IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 8 April 2023, accepted 11 May 2023, date of publication 23 May 2023, date of current version 1 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3279280

RESEARCH ARTICLE

Use of Ensemble Learning to Detect Buffer Overflow Exploitation

AYMAN YOUSSEF[1], MOHAMED ABDELRAZEK[2], AND CHANDAN KARMAKAR[1], (Member, IEEE)

[1]Faculty of Science, Engineering, and Built Environment, School of Information Technology, Deakin University, Melbourne, VIC 3125, Australia
[2]A2I2D, Applied Aritificial Intelligence Institute, Deakin University, Melbourne, VIC 3125, Australia

Corresponding author: Ayman Youssef (ayman.youssef@research.deakin.edu.au)

ABSTRACT Software exploitation detection remains unresolved problem. Software exploits that target known and unknown vulnerabilities are constantly used in attacks. Signature-based detection techniques are limited to known exploits and susceptible to circumvention. Current research on the use of Machine Learning (ML) for software exploitation detection is limited in quantity and use cases. Existing research lacks the use of public datasets, discussions of feature importance, and elaboration of parameters that affect data preparation and subsequently model performance. This paper presents ML models based on different ensemble algorithms to detect software exploitation using runtime traces. We focus on buffer overflow vulnerabilities in user-space applications within Windows Operating Systems (OS), given the prevalence of the type of vulnerability and the OS. We utilized a publicly available raw dataset of 11 Windows applications under exploitation. Multiple distinct models (based on Random Forest and XGBoost) are created and tested. Testing was performed several times using various aggregation parameters and different testing applications. Our results demonstrate that we can achieve up to 100% recall with 0% false positive rate. We report on the different parameters that must be addressed to curate runtime traces and demonstrate their impact on the performance of the ML models. We demonstrate that the proper training of models on a subset of exploitation techniques enables the model to detect techniques never seen before, such as return-oriented programming. Finally, we conclude with a discussion of the important features that had the highest impact on each of the models, along with the key takeaways.

INDEX TERMS Buffer overflow, exploitation detection, machine learning, random forests, XGBoost.

I. INTRODUCTION

In April 2022, Microsoft released a report stating that owing to the current conflict between Ukraine and Russia, the global cybersecurity community should expect an increase in the use of sophisticated attacks that utilize zero-day attacks and other novel attack techniques [1]. In its M-Trends report for 2021, Mandiant reported that 37% of the attacks investigated used software exploits as the initial attack vector [2]. Furthermore, Mandiant noticed that the median dwell time (i.e., time taken by an organization to remediate breaches) was on average 21-days. This means that attackers who utilize zero-day exploits can remain within the enterprise network for up to 21 days from the time an organization is made aware of the breach. Moreover, the most commonly used technique reported by Mandiant experts is (as per MITRE attack framework) T1027 Obfuscated Files or Information. Obfuscation is a technique used by attackers to thwart static analysis tools, further increasing the emphasis on the need for dynamic analysis techniques [3].

Several techniques have been embedded in modern Operating System (OS) defenses to help protect user-space applications but have proven to be insufficient. Techniques such as Address Space Layout Randomization (ASLR) [4] and Data Execution Prevention (DEP) [5] have been circumvented [6]. Furthermore, exploitation detection frameworks that have been proposed based on signatures or heuristic-based rules [7] fail to cope with newer-evolved exploitation techniques [8].

In 2011, Lockheed Martin developed a model called The Cyber Kill Chain® which outlines the different stages of cybersecurity attacks [9]. The Cyber Kill Chain® comprises

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

seven stages: these are, reconnaissance, weaponization, delivery, exploitation, installation, command and control (C&C), and actions on objectives [10]. Exploitation refers to the process of deploying/executing an exploit on a target system and initiating communications with a command and control server (C&C) [11]. Cisco defines an exploit as "a program, or piece of code, designed to find and take advantage of a security flaw or vulnerability in an application or computer system, typically for malicious purposes such as installing malware. An exploit is not malware itself, but rather it is a method used by cybercriminals to deliver malware" (please refer to the Background section for differences between exploitation detection and malware, intrusion and vulnerability detection) [12].

For the purpose of this research, exploitation detection refers to the ability to detect an attacker exploiting a vulnerable program based on runtime traces. Machine Learning (ML)-based exploitation detection research is limited in terms of quantity and content. To the best of our knowledge, we were able to find only 14 papers [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] that discussed the use of ML for exploitation detection. None of the 14 papers publicly presented their datasets. These research papers were segmented into seven that relied on runtime traces [13], [14], [15], [16], [17], [24], [25], four that relied on network traces, [18], [19], [20], [26] and three that relied on static analysis of the application under test (AUT) (i.e., vulnerable program) [21], [22], [23]. Of the 14 papers, 13 did not discuss feature importance and only one paper discussed feature importance by providing a single table with the Fisher score of each feature [16]. Furthermore, none of the 14 papers discussed the aggregation parameters and their impact on the performance of different algorithms.

The aim of this study is to provide guidance on feature importance and engineering to enable strong exploitation detection algorithms. We curated a raw dataset based on 11 Windows applications to train two ensemble machine-learning (ML) algorithms. Each application has a buffer overflow vulnerability, which is exploited. Table 1 lists the AUTs, the "Exploit Technique" used with each application and the size (in terms of assembly instructions) of each raw trace under column header "No. Trace Inst." The column "No. Payload Inst." lists the number of assembly instructions that are part of the exploit payload found in the raw dataset. Furthermore, we classified the category of each application to further enrich the reporting of ML performance.

Our choice of Windows OS is driven by the fact that Windows is the most popular desktop OS, based on market share [27]. The choice of buffer overflow vulnerabilities is driven by their prevalence, as demonstrated in MITRE's 25 top Common Weakness Enumeration (CWE) weaknesses for the 2022 report [28]. The report shows that three of the top 25 spots were due to buffer errors (CWE-119, CWE-787, and CWE-125), taking the spots for 1st, 5th, and 9th most dangerous CWEs. Furthermore, a review of all vulnerabilities announced by the National Vulnerability Database (NVD) from January 1st, 2008, to 22nd December 22, 2022, [29] shows that 162,806 vulnerabilities were announced during that period. Of this total value, 22,776 vulnerabilities (14.20%) are attributed to one of the three buffer-related CWEs of MITRE's 2022 top 25.

TABLE 1. List of applications used in the dataset.

App Name	App Category	Exploit Technique	No. Trace Inst.	No. Payload Inst.
Easy WMV/ASF/ASX to DVD Burner 2.3.11	Media Converter	SEHOP Overwrite	809	291
Easy Video to PSP Converter 1.6.20		SEHOP Overwrite	566	291
MP3 WAV to CD Burner 1.4.24		SEHOP Overwrite	633	291
Easy AVI DivX Converter 1.2.24		SEHOP Overwrite	684	287
Free MP3 CD Ripper 2.6		SEHOP Overwrite	146	0
Disk Pulse Enterprise 9.9.16	Disk Mgmt.	Stack Smashing	202	96
Disk Sorter Enterprise 9.5.12 (mem trace only)		Stack Smashing	0	0
VUPlayer 2.49	Media Player	ROP Chain	503	381
ALL Player 7.4		SEHOP Overwrite	120	0
Publish-It 3.6d	Citation Mngmt.	SEHOP Overwrite	396	391
T-Mobile Internet Manager	Comms	SEHOP Overwrite	1191	0
Total No. of Instructions			5250	2028

Our selection of ensemble ML models is driven by two aspects: performance and ability to report feature importance. We selected two algorithms, from one of the two main types of ensemble algorithms: bagging and boosting. Stacking algorithms were excluded from our research because of their inability to report feature importance [30].

For the bagging algorithm we selected Random Forests (RF) [31] and for the boosting algorithm we selected XGBoost [32]. Both algorithms have been utilized in recent studies and have provided advantageous results in various areas [33], [34], [35], [36].

To summarize, our contribution in this paper is:

- Provide a design (covering details of data preparation) for the use of ML in exploitation detection based on runtime traces.
- Discussion of the different features and their relative importance to be able to use them for ML.
- Comparison of two ensemble classifiers that utilize two different techniques (bagging and boosting) based on selected performance criteria.

In the remaining parts of this research, Section II provides background information on the differences between exploitation research and malware, intrusion and

vulnerability detection research. Section III describes related exploitation detection research performed on runtime traces. Section IV describes the dataset and Section V presents the methodology used for the feature engineering, building, and tuning of ML models. Section VI discusses the experiment results and feature importance. Section VII provides a summary of the key takeaways based on the experimental results and Section VIII discusses research limitations and future work. Finally, we conclude the study in Section IX.

II. BACKGROUND

This section highlights the differences between exploitation detection and malware, intrusion and vulnerability detection. Each of these domains intersects with exploitation detection; however, they differ in their research approach.

Malware detection fits in the fifth stage of the Cyber Kill Chain®, that is, installation. Malware detection engines can sometimes detect files that contain exploits as malware of type ''exploit'' [21]. However, as illustrated below, this is not the core competency of anti-malware and is part of the gray area where the line between exploits and malware is blurred.

Intrusion detection targets several stages of the Cyber Kill Chain®, including reconnaissance, delivery, exploitation, installation, C&C, and actions on objectives [37], [38], [39], [40], [41]. Nevertheless, this broad feature set makes current research and current datasets too coarse-grained to be effective in exploitation detection.

Finally, vulnerability detection research is a precursor to exploitation. Vulnerability detection typically occurs at the reconnaissance stage of the Cyber Kill Chain® where attackers search for vulnerabilities to target. Vulnerability detection research aims to identify and locate vulnerabilities in software before a malicious actor exploits such vulnerabilities. Exploitation detection targets malicious actors in the act of exploiting a vulnerability. Hence, despite the intersection of two the research directions, they address the same problem from two different angles with different approaches and in different scenarios.

A. EXPLOITation DETECTION VS. MALWARE DETECTION

We first highlight that the word ''malware'' used in mainstream research is sometimes confused with ''exploits.'' Wikipedia defines malware as ''any software intentionally designed to cause disruption to a computer, server, client, or computer network, leak private information, gain unauthorized access to information or systems, deprive access to information, or which unknowingly interferes with the user's computer security and privacy.'' Reference [42] Although software that has exploits embedded into it is a malicious software (i.e., ''malware'') researchers tend to exclude ''exploits'' from ''malware'' and define it as an independent category [12]. The behavior of ''exploits'' is significantly different from that of mainstream malware families. Mainstream malware usually relies on destructive activities such as exfiltrating data, wiping data, and encryption for ransom purposes. Furthermore, malware activities involve reading files, changing registry keys, and accessing protected files [43].

Exploitation, on the other hand, is considered the fourth stage of the Cyber Kill Chain®, a stage prior to the installation of malware and the establishment of a command channel [10]. An exploit is usually a group of bytes (i.e., shellcode) sent remotely or embedded in a file; when consumed by a vulnerable application, it causes the application to behave in a manner not intended by its creators. Examples of such behaviors include downloading destructive malware, connecting with remote command and control (C&C) servers, or simply causing an application to crash, resulting in a denial of service. Each exploit can be created by using different coding techniques and payloads [2]. Payloads can be of different types, such as command execution, creating reverse TCP/UDP connections, and binding the application to a specific port [44]. All of these could exhibit different behaviors, and hence are not properly covered by the existing body of research on malware detection.

B. EXPLOITation DETECTION VS. INTRUSION DETECTION

The current intrusion detection research includes a wealth of research on the use of ML for intrusion detection [45], [46], [47]. The key differentiator between exploitation detection and intrusion detection research (based on ML models) is the dataset used. IDS datasets focus on the context of an attack, including several steps of the attack lifecycle (a.k.a. Cyber Kill Chain®). This is different from exploitation detection datasets, which focus solely on the behavior of exploits. Furthermore, IDS datasets do not focus on reporting the types of vulnerabilities exploited nor do they focus on including a large number of exploitable vulnerabilities within each category. Unlike Exploitation Detection datasets, which document the type of vulnerabilities being exploited, how exploitation occurs (i.e., locally or remotely) and the payload content being used [48].

To elaborate further on the shortcomings of IDS datasets for support of exploitation detection research, we sampled some of the highest-cited IDS datasets (as per the Google Scholar search engine[1]). The ADFA-WD and ADFA-WD:SAA datasets are host-based. The ADFA-WD and ADFA-WD:SAA were published in [37] and has 56 citations. The ADFA-LD was also published by the same lead author and is similar to ADFA-WD, except that it targets the Linux OS. ADFA-LD has 303 citations in Google Scholar. Both datasets were part of the thesis published in [38] which has been cited 94 times. In ADFA-WD, the author explains that there are 12 vulnerabilities exploited in attack scenarios that mimic stealth attacks, data exfiltration, and Distributed Denial of Service (DDoS) attacks. The dataset was limited to 9 DLLs. The dataset is composed of Windows audit logs. The dataset was created with anomaly detection in mind, and anomaly-detection algorithms were used to validate its usefulness. As for the ADFA-LD, the authors included a

[1]Citation counts as of 22nd December 2022

variety of attacks similar to the ADFA-WD, but with a much lower number of vulnerabilities and only two exploits. The ADFA-LD gathered only system call traces.

As shown in both datasets (ADFA-WD and ADFA-LD), traces were limited in type (only audit logs of system calls or DLL calls). This is in contrast to the exploitation based dataset presented in [48] and shown in Fig.1 and 2, which contains a greater level of detail that enables researchers to perform in-depth investigations of feature importance across different ML models. As shown in the Experiments section, different features have different importance levels depending on the algorithm and aggregation parameters. Furthermore, the author did not report the effectiveness of the algorithms based on vulnerability types. In this exploitation-detection-focused study, we clarified the vulnerability type, exploitation technique, and payload content.

The NGIDS [41] dataset has 162 citations and is considered a hybrid dataset containing network and host-based traces. NGIDS was created in a cyber range that aims to mimic real-world scenarios. The NGIDS dataset comprises network packet captures and system logs that represent a system's reaction to network-based attacks. The authors used seven categories of attacks, two of which are titled ''exploits'' and ''shellcodes''. The authors did not report on the Common Vulnerabilities and Exposures (CVEs) that were targeted or elaborate on the differences between ''exploits'' and ''shellcode''.

The same pattern can be observed in different datasets, where the focus is on the entire kill chain, instead of a specific type of vulnerability. In UNSW-NB15 [39], which has 1721 citations and contains network traces, the authors reported 10 attack categories. Two of them were also labelled ''exploits'' and ''shellcode.'' No clarification of targeted CVEs or vulnerability types Similarly, CICIDS 2017 [40] contains 2007 citations and network-based traces. Of the 11 attack types described, only 3 exploits were included. Of the three exploits elaborated, only two had CVEs explaining the type of vulnerability. As noted above, both network-based datasets (UNSW-NB15 and CICIDS 2017) have a much larger number of citations (1721 and 2007, respectively) than host-based datasets (ADFA and NGIDS) combined with 615 citations.

Hence, Intrusion Detection datasets are not suitable for exploitation detection research, and research performed on them does not provide sufficient insight into their effectiveness against specific types of exploits.

C. EXPLOITATION DETECTION VS. VULNERABILITY DETECTION

The terms vulnerability and exploit are often confused in research papers [49]. To achieve a comprehensive definition of vulnerability, Zeng et al. [50] surveyed the definitions in seminal work. Zeng et al.'s noted the interchangeable use of keywords in each definition. These keywords were an error, a fault, and a mistake. Based on their analysis and comparative investigation of the meaning of these keywords from the IEEE Standards Glossary of Software Engineering [51], they came up with a definition and an explanation of vulnerability. In their explanation, a vulnerability is a mistake made by a human in the writing of the software code. This mistake resulted in a fault in the software. The execution of faulty program statements does not necessarily result in a violation of the security policies governing the program. However, upon processing specially crafted data (i.e., the exploit) by the faulty statements, a security violation occurs (i.e., exploitation resulting in a security failure) [50]. Hence, Vulnerability detection research is focused on identifying the type of mistake, and the location of the faulty statement(s) before it is exploited.

Vulnerability detection research can be broadly divided into three types: static, dynamic and hybrid (using a mixture of static and dynamic) [52]. Static analysis based vulnerability detection is based on the analysis of source code and binary files [53], [54], [55]. Binary file analysis is performed after disassembling the files and converting the assembly code to a high level code for analysis [56]. This study focuses on exploitation detection based on runtime traces; hence, static analysis is not relevant in this comparison.

Dynamic analysis techniques involve inspecting the program behavior during runtime. Several techniques are based on dynamic taint analysis [49], symbolic code execution [57] and fuzzing [58]. For dynamic taint analysis and symbolic code execution techniques, binary instrumentation is necessary to apply the rules necessary to identify the presence of vulnerabilities. Fuzzing, however, can be segmented into black, white, and gray box fuzzing [59]. White and gray box fuzzing are techniques that rely on prior knowledge of the AUT [60], [61]. This information is used to guide the test cases that will be presented to the program. Black box fuzzing does not include any knowledge of the AUT [58]. In all the fuzzing situations explained above, the main idea is that there is a random (in the case of the black box) or valid (in the case of the grey and white boxes) seed input provided to the AUT; then, this seed input is mutated with every testing cycle. This mutation process could be governed in gray/white box testing with analysis of the program's behavior using different instrumentation techniques [62]. The black box approach does not include any knowledge of the AUT, therefore, the mutation is performed without specific feedback about the AUT's internal state [58].

The aforementioned research differs from our research direction for exploitation detection in the following ways.

- Object of detection: In dynamic analysis techniques for vulnerability detection, the object of detection is the fault. Dynamic taint analysis and symbolic execution techniques detect faults by using expertly crafted rules. Fuzzing techniques detect faults by the program state (for example, program crashing or hanging). Not all detections in vulnerability detection are exploitable, and results usually require manual vetting or the use of automated signature generation systems [63]. Exploitation

detection does not detect the fault but detect the pattern of exploitation and report the incident.
- Technique of detection: Dynamic analysis techniques for vulnerability detection (with the exception of black box fuzzing) perform some form of instrumentation on the AUT. Exploitation detection aims to detect the exploitation based on the runtime traces.
- Methods used: Vulnerability detection relies on repetitive launching of the AUT and feeding it random/crafted messages to test whether it will crash. Exploitation detection does not benefit from repetition to detect the same exploit. If the exploit is not detected from the first pass, then the ML model (or the expert rules used) must be modified.
- Output of detection process: The output in vulnerability detection is the type and location of the vulnerability in the AUT code. However, the output of vulnerability requires further validation to confirm whether the vulnerability is exploitable. The output of exploitation detection is the notification (and potential prevention) of the exploitation process which may result in further actions on the target.

III. RELATED WORK

Existing techniques can be divided into two broad categories: signature/heuristic-based and ML-based. Signature/heuristic-based techniques have been proven to have a limited capability to adapt to changes in attack techniques [8], [64], [65].

This review of the existing literature on the use of ML for exploitation detection focuses on papers that use runtime traces from processors as signals. Research that uses network traffic, such as [18], [19], [20], [26] static file analysis, [21], [22], [23] is excluded from the scope of this literature review. Furthermore, research on the use of heuristic-based techniques is also excluded.

To the best of our knowledge, and up to the date of this paper, there are only 7 research papers on the topic of using ML for exploitation detection based on runtime traces [13], [14], [15], [16], [17], [24], [25]. Of those seven, three relied completely on hardware performance counters (HPC) as the signal source for the feature set [15], [25], [66]. Two of the seven relied on Intel's PTrace traces [17], [24]. One paper is based on Android-specific system events, [13] and another includes a combination of HPCs and independent micro-architectural features [16]. We segment the research into three categories, based on the source of datasets. Datasets that are entirely composed of HPCs are categorized as ''HPC based datasets''. While, datasets that are largely or purely based on Intel Ptrace are called ''Intel's PTrace based datasets''. Third category is based on datasets that are generated based on binary instrumentation, and this category is titled ''Binary instrumentation based datasets''.

A. HPC BASED DATASETS

In Omotosho et al. [15] the authors provide a technique for launching Return-Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on embedded systems that use an Xtensa processor. The Xtensa processor is widely used in the Internet of Things (IoT). The authors attempt to evaluate several HPC and identify those that are best to describe/provide patterns for detecting attacks. The authors created vulnerable programs and launched attacks. The final dataset created by the authors contained 6061 rows and 30 features. A Support Vector Machine (SVM) was used as the ML classifier in this study. The authors opted for the use of precision and recall as the main performance measurement indicators and reported results of up to 84% recall for some test instances and 100% precision for others.

Torres et al. [25] attempted to detect data-only attacks by using an SVM for anomaly detection. They used data from the HPC and a vulnerable application with heartbleed vulnerability to test their models. Single and multiclass SVMs were created, and single/multiple event thresholds were used as comparative techniques to evaluate the performance of the model. Although the authors claimed a detection accuracy of over 92% using a two-class SVM, they also concluded that HPCs have a limited ability to detect leakage attacks that are less than 8 KBs. The authors recommended that greater architectural support for HPCs be incorporated into Central Processor Units (CPUs) to enable researchers to capture more events with higher fidelity.

In Liu et al. [14] the authors targeted the detection of data attacks using a variety of algorithms trained on traces received from the HPCs. The authors set up a Nginx server with two vulnerabilities known as 'rootdir,' and 'keyleak.' Furthermore, a third platform is used for an encryption downgrade attack based on a vulnerability known as 'FREAK.' The authors trained models based on Linear Regression (LR), autoencoder (AE), Stacked Denoising Autoencoder (SDA) and Echo State Networks (ESN). Five datasets were created for each exploitation hltype. Each subset was segmented into 80%-20% training and testing. The class distribution in the test samples was 50% benign and 50% malicious, respectively. The classification results demonstrated that almost all classifiers had high accuracy rate 100% for 'FREAK' and 'rootdir' exploits. While the 'keyleak' exploit was the trickiest to detect with high accuracy across all algorithms. SDA and ESN both performed very well with 'keyleak' while the rest of the algorithms performed at considerably lower performance.

This hlstudy differs from the above in that it does not rely on HPCs. Although HPCs offer a convenient means of gathering runtime information, they do not provide a comprehensive view of hlthe runtime traces. HPCs are not uniformly defined across all CPU vendors even within the same vendor, different models may have different performance counters. Furthermore, the number of performance counters is usually limited by the processor type. In this study, all traces were based on debugging. Although not as practical as HPCs, the aim is not to provide what is practical, but to provide knowledge of how efficient frameworks should work. This is achieved through the wealth of information provided by debuggers and

the flexibility in curation (i.e., data preparation) techniques that can be applied.

B. INTEL'S PTRACE BASED DATASETS

In Chen et al. [17] The authors proposed HeNet. HeNet is a framework that uses packets from Intel's PTrace and applies an ensemble of deep learning to detect ROP attacks against Adobe Reader 9.3 on a Windows 7 32-bit platform. The authors used the Taken-not-Taken (TNT) and Target IP (TIP) packets from Intel's PTrace as the input source. This approach involves converting the trace of the TNT and TIP packets into a 1xD image. Subsequently, a low-level Deep Neural Network (DNN) model is applied. They used transfer learning and deep DNNs trained on large image datasets. The low-level DNN produces a probability for each stream of images. This probability is fed into an upper-layer ensemble model that aggregates the probabilities and applies a preconfigured threshold to identify what is considered malicious or benign. The HeNet model produced 98% accuracy and a 0.73% false-positive rate (FPR).

C. BINARY INSTRUMENTATION BASED DATASETS

In Elsabagh et al. [16] the authors proposed detecting ROP attacks using what they call EigenROP. EigenROP is an unsupervised anomaly-detection approach based on a novel algorithm developed by the authors. The algorithm maps the data into a high dimensional space, then extracts the principal components using Kernel Principal Component Analysis (KPCA). A representative direction was then extracted from all principal components during the learning phase. During the testing phase, the distance between the direction and the principal components of the testing sample was measured. If the measurement exceeded a certain threshold, the test sample was considered an anomaly.

Furthermore, the authors used signals based on independent micro-architectural characteristics, as well as HPCs. They calculated the Fisher score for a group of 15 characteristics to identify the ones to be used. Twelve Linux applications were used to train the model. Testing was performed on two applications: Linux Hex Editor version 2.0.20, and PHP version 5.3.6. The results demonstrated that EigenROP achieved an overall accuracy rate of an 81% (based on AUC), with up to 80% True Positive Rate (TPR) and 0.8% FPR. The authors also noted that their testing focused mainly on detecting the ROP gadgets only, without detecting the underlying vulnerabilities.

This study differs from others presented in [16] in several respects. First, we did not perform anomaly detection, but attempted to classify benign and malicious samples in a binary classification problem. Second, we steer away from unsupervised learning algorithms and focus on using algorithms that are more aligned with the explainable algorithms. Third, we did not include HPC and included instruction categories rather than actual instructions (as used in [16]). Fourth, we include traces that cover the entire shellcode of the payloads, not just ROP code gadgets, and we focus on buffer overflow attacks, which include ROP-based shellcodes in addition to other techniques.

Suarez-Tangil et al. [13] performed anomaly detection on Android Mediaserver components. The objective is to detect the exploitation of a vulnerability called ''Stagefright.'' The exploit targets the mediaserver, which is responsible for playing the media files in the Android OS. The experiment involved playing a large set of benign media files to establish the mediaserver's normal behavior. Several variations of the ''Stagefright'' exploits have been used for testing purposes. Suarez-Tangil used a dynamic instrumentation platform for Android, called CopperDroid. Anomaly detection was based on multivariate statistical network monitoring (MSNM) to extract events and features using principal component analysis (PCA) techniques.

This study differs from that of Suarez-Tangil et al. in several respects. This research is based on the Windows platform as opposed to the Android OS. Our work focuses on a class of vulnerabilities (i.e., buffer overflows) and not on a specific exploit. Hence, our work should be suitable for use across any user-space application that might be subject to exploits targeting such vulnerabilities. Finally, our work is based on the classification of malicious behavior, as opposed to anomaly detection.

IV. DATASET STRUCTURE

The dataset generation was based on the methodology and toolset presented in [48]. The exploits were downloaded from exploit-db.com and tested on Windows 7 OS. Each application was traced using tracer software, as described in [48]. The list of applications was further expanded in this study to cover all the applications listed in Table 1.

The data trace for each application in Table 1 comprises two files. One file is for the control flow, and the second file is for the memory traces. Both types of files were in JavaScript Object Notation (JSON) format. For the control-flow trace, each instruction traced is represented by a single JSON object that contains the elements in Fig. 1-a. A JSON object is present in the memory file for each instruction that performs the memory-altering operations. Memory altering instructions were classified into four categories. The first category is for instructions that cause a change in the stack but do not necessarily involve operations on memory content (e.g., call and return instructions). These instructions fit into the element ''stack_change,'' as sown in Fig. 2-b. The second category of memory-altering instructions are instructions that execute certain operations on the memory content (e.g., mov and add). These instructions fit into the element ''mem_change,'' as sown in Fig. 2-b.

The third category includes instructions that include a call instruction for a module that is not part of the list of modules within the AUT, such as system DLL and compiler modules. The fourth category pertains to the loops. When a loop is detected in a trace, the tracer iterates through the loop twice and skips it. Loop skipping is performed by creating breakpoints for all the possible branches that exit the loop.

{'AUT_modules', 'Debugged_Name', 'Debugged_path', 'action', 'breakpoints', 'call_target_module', 'command', 'count', 'current_aut', 'current_module', 'current_status', 'debug_messages', 'eax', 'ebp', 'ebx', 'ecx', 'edi', 'edx', 'eip', 'esi', 'esp', 'jmp_target', 'opCode', 'opCode_type', 'status', 'target_aut', 'target_module', 'target_verification', 'thread_Id', 'true_target_module'}

(a)

'current_module'	'current_aut'	'eip'	'esp'	'command'
String name of module	Boolean value	number	number	String command syntax

(b)

Kernel32.dll	Aut modules	System modules	Compiler modules	Data transfer	Binary arithmetic	logical	Shift Rotate	Bit Byte	Control Transfer	String	Flag Control	Misc.

(c)

FIGURE 1. Control Flow Dataset. (a) Raw data format (b) Selected data (c) CF Feature set.

{'mem_change', 'eip', 'stack_change', 'count', 'mem_dump'}

(a)

Stack mem change	Size of Stack mem change	Mem change	Size of Mem change	Memory dump change	Size of mem dump change

(b)

FIGURE 2. Memory Dataset. (a) Raw data format (b) Memory feature set.

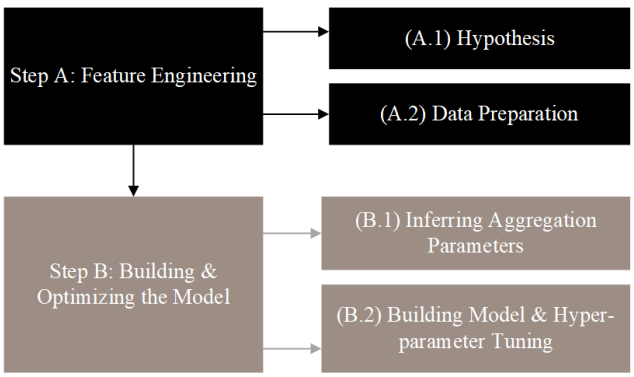

FIGURE 3. Overview of approach to build the ML.

For the first two categories, only the contents of the addresses were recorded in the trace file. For the latter two categories, a dump in the entire memory is captured in the element ''mem_dump,'' as shown in Fig. 2-b.

V. METHODOLOGY

The approach to building a machine learning model requires two main phases: feature engineering and building and tuning the ML model (see Fig. 3). The most important aspect of feature engineering is feature extraction, that is, identifying the feature space. Feature extraction is driven by they hypothesis of the most important features.

Data preparation involves performing the necessary computing/calculation/analysis steps on each group of rows and then aggregating the results into a specific feature. This intermediate step of performing aggregation is often overlooked in reporting of ML research when building and tuning models. We believe that this step affects the performance of the model and should be a part of the reporting of ML performance/tuning experiments.

Therefore, the first step in building an ML model is to infer the aggregation parameters most likely to yield high-performance models. The second step is to build actual models and perform hyperparameter tuning. We illustrate these steps in greater detail in Fig. 4, and detail each step in the following sections.

A. FEATURE ENGINEERING

1) HYPOTHESIS

To explain this hypothesis, an example of buffer overflow exploitation is first presented. For stack smashing exploits, the attacker would provide the program with a large input (larger than the buffer size) and would engineer the input such that when the current function returns, a specially crafted address would replace the current frame return address. This address is found somewhere within the buffer that the attacker supplies to the program. Usually, attackers would populate the exploit payload with NOPS (no operation). NOPS acts as both a buffer filler and at the same time a sledge for the jump address to accommodate minor changes in memory layouts from one machine to another.

Similarly, for SEHOP Overwrite, the attackers exceed the buffer until the SEHOP address is overwritten and an exception is raised. Subsequently, the SEHOP address points towards the attacker's payload. In examining this type of behavior, our hypothesis of the most important features is as follows:

- **EIP distance:** how much distance is travelled in terms of executable instructions.
- **ESP distance:** how much data were pushed or popped out of the stack. Distance travelled within the stack for either frame switching or for other reasons.
- **Instruction category:** How frequently are the data movement instructions? How frequently are control-flow instructions? The frequency of use of each instruction category was recorded during the learning process.
- **Instruction category:** How frequently are the data movement instructions? How frequently are the control-flow instructions? The frequency of each instruction category was recorded during the learning process.
- **Memory impact:** Does the instruction affect the memory? Did this cause a change in memory content? How large is the change in the memory content? What kind of memory change occurred in the stack or in other areas of memory?

We assume that the major obstacle to incorporating new technologies into production environments is the fear that new technologies would break what is currently running. Endpoint security with a high FPR would halt productivity. We also assume that the second highest priority is how good the solution is at detecting exploits. This position is also reciprocated by technical evaluators in the industry, such as av-comparatives.com [67] and av-tests [68].

Hence, FPR and recall are the main scoring metrics used to evaluate the best-performing models. The FPR is calculated as all false positive (FP) instances divided by the sum of FP and true negative (TN) instances. Recall, is calculated as all true positive (TP) instances divided by the sum of TP and false negative (FN) instances. Hence, the use of both FPR and recall involves all possible instances (TP, FN, FP and TN) of a model's decision. While the use of other metrics, such as recall and precision (which is calculated as TP/TP + FP), will not include all parameters, as it will exclude TN. Nevertheless, other metrics such as precision and accuracy, were also included to provide overall indicators of the health and viability of the model.

2) DATASET PREPARATION

As shown in Fig. 4, we elaborate on the preparatory steps in the following subsections.

a: CONTROL FLOW (CF) DATASET PREPARATION

The raw data selected for this model are shown in Fig.1-b. To use these as features, we expanded them to the list shown in Fig. 1-c. For each feature in this, a counter was used to represent the number of instructions that the feature was true. If no instructions are executed within that module, the counter is zero. The entire aggregated row is then labeled as either benign or malicious, as elaborated on in the following paragraphs.

b: EIP DISTANCE

This feature measures the distance covered in terms of the EIP addresses within a series of instructions aggregated in a window. For each instruction, its EIP address is evaluated if it is higher or lower than the address of the previous instruction. If it is higher, only the difference in the distance from the previous instruction is added to the EIP distance counter. If the distance was lower, the difference was subtracted.

c: ESP DISTANCE

The same calculations used for the EIP distance are applied to the ESP distance. The main difference is that the ESP distance increases with lower ESP values; hence, the opposite is performed. Instruction category: The categorization of the instructions was performed based on Intel Developer's guide [69]. The lists for each category were copied from the guide and used to automatically identify the category to which each instruction belonged. Subsequently, for each category, a feature column was created in the dataset. The number of instructions belonging to each category represents the final instruction count belonging to that category in each window. As ''Kernel32.dll'' is heavily utilized within the Windows API, it is treated as a separate category (i.e., a special case of instruction categories).

d: MODULE

The module to which the instruction belongs has four separate features. These are the Kernel32.dll'', ''AUT'', ''System modules'', and ''Compiler modules''. ''Kernel32.dll'' refers to the DLL library called KERNEL32.DLL. AUT modules refer to either the main AUT thread or other libraries or software modules indigenous to the AUT. The ''System modules'' category refers to all other DLLs other than KERNEL32.DLL, such as KERNELBASE.DLL, NTDDLL.DLL, OLE32.DLL among others. Finally, the ''Compiler modules'' refers to compiler created modules, such as MSVCR71.DLL, MSVCR80.DLL, EXPLORERFRAME.DLL among others.

e: MEMORY DATASET PREPARATION

To capture memory traces, a 'memory image' was first created. A 'memory image' is composed of a JSON file, where each single byte-level address within the memory is an element that contains a dictionary of instruction IDs. The instruction ID acts as a unique identifier for each execution instruction. Each instruction ID that resulted in a change in memory content was then included along with the new value of the memory after executing the instruction.

For each instruction ID, the 'memory image' file is checked. If there is a new value in, then it is mapped to the right category and the amount of change from the previous value is calculated. The features populated in the final dataset are the three types of memory changes, as shown in Fig. 2-b. For each type, there is a column to identify whether it was indeed the type of change. This is in addition to another column that captures the change in content size.

f: DATA LABELLING

We define malicious data as instructions that are part of the exploit payload and are executed by the CPU within the

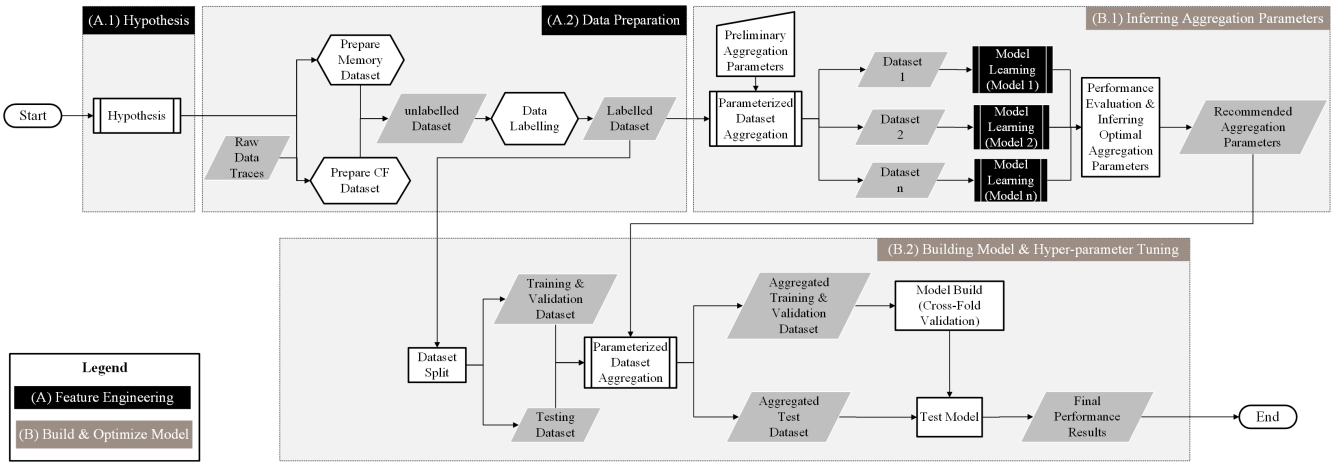

FIGURE 4. Flowchart of experimentation steps.

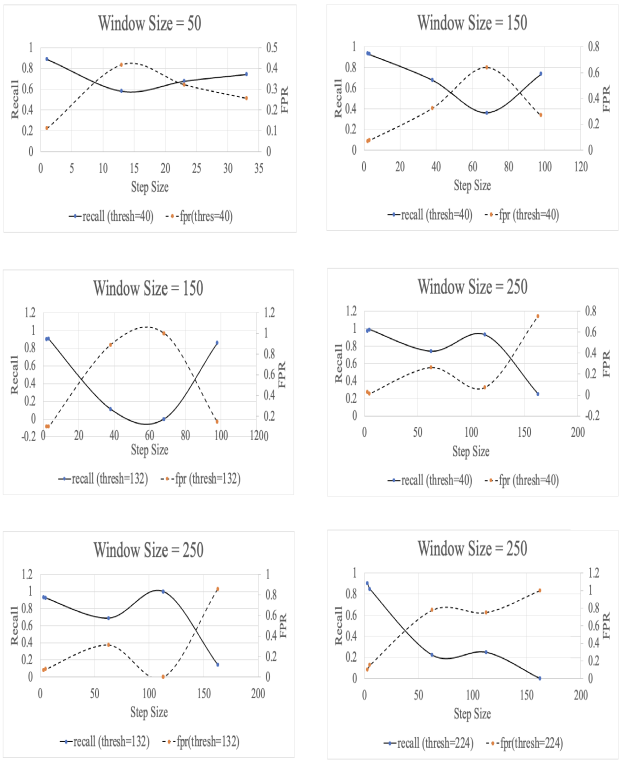

FIGURE 5. XGBoost Aggregation Parameters Performances.

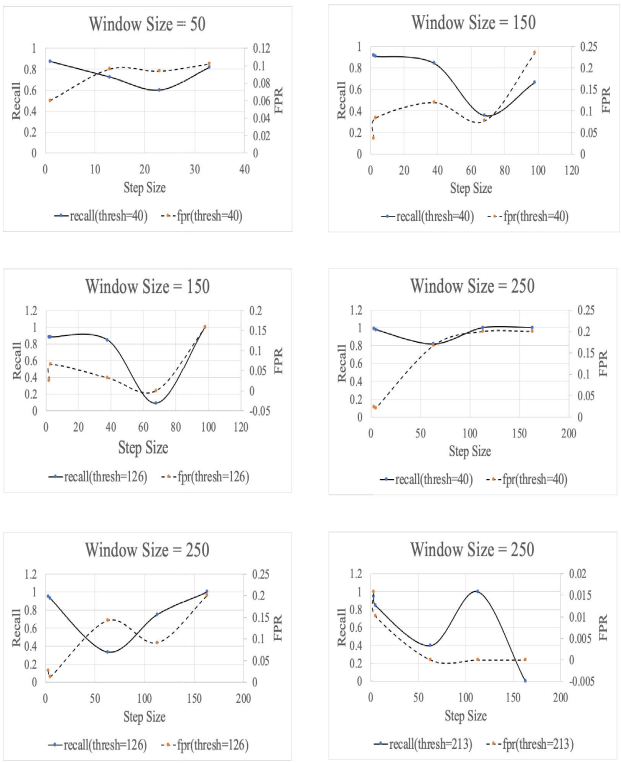

FIGURE 6. RF Aggregation Parameters Performances.

context of the AUT process. The exploits downloaded from exploit-db.com are in raw format, that is, they are composed of Python files. Each raw exploit file contains code to create a binary version of the exploit. Because this study focuses on buffer overflows, the binary version of the exploits is designed to be consumed by the AUT to fill a data variable (i.e., buffer). The exploit will result in overflowing the boundaries allocated to that variable and result in command execution of the process of choice. The exploit structure is composed of three components; shellcode, NOPS/junk, and a specific jump address to overwrite the return address/SEHOP address after overflowing. The shellcode contains opcodes for command execution. The opcodes were included in the Python file as hex digits. Junk bytes are used to help fill up the buffer and cause buffer overflow, and they are usually composed of random string values. The NOPS (represented by the opcode of hex byte 90) bytes provide the CPU with a sledge. The sledge's job is to act as a landing zone for the

last component of the exploit: the jump instruction. Sledges are needed because the program memory might differ slightly from one computer machine to another, and a single byte might perturb the exploitation. NOPS are used as sledges because they do not impact the state of the CPU nor the control flow. NOPS are placed in abundance to ensure that the target address of the jump instruction is somewhere in the middle of a sequence of NOPS. The shellcode immediately follows the NOPS sledge.

Therefore, to identify the first instruction executed within the payload (i.e., the first malicious instruction), a string search was conducted on the EIP field of each instruction in the raw trace files. The string search keyword was the jump address value in the Python file. Hence, the instruction that has an EIP address equivalent to the jump address is identified as the first instruction in payload execution. The unique ID of the instruction was recorded. The exploitation targets control flow hijacking and were executed linearly, that is, normal operation did not resume until the exploitation was fully executed. Hence, all the instructions executed after the first instruction are malicious until the last instruction in the exploit payload. To validate that we identified the start of the malicious payload, we compared the opcodes of a group of instructions that directly followed the first instruction with the hex codes that were at the start of the payload in the Python file. If they match, the first payload instruction is successfully identified.

To identify the last instruction in the stream, a search was conducted for the last opcode values in the payload (based on hex digits in the Python file). Once found in the raw trace file, the instruction ID was recorded. To validate that the instruction was the last instruction, we compared a group of opcodes before the last instruction with the hex values in the Python file. If matched, we record the instruction's unique ID in the trace. The results were further validated by converting the opcodes found in the Python file into an assembly language. The instructions were then compared with those found in the traces. Starting with a search from the first malicious instruction, each instruction is recorded until the search reaches the last malicious instruction. All instructions between the first and last were marked as malicious. The size of this task can be determined by reviewing Table 1. In Table 1, the column titled ''No. Trace Inst.'' represents the number of instructions to be searched for the jump address. Each instruction is composed of a JSON object, and hence, the search was only limited to the 'EIP' field. Then after the instruction was found, the field of 'opcode' is inspected versus the values found in the python file. The final series of instructions that were found between the first and the last instruction is documented in the column ''No. Payload Inst.'' The string search operations and comparisons of opcodes and instructions were done manually. Text editors were used to perform the searches on the trace files. Once the first and last instructions were identified, the process of marking was done programmatically in a Jupyter notebook.

We noted that there are optimizations that are performed by the CPU on the assembly instructions as they are being pre-fetched and executed. These optimizations modified the assembly instructions at certain segments of the payload execution. However, these optimizations did not impact the first nor the last group of instructions in the payload execution nor did they result in changes of the raw payload structure found in memory. These optimizations did not also perturb the execution of the exploit.

B. BUILD AND OPTIMIZE MODEL

1) INFERRING AGGREGATION PARAMETERS

When preparing the dataset, several factors affect the final dataset. These factors are as follows:

- **Window size:** refers to the number of instructions aggregated at any given time.
- **Step size (or overlapping):** refers to the percentage of overlapping instructions between consecutive windows. Step size refers to the number of instructions on which the window slides. The overlapping percentage refers to how many instructions are common between different windows. We used both terms interchangeably, noting that they have an inverse relationship with the same meaning.
- **Threshold:** The minimum number of payload-related instructions that classify a data row as malicious.

The experiment for inferring the best aggregation parameters begins by creating parameterized aggregation datasets. These datasets had different window sizes, thresholds, and step sizes (aggregation parameters). Each combination of the window size, threshold, and step size represents a unique and distinct dataset. Two classifiers (RF and XGBoost) were trained on each dataset. Scorings were collected for each set of aggregation parameters and analyzed. Guidelines for selecting aggregation parameters were then inferred and used to build the model.

2) BUILDING MODEL AND HYPER-PARAMETER TUNING

The first step was to identify a testing strategy. For each application category stated in Table 1 in the dataset (excluding 'Communication' category), an application will be selected for testing. This application was not part of the training and validation of the model. The objective was to identify the effectiveness of the model when trained on specific application categories to detect exploits in different categories. This also ensures that there is no leakage of the test data into training. This strategy also demonstrates the effectiveness of detecting techniques that might not be present in the training dataset, for example, return-oriented programming (ROP).

Four raw datasets were created by removing one application from the identified categories (please refer to sub-section VI-B and Table 2 for complete details of the datasets used in testing). Datasets were created based on the inferences made regarding the aggregation parameters. Then,

TABLE 2. Models test results.

								Random Forest				XGBoost			
Test Application	Window Size	Threshold	Step Size	Total dataset size	Malware Sample Size	Testing sample size	Malware size within test	ACC	PREC	RECALL	FPR	ACC	PREC	RECALL	FPR
Publish-It 3.6d	100	15	1	5108	1685	298	298	0.570	1.00	0.570	0.00	0.621	1.00	0.621	0.00
	100	80	1	5108	1192	298	298	0.564	1.00	0.564	0.00	0.047	1.00	0.047	0.00
	250	40	3	1239	505	50	50	1.00	1.00	1.00	0.00	0.960	1.00	0.960	0.00
	250	200	3	1239	213	50	50	0.600	1.00	0.600	0.00	0.640	1.00	0.640	0.00
Disk Pulse Enterprise 9.9.16	100	15	1	5303	1899	103	84	0.184	0.00	0.00	0.0	0.184	0.00	0.00	0.00
	100	80	1	5303	1471	103	19	0.816	0.00	0.00	0.0	0.816	0.00	0.00	0.00
	250	40	3	1288	554	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	250	200	3	1288	263	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
VUPlayer 2.49	100	15	1	5003	1614	403	369	0.797	1.00	0.778	0.0	0.893	1.00	0.883	0.00
	100	80	1	5003	1186	403	304	0.988	1.00	0.984	0.0	1.00	0.993	1.00	0.020
	250	40	3	1204	470	85	85	0.918	1.00	0.918	0.0	0.835	1.00	0.835	0.0
	250	200	3	1204	201	85	62	0.918	0.899	1.00	0.304	0.918	0.982	0.903	0.043
Easy AVI DivX Converter 1.2.24	100	15	1	4820	1709	586	274	0.584	1.00	0.109	0.00	0.532	0.00	0.00	0.00
	100	80	1	4820	1281	586	209	0.592	0.00	0.00	0.080	0.643	0.00	0.00	0.00
	250	40	3	1143	472	146	83	0.705	1.00	0.482	0.00	0.685	1.00	0.446	0.00
	250	200	3	1143	233	146	30	0.918	1.00	0.600	0.00	0.795	0.00	0.00	0.00

a grid search and cross-fold validation were performed to identify the best hyper-parameters. After tuning the model, testing was performed, and the scoring performance was recorded.

VI. EXPERIMENTS

This section elaborates how the methodology for ''Build & Optimize Model'' in section V is executed. The experiments were conducted on data prepared as elaborated in Section V (please refer to Fig. 1 and Fig. 2 for elaboration on the dataset structure). The objective of the experiments was twofold. First, we tested different window sizes, step sizes, and thresholds to infer the range of best values (Phase A as per the experimentation steps shown in Fig. 4). Second, different models are built based on the inferred recommendations for aggregation parameters. These models are evaluated primarily based on recall and FPR. Furthermore, we used the ML algorithm's ability to report on feature importance to compare feature usage among the two chosen models (Phase B as per Fig. 4).

A. INFERRING AGGREGATION PARAMTERS

1) EXPERIMENTAL DESIGN

Several datasets were created using different values for each aggregation parameter. The selected window sizes are 50, 150, and 250. Thresholds from the list of 40, 126, and 213 (each approximately 80-85% of the size of the corresponding window size) were applied. Each threshold was iterated across all window sizes, excluding thresholds that were larger than the window size. The reason for this exclusion is that it results in biased datasets that do not contain malicious samples. Window sizes were selected so that they did not exceed 50% of the largest application dataset. For each window size and threshold pair, an iteration over a list of step size percentages was performed, starting from 1%, 2%, 25%, 45%, and 65% (percentages of the window size). After creating a dataset for every set of aggregation parameters, the model was trained on each.

Each dataset was divided into 70% for model training and 30% for validation. The resulting total sample size, malicious sample size, accuracy, recall, precision, and FPR are recorded. Based on the recall and FPR results, the relationship between the different aggregation parameters and the scoring metrics can be mapped, and guidelines for selecting the aggregation parameters can be inferred.

2) EXPERIMENTAL RESULTS

As shown in Fig. 6 for the RF and Fig. 5 for the XGBoost graphs, there are two curves for each combination of the threshold and window size. The solid curve represents the recall, whereas the dotted curve represents the FPR. For each graph, there are two axes, one for the FPR and the other for recall. The graphs show how the recall/FPR changes as the step size increases for a specific window size and threshold pair.

It is important to note that FPR and recall graphs should be interpreted differently. Recall values were preferred to be high, whereas FPR values were preferred to be low. Hence, the ideal step size candidate for each graph would be at a point that represents the maximum recall curve and, at the same time, a minimum point in the FPR curve.

To better explain the different graphs presented in Fig. 6 and 5, two additional graphs (Fig. 7) were created to capture the overall pattern characteristics of the FPR and recall. Patterns were defined based on four characteristics. The first is the maximum value, which is the point that contains the largest value in the FPR/recall. Second, the minimum value is the lowest FPR/recall. The third is the local maximum, which is a point that has a visible curvature where the curve changes direction from increasing the value of FPR/recall to decreasing the value. Finally, is the local minimum, and these are the inflection points where the curve moves from

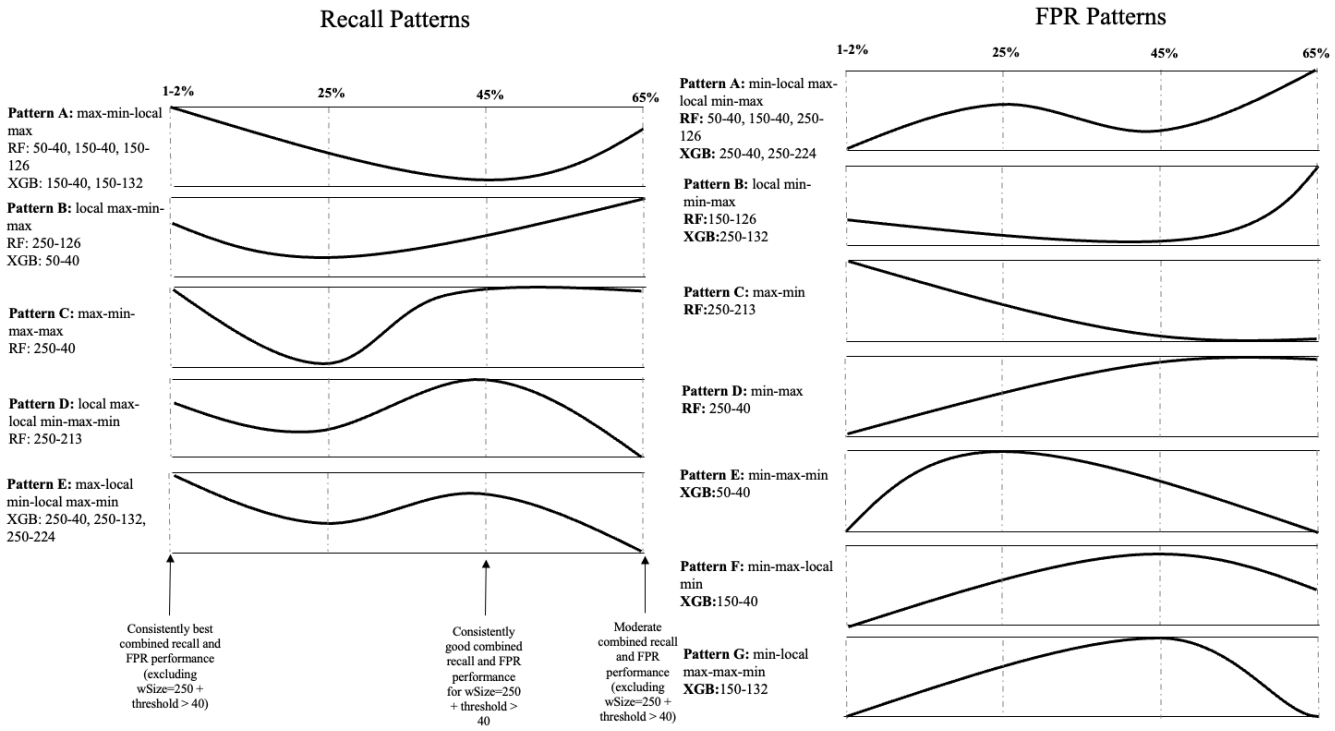

FIGURE 7. Overview of Recal/FPR Patterns.

decreasing values for the recall/FPR to increasing values. Our key points were measured at each step size percentage marker. Graphs that are part of a specific pattern are labelled using window size threshold pairs (i.e., window size = 50 and threshold = 40 are represented as 50-40). The reader can check the actual performance graphs in Fig. 5 and 6 by looking for the specific window size threshold numbers, noting that the pattern graphs are approximations and are not intended to provide highly accurate mapping.

Patterns within the recall graphs across RF and XGBoost were categorized into five generic patterns. As shown in Fig. 7, recall pattern "A" contains the largest number of datasets (3 in the RF and 2 in the XGB), followed by pattern "E" with 3 datasets in XGB and then pattern "B" with one dataset from each ensemble. The remaining patterns contain a single dataset. As the figure demonstrates, the highest recall value is obtained with a step size of 1-2% of the window size. As the step size increased, the recall decreased until a step size of 25% or 45%, after which it increased again. For Patterns D and E represent an exception. The exception is that they decrease again after the 45% mark.

Fig. 7 also contains eight FPR patterns. The largest pattern is pattern "A" which contains 5 datasets (3 from RF and 2 from XGB). As shown in the FPR patterns, most graphs start at step size of 1-2% with a very low FPR and increase at either 25% or 45% before decreasing again at 65% (but not as low as the values at–1-2%). Two patterns represent the exception. Pattern B and Pattern C. Pattern B the FPR at 1-2% is low but not the lowest. The FPR decreases until the 45% mark then it starts increasing. Pattern C the 1-2% has the highest FPR which decreases until the 65% mark. Given that the best results are those with the highest recall and lowest FPR, we conclude that most datasets with small/medium window sizes (i.e., < 250) would produce the best results with high overlapping of the windows (i.e., step size 1-2%). Datasets with a high window size (>= 250) will most likely produce good results with high overlapping (i.e., step size of 1-2%) and 50% overlapping percentage (i.e., 50% step size).

B. BUILDING AND TUNING MODEL

1) EXPERIMENT DESIGN

Based on the guidelines discussed in the previous section, we trained our model using two window sizes: 100 and 250. We applied thresholds of 20 and 80 for a window size of 100. For a window size of 250, thresholds of 50 and 200 were applied. The step size percentage was fixed at 1% for both aspects of the experiment, as it has consistently proven to be the best overlap percentage.

2) EXPERIMENTAL RESULTS

A complete list of the results is presented in Table 2. We started with the 'Citation Management' category and set aside 'Publish-It 3.6d'. The best RF and XGBoost results were achieved with a window size of 250 and threshold of 40. These parameters resulted in 100% recall and 0% FPR in RF, and 96% recall and 0% FPR in XGBoost. We note that the accuracy in the RF test was 100%, whereas that in the

TABLE 3. Top 5 features by random forest.

Test Application	Window Size	Threshold	Top 5 Features by RF					Aggregated Impact
			1	2	3	4	5	
36104_Publish-It 3.6d - Local Buffer Overflow (SEH)	100	15	esp_distance	eip_distance	bit_byte	string	aut_modules	0.608446161
	100	80	esp_distance	eip_distance	bit_byte	string	misc	0.596516309
	250	40	esp_distance	string	shift_rotate	bit_byte	kernel32.dll	0.471340564
	250	200	bit_byte	control_transfer	esp_distance	shift_rotate	binary_arithmetic	0.520393923
42536_Disk Pulse Enterprise 9.9.16 - 'Import Command' Local Buffer Overflow	100	15	esp_distance	eip_distance	bit_byte	string	shift_rotate	0.642203928
	100	80	esp_distance	eip_distance	bit_byte	string	shift_rotate	0.608523731
40018_VUPlayer 2.49 (Windows 7) - .m3u Local Buffer Overflow (DEP Bypass)	100	15	eip_distance	string	esp_distance	bit_byte	shift_rotate	0.65158037
	100	80	esp_distance	eip_distance	string	bit_byte	misc	0.608042843
	250	40	esp_distance	shift_rotate	flag_control	string	misc	0.486198522
	250	200	bit_byte	control_transfer	esp_distance	string	shift_rotate	0.630999307
42549_Easy AVI DivX Converter 1.2.24 - Local Buffer Overflow (SEH)	100	15	esp_distance	eip_distance	bit_byte	control_transfer	string	0.689475008
	100	80	esp_distance	eip_distance	string	bit_byte	misc	0.608042843
	250	40	esp_distance	eip_distance	flag_control	shift_rotate	bit_byte	0.520727088
	250	200	bit_byte	control_transfer	esp_distance	string	shift_rotate	0.630999307

XGBoost was 96%. However, the dataset that produced these results was composed of only malicious samples. Moreover, the accuracy and precision were 100% for RF and 96% and 100% for XGBoost, respectively. The second category is 'Disk Management', from which 'Disk Pulse Enterprise 9.9.16' was selected as the testing app. For a window size of 100, and for both thresholds of 15 and 80, the recall and FPR were 0.0%. For threshold 15, the accuracy was 18%, and for threshold 80, the accuracy was 81%. For a window size of 250 the samples were too small to be tested. Therefore, it is not possible to aggregate these applications for these parameters.

For the 'Media Player' category, 'VUPlayer 2.49' was used as the testing app. In RF, this testing app had 0% FPR for all aggregation parameters except for window size 250 and threshold 200, which resulted in a 30% FPR. Furthermore, for the window sizes of 100 and 80, the recall was 98%. Although the testing results of window size 250 and threshold 200 produced 100% recall, we elected to use the test results of window size 100 and threshold 80 as the best results in this category. This is because, for window sizes of 100 and 80, the accuracy is 98.76%, which is 7% higher than that of window size 250 and threshold 200. Furthermore, the FPR for window sizes of 100 and 80 was 0%, compared to 30.43% for window sizes of 250 and 200.

Finally, for the 'Media Converter' category, 'Easy AVI DivX Converter 1.2.24' was selected as the testing application. The recorded FPR was 0% for all traces, except for window size 100 and threshold 80, which were produced in RF 7.9%. Moreover, the highest recall across all datasets was 60% for RF with a window size of 250 and threshold of 200. The highest recall for XGBoost was 44.58% for a window size of 250 and threshold of 40.

3) FEATURE IMPORTANCE

It is worth noting that the performance of the training set when it does not contain any exploit technique other than SEHOP overwrite and stack smashing is still capable of accurately detecting ROP-based attacks, as demonstrated in the testing of app 'VUPlayer 2.49'. Another point to note is that testing on 'Disk Pulse Enterprise 9.9.16' produced the worst results across all testing datasets followed by testing on ''Easy AVI DivX Converter 1.2.24''. Finally, it can be seen from the overall results that the window size and threshold combinations that perform the best in RF also perform the best in XGBoost with one exception. This exception is in the testing of ''Easy AVI DivX Converter 1.2.24', where the best results were indeed in the same window size of 250, but different thresholds. This further highlights the importance of tuning the selection of the window size and threshold.

The models described above are based on datasets that contain 21 features, 15 relating to instruction and module categorization, while memory metadata are presented by six features (please refer to Fig. 1 and 2). The RF and XGBoost algorithms provide an option for reporting feature importance. The importance rating for each feature was reported as a fraction smaller than or equal to one. The combined value of all feature importance measurements reported by each model was up to 1.

The top five features for each of the testing datasets are reported in Table 3 (for RF) and Table 4 (for XGBoost). In addition, Fig. 8 depicts the average importance rating for each feature as reported by each algorithm. As shown in Fig. 8, each algorithm selects a different set of features to be the most important. In XGBoost, the highest average of important features is instructions that related to categories ''shift_rotate'' and ''bit_byte.'' The ''shift_rotate'' and ''bit_byte'' refers to ''Shift and Rotate,'' and to ''Bit and Byte'' instructions categories respectively as identified in Intel Developer's guide [69]. In RF, the highest average importance is ''eip_distance'' and ''esp_distance'' and thirdly ''bit_byte'' instruction categories.

As shown in Table 4, the 'esp_distance' feature is the most important feature in RF, occurring in the top five across all

TABLE 4. Top 5 features by XGBoost.

Test Application	Window Size	Threshold	Top 5 Features by XGBoost					Aggregated Impact
			1	2	3	4	5	
36104_Publish-It 3.6d - Local Buffer Overflow (SEH)	100	15	kernel32.dll	aut_modules	system_modules	compiler_modules	eip_distance	0.689429983
	100	80	system_modules	bit_byte	kernel32.dll	binary_arithmetic	data_transfer	0.723595388
	250	40	system_modules	flag_control	control_transfer	compiler_modules	string	0.604715295
	250	200	kernel32.dll	data_transfer	bit_byte	control_transfer	flag_control	0.671391897
42536_Disk Pulse Enterprise 9.9.16 - 'Import Command' Local Buffer Overflow	100	15	system_modules	flag_control	aut_modules	eip_distance	kernel32.dll	0.568424568
	100	80	system_modules	kernel32.dll	compiler_modules	eip_distance	esp_distance	0.595124137
40018_VUPlayer 2.49 (Windows 7) - .m3u Local Buffer Overflow (DEP Bypass)	100	15	system_modules	aut_modules	compiler_modules	control_transfer	eip_distance	0.756305698
	100	80	kernel32.dll	esp_distance	compiler_modules	eip_distance	control_transfer	0.674772635
	250	40	flag_control	system_modules	kernel32.dll	control_transfer	string	0.777830083
	250	200	kernel32.dll	compiler_modules	flag_control	system_modules	binary_arithmetic	0.567752525
42549_Easy AVI DivX Converter 1.2.24 - Local Buffer Overflow (SEH)	100	15	kernel32.dll	aut_modules	eip_distance	esp_distance	data_transfer	0.640627149
	100	80	system_modules	kernel32.dll	aut_modules	binary_arithmetic	control_transfer	0.716815941
	250	40	flag_control	system_modules	control_transfer	kernel32.dll	string	0.64413467
	250	200	kernel32.dll	binary_arithmetic	eip_distance	system_modules	compiler_modules	0.645647749

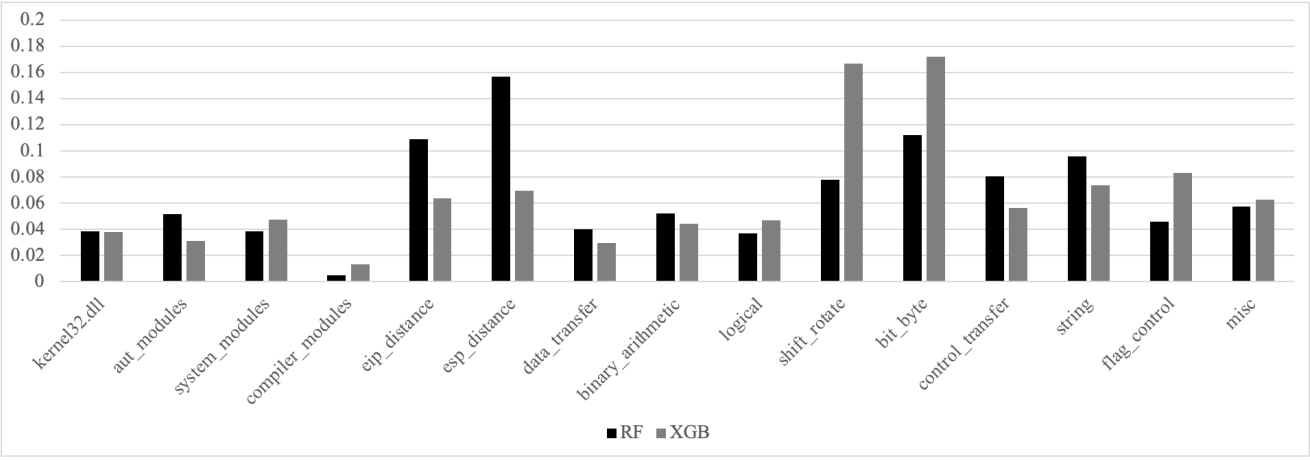

FIGURE 8. Comparison of Feature Importance across all tests (by average value).

the datasets. Followed by 'bit_byte' appearing in 15 out of the 16 datasets within the top five. The third most common is the 'string' (stands for ''String Instructions'' category), which appeared 14 out of 16 times. Following is 'shift_rotate' (stands for ''Shift and Rotate Instructions'' category) and 'eip_distance' each appearing 11 and 5 times respectively.

While for the XGBoost, the most common feature among the top 5 is 'kernel32.dll' which appeared 12 times out of 14. Second most common feature is 'system_modules' which appeared 10 times out of 14. Finally, 'compiler_modules' and 'eip_distance' are both the third most common feature where each appeared 7 times out of 14. Interestingly, the top three within the RF test results appeared within the top five of all datasets, but with a low frequency. The top feature in RF, 'esp_distance,' appeared only twice among the top five in XGBoost. While 'bit_byte' and 'string' each appeared 2 and 3 times respectively. Furthermore, by reviewing the data in Table 3 and Table 4, we note that there are 12 unique features among the top five features of XGBoost, whereas in RF, there are 11 unique features. As shown in Fig. 9, from the 12 unique features in XGBoost, three did not appear in the list of the top five in any of the datasets in RF. Of the 11 unique features of RF, two do not appear in any of the top five of XGBoost. Most notable in both RF and XGBoost is the complete disregard of memory-related features (except for 'eip_distance' and 'esp_distance'). The reported feature importance for the memory features depicted in Fig. 2 is zero across all the datasets. This is in complete contrast to other signature-based techniques that utilize the contents of memory heavily, and it further shows that memory metadata do not contain enough entropy to be used for detection.

VII. KEY TAKEAWAYS

Below is a summary of the key takeways from the above results:

- **Takeaway (1):** *Data aggregation parameters affects models' performance more than hyper-parameter tuning.*
  As shown in Fig. 6 and 5, good aggregation parameters can provide performance results across recall and FPR

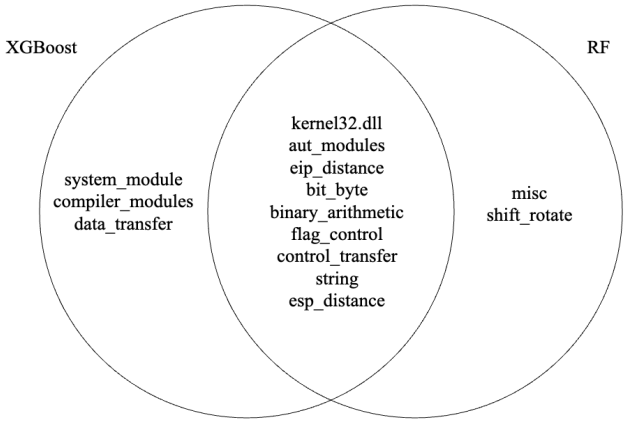

FIGURE 9. Venn Diagram of Unique Top 5 Features in RF and XGBoost.

that are of acceptable quality (i.e., recall >= 90% and FPR <= 10%).

- **Takeaway (2):** *Good data aggregation parameters perform equally well across different classifiers (and vice versa).*
  As shown in the testing results table (Table 2), good performing aggregation parameters performed well for both XGBoost and RF (except for the 'Easy AVI DivX Converter 1.2.24,' where the threshold differed). Similarly, the aggregation parameters produced poor results.
- **Takeaway (3):** *Training on specific vulnerabilities enables detection of never-before-seen techniques.*
  As shown in Table 1, the application ''VUPlayer 2.49'' is the only application in the dataset exploited using an ROP chain. However, training a model on the dataset excluding this application still enabled the model to detect exploitation traces with high recall (98.36% in RF and 100% in XGB) along with low FPR (0% in RF and 2.02% in XGB).
- **Takeaway (4):** *Feature engineering is model specific.*
  As shown in Table 3 and Table 4, the top 5 important features differed significantly between RF and XGBoost. Furthermore, the importance of the common features shown in Fig. 9 varies significantly, as demonstrated by the bar charts in Fig. 8.
- **Takeaway (5):** *Memory metadata does not provide enough information for exploitation detection.*
  As elaborated in the feature importance discussion, the memory features shown in Fig. 2 had zero importance assigned to them by both models.

VIII. RESEARCH LIMITATIONS AND FUTURE WORK

There are several limitations in this research that could be addressed in future research.

- Robustness to adversarial attacks: How can these models be evaluated against adversarial attacks and mitigated against adversarial attacks? Further evaluation of adversarial attacks is required to validate the effectiveness of each model or ensemble of models.
- The dataset is based on single threaded applications or applications that contain all vulnerable codes within a single thread. Would the models produce similar results if the dataset containes vulnerable code (or exploit execution) across two threads?
- The dataset is based only on buffer overflow vulnerabilities. How do we create datasets with greater variance in the targeted vulnerabilities? Capturing runtime traces of software under exploitation is a fragile process because exploits are easily perturbed by code instrumentation at any stage; hence, it is a trial and error process.
- Evaluation of important features with exploits targeting other vulnerability types. Would the same sets of features be used for the same models or would they differ? How large is the difference?
- ML learning is based on the execution of buffer-overflow exploits that employ shellcode for command execution. What will be the model's behavior if the payloads are changed to other techniques such as reverse shell, TCP binding or other techniques? Given that maliciousness is based on the start of the actual exploit payload, a change in payload would result in circumventing the model. Hence, there is a need to expand the dataset further to include other payloads.
- Identifying a program tracing framework that would be sufficiently light, and at the same time, can capture the most important features for multiple ML models. This enables prototyping of this solution using an ensemble that focuses on different mixture of features important in defense against adversarial attacks.
- Current labelling process is manual. In order to expand the number of exploitation techniques and targetting vulnerabilities it will be difficult to use manual labelling techniques. Therefore, unsupervised and semi-supervised algorithms should be explored to address this issue. In addition, increasing number of exploitations and vulnerabilities will also increase the feature dimensions. Therefore, feature selection and dimensionality reduction techniques should be explored in future work.

IX. CONCLUSION

We demonstrated that runtime traces based on assembly instructions present sufficient entropy for RF and XGBoost classifiers to detect the exploitation activity. We provide guidelines for the data preparation of runtime traces along with the recommended aggregation parameters based on the available dataset. We noted differences in feature importance across a range of 21 features. Furthermore, we provide a comparative analysis of the differences in the feature importance between the models. We demonstrated that meta-information about memory changes does not contain enough entropy to enable an ML model to detect exploitative activities. We also demonstrate that it is possible to detect ROP-based attacks

that target buffer overflow vulnerabilities without using ROP gadgets in the training dataset.

REFERENCES

[1] Digital Security Unit, "Special report: Ukraine," One Microsoft Way, Redmond, WA, USA, Tech. Rep., 2022.
[2] P. Kumar, N. Chowdary, and A. Mathuria, "Alphanumeric shellcode generator for ARM architecture," in *Proc. Int. Conf. Secur., Privacy, Appl. Cryptogr. Eng.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 8204, 2013, pp. 38–39.
[3] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Impact of code obfuscation on Android malware detection based on static and dynamic analysis," in *Proc. 4th Int. Conf. Inf. Syst. Secur. Privacy (ICISSP)*, Jan. 2018, pp. 379–385.
[4] S. Bhatkar, R. Sekar, and C. D. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th USENIX Secur. Symp.*, 2005, pp. 255–270.
[5] *Data Execution Prevention—Win32 Apps | Microsoft Docs*, Microsoft. Accessed: May 27, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention
[6] *Bypassing DEP With VirtualProtect (x86). Vulndev*, Vulendev. Accessed: May 27, 2023. [Online]. Available: https://vulndev.io/2022/06/14/bypassing-dep-with-virtualalloc-x86/
[7] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, Mar. 2011, pp. 40–51.
[8] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-call oriented programming (PCOP): Chaining the gadgets using call instructions," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 2, pp. 139–156, May 2018.
[9] *What is the Cyber Kill Chain? Introduction Guide | CrowdStrike*, CrowdStrike, Sunnyvale, CA, USA. Accessed: Mar. 6, 2023. [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/cyber-kill-chain/
[10] L. Martin, *Cyber Kill Chain®*, Lockheed Martin, Rockledge, MD, USA. Accessed: Jun. 27, 2022. [Online]. Available: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html
[11] *Gaining the Advantage—Applying Cyber Kill Chain Methodology to Network Defense*, Lockheed Martin Corporation, Bethesda, MD, USA, 2015, pp. 1–13.
[12] *What is an Exploit?—Cisco*, San Jose, CA, USA. Accessed: Feb. 28, 2023. [Online]. Available: https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-exploit.html
[13] G. Suárez-Tangil, S. K. Dash, P. García-Teodoro, J. Camacho, and L. Cavallaro, "Anomaly-based exploratory analysis and detection of exploits in Android mediaserver," *IET Inf. Secur.*, vol. 12, no. 5, pp. 1–10, 2018.
[14] C. Liu, Z. Yang, Z. Blasingame, G. Torres, and J. Bruska, "Detecting data exploits using low-level hardware information: A short time series approach," in *Proc. 1st Workshop Radical Experiential Secur. (ASIA CCS)*, May 2018, pp. 41–47.
[15] A. Omotosho, G. B. Welearegai, and C. Hammer, "Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2022, pp. 510–519.
[16] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou, "Detecting ROP with statistical learning of program characteristics," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, New York, NY, USA, Mar. 2017, pp. 219–226.
[17] L. Chen, S. Sultana, and R. Sahita, "HeNet: A deep learning approach on Intel processor trace for effective exploit detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 109–115.
[18] M. Masud, L. Khan, B. Thuraisingham, X. Wang, P. Liu, and S. Zhu, "Detecting remote exploits using data mining," in *Proc. IFIP Int. Conf. Digit. Forensics*, in IFIP—The International Federation for Information Processing, vol. 285, 2008, pp. 177–189.
[19] G. Yang, X. Liu, and C. Tang, "Horus: An effective and reliable framework for code-reuse exploits detection in data stream," *Electronics*, vol. 11, no. 20, p. 3363, Oct. 2022.
[20] X. Li, Z. Hu, Y. Fu, P. Chen, M. Zhu, and P. Liu, "ROPNN: Detection of ROP payloads using deep neural networks," 2018, *arXiv:1807.11110*.
[21] X. Zhou and J. Pang, "Expdf: Exploits detection system based on machine-learning," *Int. J. Comput. Intell. Syst.*, vol. 12, no. 2, pp. 1019–1028, 2019.
[22] H. Wang and P. Liu, "Tackling imbalanced data in cybersecurity with transfer learning: A case with ROP payload detection," May 2021, *arXiv:2105.02996*.
[23] S. Yoo, S. Kim, and B. B. Kang, "The image game: Exploit kit detection based on recursive convolutional neural networks," *IEEE Access*, vol. 8, pp. 18808–18821, 2020.
[24] L. Cheng, D. Yao, and G. Wang, "Program anomaly detection against data-oriented attacks," Ph.D. dissertation, Faculty Viriginia Polytech. Inst. State Univ., Blacksburg, VA, USA, Jul. 26, 2018.
[25] G. Torres and C. Liu, "Can data-only exploits be detected at runtime using hardware events? A case study of the Heartbleed vulnerability," in *Proc. Hardw. Architectural Support Secur. Privacy*, 2016, pp. 1–7, doi: 10.1145/2948618.2948620.
[26] S. Harnmetta and S. Ngamsuriyaroj, "Classification of exploit-kit behaviors via machine learning approach," in *Proc. 20th Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2018, pp. 468–473.
[27] *Desktop Operating System Market Share Worldwide | Statcounter Global Stats*, Statcounter GlobalStats, Dec. 2022. [Online]. Available: https://gs.statcounter.com/os-market-share/desktop/worldwide
[28] *CWE—2022 CWE Top 25 Most Dangerous Software Weaknesses*, MITRE, McLean, VA, USA, 2022.
[29] *NVD—Search and Statistics*. Accessed: Dec. 22, 2022. [Online]. Available: https://nvd.nist.gov/vuln/search
[30] I. D. Mienye and Y. Sun, "A survey of ensemble learning: Concepts, algorithms, applications, and prospects," *IEEE Access*, vol. 10, pp. 99129–99149, 2022, doi: 10.1109/ACCESS.2022.3207287.
[31] L. Breiman, "Random forests," *J. Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[32] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, vols. 13–17, Aug. 2016, pp. 785–794.
[33] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *Proc. IEEE 3rd Int. Conf. Data Sci. Cyberspace (DSC)*, Jun. 2018, pp. 153–160.
[34] I. Ahmad, M. Basheri, M. J. Iqbal, and A. Rahim, "Performance comparison of support vector machine, random forest, and extreme learning machine for intrusion detection," *IEEE Access*, vol. 6, pp. 33789–33795, 2018.
[35] P. A. A. Resende and A. C. Drummond, "A survey of random forest based methods for intrusion detection systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–36, 2019.
[36] S. Dhaliwal, A.-A. Nahid, and R. Abbas, "Effective intrusion detection system using XGBoost," *Information*, vol. 9, no. 7, p. 149, Jun. 2018.
[37] W. Haider, G. Creech, Y. Xie, and J. Hu, "Windows based data sets for evaluation of robustness of host based intrusion detection systems (IDS) to zero-day and stealth attacks," *Future Internet*, vol. 8, no. 4, p. 29, Jul. 2016.
[38] G. Creech, "Developing a high-accuracy cross platform host-based intrusion detection system capable of reliably detecting zero-day attacks," Ph.D. thesis, Univ. New South Wales, Sydney, NSW, Australia, 2014.
[39] N. Moustafa and J. Slay, "UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proc. Mil. Commun. Inf. Syst. Conf. (MilCIS)*, Nov. 2015, pp. 1–6.
[40] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proc. 4th Int. Conf. Inf. Syst. Secur. Privacy (ICISSP)*, 2018, pp. 108–116.
[41] W. Haider, J. Hu, J. Slay, B. P. Turnbull, and Y. Xie, "Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling," *J. Netw. Comput. Appl.*, vol. 87, pp. 185–192, Jun. 2017.
[42] *Malware—Wikipedia*. Accessed: Mar. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Malware
[43] L. Wu, R. Ping, L. Ke, L. Xing, W. Jian-Ping, and L. Ke, "Analysis and forensics for behavior characteristics of malware in Internet," in *Proc. IEEE Int. Conf. Digit. Signal Process. (DSP)*, Oct. 2016, pp. 545–549.
[44] *Generating Payloads—Metasploit Unleashed*, Offensive Secur. Accessed: Jun. 24, 2022. [Online]. Available: https://www.offsec.com/metasploit-unleashed/generating-payloads/
[45] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, Dec. 2019.
[46] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. Chen, "A survey of intrusion detection systems leveraging host data," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–35, Nov. 2020.

[47] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–36, Sep. 2019.
[48] A. Youssef, M. Abdelrazek, C. Karmakar, and Z. Baig, *Tracing Software Exploitation* (Lecture Notes in Computer Science), vol. 13041. Springer, 2021.
[49] H. Tang, S. Huang, Y. Li, and L. Bao, "Dynamic taint analysis for vulnerability exploits detection," in *Proc. 2nd Int. Conf. Comput. Eng. Technol. (ICCET)*, vol. 2, 2010, pp. 215–218.
[50] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020.
[51] *IEEE Standard Glossary of Software Engineering Terminology*, Standard 610.12-1990, Institute of Electrical and Electronics Engineers (IEEE), 1990.
[52] H. Hanif, M. H. N. M. Nasir, M. F. A. Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *J. Netw. Comput. Appl.*, vol. 179, Apr. 2021, Art. no. 103009.
[53] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *Proc. Int. Conf. Availability, Rel. Secur. (ARES)*, Sep. 2013, pp. 446–454.
[54] A. Ibrahim, M. El-Ramly, and A. Badr, "Beware of the vulnerability! How vulnerable are GitHub's most popular PHP applications? in *Proc. 16th ACS/IEEE Int. Conf. Comput. Syst. Appl. (AICCSA)*, Nov. 2019, pp. 1–7.
[55] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.
[56] A. J. Harer, Y. L. Kim, L. R. Russell, O. Ozdemir, R. L. Kosta, A. Rangamani, H. Lei Hamilton, I. G. Centeno, R. J. Key, M. P. Ellingwood, E. Antelman, A. Mackay, W. M. McConley, M. J. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," Feb. 2018, *arXiv:1803.04497*.
[57] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, May 2012.
[58] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.
[59] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020.
[60] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2329–2344.
[61] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 474–484.
[62] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 725–741.
[63] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2005, pp. 1–14.
[64] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23rd USENIX Conf. Secur. Symp.*, 2014, p. 256.
[65] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, Oct. 2010, p. 559.
[66] C. Liu, Z. Yang, Z. Blasingame, G. Torres, and J. Bruska, "Detecting data exploits using low-level hardware information: A short time series approach," in *Proc. 1st Workshop Radical Experiential Secur. (ASIA CCS)*, May 2018, pp. 41–47.
[67] *Summary Report 2022—AV-Comparatives*, AV Comparatives, Grabenweg, Austria, Jan. 2023.
[68] *AV-TEST Seal of Approval | AV-TEST Institute*. Accessed: Mar. 12, 2023. [Online]. Available: https://www.av-test.org/en/about-the-institute/certification/
[69] *Intel® 64 and IA-32 Architectures Software Developer Manuals*, Intel Corp., Santa Clara, CA, USA, Mar. 2023.

AYMAN YOUSSEF received the Master of Science degree in information security from Nile University, in 2018. He is currently pursuing the Ph.D. degree with Deakin University, Australia. He has more than ten years of practical experience working as a security consultant in several regions around the globe. His research interests include software and computer security.

MOHAMED ABDELRAZEK is currently an Associate Professor in software engineering and the IoT with Deakin University, Australia. Before joining Deakin University, Australia, in 2015, he was a Senior Research Fellow with the Swinburne University of Technology, Australia, and the Swinburne-NICTA Software Innovation Laboratory (SSIL). Before 2010, he was the Head of the Software Development Department, Microtech. For more information visit the link (https://sites.google.com/site/mohamedalmorsy/).

CHANDAN KARMAKAR (Member, IEEE) received the B.Sc.Eng. degree in computer science and engineering from the Shahjalal University of Science and Technology, Sylhet, Bangladesh, in 1999, and the Ph.D. degree from The University of Melbourne, Melbourne, VIC, Australia, in 2016. He joined the School of Information Technology, Deakin University, Geelong, VIC, Australia, in 2018, as a Lecturer. He has published one book and more than 170 research articles, including 82 journal articles. His research interests include biomedical devices and signal processing, cardiovascular and neural systems related to sleep-disordered breathing, human gait dysfunctions, cardiovascular diseases, and diabetic autonomic neuropathy.