IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 6, 2022, accepted April 19, 2022, date of publication April 25, 2022, date of current version May 3, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3170479

Identification of Return-Oriented Programming Attacks Using RISC-V Instruction Trace Data

DANIEL F. KORANEK, SCOTT R. GRAHAM, (Senior Member, IEEE), BRETT J. BORGHETTI, AND WAYNE C. HENRY, (Member, IEEE)

Graduate School of Engineering and Management, Air Force Institute of Technology, WPAFB, Dayton, OH 45433, USA

Corresponding author: Daniel F. Koranek (daniel.koranek@us.af.mil)

ABSTRACT An increasing number of embedded systems include dedicated neural hardware. To benefit from this specialized hardware, deep learning techniques to discover malware on embedded systems are needed. This effort evaluated candidate machine learning detection techniques for distinguishing exploited from non-exploited RISC-V program behavior using execution traces. We first developed a dataset of execution traces containing Return Oriented Programming (ROP) exploitation on the RISC-V Instruction Set Architecture (ISA) and then developed several deep learning bidirectional Long Short-Term Memory (LSTM) models capable of distinguishing exploited traces from non-exploited traces, each using subsets of features from the execution traces. An objective of this effort was to evaluate which features (instruction addresses and immediate values) from an execution trace are application-specific, which features (opcodes and operands) are application-agnostic, and how these subsets of features affect model performance. Application-agnostic features allow a model to generalize its detection capability to detecting ROP in previously unseen applications. The model using opcode and operand sequences obtained 98.21% cross validation accuracy and 97.94% test accuracy. In contrast, a model using address values obtained 92.79% cross validation accuracy with 99.59% test set accuracy. This research also analyzed whether ROPs exploitation significantly affects branch prediction; experimental evidence suggests that it does. Thus, branch prediction behavior could be a valuable feature in detecting ROPs exploits.

INDEX TERMS Computational and artificial intelligence, recurrent neural networks, computer viruses, embedded software, computer architecture, RISC-V.

I. INTRODUCTION

Embedded systems, such as Internet-of-Things (IoT) devices, constitute an increasing percentage of computing systems. These systems are characterized by their dedication to a single purpose or very few dedicated functions, their use of nonstandard processors, and their use of real-time operating systems to ensure that tasks meet timing and resource constraints. These devices are becoming a target for cyber attack, and increasingly need malware protection.

One difficulty in protecting embedded systems is that each device often requires specialized configuration, which may also cause a potential attack to require unique consideration. Detecting unique attacks requires a unique signature that malware detection systems would have to recognize. A virus to infect a router, for example, may leverage a vulnerability that only appears in a specific firmware version. Malware authors can mitigate this increased effort by constructing toolkits that allow them to reuse components that target particular architectures, keeping device-specific components but changing the specific exploit. Generally, malware discovered on embedded systems will be device-targeted and novel [1].

Recent advances using machine learning (ML) in malware detection offer some new possible solutions. As will be discussed in Section II-C, there are many existing strategies for detecting malware. Li et al [2] note that traditional methodologies have shortcomings which are addressed by deep learning methodologies. For instance, signature-based methodologies require expert knowledge for signature development. Some techniques, like those based on hardware performance counter data, are capable of detecting the presence of malware without specifically identifying it. This motivates a machine learning approach that infers malware from its code features.

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

A subset of artificial intelligence (AI), MLs can be used to predict malware evolution and is also being used in detection engines. Cylance is one example of a "smart" antivirus advertised to use AI components [3]. MLs techniques assist researchers with identifying patterns and generalizing trends. The explosion in the use of MLs techniques has led to the inclusion of dedicated hardware for deep learning and machine learning on many platforms. Smartphones, for instance, increasingly include neural hardware to accelerate onboard camera image processing. Harnessing this hardware for malware detection requires novel MLs techniques.

The hypotheses for this work were that 1) exploitation causes statistically significant deviation from normal control flow and branch prediction, and that 2) a machine learning model trained on execution traces with examples of benign and exploited behavior will be able to predict the classification of the trace. Underlying these hypotheses is the assumption that there exist specific trace-based features for which the model will not require any knowledge of the underlying application - application-agnostic features.

This research applies to a scenario in which a machine learning model is deployed on an embedded system and trained using malware examples ported to that architecture. The models developed in this research will perform best when applied to problems saturated with real data samples, but because few data samples exist for the RISC-V ROP problem [4], this research generated synthetic data.

One desirable characteristic of malware detectors is interpretability [5]. Many MLs methods are treated as a "black box," where model success or failure on specific samples cannot be attributed to specific causes. This research performs feature exploration both for the purpose of addressing the need for interpretability and for the purpose of constructing a robust model.

The RISC-V ISA was used in this research because RISC-V's open nature facilitates processor modifications and extensions. In addition, the open standard requires no licensing fees, fueling industry interest. We anticipate a significant increase in the use of RISC-V processors in embedded systems which could benefit from generalized and explainable ML-based malware detection systems.

The contributions of this paper are:

- A novel methodology for constructing an execution trace dataset to study the RISC-V ROP problem. To the best of our knowledge, no similar research develops an extensible RISC-V ROP execution trace dataset.
- A deep learning methodology for detecting RISC-V ROP in execution traces, along with an analysis of how RISC-V assembly features affect learning and detection performance. The results show how feature combinations affect model performance, highlighting several high-performing model configurations.
- An analysis of how ROP affects control flow and branch prediction accuracy in the execution trace dataset.

II. BACKGROUND

Malware development for this type of effort depends upon several computing concepts, briefly reviewed here. A RISC-V emulation environment and a cross-compilation toolchain facilitates dataset development. Analysis of branch prediction is tied to RISC-V execution pipeline concepts. Also reviewed here are concepts related to ROP and the machine learning layer types used in this research's detection model.

A. BRANCH PREDICTION

Many processors pipeline the execution of instructions for efficiency. The RISC-V processor uses a five stage pipeline [6] consisting of:

1) Fetch instruction from memory
2) Read registers and decode the instruction
3) Execute the operation or calculate an address
4) Access an operand in data memory
5) Write the result into a register

Occasionally, a result from a later stage, for example, Execute, is required for a subsequent instruction currently in an earlier stage, perhaps Instruction Fetch, which may need to begin operation before the previous result in Execute has completed. This is the case with many branch instructions which must compute the address to branch to. Branch instructions are the assembly language equivalent of an if-then-else statement and are given in the form of "jump to address X if condition Y is met." A branch instruction is a special case where a subsequent instruction depends on the outcome of a conditional statement. To lessen the impact of the delay in waiting until the later stage is complete, processors use branch prediction to guess in advance what the outcome of a branch will be, and begin execution of the subsequent instruction based upon that guess.

The outcomes of a branch prediction may or may not be correct, with each prediction contributing to execution trace data that may be useful as part of a MLs based malware detection system.

B. EXECUTION TRACING

This research utilized RISC-V execution traces as a data source. Many processors allow the monitoring of program execution for debugging and performance enhancement reasons. The Linux `perf` tool is a performance counter utility that uses hardware program monitoring to profile commonly executed code regions [7]. Similarly, Intel Processor Trace [8] and ARM CoreSight [9] are technologies built into their respective architectures that allow execution tracing, one of several forms of processor telemetry. QEMU [10] provides virtualization for a wide range of processor types, including RISC-V. Execution tracing is a feature of QEMU, which allows capture of telemetry in a virtual environment that may not yet be possible on real hardware.

Some execution trace frameworks record only control flow changes for efficiency. This research considered whether

control flow information in the absence of specific assembly features was a useful indicator of ROP exploitation. Assembly language and high-level languages both contain control structures with conditional statements indicating how code flow will behave and it is not known *a priori* how execution will occur. In an execution trace, however, those conditional statements take on a definite "taken or not taken" value indicating how the code behaved retrospectively.

C. ROP DETECTION

Return-Oriented Programming (ROP) is a specific exploit technique to obtain control of a program [11], [12]. On some platforms, it is possible to put executable code on the stack as part of a buffer overflow, then point the return address to the buffer overflow region, resulting in execution of custom code, as depicted in Figure 1. To counter this threat, some platforms are designed so that the stack is not executable. In response, ROP circumvents a non-executable stack by writing a sequence of return addresses onto the stack, each pointing to the tail end of an existing function in the original program. As each function tail is executed, the processor returns to the next return address the attacker placed on the stack. It is possible to build a complete program by cleverly chaining together functionality present immediately before `return` instructions. These function tails are called *ROP gadgets*. Some libraries like `libc` provide a Turing-complete collection of ROP gadgets.

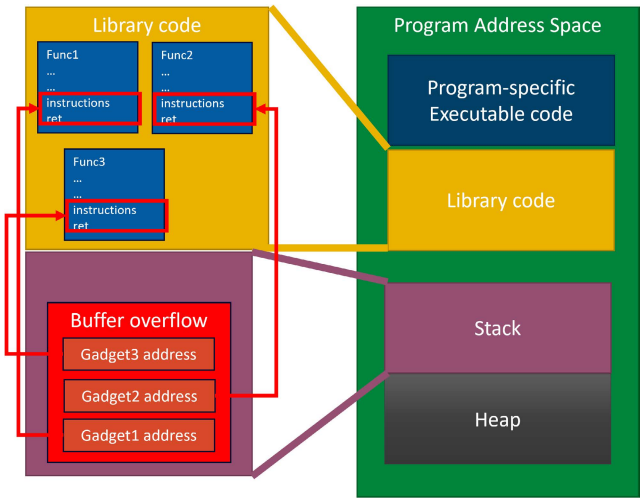


FIGURE 1. Illustration of a ROP attack using a buffer overflow.

This effort specifically targets the identification of ROP attacks. Several ROP detection efforts are reviewed in this section, along with an assessment of each effort's applicability to RISC-V.

1) CATEGORIZING MALWARE DETECTORS

Malware detection approaches can be placed in the following general categories:

- *Signature-based*. Malware may be identified using specific features that indicate its presence. For example, the ClamAV open source antivirus tool [13] allows the definition of signatures that use file hashes, byte patterns, and header values. If a particular website address points to a malware server, then a signature could be written to treat programs containing that text as malware.
- *Anomaly-based*. Malware may be identified by how it makes a system deviate from normal behavior, if that deviation is sufficiently large or somehow detectable.
- *Static analysis/detection*. Some malware may exhibit in-memory artifacts with static features that may be used to determine if the memory artifact is malware. The artifact could be a program or a file.
- *Dynamic analysis/detection:* Some malware may be distinguishable through features that manifest during performance or specific run-time behavior of a system. Behavior could be process-level or system-level.
- *Control flow-based*. Deviation from normal control flow may be used to determine if something is malware. The compiler may be able to identify normal control flow at compile time or a separate utility can identify it after the binary is produced. *Control Flow Integrity (CFI)* is a specific technique in this family.

This research effort is primarily concerned with dynamic detection, and incorporated the use of control flow information.

2) HadROP

Pfaff *et al.* [14] constructed a ROP detector called HadROP that uses statistical analysis of hardware performance counters to classify exploited from non-exploited program runs, and did this by constructing a *Support Vector Machine (SVM)* model on hardware performance counter data. A technique based on hardware performance counters could be used on RISC-V in some configurations because version 1.9.1 of the RISC-V Privileged Architecture [15] adds support for hardware performance monitors, including 29 event counters. However, Pfaff *et al.* observed a 5-8% performance overhead in their HadROP monitor.

3) EigenROP

Elsabagh *et al.* [16] use program characteristics like memory reuse distance, register traffic load, and memory locality - incorporating them into a protection technology called *EigenROP.* It would be possible to apply all of the metrics used by Elsabagh *et al.* to anomaly detection on a RISC-V chip. These metrics are:

1) *Branch predictability*: the degree to which a specific predictor can determine ahead of time which branch will be taken [6].
2) *Instruction mix*: the relative frequency of different classes of instructions like arithmetic, branch, call, and return instructions.
3) *Memory locality*: the address differences between a given memory access and subsequent memory accesses.

4) *Register traffic*: the number of register input operands or the distance between reused registers.
5) *Memory reuse*: the number of unique cache blocks between memory reads.

Elsabagh *et al.* collect each of these features using a tool called MICA, a part of the Pin tools. (The methodology for the Pin toolsuite appears in publications by Hoste and Eekhout [17] and Luk *et al.* [18].) The above metrics are measured using instrumentation which is inserted by the Pin toolsuite's just-in-time compiler.

Determining these metrics after execution is straightforward using trace analysis. However, analyzing these metrics in real-time would require modification to an emulator or to a RISC-V design. It would also require the Pin toolsuite be reimplemented for RISC-V.

4) ROPNN

Li *et al.* [2] developed a detection technique called ROPNN that increases performance over EigenROP. ROPNN searches for the ROP chains themselves directly in program data. The authors use Address Space Layout-guided disassembly whereby addresses in data are checked to see if they point to gadget-like instruction sequences. This process is used to generate the raw input data for testing the neural ROP detector for the end goal of distinguishing actual ROP chains from ROP-like gadget chains. Assembly instructions from real and ROP-like chains are adapted using a one-hot encoding scheme to be classified using a *Convolutional Neural Network (CNN)*.

There are no RISC-V specific barriers to adapting similar techniques to RISC-V. The primary difficulty is that ROPNN uses many utilities for assembly language processing and ROP gadget chaining that do not support RISC-V yet.

5) DeepCheck

Zhang *et al.* [19] developed a CFI technique for mitigating ROP that used a deep learning approach called DeepCheck to classify Intel x64 execution trace data created by the `perf` utility, consisting of *Taken/Not Taken (TNT)* and *Target Instruction Pointer (TIP)* packets. This trace allowed the complete reconstruction of the execution of a program. The DeepCheck classifier ingested sequences of gadget chains, padded by 0×90 NOP instructions to make every gadget the same length. This classifier then identified whether all transitions between gadgets are allowed transitions.

DeepCheck operates on one of the the same types of data used in this research effort: TNT and TIP-style execution trace data. This data is available by parsing existing assembly traces from QEMU and also available from RISC-V chips once the processor trace specification is implemented.

6) ROPDefender

Davi *et al.* [20] developed a ROP tool called ROPDefender using the same Pin framework used by Elsabagh *et al.* [16], and mitigated ROP using a shadow copy of the program stack. The tool later compares the return address on the shadow stack and the program stack. This tool does not use deep learning. It is based on instrumenting a program at runtime or compile time with additional code to create trampoline instructions that generate a shadow copy of the stack. As noted above, no toolsuite similar to the Pin tools currently exists for RISC-V.

7) MATANA

Mao *et al.* [4] constructed a reconfigurable hardware and software framework for assessing attack detection mechanisms. This framework ingests microarchitectural effects - specifically, signals from instruction flow and jump instruction execution. Their framework prototype uses a RISC-V softcore processor for the purpose of detecting ROP attacks on their microarchitecture.

The dataset for the MATANA effort is comprised of synthetic ROP attacks constructed using gadgets distributed in advance through the target executable. This is similar to the approach used in our dataset, covered in Section III.

D. MACHINE LEARNING

This effort addresses the classification of exploited/nonexploited execution traces using machine learning algorithms, which are often used in classification tasks. ML algorithms build a model that improves upon experience, learning how to weight the features required for classification tasks. Machine learning algorithms undergo training, validation, and testing before deployment, with separate data sets for each phase to ensure an algorithm has learned a model that generalizes beyond its training data.

Models in this research effort utilized the convolutional and LSTM layers reviewed here to learn the exploited/nonexploited classification task.

1) CONVOLUTIONAL NEURAL NETWORKS (CNNs)

Artificial Neural Networks are a type of machine learning model which attempts to mimic the behavior of biological neurons. *Deep Learning* refers to an artificial neural network with three or more layers.

Convolutional Neural Networks (CNNs) are a type of artificial neural network that accomplish machine learning by training three types of layers: convolutional layers, pooling layers, and fully connected layers. [21] Training uses two computational passes - a forward pass which determines the output or activation of each layer and then a backward pass which determines the gradient of loss for each layer with respect to its outputs. The gradients for a layer are used to determine how to update the weights for that layer.

2) RNN AND LSTM ARCHITECTURES

Recurrent Neural Networks (RNNs) are a type of deep learning network where network nodes feed forward in a directed graph, making the network better tailored to handle temporal data like text or signal waveforms. These networks can be configured in many input-to-output configurations.

Long Short-Term Memory (LSTM) networks are a form of RNNs constructed with the ability to both retain memory from temporally-previous inputs and forget memory from previous inputs. [22], [23] They are both feedforward and feedback networks. LSTMs are often used in handwriting and speech recognition.

LSTM cells are gated, using the `tanh` and `sigmoid`(σ) functions to control how information from previous cells affects future cells. From an information theoretic-point of view, it is possible for an LSTM network to make a translation or classification where nothing from a future timestep in the input affects a previous timestep in the output. It is also possible for the output at a time step t to depend upon a time step prior to t while possibly forgetting previous information.

3) MALWARE DETECTION EFFORTS USING DEEP LEARNING

As previously mentioned, there have been many applications of machine learning to the malware detection problem, examples of which are found in Table 1.

Making realistic datasets containing malware is complicated by the diversity of existing malware and its sheer volume. In addition, malware requires special handling to prevent accidental malware infections.

Table 2 overviews malware detection research efforts with a focus on the datasets that were used. Two observations that can be drawn from Table 2 are that malware datasets in previous research have 1) contained benign and malicious samples and 2) have contained static and dynamic data.

III. MALWARE DATASET METHODOLOGY

A hypothesis of this research is that a machine learning model could use execution traces to distinguish exploited from non-exploited applications, which requires a data set containing key data from execution traces. Because real data does not yet exist in this area, a synthetic data set was designed to evaluate this hypothesis. As noted in Table 2, a dataset to investigate the "malware/not malware" problem may contain data separated along two dimensions: whether a dataset contains real or synthetic samples and whether the data is static or dynamic. This research only examines exploitation detection on synthetic data.

First, a sample set of RISC-V executables was generated for this task. These executables accomplished small tasks mimicking standard Linux utilities like file hashing, checksumming, content searching, and others. The executables were examined for ROP gadgets using custom scripts for Ghidra [35]. An auxiliary library was included to increase their attack surface and the number of ROP gadgets. These executables were then executed in the RISC-V QEMU static emulator using semi-random input both with and without having an exploit present in their input.

All executions were recorded, yielding instruction traces for ingestion by a deep learning algorithm. These execution recordings were then post-processed by a Python script simulating branch prediction and evaluating whether each branch was predicted correctly or mispredicted.

A. REQUIREMENTS FOR SAMPLE EXECUTABLE CONSTRUCTION

A dataset of meaningful traces requires a diverse dataset of sample executables. The criteria for the sample executables was the following:

1) Represented real Linux utilities or other real-world programs
2) Had execution which depended upon program input
3) Were simple enough that an execution recording could be ingested by a machine learning algorithm
4) Possessed an attack surface that could be exploited by shellcode

One constraint of using the QEMU RISC-V static emulator is that it does not completely implement a Linux environment [10]. This constraint made network functionality unavailable, as well as some other functionality an attacker might use. Also, this constraint is specific to the QEMU static emulator and not QEMU as a whole. In contrast, an OS or application running on full QEMU in RISC-V could have access to networking.

Because file system access and standard I/O functionality were available in the QEMU static emulator, the sample executables use file I/O and command line input. It was also possible to script the utilities to obtain execution traces automatically.

Each of the following algorithms was implemented in RISC-V; one algorithm (the `cat` utility mimic) had no exploit developed for it. There were 9 unique exploits (where one program received 3 exploits), and the dataset was balanced to ensure that it contained an equal number of exploited to non-exploited traces:

1) *SHA-256 implementation [36]*
2) *MD5 implementation [36]*
3) *CRC32 implementation [37]*
4) *Base64 conversion implementation (encode and decode) [36]*
   Some of these executables were compiled from an open-source cryptographic library retrieved from a repository [36]. They each compute over a sample set of input data files and produce hashes or checksums that verify file contents. Neither OpenSSL or other reference library was used due to the complexity of compiling OpenSSL for the target platform.
5) *File read utility*. This mimics the Linux `cat` utility.
6) *Dictionary word sorting algorithm*. This algorithm uses quicksort, insertionsort, or mergesort to sort a list of words, which are read in from a file using an importer that mishandles strings larger than 100 bytes.
7) *Fibonacci number computation*

B. EXPLOIT DEVELOPMENT

Several tools were used to develop exploits (a malicious task using ROP) for each of the sample executables. Due to constraints on exploit development time it was not possible to develop very complex ROP chains.

TABLE 1. Overview of recent machine learning malware detection research efforts.

Authors	Year	Methods	Data Adaptation
Huang and Stokes [24]	2016	MtNet: a custom architecture using dense layers with softmax and ReLu activation functions	API call sequence events and their parameters; null-terminated data objects from system memory. Adapted to sparse binary features representing whether a null terminated object is present or not, and a feature set from the API and parameter stream. API calls were combined into a trigram stream consisting of three successive API call events.
Hardy et al. [25]	2016	Stacked auto-encoders (SAEs)	Feature extraction using PE parser and malware decompressor
Raff et al. [26]	2018	MalConv: Embedding + 1D Convolution + Fully connected layer	Embedding of byte values
Le et al. [27]	2018	CNN + bidirectional LSTM	Converting binary file to 1D representation
Li et al. [28]	2021	IFFNN: an interpretable feed-forward neural network architecture created for the research effort	Software is compiled at five obfuscation levels and four different optimization levels. Software is disassembled using IDA Pro and model is trained on function clone pairs. Malware detection set is from MalShare, VirusShare, and benign software installations. Numeric fields are extracted from PE headers and used as features in addition to graph representation of software.
Narayanan and Davuluru [29]	2020	Ensemble of LSTM and CNN networks	Word embedding of opcodes
Nagaraju and Stamp [30]	2021	Auxiliary-classifier Generative adversarial network (AC-GAN) using a custom Extreme Learning Machine(ELM)/CNN architecture	MalImg and MalExe datasets, converted into 2D image representations
Moti et al. [31]	2021	CNN and LSTM/Generative adversarial network (GAN)	Feature extraction by CNN from PE and ELF headers
Darem et al. [32]	2021	Convolutional neural network (CNN)	Convert disassembled opcodes into 1-gram, 2-gram, ... 4-grams and convert these into an image.
Tupadha and Stamp [33]	2021	Hidden Markov Models, Logistic Regression	PE header features embedded using Word2Vec
Mao et al. [4]	2022	Instruction timing and memory access patterns	Processor signals identified within the RISC-V hardware

Exploit development made use of a ROP gadget finder developed by Tactical Network Solutions [38]. However, the original TacNetSol gadget finder did not support RISC-V so this research effort extended it to RISC-V, resulting in a tool called RISCVRopGadgetFinder.

To construct a realistic scenario accomplishing a malicious function through a ROP exploit, each sample executable needed to contain a large enough number of functions. This was accomplished by adding a primitive auxiliary function library, `auxiliary_functions.c`, that represented functions a ROP exploit might have used. The auxiliary functions perform math functions and file operations and imitate system calls and network operations by making print statements indicating what they would do, since the sandbox lacked the ability to represent certain malicious functionalities. When a program loads a ROP exploit, the exploit contains the stack values necessary to return into the appropriate functions, combining them into a malicious activity. Some of the example auxiliary functions are:

1) Write data to a file.
2) Read data from a file.
3) Retrieve data from a URL.
4) Post data to a URL.
5) Execute a system command.
6) Terminate.

C. EXPLOIT LIMITATIONS AND CONSTRAINTS

An attacker could get functions to execute in an arbitrary sequence by pushing the addresses for these functions onto the stack in succession during a buffer overflow. Since the auxiliary functions are in every executable, it was guaranteed that every executable had the gadgets necessary to perform these behaviors. An exploit which uses an auxiliary library is a close analogy to ROP because no code is being executed on the stack.

A common attacker technique is to use the gadgets available in `libc` (the library of basic C functions used by many Linux system executables), in a *return-to-libc* attack [39], [40]. A return-to-libc attack could have been constructed within the cross-compiled static QEMU environment given enough time, but time constraints did not permit it.

RISC-V function arguments present a processor-specific constraint. *Calling conventions* are a technique used on a per-ISA basis to organize the context switching between a function and the code that called it. They describe where a function can expect function arguments to be stored and also where return values are stored. The RISC-V architecture manual describes RISC-V calling conventions, and RISC-V compilers follow these conventions. Under these conventions the first eight arguments are stored in hardware registers `a0` through `a7`, and the stack stores all successive arguments. From an attacker's perspective, to call a function with an argument, control over the `a0` register is also required. One common attack is to use ROP to open up a command prompt by making a system call that executes `/usr/bin/bash` or `/bin/sh`, running those applications at the privilege level of the exploited application. Such an attack would require passing the name of the shell in the `a0` register.

A constraint specific to the experimental environment is that null bytes are often stripped from user string input. For example, if a program is executed using a command line argument containing shellcode, the Bash shell strips out the null bytes from the argument. Null bytes are part of many

TABLE 2. Example dataset approaches in malware detection research efforts.

Authors	Year	Protection	Real data	Synthetic data	Static/Dynamic	Features
Davi et al. [20]	2011	ROPDefender	Adobe reader exploit	Normal Adobe execution	Dynamic	PinTool instrumentation
Pfaff et al. [14]	2015	HadROP	Adobe flash player exploit, ROP of nginx web server	25 payloads generated by ROP payload generator Q	Dynamic	Hardware performance counter (HPC) profiles
Hardy et al. [25]	2016	DL4MD (SAE method)	Malware and benign data set from Comodo Cloud Security Center	None	Static	Windows API calls
Elsabagh et al. [16]	2017	EigenROP	Publicly available ROP exploits for `hteditor` and PHP	Exploits generated by ROP gadget finder and compiler ROPC	Combination	Branch predictability, instruction mix, memory locality, register traffic, and memory reuse
Pendlebury et al. [34]	2017	TESSERACT	AndroZoo dataset of Android apps	None	Static	Authors do not indicate what features are used
Raff et al. [26]	2018	MalConv	VirusShare executables and benign Windows programs	None	Static	Program bytes
Li et al. [2]	2018	ROPNN	None	ROP gadget chains generated by the ROPGadget tool, validated using CPU emulator Unicorn and targeted against nginx, apache, proftpd, vsftpd, and ImageMagick	Combination	Input data for each program, respectively. For instance, for ImageMagick, the data would be images containing ROP gadgets or benign images
Le et al. [27]	2018	CNN + biLSTM	Microsoft Malware Classification Challenge dataset from Kaggle	None	Static	Binary data from files
Zhang et al. [19]	2019	DeepCheck	Existing exploits for Adobe Flash Player, Nginx, proftpd, and Moxilla Firefox. Also, profiles collected in perf from benign execution of each program.	ROP exploits generated for the four test programs, using ROPGadget and Ropper	Dynamic	Branch information: taken/not taken data and control flow graphs of benign execution
Tupadha and Stamp [33]	2021	HMM and Word2Vec method	Existing examples of PE malware from the malicia and VirusShare datasets	None	Static	Opcodes from malware binary files
Mao et al. [4]	2022	Custom hardware and software detection module	None	Custom Python ROP gadget/attack generator	Dynamic	RISC-V processor signals

addresses, especially in a large address space like the 64-bit RISC-V one. This means that an attacker needs to leverage an input mechanism that copies byte-for-byte. The limited number of these mechanisms that exist limits the diversity of the exploits that can be produced. The ROPNN authors reviewed in Section II-C4 focused specifically on input attacks of this kind [2].

In summary, to achieve a command prompt, the simulated attacker must have 1) the memory address of a string that states the desired application to run, and 2) access to primitives that put that string into a function argument that make a `system()` or similar call. To simplify attack development, the auxiliary functions library includes functions that reduce the number of primitives needed. One such example is shown in Figure 2, which contains a function that deliberately includes a primitive to ensure that a value will be copied off the stack into a register. The auxiliary functions library also included the string `/usr/bin/bash` and a malicious URL. This ensured that the shellcode had access to strings necessary to perform something malicious.

```
1 char* return_arg(char* a, char* b,
2                   char* c, char* d,
3                   char* e, char* f,
4                   char* g, char* h,
5                   char* i){
6     printf("Innocuous function\n");
7     return i;
8 }
```

FIGURE 2. A function crafted to ensure that there is a ROP gadget that copies a stack value into register `a0`.

D. TRACE GENERATION

A tool was constructed to automate shellcode generation. It searched target applications for the addresses of auxiliary functions and arranged the addresses of the target functions into exploits.

The final data generation process yielded the ability to generate an arbitrary number of new traces dependent solely upon identifying new sample text files to use as input to the sample executables. Given the requirement for new algorithms to trace, it would be straightforward to include `auxiliary.o` and a vulnerable input into a new program, yielding the ability to generate traces from more executables. In addition, using the developed script, it would be possible to automatically generate new ROP chains that consist of arbitrary function sequences.

Table 3 illustrates the dataset variables. The dataset consisted of 12,885 unique files, and 2570 different program runs. This is the result of executing 15 different algorithm/exploit configurations on 18 text input files of different contents and input sizes. A straightforward way to expand the dataset is to run all algorithm configurations on additional text samples, though to increase diversity in the dataset, it would be more meaningful to increase the number of unique exploits and algorithms. Figures 3 through 7 show samples of the data obtained in this effort.

Equation 1 expresses the formula for counting the number of unique traces generated:

$$wxy + 2z \tag{1}$$

where $w = 146$ is the number of text samples, x is the number of unique programs, and y_x is the number of exploits available for each program. The value z represents a specific program (Fibonacci number generation) for which new traces could

```
0x000000400084f0ea:  9ee23823          sd              a4,-1552(tp)
0x000000400084f0ee:  9782              jalr            ra,a5,0
0x0000000000011452:  7105              addi            sp,sp,-480
0x0000000000011454:  ef86              sd              ra,472(sp)
0x0000000000011456:  eba2              sd              s0,464(sp)
0x0000000000011458:  1380              addi            s0,sp,480
0x000000000001145a:  87aa              mv              a5,a0
0x000000000001145c:  e2b43023          sd              a1,-480(s0)
0x0000000000011460:  e2f42623          sw              a5,-468(s0)
0x0000000000011464:  e2c42783          lw              a5,-468(s0)
0x0000000000011468:  0007871b          sext.w          a4,a5
0x000000000001146c:  4785              addi            a5,zero,1
0x000000000001146e:  00e7c963          bgt             a4,a5,18        # 0x11480
0x0000000000011480:  e2043783          ld              a5,-480(s0)
0x0000000000011484:  07a1              addi            a5,a5,8
0x0000000000011486:  6398              ld              a4,0(a5)
0x0000000000011488:  67c9              lui             a5,73728
0x000000000001148a:  98878593          addi            a1,a5,-1656
0x000000000001148e:  853a              mv              a0,a4
0x0000000000011490:  b40ff0ef          jal             ra,-3264        # 0x107d0
0x00000000000107d0:  00003e17          auipc           t3,12288        # 0x137d0
0x00000000000107d4:  900e3e03          ld              t3,-1792(t3)
0x00000000000107d8:  000e0367          jalr            t1,t3,0
0x0000004000881d7a:  4605              addi            a2,zero,1
```

FIGURE 3. Beginning of main() function assembly-level trace.

```
0x000000400084f0ee
0x0000000000011452
0x0000000000011454
0x0000000000011456
0x0000000000011458
0x000000000001145a
0x000000000001145c
```

FIGURE 4. Beginning of main() function address trace.

```
0x4000801ee0
0x2
0x696
0x2
0x2
0x2
0x2
0x2
0x2
0x2
0x2
0x4
```

FIGURE 5. Beginning of main() function offset trace.

```
NOT TAKEN: 0x400084f0ca
TIP: 0x11452
TIP: 0x107d0
TIP: 0x4000881d7a
TIP: 0x400084ec50
TIP: 0x400084ec20
TAKEN: 0x400088ff00
NOT TAKEN: 0x400089014c
TIP: 0x400088d054
NOT TAKEN: 0x4000802642
NOT TAKEN: 0x400080264a
TAKEN: 0x4000802654
```

FIGURE 6. Beginning of main() function control flow trace. `TIP` indicates a control flow redirect that was unconditional.

```
P: 0x400084f09e
P: 0x11788
MP: 0x1086c
P: 0x117a2
P: 0x400084f0bc
P: 0x400084f0c0
P: 0x400085ca28
P: 0x400084f0ca
MP: 0x400088ff00
```

FIGURE 7. A sample prediction/misprediction trace from the main() function.

be generated simply by giving another integer input. Overall, there were 17 unique program runs across 146 input files, 88 different Fibonacci traces, 7 traces from unique input to the base64 algorithm. The dataset had a proportion of 52.96% non-exploited traces to 47.04% exploited traces.

E. DATASET LIMITATIONS

Applying this methodology yields a dataset of dynamic trace information to train a classifier that identifies malicious execution. While this dataset enables some meaningful analysis, it also presents some concerns.

The first concern is common to any synthetic dataset. Any methodology for generating synthetic data will generate only data that fit within certain parameters. In this effort exploit development was restricted to only ROP exploits that used the `auxilary.o` library. The gadgets in this library were not as complete as `libc`. Because the exploits in this dataset mimicked the first stage of an attack to launch other code, their runtime was often a small portion of the overall application runtime.

A second concern is the limited number and type of vulnerabilities present. There is a significant development cost associated with increasing the vulnerability and exploit diversity, even with automation. This dataset limitation would be expected to limit the accuracy of any models trained from the dataset.

When an execution trace is recorded, a large part of the recording is low-level library execution. Some low-level execution occurs prior to the `main()` function and some of it occurs during library calls. The additional search area can make it more difficult for a human to identify relevant parts of the trace. Low-level library execution increases the information an ML algorithm must search through to identify malicious execution.

IV. EXPERIMENTAL SETUP FOR EVALUATING EFFECT ON BRANCH PREDICTION

QEMU was set to evaluate instructions sequentially to mimic a scalar processor. To evaluate the effect of ROP on branch prediction, this research implemented two branch predictor models: a local model and a global model. The local model retained a unique 2-bit predictor state for every instruction address, while the global model retained a table of 4096 predictors indexed by the taken/not taken status of the last twelve predictions. The 2-bit branch prediction strategy is diagrammed in Figure 8.

The following statistics were extracted from the execution trace recordings:

1) The proportion of Taken/Not-Taken/Unconditional branches in traces overall, and separated by classes of exploited/not exploited.
2) For a local history 2-bit predictor, the proportion of Predicted/Not predicted in traces overall, and separated by classes of exploited/not exploited.
3) For a global history 2-bit predictor, the same metrics

To generate these statistics, a Python script post-processed each trace. It used regular expressions to identify whether the currently executed instruction was a branch instruction

TABLE 3. Four labels applied to each data file, indicating what type of trace it contains and what occurred in the recording.

Types of Trace Data	ROP Options	Algorithms	Input to Algorithms
Sequences of executed RISC-V assembly Address trace Address offset from previous instruction Control flow changes Branch prediction correct or not	Executable exploited Executable not exploited	SHA256 MD5 CRC32 cat file Base64 encode/decode	Unique text files

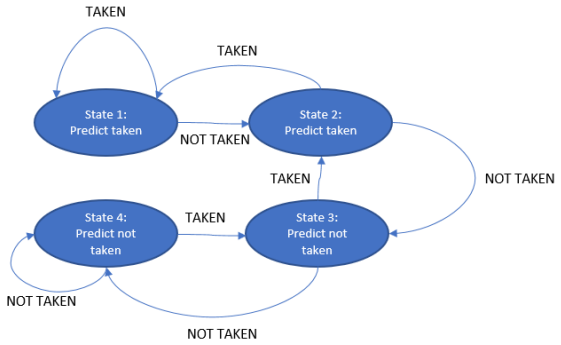


FIGURE 8. The state diagram for a 2-bit branch predictor [6].

or a control flow instruction. Each branch was initialized to state 2, predicting Taken. After the branch is taken or not taken, the 2-bit predictor state was then updated based upon the jump outcome.

V. DEEP LEARNING MODEL METHODOLOGY

The second phase of this research effort focused on developing a model to distinguish exploited execution traces from non-exploited ones.

As noted in Section I, this research sought specifically to address the problem of how feature selection affects the performance of deep learning models. Deep learning was chosen for its ability to infer indicators of exploitation, which is useful when those indicators are not known ahead of time. One unanswered question from the surveyed background efforts was whether specific executed instruction sequences rather than their frequency are indicators of a ROP attack. ROPNN [2] performed gadget identification using a deep learning model, suggesting that deep learning contains good candidate techniques for the target problem.

This guided model selection to a deep learning architecture capable of inferring patterns from sequential data. Recurrent network architectures (e.g., GRUs, RNNs, and LSTMs) infer patterns from sequential data well. A bidirectional LSTM was chosen for its ability to capture patterns in both forward and backward directions.

A. DATA ADAPTATION

Data adaptation was driven by the requirements of the bidirectional LSTM model architectures: 1) that model input be fixed length, and 2) that the final form of the data be in a dimensional space which would best facilitate model operation.

For all machine learning experiments, the entire collection of exploited/not-exploited traces was compressed to contain just the first instance of each address execution. Without this adaptation, the largest trace was more than 41 GB. This adaptation captured both code coverage and the order in which each address first executed.

Branch prediction data for Section IV was drawn from traces prior to compression. This was because branch predictor accuracy would have been affected by addresses being executed only once.

Further, these traces were adapted to contain 5000 samples each, where the samples were drawn from the end of both exploited and non-exploited traces. This length was chosen because it was more than half the length of the shortest traces while still fitting within the memory requirements of the computers used for training. If an application was exploited, the end of the trace will probably contain the exploit, especially if the exploit caused the application to terminate. Each sample was labeled by whether it contained exploits or not.

Researchers use varying methods for adapting text to numerical data. One of the simplest methods is to generate a *one-hot encoding*, where every possible text value is given its own feature. For diverse data samples however, the dimensionality of a one-hot encoding can make the model difficult to train.

Embeddings alleviate dimensionality problems by mapping the input space into a different dimensional space. *Word2Vec* [41] is a tool for adapting words into a vector of this kind, based upon cosine similarity between words in the dataset. The opcodes, register operands, and control flow/branch prediction data were all embedded as separate 4-dimensional embeddings. Each embedding was a separate Word2Vec model, trained on the occurrences of each value within the complete dataset. All immediate and address values were adjusted to be the same value mod 4096. This represented the lower 13 bytes of the immediate and address features. This prevented large changes in program addresses from overshadowing more common small changes in program addresses.

Figure 9 shows the data sample fields. Of these data samples, seven of them (opcode, operands 1-3, control flow, and both branch predictors) were categorical data. These four categorical data types were each projected into a 4-dimensional space using *Word2Vec*. (These embeddings are visualized in Figures 10, 11, 12, and 13 in 2 dimensions using *Principal Component Analysis (PCA)*.

The data was split into a training set of 90% of the original 2577 traces and a test set of 10%. 4-fold cross validation was used when training and validating the models.

B. MODEL CONSTRUCTION

Model design was driven by the research goal of examining the effects of different feature choices on model performance. Models were trained on several feature combinations, with generally increasing amounts of information, and these combinations were given sequential labels A through F, as shown in Table 4.

One feature, addresses, was used in Models D and F but it was not expected that using addresses would generalize to programs outside this specific dataset. Address sequences as a feature are specific to an executable and are unlikely to be meaningful in a program not in the existing dataset. For a detection technique to be useful to the malware detection community, its model needs to train on features that are not specific to the executable.

Models C, E, and F used the immediate field, which occasionally contains address values. Because of this, it also contains application-specific information. For example, the ROP exploits occasionally access text strings like `/etc/passwd` that are not accessed in non-exploited traces. The addresses of these strings appear in the immediate field and are a feature which is specific to each application.

Figure 14 shows a generic version of the deep learning model architecture used in each model. In the `input_fields` layer and the `conv1d_1` layer, the dimension of 1 was the dimension of the final data model but is set to the dimension of the data samples. 5000 represents the 5000 data samples extracted from each trace. The length was chosen to accommodate the bidirectional network size.

Categorical crossentropy rather than binary crossentropy was chosen as the training loss function. This improved the training stability of the results. Categorical crossentropy is a better loss function in situations where the model is trying to figure out which single category a data sample belongs in; binary crossentropy is better in situations where a data sample could be in multiple categories.

Hyperparameters are parameters related to the learning process itself. Examples of these include the learning rate and the batch size used for training. The learning rate was set at 0.001 initially and declined by four percent whenever validation performance did not improve after two epochs. Each model was set to train for 250 epochs and halt training if validation set accuracy did not improve after 20 epochs. As mentioned in the previous section, 4-fold cross validation was used during training to assess each model.

VI. RESULTS

A. BRANCH AND BRANCH PREDICTION STATISTICS

Figure 25 in Appendix shows scatter matrices of five different statistics and how they are affected by ROP in this dataset:

1) `accuracy_local`. The ratio $\frac{total\_correct}{total\_conditional}$ that represents the ratio of correct predictions made by the local 2-bit predictor to the total number of conditional branches.
2) `accuracy_global`. The same ratio but for the global 2-bit predictor.
3) `total_unconditional`. The total number of unconditional branches in each trace.
4) `total_conditional`. The total number of conditional branches in each trace.
5) `total_taken`. A value indicating how many conditional branches were taken.

In Figure 25, plots along the diagonal show the distribution of the populations in each category. Red points indicate traces of ROP exploitation and green points indicate no exploit occurred. It does appear that increasing the number of conditional branches increases the accuracy of a global predictor and lowers the accuracy of the local predictor, but identifying specific linear relationships was outside the scope of this effort.

Boxplots of the branch prediction data also help in evaluating questions about exploited vs. non-exploited branch prediction data. These are shown in Figures 15, 16, 18, and 17. The data from these figures is also provided in Table 5.

Figure 26 shows the distribution of the complete population from which the statistics in Table 5 are drawn. Visual inspection of Figure 25 and Figure 26 as well as Shapiro-Wilk tests indicate that all five statistics are not normally distributed. This is the case even when these statistics are separated into ROP/non-ROP populations. The Wilcoxon signed rank paired test for distribution differences [42] is applicable if pairs are constructed from program runs which were the same apart from their exploited/nonexploited status.

The branch prediction research question expressed as hypotheses were as follows. $\tilde{x}$ is used to indicate the median of a population:

$$H_0 : \tilde{x}_e = \tilde{x}_{ne} \tag{2}$$

$$H_{A+} : \tilde{x}_e > \tilde{x}_{ne} \tag{3}$$

$$H_{A-} : \tilde{x}_e < \tilde{x}_{ne} \tag{4}$$

As expressed in Equations 2, 3, and 4, this research hypothesizes that each statistic is distributed equally in ROP and non-ROP traces.

A Wilcoxon signed rank test indicates that there is sufficient evidence to reject a hypothesis (with $p = 9.49 * 10^{-7}$) that the distribution of `accuracy_global` is the same in ROP and non-ROP traces. It gives enough evidence to accept an alternative hypothesis H_{A-} that the median global predictor accuracy is higher in non-rop traces. The opposite is true for the local branch predictor. The Wilcoxon signed rank test gives evidence to reject a hypothesis that local predictor accuracy is the same in ROP and non-ROP samples, supporting instead a hypothesis H_{A+} that local predictor accuracy is lower in non-ROP traces.

The Wilcoxon signed rank test requires that ROP/non-ROP statistic differences be symmetric, which is the case

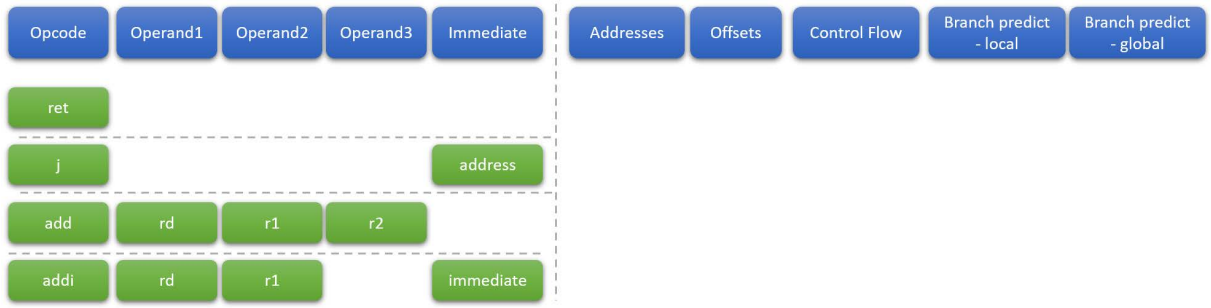


FIGURE 9. Every timestep within a trace contained these 10 features (shown as blue roundtangles).

TABLE 4. Model IDs, dimensionality, and feature set of the six models evaluated in this research.

Model ID	Dim.	Features (dimensions)					
		Opcode (4)	Operand 1 (4)	Operand 2 (4)	Operand 3 (4)	Immediate (1)	Address (1)
A	4	X					
B	16	X	X	X	X		
C	1	X				X	
D	1	X					X
E	17	X	X	X	X	X	
F	18	X	X	X	X	X	X

TABLE 5. Data on branch prediction differences between exploited and non-exploited traces.

	ROP		no-ROP	
	Mean	Std. Dev	Mean	Std. Dev
Total branches	269,802.858	249,258.512	$1.786 * 10^6$	$7.679*10^6$
Total cond. taken	147,711.790	155,080.086	304,906.9	568,120.6
Total cond. nottaken	67,190.900	60,230.415	768,674	$3.648*10^6$
Total conditional	214,902.690	202,815.609	$1.073 * 10^6$	$4.107*10^6$
Total uncond.	54,900.168	166,467.496	712,775.5	$3.595*10^6$
Total correct, local predictor	205,544.308	196,796.374	$1.023 * 10^6$	$3.945*10^6$
Total incorrect, local predictor	9358.382	6296.569	49,847.47	170,854.7
Total false positives, local predictor	6065.412	4424.946	18,535.82	54,096.84
Total false negatives, local predictor	4291.310	2071.835	31,311.65	136,945.9
Accuracy, local predictor	0.934731	0.037011	0.9318425	0.03484667
Total correct, global predictor	204,487.801	196,655.520	$1.051*10^6$	$4.091*10^6$
Total incorrect, global predictor	10,414.888	6446.008	21,682.18	29,210.83
Total false positives, global predictor	6123.578	3974.639	11,641.75	15,383.55
Total false negatives, global predictor	4291.310	2682.558	10,040.44	14,011.75
Accuracy, global predictor	0.923441	0.046102	0.927359	0.04691471
Percent if always guessing taken	0.6873426		0.284009	

for `accuracy_global` and `accuracy_local` but not for the counts of total taken, conditional, and unconditional branches. A sign test provided sufficient evidence to support alternative hypotheses that if this dataset were representative of ROP and non-ROP populations, ROP causes `total_conditional` and `total_unconditional` to shift negatively. On the other hand, a sign test provided sufficient evidence to support a hypothesis ROP caused a positive shift in `total_taken`.

Exploitation may cause early termination of a program or cause the program to continue to run beyond its normal termination. Statistics in this area are more useful when reasoning about a specific program/exploit combination rather than trends. For example, the trace dataset in this research contained recordings of an insertion sort algorithm affected by an exploit randomly placed in its input data. When exploited, this program did not exhibit its normal $O(n^2)$ runtime. Shortened runtime on this program would be an indicator of exploitation. Other programs were exploited around time of completion. For these programs continued execution is indicative of an attack.

Individual algorithms separate into distinct groups in the plots of global and local branch predictor accuracy in Figure 25. On most algorithms but not all, global and local branch prediction accuracy increase with algorithm length as represented by total unconditional and conditional branches.

One limitation of these results is that ROP attacks primarily utilize the return address stack, and returns are an unconditional jump. Branch prediction is affected only by conditional jumps. ROP attacks do often make use of gadgets

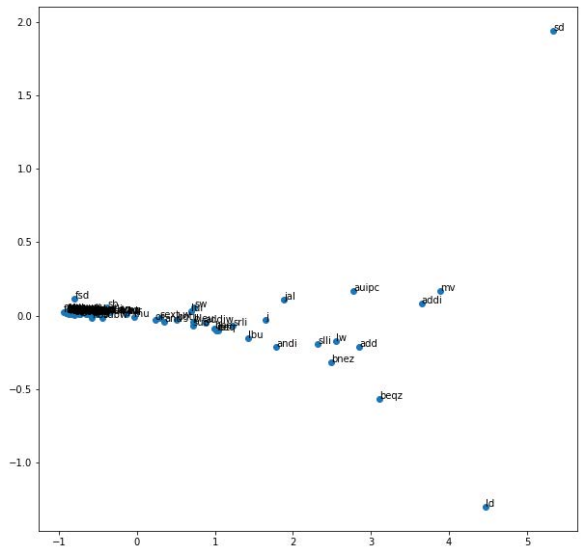


FIGURE 10. PCA 2-dimensional projection of a RISC-V opcode embedding produced by Word2Vec.

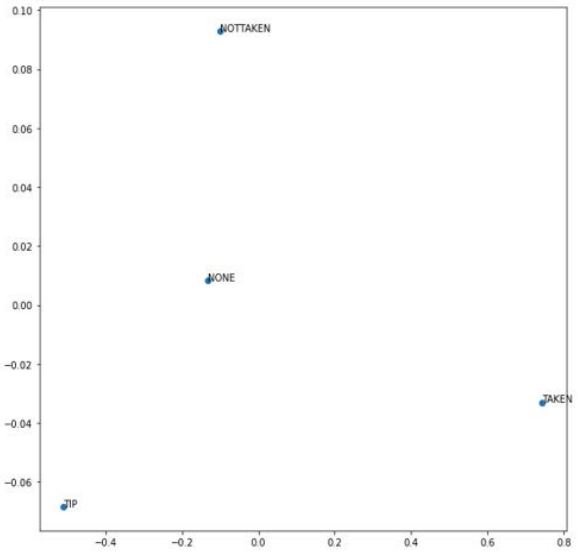


FIGURE 12. PCA of the values in the control flow embedding produced by Word2Vec.

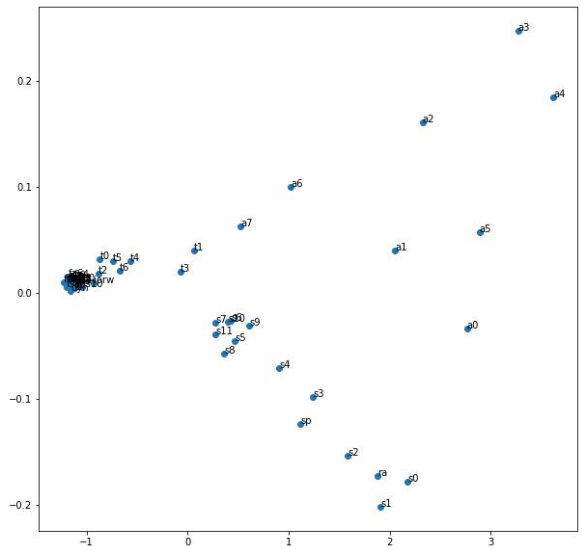


FIGURE 11. PCA 2-dimensional projection of a RISC-V operand embedding produced by Word2Vec.

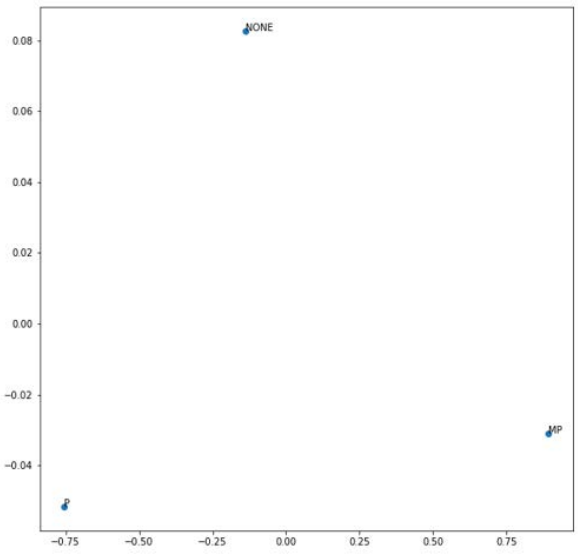


FIGURE 13. PCA of the values in the branch prediction embedding produced by Word2Vec.

and functions containing conditional jumps, however. This means that ROP attacks can still produce a measurable effect on branch prediction statistics.

B. MACHINE LEARNING MODEL RESULTS

In the experimental setup using two computers with an Intel Xeon E5-2687 processor, 64 GB of RAM, and an 8GB Nvidia Quadro M5000 GPU, it required about 21 hours for each fold of cross validation on every model, requiring about 3.5 days of training per model. Results are summarized in Table 6, and Figures 19 through 23 contain the training and validation graphs for all models. These graphs depict the stability of the training process for each configuration. In Table 6, cross validation accuracy is the best performance of each of 4 folds, averaged. Test accuracy is the performance of the best model from each cross-validation fold, tested against data not used in cross-validation.

The combination of 4-fold cross-validation and early stopping after 20 epochs of no validation accuracy improvement caused some folds of cross validation to terminate before others. Because graphs of validation performance in this section are averaged, they only show epochs which all models completed.

Model A was an opcode-only model which achieved an average of 91.24% cross-validation accuracy. The training and validation curves are shown in Figure 19. Model B used

TABLE 6. Performance of all model configurations.

Name	Features	Average Cross Val. Acc.	Test Acc.
A	Opcode only	0.9124	0.9506
B	Opcode/three operands	0.9821	0.9794
C	Immediate value only	0.9825	0.9835
D	Address value only	0.9279	0.9959
E	Opcode/three operands/immediate	0.9835	0.9835
F	Opcode/three operands/immediate/address	0.9671	1.00

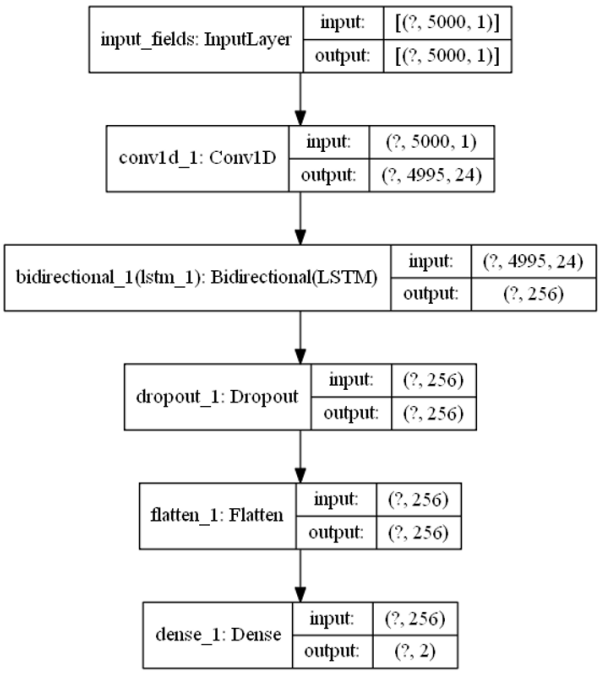


FIGURE 14. Example plot of Models C and D for classifying trace data.

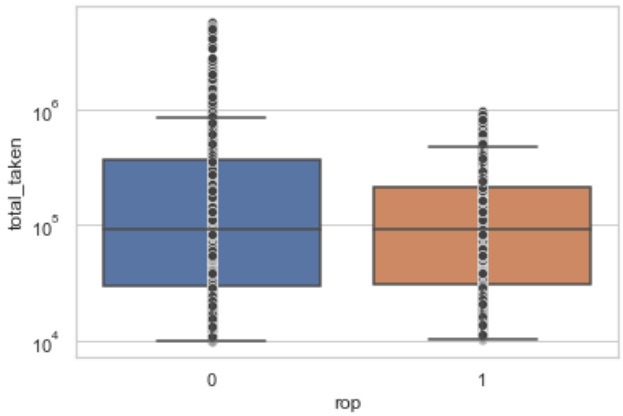


FIGURE 15. Boxplot showing the difference in distribution of number of taken branches by exploited/not exploited status.

the opcode and all three operand features, and all of these features were embedded into 4 dimensions. The model achieved 98.35% mean validation accuracy. While the training and

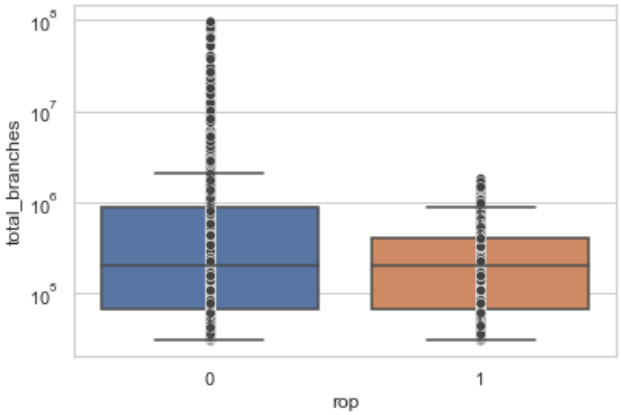


FIGURE 16. Boxplot showing the difference in distribution of number of branches by exploited/not exploited status.

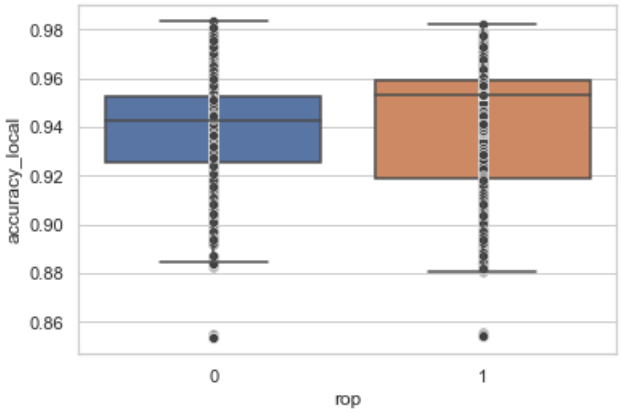


FIGURE 17. Boxplot of accuracy of the local branch predictor for rop vs.non-rop traces.

validation curves of models A and B are not very stable, there is a definite stability improvement from Model A to B. The Model B results are shown in Figure 20.

Models A and B do not contain the same application-specific information that Models C and D do. Model C was trained using a single feature of the immediate values from the assembly instruction, mod 4096. The training and validation curves from this experiment are shown in Figure 21. Model C achieved both more stable and more accurate results than Model B despite using fewer features.

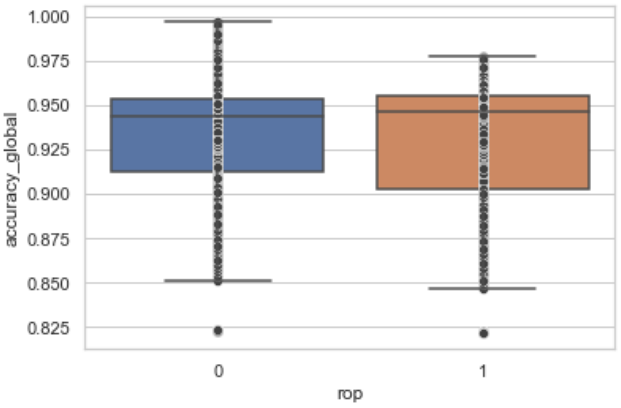


FIGURE 18. Boxplot of accuracy of the global branch predictor for rop vs.non-rop traces.

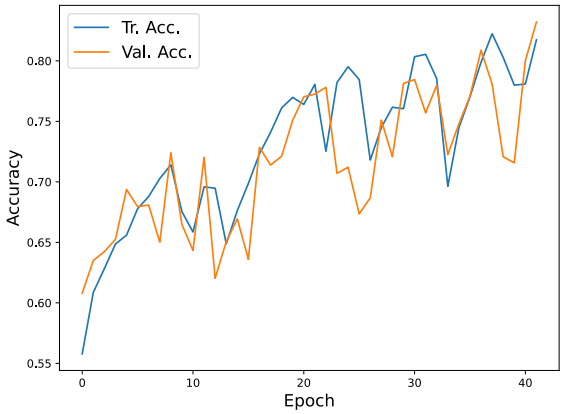


FIGURE 19. Model A average cross-validation accuracy per epoch on training using 4-fold cross validation.

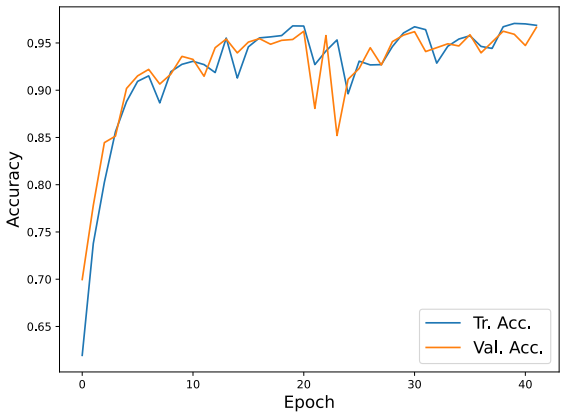


FIGURE 20. Model B average cross-validation accuracy per epoch on training using 4-fold cross validation.

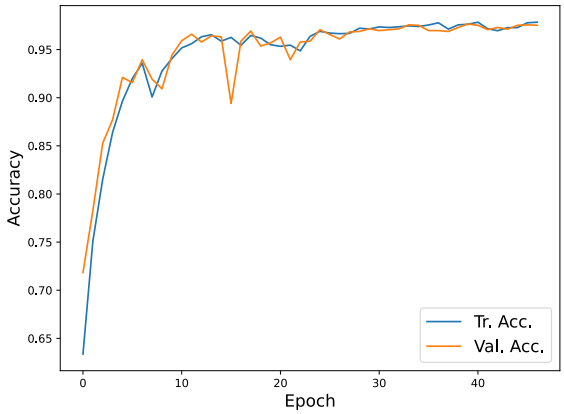


FIGURE 21. Model C average cross-validation accuracy per epoch on training using 4-fold cross validation.

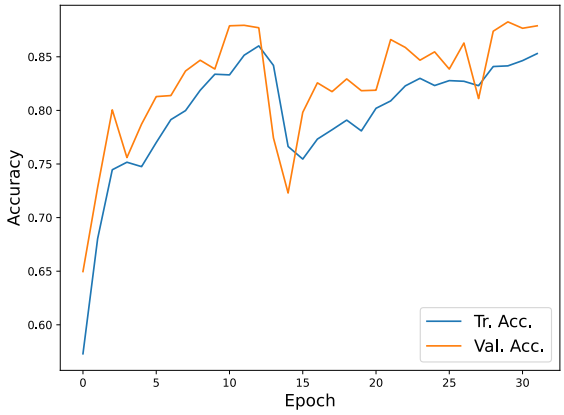


FIGURE 22. Model D average cross-validation accuracy per epoch on training using 4-fold cross validation.

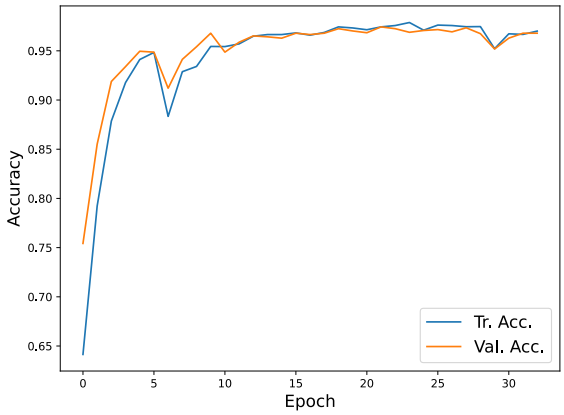


FIGURE 23. Model E average cross-validation accuracy per epoch on training using 4-fold cross validation.

Addresses appear significant in this classification task, based upon Model D and F's test set performances as well as the differences between their test set performances and cross validation performance. This most likely meant that the model identified all of the code regions where ROP gadgets are found in the trace data set, and it detects those addresses specifically. Model D's training and validation curves, shown in Figure 22, appear most unstable of all of the models that were evaluated.

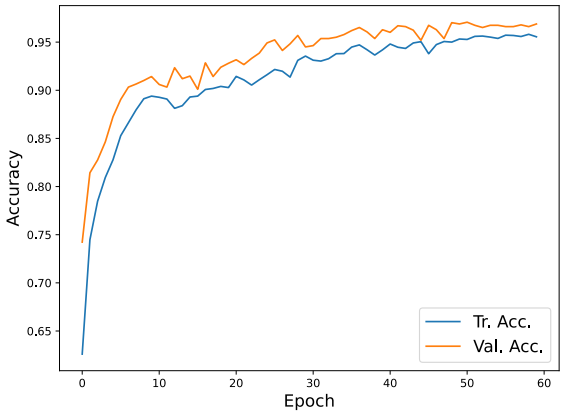


FIGURE 24. Model F average cross-validation accuracy per epoch on training using 4-fold cross validation.

As noted above, Models E and F, whose training and validation curves are depicted in Figures 23 and 24, achieved the best results. A progression through Models A, B, E, and F show how performance improves and training stabilizes through feature addition. A similar progression can be seen in Models C, E, and F or D, E, and F. The best performing model in this research, Model F, used six features and achieved test set accuracy that rounded up to 1.

Models D and F have similar diverging cross validation/test set performance. It appears that an overfitting behavior was captured by the cross-validation process.

VII. CONCLUSION

This research focused on methods for detecting when a RISC-V execution trace contains a ROP exploit. As part of this effort, several deep learning models were constructed that were able to distinguish exploited from non-exploited traces based on instruction data. Models contained a bidirectional LSTM with a 1D convolutional layer as its first layer. Models based on instruction addresses have several issues related to their portability.

Model results highlight the contributions of each feature to model performance. An embedded opcode is outperformed by an ensemble of the embedded opcode and the embedded operands, and both are outperformed by a model containing only the immediate value. Using all available features of an assembly language statement in the trace results in an even better model.

Finally, this research examined the statistical distribution of branch prediction accuracy in exploited and non-exploited traces. It found evidence to support the hypothesis that ROP exploitation affects local and global branch prediction. This effort produced a result that supports the hypothesis a ROP detector could function using just a trace of assembly instructions.

VIII. FUTURE WORK AND IMPROVEMENTS

Future research will be focused on several areas - some related to improving the dataset and some related to logical extensions of the work.

A. BETTER DATA AND REORGANIZATION OF DATASET

To improve the synthetic dataset's ability to force a model to train on ROP behaviors, the dataset should be reorganized so that the testing and validation sets each contain traces from an application not seen in the training data. This would reduce the model's ability to train on application-specific features.

The dataset would also benefit from the introduction of real-world applications and exploits from a malware tracing environment. One difficulty with this is that little real-world malware exists for RISC-V.

One improvement will be to use a Linux operating system running in QEMU and to identify a mechanism for localizing exploit execution out of the RISC-V trace. This will allow the exploits to be more representative of real-world exploits. Adapting real-world exploits for Linux will increase the strength of the dataset. It will, however, also require a sandbox that does not allow the malware to escape into the real world.

B. DEPLOYMENT

The best method for deployment of this model would involve pruning and an implementation on some form of neural-accelerated hardware. For deployment, the classification latency of this model will need to be reduced to the point that it can keep up with the real-time demands of the system where it is deployed.

On March 20, 2020 the RISC-V foundation ratified a processor trace specification for RISC-V. The specification standardizes a format for recording the types of information used in the dataset built in this effort [43]. It is expected that popular RISC-V cores will eventually implement this specification as a means of monitoring execution but it will probably be several years before the trace specification has OS support in the same areas as Intel Processor Trace and ARM CoreSight. A custom recording solution is necessary for capturing traces on non-emulated RISC-V processors. Using a hardware-based data collection method would shorten the time between data collection and malware detection.

C. MODEL IMPROVEMENTS

An adversary position will be considered in future research. Malware authors are aware of efforts to detect and analyze their work, and often incorporate methods to prevent detection and analysis. Reverse-engineers must work around anti-tamper and obfuscation to determine a piece of malware's purpose. The original dataset was highly specific to a select few data traces. In addition, the model introduced in this effort has a large latent space where the classification of unseen behaviors is undetermined. An adversary could examine the latent space of our model and ensure that an attack falls into the latent space where behaviors are classified as non-exploited. Mitigating this would be possible using a form of adversarial machine learning.

One limitation of this work was that our model design was unable to handle complete execution traces and was forced to work with compressed traces. Future work will examine

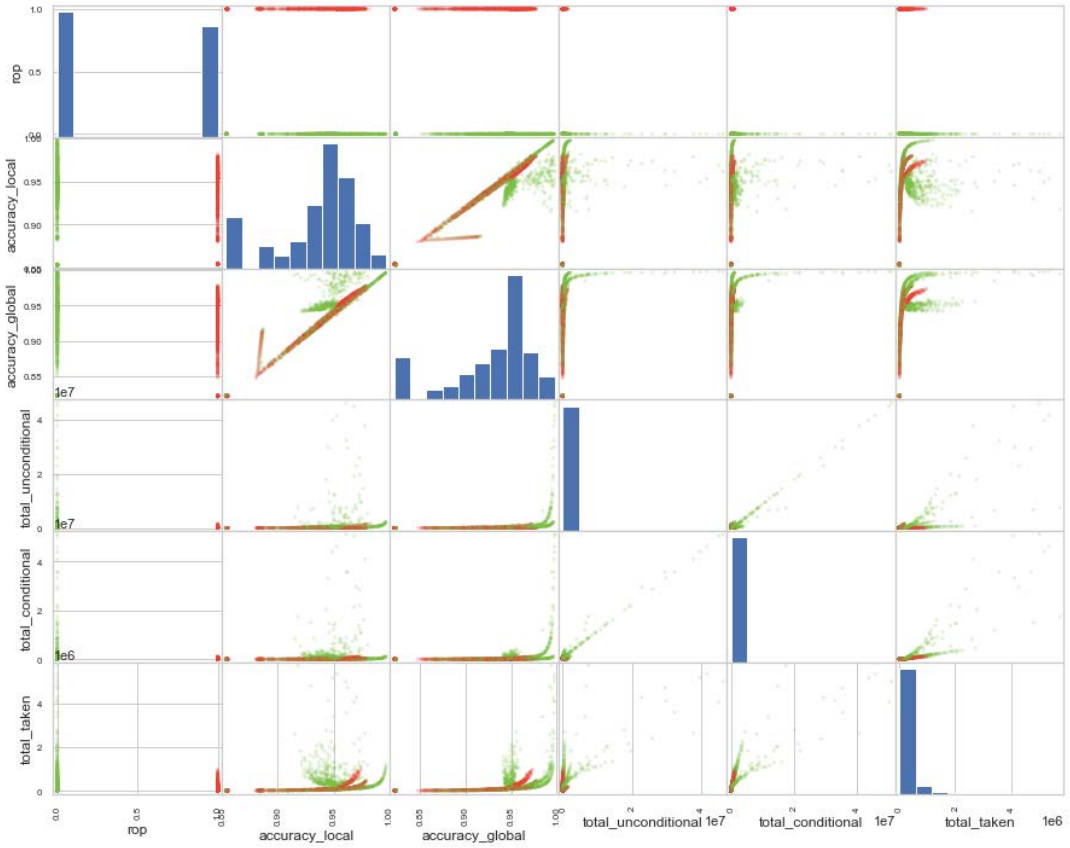


FIGURE 25. Scatter matrix for branch prediction statistics for the research dataset.

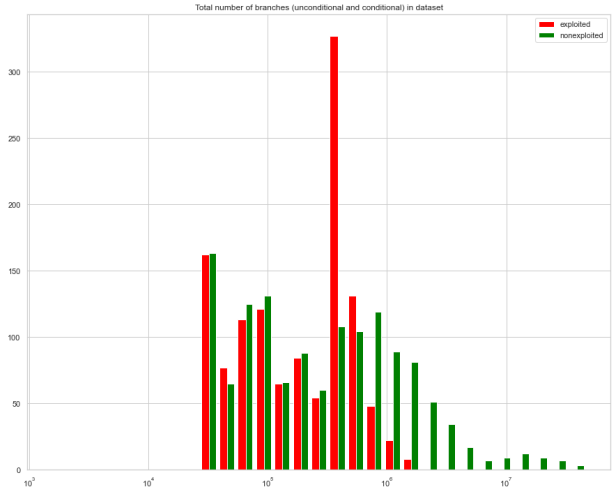


FIGURE 26. Histogram of branch count across all traces.

recurrent models which perform anomaly detection to recognize exploit behavior on segments of execution traces rather than compressed representations of the whole. This will avoid problems related to handling the entire trace at once.

Also, future research will examine methods for explaining the neural model used in this research, particularly Layerwise Relevance Propagation (LRP). LRP is a whitebox technique for explaining the relevance of specific features to a deep learning model [44]. Warnecke *et al.* [45] explored the use of this technique in security applications. Future research will augment the feature exploration performed in this research effort with LRP, allowing malware analysts to better understand what features contribute to a malware classification.

APPENDIX
ADDITIONAL BRANCH PREDICTION RESULTS

Figures 25 through 26, referenced in Section VI, contain graphical depictions of branch prediction results from this effort.

ACKNOWLEDGMENT

The views expressed in this document are those of the authors and do not reflect the official policy or position of the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2022-0051.

REFERENCES

[1] M. Al-Hawawreh, F. D. Hartog, and E. Sitnikova, "Targeted ransomware: A new cyber threat to edge system of brownfield industrial Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 7137–7151, Aug. 2019.
[2] 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*.
[3] BlackBerry Limited. (2022). *Cylance AI From Blackberry*. Accessed: Jan. 25, 2022. [Online]. Available: https://www.blackberry.com/us/en/products/unified-endpoint-security/cyl%ance-ai

[4] Y. Mao, V. Migliore, and V. Nicomette, "MATANA: A reconfigurable framework for runtime attack detection based on the analysis of microarchitectural signals," *Appl. Sci.*, vol. 12, no. 3, p. 1452, Jan. 2022. Accessed: Apr. 5, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/3/1452

[5] Y. Lin and X. Chang, "Towards interpreting ML-based automated malware detection models: A survey," 2021, *arXiv:2101.06232*.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2017.

[7] Linux Kernel Organization Inc. (2022). *Perf Wiki*. Accessed: Jan. 25, 2021. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[8] J. Reinders. (Sep. 2013). *Processor Tracing*. Accessed: Jan. 12, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/processor-tr%acing.html

[9] L. ARM. *CoreSight Architecture*. Accessed: Apr. 5, 2022. [Online]. Available: https://developer.arm.com/architectures/cpu-architecture/debug-visibili%ty-and-trace/coresight-architecture

[10] QEMU Project. *QEMU Developer's Guide*. Accessed: Apr. 7, 2021. [Online]. Available: https://github.com/multiarch/qemu-user-static/blob/master/docs/develope%rs_guide.md

[11] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-LIBC without function calls (on the ×86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security* New York, NY, USA: Association Computing Machinery, 2007, pp. 552–561, doi: 10.1145/1315245.1315313.

[12] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 1–34, 2012.

[13] Cisco Inc. (2021). *ClamAVNet*. Accessed: Aug. 25, 2021. [Online]. Available: http://www.clamav.net/

[14] D. Pfaff, S. Hack, and C. Hammer, "Learning how to prevent return-oriented programming efficiently," in *Proc. Eng. Secure Softw. Syst.*, F. Piessens, J. Caballero, and N. Bielova, Eds. Cham, Switzerland: Springer, 2015, pp. 68–85.

[15] A. Waterman, Y. Lee, R. Avižienis, D. Patterson, and K. A. Asanović, "The RISC-V instruction set manual," User-Level ISA Version 2.0, Tech. Rep. UCB/EECS-2014-54, vol. I, 2014. Accessed: Apr. 27, 2022. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf

[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*, Mar. 2017, pp. 219–226.

[17] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, May 2007.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[19] J. Zhang, W. Chen, and Y. Niu, "DeepCheck: A non-intrusive control-flow integrity checking based on deep learning," 2019, *arXiv:1905.01858*.

[20] 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)*, 2011, pp. 40–51.

[21] IBM Cloud Education. (2021). *What are Convolutional Neural Networks*. Accessed: Apr. 5, 2022. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] L. Arras, J. Arjona-Medina, M. Widrich, G. Montavon, M. Gillhofer, K.-R. Müller, S. Hochreiter, and W. Samek, "Explaining and interpreting LSTMs," Tech. Rep., 2019, pp. 211–238, doi: 10.1007/978-3-030-28954-6_11.

[24] W. Huang and J. W. Stokes, "MtNet: A multi-task neural network for dynamic malware classification," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, vol. 9721, 2016, pp. 399–418.

[25] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "DL4MD: A deep learning framework for intelligent malware detection," in *Proc. Int. Conf. Data Mining (DMIN)*, 2016, pp. 61–67. Accessed: Apr. 5, 2022. [Online]. Available: https://www.proquest.com/openview/a090ba95404b143e4bbfbb4e0b6bebab/

[26] M. Al-Fawa'reh, A. Saif, M. T. Jafar, and A. Elhassan, "Malware detection by eating a whole APK," in *Proc. 15th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2020, pp. 1–7.

[27] Q. Le, O. Boydell, B. M. Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," in *Proc. Digit. Forensic Res. Conf., (DFRWS)*, vol. 26, 2018, pp. S118–S126, doi: 10.1016/j.diin.2018.04.024.

[28] M. Q. Li, B. C. M. Fung, P. Charland, and S. H. H. Ding, "I-MAD: Interpretable malware detector using galaxy transformer," *Comput. Secur.*, vol. 108, Sep. 2021, Art. no. 102371. Accessed: Apr. 5, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821001954

[29] B. N. Narayanan and V. S. P. Davuluru, "Ensemble malware classification system using deep neural networks," *Electronics*, vol. 9, no. 5, p. 721, Apr. 2020.

[30] R. Nagaraju and M. Stamp, "Auxiliary-classifier GAN for malware analysis," 2021, *arXiv:2107.01620*.

[31] Z. Moti, S. Hashemi, H. Karimipour, A. Dehghantanha, A. N. Jahromi, L. Abdi, and F. Alavi, "Generative adversarial network to detect unseen Internet of Things malware," *Ad Hoc Netw.*, vol. 122, Nov. 2021, Art. no. 102591. Accessed: Apr. 5, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1570870521001281%?via%3Dihub

[32] A. Darem, J. Abawajy, A. Makkar, A. Alhashmi, and S. Alanazi, "Visualization and deep-learning-based malware variant detection using OpCode-level features," *Future Gener. Comput. Syst.*, vol. 125, pp. 314–323, Dec. 2021. Accessed: Apr. 5, 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X21002272

[33] L. S. Tupadha and M. Stamp, "Machine learning for malware evolution detection," 2021, *arXiv:2107.01627*.

[34] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, L. Cavallaro, C. London, and R. Holloway, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 729–746.

[35] National Security Agency. *Ghidra*. Accessed: Apr. 5, 2022. [Online]. Available: https://ghidra-sre.org/

[36] Brad Conte. *B-Con Github Repository*. Accessed: Apr. 5, 2022. [Online]. Available: https://github.com/B-Con/crypto-algorithms

[37] Björn Samuelsson. *Simple CRC32 C-Code*. Accessed: Apr. 5, 2022. [Online]. Available: http://home.thep.lu.se/~bjorn/crc/

[38] Tactical Network Solutions. *Github: Port of Devttys0's IDA Plugins to the Ghidra Plugin Framework, New Plugins as Well*. Accessed: Apr. 5, 2022. [Online]. Available: https://github.com/tacnetsol/ghidra_scripts

[39] G. Gu and H. Shacham, "Return-oriented programming in RISC-V," 2020, *arXiv:2007.14995*.

[40] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-oriented programming on RISC-V," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 471–480.

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[42] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric Stat. Methods*, vol. 751, Hoboken, NJ, USA: Wiley, 2013.

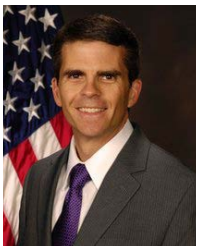
[43] G. Panesar and I. Robertson, "RISC-V processor trace version 1.0 4d009c4de4c68d547adb4adec307a438feb3d815," Tech. Rep., 2020. Accessed: Apr. 27, 2022. [Online]. Available: https://github.com/riscv/riscv-trace-spec/raw/e372bd36abc1b72ccbff31494a73a862367cbb29/riscv-trace-spec.pdf

[44] S. Bach, A. Binder, G. Montavon, F. Klauschen, K. M'uller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PLOS ONE*, vol. 10, no. 7, pp. 1–46, 2015.

[45] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Evaluating explanation methods for deep learning in security," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Sep. 2020, pp. 158–174.

DANIEL F. KORANEK received the B.S. degree in computer science from Cedarville University, and the M.S. degree in cyber operations from the Air Force Institute of Technology (AFIT), where he is currently pursuing the Ph.D. degree. He is also a Research Computer Scientist with the Air Force Research Laboratory. His current research interests include embedded systems security, malware detection, and machine intelligence.

SCOTT R. GRAHAM (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2004. He is currently an Associate Professor of computer engineering with the Air Force Institute of Technology, Wright Patterson Air Force Base, Dayton, OH, USA. His main research interests include the security of cyber physical systems, looking at the interaction of computer architecture, networks, and security.



BRETT J. BORGHETTI received the B.S. degree in electrical engineering from the Worcester Polytechnic Institute (WPI), Worcester, MA, USA, in 1992, the M.S. degree in computer systems from the Air Force Institute of Technology (AFIT), Dayton, OH, USA, in 1996, and the Ph.D. degree in computer science from the University of Minnesota, Twin Cities, Minneapolis, MN, USA, in 2008. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Graduate School of Engineering Management, AFIT. He has research experience in estimating human cognitive performance, statistical machine learning, genetic algorithms, self-organizing systems, neural networks, game theory, information theory, and cognitive science. His research interests include improving human–machine team performance in complex environments using artificial intelligence and machine learning.

WAYNE C. HENRY (Member, IEEE) received the B.S. degree in computer engineering from The Pennsylvania State University, in 2004, and the M.S. and Ph.D. degrees in computer engineering from the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, OH, USA, in 2011 and 2020, respectively. He is currently an Assistant Professor of electrical engineering at the AFIT. His research interests include cyber security, malware analysis, human–machine interaction, and information visualizations.