Chen, T. Lan, and G. Venkataramani, "DamGate: Dynamic adaptive multi-feature gating in program binaries," in *Proc. Workshop Forming Ecosyst. Around Softw. Transformation*, 2017, pp. 23–29.

[26] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "TOSS: Tailoring online server systems through binary feature customization," in *Proc. Workshop Forming Ecosyst. Around Softw. Transformation*, 2018, pp. 1–7.

[27] L. Cheng *et al.*, "Optimizing seed inputs in fuzzing with machine learning," 2019, *arXiv:1902.02538*. [Online]. Available: https://arxiv.org/abs/1902.02538

[28] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Conf. Secur. Symp., Secur.*, vol. 17, 2017, pp. 99–116.

[29] C. Cifuentes, "Partial automation of an integrated reverse engineering environment of binary code," in *Proc. WCRE*, Nov. 1996, pp. 50–56.

[30] W. W. Cohen, "Fast effective rule induction," in *Proc. Mach. Learn. Process.*, 1995, pp. 115–123.

[31] W. W. Cohen, "Learning trees and rules with set-valued features," in *Proc. AAAI/IAAI*, vol. 1, 1996, pp. 709–716.

[32] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *Proc. Reverse Eng. 16th Work. Conf. (WCRE)*, 2009, pp. 167–176.

[33] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[34] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code comparison system based on abstract syntax tree," in *Proc. 3rd IEEE Int. Conf. Broadband Netw. Multimedia Technol.(IC-BNMT)*, 2010, pp. 668–673.

[35] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.

[36] J. Dai, R. K. Guha, and J. Lee, "Efficient virus detection using dynamic instruction sequences," *JCP*, vol. 4, no. 5, pp. 405–414, 2009.

[37] M. Damashek, "Gauging similarity with n-grams: Language-independent categorization of text," *Science*, vol. 267, no. 5199, pp. 843–848, 1995.

[38] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.

[39] Y. David and E. Yahav, "Tracelet-based code search in executables," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[40] T. G. Dietterich, "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization," *Mach. Learn.*, vol. 40, no. 2, pp. 139–157, 2000.

[41] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and efficient memory protection using dynamic tainting," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 87–100, Jan. 2012.

[42] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. Hoboken, NJ, USA: Wiley, 2012.

[43] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. NDSS*, 2016.

[44] M. Everingham *et al.*, "The 2005 Pascal visual object classes challenge," in *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Tectual Entailment*. Springer, 2006, pp. 117–176.

[45] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Aug. 2008.

[46] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 346–359.

[47] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.

[48] Q. Feng *et al.*, "Learning binary representation for automatic patch detection," in *Proc. 16th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2019, pp. 1–6.
[49] I. Firdausi *et al.*, "Analysis of machine learning techniques used in behavior-based malware detection," in *Proc. 2nd Int. Conf. Adv. Comput., Control Telecommun. Technol. (ACT)*, Dec. 2010, pp. 201–203.
[50] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE Symp. Secur. Privacy*, May 1996, pp. 120–128.
[51] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *Proc. ICML*, vol. 99, 1999, pp. 124–133.
[52] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, Feb. 2007.
[53] G. Fursin *et al.*, "Milepost GCC: Machine learning based research compiler," in *Proc. GCC Summit*, 2008.
[54] D. Gavriluţ, M. Cimpoeşu, D. Anton, and L. Ciortuz, "Malware detection using machine learning," in *Proc. Int. Multiconf. Comput. Sci. Inf. Technol. (IMCSIT)*, 2009, pp. 735–741.
[55] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theor. Comput. Sci.*, vol. 38, pp. 293–306, 1985.
[56] A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proc. 3rd Int. Conf. Frontiers Intell. Comput., Theory Appl. (FICTA)*. Cham, Switzerland: Springer, 2015, pp. 113–122.
[57] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. Sixth ACM Conf. Data Application Secur. Privacy*, 2016, pp. 85–96.
[58] H. Guo, J. Pang, Y. Zhang, F. Yue, and R. Zhao, "HERO: A novel malware detection framework based on binary translation," in *Proc. IEEE Int. Conf. Intell. Comput. Intell. Syst. (ICIS)*, vol. 1, Oct. 2010, pp. 411–415.
[59] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 364–379.
[60] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.
[61] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Appl. Statist.*, vol. 28, no. 1, pp. 100–108, 1979.
[62] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit.*, vol. 1. Aug. 1995, pp. 278–282.
[63] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Proc. Scalable High–Performance Comput. Conf.*, 1994, pp. 841–850.
[64] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci. USA*, vol. 79, no. 8, pp. 2554–2558, 1982.
[65] D. D. Hosfelt, "Automated detection and classification of cryptographic algorithms in binary programs through machine learning," 2015, *arXiv:1503.01186*. [Online]. Available: https://arxiv.org/abs/1503.01186
[66] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. 25th Int. Conf. Program Comprehension*, 2017, pp. 88–98.
[67] Insights, OECD and Revenue, Increase, *The Internet of Things*, 2017.
[68] R. Islam, R. Tian, L. Batten, and S. Versteeg, "Classification of malware based on string and function feature selection," in *Proc. 2nd Cybercrime Trustworthy Comput. Workshop (CTC)*, 2010, pp. 9–17.
[69] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *J. Netw. Comput. Appl.*, vol. 36, no. 2, pp. 646–656, 2013.
[70] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 96–105.
[71] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.
[72] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 346–356.
[73] M. Kearns and Y. Mansour, "On the boosting ability of top–down decision tree learning algorithms," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 109–128, 1999.
[74] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 5, pp. 207–218, 2006.
[75] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*. Savannah, GA, USA: Springer, 2009, pp. 214–228.
[76] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. 13th Work. Conf. Reverse Eng.*, Oct. 2006, pp. 253–262.
[77] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. USENIX Secur. Symp.*, vol. 13, 2004, p. 18.
[78] Y. LeCun *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.
[79] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," 2011.
[80] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning patterns from unix process execution traces for intrusion detection," in *Proc. AAAI Workshop AI Approaches Fraud Detection Risk Manage.*, 1997, pp. 50–56.
[81] D. Levy and L. Wolf, "Learning to align the source code to the compiled object code," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 2043–2051.
[82] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "SARRE: Semantics-aware rule recommendation and enforcement for event paths on android," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 12, pp. 2748–2762, Dec. 2016.
[83] C. Liangboonprakong and O. Sornil, "Classification of malware families based on N-grams sequential pattern features," in *Proc. IEEE 8th Conf. Ind. Electron. Appl. (ICIEA)*, Jun. 2013, pp. 777–782.
[84] M. C. Libicki, L. Ablon, and T. Webb, *The Defender's Dilemma: Charting a Course Toward Cybersecurity*. Santa Monica, CA, USA: Rand, 2015.
[85] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. 11th Annu. Inf. Secur. Symp.* West Lafayette, IN, USA: CERIAS-Purdue Univ., 2010, Art. no. 5.
[86] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, 2003, pp. 290–299.
[87] Y.-S. Liu, Y.-K. Lai, Z.-H. Wang, and H.-B. Yan, "A new learning approach to malware classification using discriminative feature extraction," *IEEE Access*, vol. 7, pp. 13015–13023, 2019.
[88] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982.
[89] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.
[90] L. J. McKenzie, M. S. Trevisan, D. C. Davis, and S. W. Beyerlein, "Capstone design courses and assessment: A national study," in *Proc. Amer. Soc. Eng. Educ. Annu. Conf. Expo.*, 2004, pp. 1–14.
[91] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *Proc. Eur. Symp. Res. Comput. Secur.* Oslo, Norway: Springer, 2017, pp. 286–304.
[92] P. Miller. (2000). *Hexdump*. [Online]. Available: http://man7.org/linux/man-pages/man1/hexdump.1.html
[93] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.
[94] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Proc. Adv. Neural Inf. Process. Syst.*, 2002, pp. 849–856.
[95] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 201–210.
[96] D. Oktavianto and I. Muhardianto, *Cuckoo Malware Analysis*. Birmingham, U.K.: Packt, 2013.
[97] B. M. Padmanabhuni and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 2, pp. 450–459, Jul. 2015.
[98] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proc. Int. Conf. Knowl. Sci., Eng. Manage.* Chongqing, China: Springer, 2015, pp. 547–553.
[99] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: Scale-free or geometric?" *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, 2004.

[100] P. Pudil, F. J. Ferri, J. Novovicova, and J. Kittler, "Floating search methods for feature selection with nonmonotonic criterion functions," in *Proc. 12th IAPR Int. Conf. Pattern Recognit., Conf., B, Comput. Vis. Image Process.*, vol. 2, Oct. 1994, pp. 279–283.

[101] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2014.

[102] *Ida Pro Disassembler*, Data Rescue, 2006. [Online]. Available: https://www.datarescue.com/idabase/

[103] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, 2011.

[104] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, no. 6, p. 386, 1958.

[105] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. Int. Symp. Softw. Test. Anal.*, 2011, pp. 100–110.

[106] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Machine learning-assisted binary code analysis," in *Proc. NIPS Workshop Mach. Learn. Adversarial Environ. Comput. Secur.*, Whistler, BC, Canada, Dec. 2007, pp. 1–3.

[107] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? Identifying the authors of program binaries," in *Proc. Eur. Symp. Res. Comput. Secur.* Berlin, Germany: Springer, 2011, pp. 172–189.

[108] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2010, pp. 21–28.

[109] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proc. AAAI*, 2008, pp. 798–804.

[110] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal.*, 2009, pp. 117–128.

[111] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[112] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.

[113] B. Schölkopf *et al.*, "Comparing support vector machines with Gaussian kernels to radial basis function classifiers," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2758–2765, Nov. 1997.

[114] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Proc. Int. Workshop Inf. Hiding*. Berlin, Germany: Springer, 2011, pp. 270–284.

[115] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Comput. Surv.*, vol. 49, no. 1, 2016, Art. no. 4.

[116] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2001, pp. 38–49.

[117] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, 2002.

[118] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 94–109.

[119] J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," in *Proc. 1st Workshop Archit. Syst. Support Improving Softw. Dependability*, 2006, pp. 52–57.

[120] P. V. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," *Procedia Comput. Sci.*, vol. 46, pp. 804–811, Feb. 2015.

[121] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. USENIX Secur. Symp.*, 2015, pp. 611–626.

[122] G. Sun and Q. Qian, "Deep learning and visualization for identifying malware families," *IEEE Trans. Dependable Secure Comput.*, to be published.

[123] G. J. Tesauro, J. O. Kephart, and G. B. Sorkin, "Neural networks for computer virus recognition," *IEEE Expert*, vol. 11, no. 4, pp. 5–6, Aug. 1996.

[124] H. Theiling, "Extracting safe and precise control flow from binaries," in *Proc. 7th Int. Conf. Real-Time Comput. Syst. Appl.*, Dec. 2000, pp. 23–30.

[125] P. Trinius, C. Willems, T. Holz, and K. Rieck, "A malware instruction set for behavior-based analysis," 2009.

[126] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, Feb. 2008, pp. 173–184.

[127] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "MemTracker: An accelerator for memory debugging and monitoring," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, 2009, Art. no. 5.

[128] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "DeFT: Design space exploration for on-the-fly detection of coherence misses," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, 2011, Art. no. 8.

[129] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, and D. A. Keim, "A survey of visualization systems for malware analysis," in *Proc. EG Conf. Vis. (EuroVis)-STARs*, 2015, pp. 105–125.

[130] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 388–398.

[131] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "NeuFuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019.

[132] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," *J. Mach. Learn. Res.*, vol. 10, pp. 207–244, Feb. 2009.

[133] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2016, pp. 87–98.

[134] C. Willems, T. Holz, and F. Freiling, "CWSandbox: Towards automated dynamic binary analysis," *IEEE Secur. Privacy*, vol. 5, no. 2, pp. 32–39, Mar. 2007.

[135] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 536–546.

[136] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

[137] H. Xue, Y. Chen, G. Venkataramani, T. Lan, G. Jin, and J. Li, "MORPH: Enhancing system security through interactive customization of application and communication protocol features," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2315–2317.

[138] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "SIMBER: Eliminating redundant memory bound checks via statistical inference," in *Proc. IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*. Cham, Switzerland: Springer, 2017, pp. 413–426.

[139] H. Xue, G. Venkataramani, and T. Lan, "Clone-hunter: Accelerated bound checks elimination via binary code clone detection," in *Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, 2018, pp. 11–19.

[140] H. Xue, G. Venkataramani, and T. Lan, "Clone-slicer: Detecting domain specific binary code clones through program slicing," in *Proc. Workshop Forming Ecosyst. Around Softw. Transformation*, 2018, pp. 27–33.

[141] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 674–691.

[142] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 359–368.

[143] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 797–812.

[144] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 499–510.

[145] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 168–179.

[146] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani, "StatSym: Vulnerable path discovery through statistics-guided symbolic execution," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 109–120.

[147] F. Yao, G. Venkataramani, and M. Doroslovački, "Covert timing channels exploiting non-uniform memory access based architectures," in *Proc. Great Lakes Symp. VLSI*, 2017, pp. 155–160.

[148] Oleh Yuschuk. (2007). *Ollydbg*. [Online]. Available: http://www.ollydbg.de/
[149] R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?" in *Proc. 12th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2017, pp. 109–118.
[150] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 487–498.
[151] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*. [Online]. Available: https://arxiv.org/abs/1808.04706

HONGFA XUE received the B.S. degree from the Beijing University of Posts and Telecommunications, Beijing, China. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, The George Washington University. His research interests include system/software security and machine learning optimization.

SHAOWEN SUN received the B.E. degree from the North China University of Technology, Beijing, China, in 2017. He is currently pursuing the M.S. degree with the Department of Electrical and Computer Engineering, The George Washington University. His research interests include networking and system security.

GURU VENKATARAMANI (SM'15) received the Ph.D. degree from the Georgia Institute of Technology, Atlanta, in 2009. He has been an Associate Professor of electrical and computer engineering with The George Washington University, since 2009. His research interests include computer architecture, hardware support for energy/power optimization, debugging, and security. He was a recipient of the NSF Faculty Early Career Award, in 2012. He was a General Chair for HPCA'19.

TIAN LAN received the Ph.D. degree from the Department of Electrical Engineering, Princeton University, in 2010. He joined the Department of Electrical and Computer Engineering, The George Washington University, in 2010, where he is currently an Associate Professor. His research interests include mobile energy accounting, cloud computing, and cyber security. He received the Best Paper Award from the IEEE Signal Processing Society 2008, the IEEE GLOBECOM 2009, and the IEEE INFOCOM 2012.