DEEPRETURN: A Deep Neural Network Can Learn How to Detect Previously-Unseen ROP Payloads without Using Any Heuristics

Xusheng Li [a], Zhisheng Hu [b], Haizhou Wang [a], Yiwei Fu [c], Ping Chen [d], Minghui Zhu [e], and Peng Liu [a]

[a] *College of Information Sciences and Technology, Pennsylvania State University, PA, USA*
E-mails: xul200@psu.edu, hjw5074@psu.edu, pliu@ist.psu.edu
[b] *Baidu Security, CA, USA*
E-mail: zhishenghu@baidu.com
[c] *GE Research, NY, USA*
E-mail: yiweifu1@gmail.com
[d] *JD.com American Technologies Corporation, CA, USA*
E-mail: ping.chen@jd.com
[e] *School of Electrical Engineering and Computer Science, Pennsylvania State University, PA, USA*
E-mail: muz16@psu.edu

Abstract. Return-oriented programming (ROP) is a code reuse attack that chains short snippets of existing code to perform arbitrary operations on target machines. Existing detection methods against ROP exhibit unsatisfactory detection accuracy and/or have high runtime overhead.

In this paper, we present DEEPRETURN, which innovatively combines address space layout guided disassembly and deep neural networks to detect ROP payloads. The disassembler treats application input data as code pointers and aims to find any potential gadget chains, which are then classified by a deep neural network as benign or malicious. Our experiments show that DEEPRETURN has high detection rate (99.3%) and a very low false positive rate (0.01%). DEEPRETURN successfully detects all of the 100 real-world ROP exploits that are collected in-the-wild, created manually or created by ROP exploit generation tools. DEEPRETURN is non-intrusive and does not incur any runtime overhead to the protected program.

Keywords: Return-Oriented programming, Intrusion Detection System, Disassembly, Convolutional Neural Network

1. Introduction

Due to broad deployment of W⊕X or Data Execution Prevention (DEP) [1, 2], code injection attacks (e.g., shellcode injection) are no longer very viable. Code reuse attacks, especially return-oriented programming (ROP) attacks [3–6], have recently risen to play the role code injection attacks used to play. A ROP attack proceeds with two phases. First, the attacker identifies a set of particular machine instruction sequences (i.e., “gadgets") that are elaborately selected from a binary executable or a shared library. Each gadget typically ends in a return instruction. Second, enabled by exploiting a (buffer overflow) vulnerability, the attacker overwrites part of the stack with the addresses of these gadgets; the addresses and register operands are placed at particular locations (on the stack) so that these gadgets will be executed

Detection Methods	R1	R2	R3	R4
Heuristic-based	×	✓	✓	×
Control Flow Integrity (CFI)	✓	✓	×	×
Signature-based	×	✓	✓	✓
Speculative code execution	✓	✓	×	×
Searching code pointers in the data region	×	✓	✓	✓
Statistical-based detection	×	✓	✓	×

Table 1
Limitations of existing detection methods against ROP attacks.

sequentially if the control flow is directed to the first one. By chaining gadgets together, the attacker is often able to perform arbitrary operations on the target machine [6].

Since ROP attacks are major threats to business-critical server programs, extensive researches have been conducted to defend against ROP attacks. The existing defenses focus on two perspectives, namely prevention and detection. ROP prevention methods aim to make launching a ROP attack itself very difficult. For example, ASLR (Address Space Layout Randomization) randomizes the memory addresses to make it difficult for an attacker to accurately locate any ROP gadgets. However, ASLR suffers from various forms of information leakage [7–9], code inference [10] and indirect profiling attacks [11], which significantly undermine the protection.

On the other hand, ROP detection methods aim to catch the ROP attack in action and then stop it. For example, CFI (Control Flow Integrity) checks whether a control flow transfer in a running program violates the CFG (Control Flow Graph). If so, the program is terminated. Existing ROP detection methods can be divided into six classes: (1A) Heuristics-based detection [12–14]. (1B) Fine-grained CFI [15–17]. (1C) Signature-based detection [18]. (1D) Speculative code execution [19]. (1E) Searching code pointers in the data region [20]. (1F) Statistical-based detection [21, 22].

Unfortunately, these methods are still quite limited in meeting four highly-desired requirements: **(R1)** high detection rate for ROP attacks; **(R2)** close to zero false positive rate; **(R3)** acceptable runtime overhead; **(R4)** minimal changes to protected programs and running environments.

In fact, (a) Class 1A and 1C detection methods could result in low detection rates; many heuristics or signatures are found not very hard to bypass. (b) Class 1B and 1D detection methods could cause substantial runtime overhead. (c) Class 1A, 1B, 1D, and 1F detection methods may cause substantial changes to existing (legacy) application software and even the running environment, thus they are not transparent. (Please refer to Section 2.2 for detailed discussion.) As a result, none of the existing ROP detection methods satisfy the above four requirements.

In recent years, deep neural network sees applications in the security field, e.g., fuzzing [23], log analysis [24], memory forensic [25], etc. Deep neural network has several clear advantages over traditional machine learning methods, for example it provides better accuracy than conventional models such as Support Vector Machine (SVM); it does not require expert knowledge to set thresholds for classification (detection) criteria; it requires less efforts on feature engineering and can be trained end-to-end using minimally preprocessed data.

In this paper, we study whether these advantages could be leveraged to address the aforementioned limitations of existing ROP payload detection methods. In particular, we propose a new ROP payload detection method, DEEPRETURN, which is the first to satisfy all of the above four requirements via deep learning. Regarding how DEEPRETURN works, firstly, our method is used as a "classification engine" to build a new kind of network Intrusion Detection System (IDS). It can be deployed in the

same way as a conventional network IDS such as Snort. Secondly, once deployed, our method works in the following manner: when network packets arrive and a reassembled protocol data unit (PDU) is obtained, our method takes two steps. (Step 1) Our method does ASL-guided PDU (i.e., application input data) disassembly and obtains a set of potential gadget chains. (Step 2) The potential gadget chains obtained in Step 1 are fed into a neural network classifier. The classifier identifies each potential gadget chain as either "ROP payload" or "benign data".

As we will show shortly in Section 7, DEEPRETURN achieves very high detection rate (99.3%) and very low false positive rate (0.01%), which satisfy R1 and R2. DEEPRETURN can also detect previously unseen ROP attacks. By "previously-unseen", we mean the malicious real-world ROP payloads which were not "seen" or used during the training process of the deep neural networks. Meanwhile, since DEEPRETURN can be deployed on a separate machine other than the protected server, it requires no changes to the protected program or the running environment (R4). DEEPRETURN also has no runtime overhead for the protected program (R3), which is an advantage over many other methods.

Despite the successful applications of deep neural network in other security problems [23–25], DEEPRETURN still faces several unique domain-specific challenges. Firstly, a deep neural network must be trained with proper data. Since ROP payloads only contain addresses of ROP gadget chains (please refer to Section 2.1), we should not train a classifier to directly distinguish ROP payloads from benign data. Otherwise, the signal-to-noise ratio is so low that the accuracy can be very poor where low signal-to-noise ratio means that most of the bytes in a packet payload are actually noise and have nothing to do with the ROP attack. Instead, we propose ASL-guided disassembly (Section 4) and create gadget-chain-like instruction sequences based on the addresses identified in benign data. In section 5, we also propose a viable method to generate sufficient real gadget chains. Simply put, the two datasets are both a set of instruction sequences and we train a classifier to distinguish the two. Also, to obtain comprehensive and representative benign training samples, we do ASL-guided disassembly on TB-level amount of raw input data (HTTP traffic, images, PDFs). We obtain 26k-105k benign training samples for different programs.

Secondly, we need to design a customized deep neural network for the detection of ROP payloads. We propose to use a convolutional neural network (CNN) as our classifier as it is good at capturing spatial structure and local dependencies. This corresponds to the nature of a ROP gadget chain that gadgets are chained with orders and adjacent instructions in a chain have meaningful connections with each other. These orders and connections in turn indicate whether an instruction sequence is indeed a real gadget chain or is formed merely due to the coincidental addresses in the data.

In summary, the contributions of this paper are as follows:

- We propose DEEPRETURN, a novel method for a network IDS to use in detecting ROP payloads. It combines ASL-guided disassembly and deep learning to classify reassembled PDU into either "ROP payload" or "benign data".
- To the best of our knowledge, DEEPRETURN is the *first* intrusion detection method that applies deep learning to mitigate the threat of ROP attacks. It sheds light on the applicability of deep learning to software system security problems.
- We have designed and implemented DEEPRETURN on Linux. We test it with several programs (e.g., Nginx). The evaluation results show that DEEPRETURN achieves very high detection rate (99.3%) and very low false positive rate (0.01%). More importantly, it can detect real-world ROP exploits collected in-the-wild and generated by widely-used exploit generation tools. We collect and create 100 real-world ROP exploits for a total of five vulnerable programs and DEEPRETURN successfully detects all of them. Meanwhile, DEEPRETURN does not require changes to the program or the running environment. It also incurs no runtime overhead to the protected program.

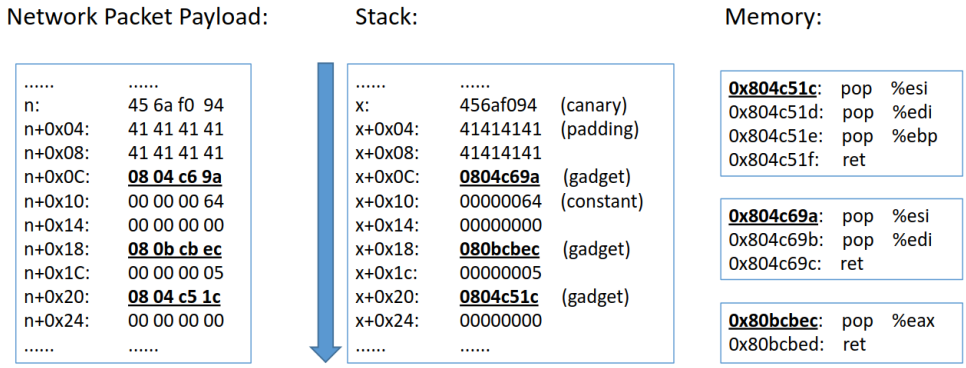

Fig. 1. Relationship between network packet payload, stack layout, and address space layout

2. Background

2.1. ROP Attacks

Despite years of effort to mitigate the threat, ROP attack remains a widely used method to launch exploits [26–29]. In a typical ROP attack, an attacker carefully crafts a gadget chain and embeds the addresses of the gadgets in a network packet. This packet exploits a vulnerability in the server program and causes the gadget to be executed one by one. Figure 1 shows the relationship of the network packet payload, the overflowed stack, and the program's memory address space layout. In fact, this comes from the blind ROP attack [30], which exploits a stack buffer overflow vulnerability in Nginx 1.4.0. As we can see, the addresses of gadgets (e.g., 0x804c51c, 0x804c69a) are embedded in the network packet. This relationship is an intrinsic characteristic of ROP attacks and it motivates us to search for addresses of potential gadgets in input data and then disassemble the corresponding instructions.

On the other hand, since only the address of ROP gadgets are contained in the ROP payloads, we should not directly label ROP payloads and benign data and then train a classifier on the two – because the signal-to-noise ratio is too low and it is difficult to train an accurate classifier.

2.2. Existing Detection Methods against ROP Attacks

In this subsection, we classify the existing detection methods against ROP attacks into six categories and explain why they fail to meet the four requirements set in Section 1.

Heuristic-based detection methods look for abnormal execution patterns. DROP [12] checks whether the frequency of executed return instructions exceeds a certain threshold. kBouncer [14] and ROPecker [13] detect abnormal pattern of indirect branches. Although these methods can detect conventional ROP attacks, they leverage certain heuristic-based thresholds as critical detection criteria. Göktaş et al. point out in [31] that they can be bypassed by carefully constructed gadget chains which violate the defender's assumptions.

Control Flow Integrity (CFI) [32–36] involves two steps, i.e., detection and response. CFI detects whether any control flow transfer violates the pre-defined control flow graph (CFG). If so, it terminates the program. Theoretically, CFI can detect all ROP attacks since they inevitably violate the CFG. However, due to the difficulty of point-to analysis, obtaining a perfect CFG is difficult (if not impossible) [37]. In practice, coarse-grained CFI implementations adopt an approximated CFG, which is not as secure as

it should be. Consequently, attackers can leverage the extra edges in the CFG to bypass them [34, 38–40]. On the other hand, fine-grained CFI provides a strong security guarantee at the cost of significantly high runtime overhead; e.g., 19% in [16] and 52% in [17], which is impractical. Meanwhile, CFI requires either instrumentation to the program binary or modifications to the compiler.

There are also signature-based detection methods against ROP, i.e., n-ROPdetector [18]. n-ROPdetector checks whether a set of addresses of API functions appear in network traffic. While this method can detect some ROP attacks, the address of APIs can be masqueraded to evade the detection. For example, an attacker can represent an address as the sum of two integers, and calculate the address at runtime.

Speculative-code-execution-based detection (ROPscan [19]) searches network traffic or files for any code pointers and starts speculative code execution from the corresponding instruction. If four or more consecutive gadget chains can be successfully executed, ROPscan considers the input data as a ROP exploit. Note the threshold four is empirically determined in the experiment; however, check my profile [20] finds that benign data can produce a gadget chain that has up to six gadgets. Meanwhile, real gadget chains need not to be very long to be useful. Thus the selection of the threshold is challenging. In fact, this also motivates the use of deep neural network as a classifier because no such thresholds need to be provided.

Check my profile [20] statically analyzes the data region and looks for any code pointers that form potential gadget chains. To avoid false positives, it checks whether the gadget chain eventually calls an API function – which most useful exploits will do. Unfortunately, due to the nature of static analysis, the detection can be bypassed when the address of API function is masqueraded, e.g., as a sum of two integers.

Statistical-based detection methods [21, 22] first extract certain meta-information (e.g., micro-architectural events) about the program execution and then train a statistical machine learning model (e.g., SVM) to distinguish ordinary execution from ROP gadget chain execution. They can detect various types of ROP attacks. However, they require instrumentation to acquire such information. Moreover, they use relatively small training dataset, which leaves plenty of spaces for improvement. For example, EigenROP [22] achieves 80% detection rate and 0.8% false positive rate, whereas we get 99.3% detection rate and 0.01% false positive rate. Note our false positive rate means 0.01% of the potential gadget chains, and the number of such chains is significantly smaller the number of program inputs (see Section 7), so our accuracy is much better than EigenROP.

3. Overview

If a PDU does not contain any 4-byte (8-byte) value which points to an instruction in a 32-bit (64-bit) system, the PDU is certainly not a ROP payload. Otherwise, the PDU is suspicious. A main challenge for detecting ROP payloads is that most suspicious PDUs are actually benign. Now we need to accurately judge if a suspicious PDU is a ROP payload or not. Since creative zero-day ROP attack scripts could be invented at any time, no IDS knows all the essential features of ROP attacks. This key observation reveals a fundamental limitation of the ROP detection methods that rely on known features (e.g., abnormal patterns of indirect branches, addresses of API functions being contained in a PDU); and DEEPRETURN makes it easier for security experts to conduct feature engineering. A main reason on why we leverage deep learning is that such learning models do not require any attack features to be explicitly identified.

Figure 2 illustrates the workflow of our proposed DEEPRETURN. Overall, our approach has two phases: the training phase and the production phase. During the training phase, we first collect two

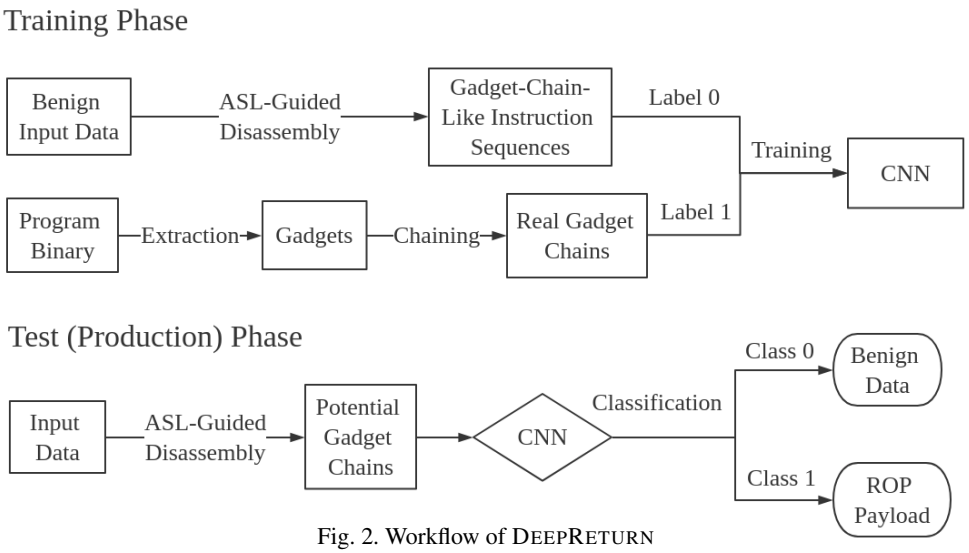

Fig. 2. Workflow of DEEPRETURN

training datasets, namely the gadget-chain-like instruction sequences and the real gadget chains. We get the gadget-chain-like instruction sequences by Address Space Layout (ASL)-guided disassembly (of the protected program's memory dump) based on the valid addresses identified in the benign input data. We extract real gadget chains from the protected program by chaining individual gadgets. Note by instruction we mean both opcode and operands (if any). We then train a deep neural network to classify the two datasets. Specifically, we choose to use a three-layer convolutional neural network (CNN) and represent the instructions byte-by-byte using one-hot encoding (Section 6.1).

During the production phase, after a PDU is obtained (re-assembled), we first identify any valid addresses contained in the PDU. Then we use the addresses to perform ASL-guided disassembly, which will result in a set of potential gadget chains. After that, we send each of them to the trained CNN. A warning is raised if any of these potential gadget chains is classified as real ROP attack. Conversely, if the ASL-guided disassembly does not produce a potential gadget chain, or the chain is classified as benign by the CNN, the input data is considered benign.

There will be no overhead on the production server since each trained deep learning model is deployed to achieve Intrusion Detection instead of Intrusion Prevention, it does not introduce any delay to network packets if the (HTTPS) reverse proxy forwards a copy of each decrypted payload to the IDS VM that runs the deep learning model. It should be noticed that the IDS VM and the production VM that runs the being protected program are usually two completely separate VMs.

Note we do need a classifier as part of the DEEPRETURN because if we treat all of the inputs that have addresses in it as ROP payloads, we can cause high false positive rate and substantial denial of service. Meanwhile, the ASL-guided disassembly is also necessary because otherwise the signal-to-noise ratio is very low. An HTTP request can be as large as a 2MB image. However, suppose a ROP gadget chain has 20 gadgets, then only 20 addresses are present in the payload, which means 80 bytes. Viewing ROP payloads as a "haystack", the addresses embedded in by an attacker are in fact some "needles". If we treat a whole payload as a training sample, then any trained neural network will very likely be primarily capturing the features of the haystack instead of the needles. If this is the case, a very low classification accuracy is unavoidable.

We choose CNN as the classifier because ASL-guided disassembly outputs instruction sequences which have strong spatial structures and local dependencies. The appearance of pairs of instructions and the orders of instructions could indicate different nature of an instruction sequence. CNN can represent the abstract nature (e.g., benign or real gadget chain) into compositions of simpler concepts (e.g., locations of instructions), which is important for the classification.

This unique combination of ASL-guided assembly and CNN enables us to solve the ROP payload detection problem without any software instrumentation. The only thing that the DEEPRETURN needs to know about is a memory dump of the protected program. If we train a classifier to distinguish ordinary execution trace from ROP gadget chain execution trace, we need software instrumentation to monitor the execution trace, which is against our goal.

4. ASL-Guided Disassembly and Gadget-Chain-Like Instruction Sequences Generation

This section describes the details about the ASL-guided disassembly. ASL-guided disassembly treats bytes in data as addresses and checks if they point to gadget-like or gadget-chain-like instruction sequences. In the training phase, we use ASL-guided disassembly to collect gadget-chain-like instruction sequences as training data for the neural network. In the production phase, we use ASL-guided disassembly to identify any potential gadget chain in input data.

It is important that ASL-guided disassembly works as closely as possible to how ROP attacks work. ROP attacks work by chaining multiple gadgets together. Each of these gadgets ends in an indirect branch, i.e., `ret`, `jmp`, or `call`, which starts the execution of the next gadget. The proposed ASL-guided disassembly examines the given payload data, and checks if they can form a valid gadget chain.

An observation of the stack layout related to ROP attacks is that ROP attacks do not put gadgets (instructions) themselves on the stack. Instead, they put addresses of the gadgets on the stack, as well as any constant values that are going to be popped from the stack and used by the gadget chain[41]. Since we do not know where the first address of the gadget starts in the data, we process every byte and interpret the starting four bytes as an address, and check if it points to a valid gadget. Details are discussed in Section 4.1.

After we identify an address and the corresponding valid gadget, we need to look for other ones after it. It is tempting to track the stack pointer changes caused by the current gadget and find the next possible address directly. However, an ROP attack is capable of leveraging indirect function calls, e.g., `call eax`, where the code pointed to by register `eax` is hard to determine statically. So instead of tracking the stack pointer changes, we consider every four bytes following the identified address in the given data to see if it points to another gadget. In this way, we can capture all the gadgets, if the payload indeed intends to execute a gadget chain. Details are discussed in Section 5.

4.1. Disassembly of Individual Addresses

We first create a memory dump of the protected program. The memory dump contains the addresses and contents of all memory pages that are marked as executable. Oftentimes, these pages are mapped from the `text` segment of the protected program or any loaded shared libraries (e.g., libc.so, etc). The ROP gadgets must fall inside the memory dump, otherwise it is not executable and the attack will fail.

Then we consider every four bytes (on 32-bit system)[1] in the input data as an address. If an address does not appear inside any one of the dumped pages, we ignore it because it cannot be the start of a ROP

[1] In this paper, we only consider 32 bit systems and leave it for a future work to experiment with 64 bit systems.

gadget. Here we do not limit our search space to non-randomized modules because attackers can bypass the ASLR in multiple ways.

We start disassembling from any identified addresses using Capstone [42]. The disassembling can stop in two ways: (1). An invalid or privileged instruction is encountered or the disassembly reaches the end of code segment. In this case, the current address is ignored in subsequent analysis. (2). An indirect branch, i.e., `ret`, `jmp`, or `call` is encountered. Then the instruction sequence (from the instruction at the starting address all the way to the indirect branch) is considered a gadget-like instruction sequence.

4.2. "Chaining" of Gadget-Like Instruction Sequences

Having obtained the set of gadget-like instruction sequences, we need to figure out how they could be chained together, in a similar way as an attacker chains gadgets into a gadget chain. Specifically, if we find an address at offset n in the data which points to a gadget-like instruction sequence, we check if any one of the next ten addresses, i.e., address at $n + 4$, $n + 8$, $n + 12$,...,$n + 40$ in the data also points to a gadget-like instruction sequence. If so, we "chain" it together with the previous gadget-like instruction sequence and repeat the process. Otherwise, we end the "chaining" process.

For any "chain" that has at least two addresses, we collect the corresponding gadget-like instruction sequences and concatenate them into a gadget-chain-like instruction sequence.

Note the maximum ten is determined by the observation that most gadgets in our dataset only `pop` less than five integers from the stack to the registers, and all of them pop less than ten integers; so ten is sufficient to capture the next address in a "chain".

In other words, a ROP attack cannot spread the addresses of gadget chains in the payload arbitrarily; otherwise, even if the control flow is successfully hijacked, the subsequent gadgets will not execute one by one – because upon return from the previous gadget, the address of the next address is not on the top of the stack.

When we look for the next "chain", we skip the addresses and the corresponding instruction sequences that are already part of a "gadget chain". For example, if a "chain" contains five "gadgets", next we start from the sixth address and repeat the "chaining" process. We repeat the process on every address to collect all possible gadget-chain-like instruction sequence. In this way, we get the first training dataset.

To efficiently implement the above algorithm, we start multiple parallel threads to analyze different addresses. Moreover, if an address is already examined and found to be pointing to a gadget-like instruction sequence, we cache it in a global table. In this way, the disassembly and analysis is not repeated on the same address. Besides, we also process multiple inputs simultaneously to utilize all CPU cores.

The reader may take Figure 1 as an example of the ASL-guided disassembly during the production phase. Suppose we start from offset n of the data, we first encounter a candidate address 0x456af094. We find this address is not inside the memory dump, which indicates it is not mapped or not marked as executable in the protected program, so we move on to the next 4 bytes starting at offset n+0x04. The next two addresses are 0x41414141 and they both do not lead to a potential gadget chain. Next, we move to offset n+0x0c and process 0x0804c69a. Note this address is inside the memory dump and we start disassembling from it. The result is a potential gadget with three instructions: `pop esi; pop edi; ret;`. We continue the process and then identify two other addresses (0x080bcbec, 0x0804c51c) and their corresponding instructions. Eventually, we end up with a potential gadget chain with three gadgets (`pop esi; pop edi; ret; pop eax; ret; pop esi; pop edi; pop ebp; ret;`). In the training phase, however, when we collect gadget-chain-like-instruction sequences, we process benign input data using the same approach.

It is noteworthy that we start ASL-guided disassembly from EVERY byte of the input data. This is because we are dealing with data, so code or memory alignment actually does not apply, and any four-bytes can be an address. In fact, in Figure 1, we also treat the four-byte data at n+1 (0x6af09441), n+2(0xf0944141) and n+3(0x94414141) as addresses and start ASL-guided disassembly. Though they do not lead to gadget-like instruction sequences and are ignored in the subsequent analysis.

5. Real Gadget Chain Generation

The real gadget chain dataset is created by chaining real individual gadgets together. There are several existing tools to automate gadget chains generation; e.g., rp++ [43], ROPgadget [44], ropper [45], PSHAPE [46], ROPER [47]. However, the existing tools cannot be directly used to generate the real gadget chain dataset due to three main reasons: 1. The number of generated gadget chains is small. Many existing tools usually build ROP exploits for one specific scenario; e.g., `execve` or `mprotect`, which leads to a small number of real gadget chains. 2. Existing tools usually use gadgets whose lengths are as short as possible to reduce side-effects on other registers, the stack, or flags. This makes the dataset not comprehensive. 3. The generated gadget chains might cause crashes due to accesses of unmapped memory.

We decide to generate the real gadget chains in a new way. Our idea is to first generate a lot of candidates, and then filter out those invalid ones. We use ROPgadget [44] to extract individual gadgets from the program binary. Then the gadgets are added to the chain in such a way that every register is initialized before the chain dereferences it, as the execution may otherwise lead to a crash. To avoid crashes, we have to solve the side effect of gadgets. Take two gadgets "mov [esi], 0x1; ret;" and "mov eax, 0x1; jmp [esi];" for instance. There exists a side effect caused by the gadget "mov eax, 0x1; jmp [esi];" unintendedly changing the value of EAX, and a register usage conflict for ESI, which is used for setting a memory as 0x1 and setting the target address for jump instructions. To solve the side effects, we remove all the gadgets that contain the memory usages, and make sure that no two gadgets read one register without write operation between them. To make the dataset more comprehensive, we also combine both short and long gadgets.

After that, we use the CPU emulator Unicorn [48] to validate the generated gadget chains. We first analyze how the individual gadgets interact with the stack pointer. For example, if a gadget pops two integers into registers, we know that after the execution of this gadget, the new `esp` value will become `esp+0xc`. We arrange the addresses of gadgets on the stack inside Unicorn according to their stack interaction. We then start emulation from the first gadget and observe if the gadgets can be executed one-by-one correctly. If not, this gadget chain is filtered out. One exception to the emulation is all function calls (e.g. `call eax`) are assumed successful and the corresponding function call is skipped. In this way, we make sure the generated gadget chains are all valid (they are not necessarily useful for attackers).

We can generate a huge amount of real gadget chains in this way. Although the rule of the thumb for training a neural network classifier is to ensure the dataset is balanced, in our problem domain generating benign gadget-chain-like instruction sequences takes significantly more resources than generating real gadget chains. Therefore, we first generate more real gadget chains than gadget-chain-like instruction sequences, and then conduct experiments to see if DEEPRETURN can handle unbalanced data.

Furthermore, we make sure the length (in bytes) distribution of the real gadget chain datasets is similar to that of the gadget-chain-like instruction sequence dataset; otherwise, even if the classifier does a good

job to distinguish the two datasets, it may leverage the length information too much rather than learn anything about the data.

6. Neural Network Classification

We formulate the ROP payloads detection problem as a classification problem where our goal is to discriminate ROP gadget chains from gadget-chain-like instruction sequences. We now need to tackle two main challenges in order to successfully apply deep neural networks. First, deep neural networks require a huge amount of training data to perform well. For example, deep neural networks perform worse than some traditional classifiers when the dataset has less than 100 samples [49]. As mentioned in Section 4.2 and Section 5, it is challenging to get a large number of gadget-chain-like instruction sequences. In fact, we use ASL-guided disassembly to disassemble TB level of raw input data to generate enough instruction sequences (please refer to Section 7.1). Second, different types of deep neural networks are suitable for different types of data; e.g., convolutional neural networks (CNN) work well in image classification since the data has spatial structures [50], and recurrent neural networks (RNN) with long short-term memory (LSTM) [51] can deal with temporal structures in spoken language. It is very important to design the correct architecture of the deep neural network according to the nature of instruction sequence data in our task.

6.1. Data Representation and Preprocessing

In order to make our two datasets (i.e., the gadget-chain-like instruction sequences and the real gadget chains) tractable for the neural network, we need to convert every instruction in them into numerical data. Recall that instructions are binary data by nature. We first convert every byte in an instruction sequence into its hex value (0-255). For example, the instruction sequence "mov eax, 0x1; pop edx; ret" is converted to "[0xb8, 0x01, 0x00, 0x00, 0x00, 0x5a, 0xc3]".

We can simply scale these values to 0-1 by a division of 255 and then input them into the neural network. However, this is inappropriate for our data. Since neural networks multiply the input data with certain weight parameters, such representation leads to an implicit relative order between different byte values. For example, in the above example, the neural network implicitly assumes the "ret" (0xc3) is larger than "pop edx" (0x5a), which is meaningless regarding instructions.

To address this problem, we instead use one-hot encoding. We represent each byte value by a 256×1 vector, with all but one positions to be zero, and the position that corresponds to the byte value to be one. For example, the "ret" (0xc3, 195) will be represented by {0,...,0,1,0,...,0}, where we have 195 zeros in front of the one, and 60 zeros behind the one.

In other words, an instruction sequence that has n bytes is represented by $X = \{\overrightarrow{X_1}, \overrightarrow{X_2}, ..., \overrightarrow{X_n}\}$, where $\overrightarrow{X_i}$ is a 256×1 one-hot vector. Alternatively, one can view the input as a n by 256 matrix.

Since instruction sequences usually have different lengths, padding is applied to make them have the same length. We first find the longest instruction sequence (in bytes). For shorter ones, we append the one-byte `nop` (0x90) instruction at the end of them until they all reach the same length.

An alternative way to preprocess the data is to do word embedding [52], which is widely used in Natural Language Processing (NLP). Word embedding maps every word in the data to a fixed length vector. In the evaluation section, we show that embedding, compared to one-hot encoding, provides similar accuracy while requires longer training time. Consequently, one-hot encoding is more suitable for our task.

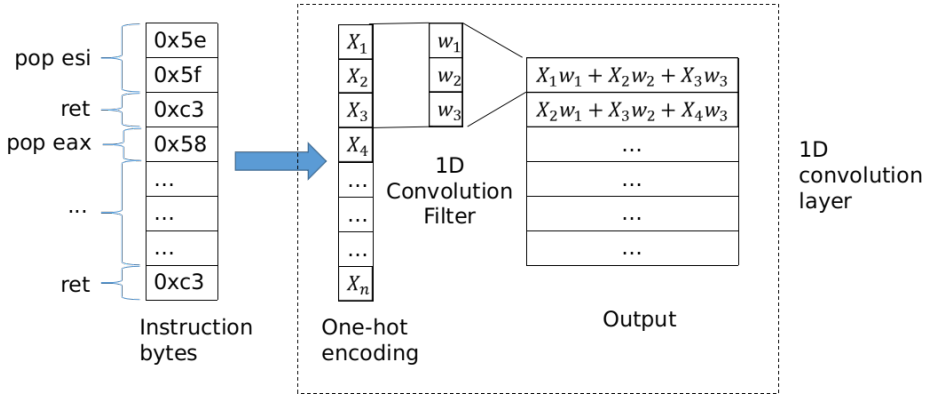

Fig. 3. Data Representation and the first 1D Convolution Layer

After the data preprocessing, we use a customized neural network to classify whether a potential gadget chain is benign or real. The architecture of our neural network (shown in Fig. 4) is a convolutional neural network (CNN), which is a feed-forward neural network that is good at learning data with spatial structures and capturing their local dependencies. Typically, a CNN has one input layer, multiple convolution layers, multiple activation layers, and one output layer. In particular, all of the convolution layer in the network is 1D convolution layer which is shown in Figure 3.

The 1D convolution layer involves a mathematical operation called `convolution`. Suppose X represents the one-hot encoded instruction bytes, w represents the convolution kernel (weight vector) whose length is m. Then the convolution between X and w is a matrix $X * w$, whose *ith* column can be written as:

$$(X * w)_i = \sum_{j=1}^{m} X_{i-j+m/2} \cdot w_j \tag{1}$$

The convolution aggregates information from adjacent bytes. This information includes certain byte values, the ordering of bytes, etc. The convolution layer is followed by a nonlinear activation layer (e.g., ReLU) to denote whether certain information is present or not. We stack three layers of convolution and activation to gradually capture higher level of information of the input bytes, e.g. the presence of a certain gadget, the ordering of different gadgets, the repetition of certain patterns, etc. The higher level of information is more abstract and difficult to extract or represent (similar to the case in image classification), but it is more related to the classification task, i.e., whether the chain is benign or real. The last activation layer is followed by a fully connected layer and another activation layer output to give a classification output, benign (0) or real (1).

Since the input X is fixed, only the weights w influence the output. These values are not determined via heuristics or expert knowledge, as in many previous ROP detection methods. Instead, they are trained (recursively updated) by minimizing the differences between the true labels and the network's outputs. For details about CNN, please refer to [53, 54].

The architecture of our neural network is illustrated in Figure 4, which is a zoom-in version of the CNN in Figure 2. We first use a convolutional layer with 64 convolution filters with the kernel size of 7 (length of the convolution filter is 7). Then we perform batch normalization (BN) before a nonlinear activation

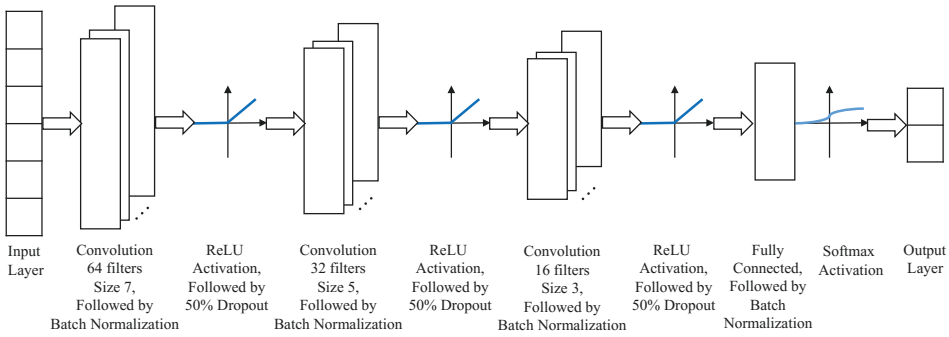

Fig. 4. Architecture of the CNN used in DEEPRETURN

function, which in this case is a rectified linear unit (ReLU). ReLU is a simple rectifier with the form of $f(x) = \max(x, 0)$. After the ReLU activation, we apply a 50% dropout to prevent overfitting. Then we repeat this Convolution-BN-ReLU-Dropout structure for two more times. In particular, we use 32 convolutional kernels with the size of 5 and 16 convolutional kernels with size 3 for the next two layers. Then the output from the last Convolution-BN-ReLU-Dropout structure is feed into a fully connected layer. Finally, we use the softmax activation function as a classifier. It is worth mentioning that we do not include pooling layers which are widely used in image classification tasks, because our entire input is meaningful and downsampling our input vector would yield a completely different gadget chain.

We use a grid search [55] to fine-tune the above configurable hyper-parameters (e.g., dropout rate, the filter sizes, etc) in our model and find the best set of values. Details are provided in Section 7.

This network includes some modern techniques in deep learning such as BN [56] and dropout [57] to improve the performance. BN is a method to reduce internal covariance shift by normalizing the layer inputs in the neural network [56], and has been shown to improve learning. Dropout is a simple way to prevent a neural network from overfitting by randomly dropping a percentage of neurons during the training phase. These methods have been widely used in recent years and seen great successes.

6.2. Training CNN

We train the CNN after collecting training samples (gadget-chain-like instruction sequences and real gadget chains) for a specific program. We use an optimization procedure called stochastic gradient descent (SGD) with learning rate 0.1 and momentum [58]. In particular, we input a mini-batch of the training samples into the network, then compute the outputs, the errors and the gradients for the input. The gradients are calculated by backpropagation to determine how the network should update its internal weights; e.g., w_1,w_2 and w_3 in Figure 3, which are used to compute the representation in each layer from the representation in the previous layer. We repeat this process for many mini-batches until the errors converge or the maximum training epoch is reached. In practice, we also test another optimizer, i.e., Adam, and both the accuracy and the convergence speed are similar to those of the SGD, so we choose to use SGD as our optimizer.

Note we set forth a goal (R2) to reach close to zero false positive rate for the DEEPRETURN. This is motivated by the fact that false positives can really irritate the system admin and undermine the usefulness of the detection method, or even impact its security. To reduce the false positive rate, we use a common technique, i.e., penalizing false positives more than false negatives. Specifically, we use the notation of "penalizing factor" to describe how much more a false positive contributes to the loss

function compared to a false negative. A penalizing factor 1 means no difference between a false positive and a false negative, while a penalizing factor of 5 means a false positive contributes to the loss 5 times more than a false negative does. In the evaluation section, we empirically decide a value to achieve a very low false positive rate and a still high detection rate.

7. Evaluation

In this section, we evaluate DEEPRETURN on real-world programs. In particular, we want to answer the following questions. **1.** Does DEEPRETURN has high detection rates and low false positive rates for multiple programs? **2.** Can DEEPRETURN detect real-world ROP exploits? **3.** Can DEEPRETURN detect ROP exploits that can bypass other ROP detection methods, e.g., CFI? **4.** Is the speed of DEEPRETURN satisfiable?

We train a CNN for each of the five tested programs, namely nginx 1.4.0, apache 2.4.18, proftpd 1.3.0a, vsftpd 3.03, ImageMagcik 7.08, respectively. The tested programs are widely adopted server-side applications (i.e., web servers, ftp servers, image processor). On one hand, we admit that one limitation of this work is that only five applications are used in the evaluation experiments. On the other hand, we note that it is extremely difficult to obtain the benign training data examples for a large number of programs. For example, in order to obtain the benign training data examples for the five applications, we need to use around 1TB data in the ASL-guided disassembly stage of our work flow. Without using a huge amount of program input data, the number of benign training data examples one can obtain would fall short of the requirements of deep learning. Also, it is very costly when 1TB data must be processed; for example, we paid around $1,000 to rent around 100 virtual CPU cores from a cloud service provider. Since we could not afford to rent the 100 virtual CPU cores for a long period of time, we were unable to evaluate the proposed approach against a large number of programs.

Besides, in terms of the representativeness of the five applications, we found that since the representativeness is closely related to the particular business/application logic of an enterprise, it is actually difficult to evaluate the representativeness objectively. For example, the real HTTP traffic used in our throughput experiments was collected in our research lab. Although one could argue that the HTTP traffic is representative in a university environment, it is hard to generalize such an argument.

We compile, execute and memory-dump them on Ubuntu 16.04 32bit system. Note one specific CNN should be trained for one program because the (training + testing) dataset for one program is different from that for other programs. We collect and create 20 working ROP exploits for each of the five tested programs and DEEPRETURN successfully detects all of them. DEEPRETURN can also detect all 10 ROP exploits that bypass CFI.

7.1. How is Big Data Used to Generate Benign Training Dataset?

Collecting abundant training data is one of the most challenging tasks when utilizing the neural network. We use the following three input datasets to generate the gadget-chain-like instruction sequences: (A) a medium size HTTP traffic dataset in [59]; (B) all images in ImageNet [60] (1.2 TB in total); (C) all PDF files from arXiv [61] (800 GB in total). We use all (A)(B)(C) for the two web servers, i.e., nginx and apache, to do ASL-guided disassembly (described in Section 4) and generate gadget-chain-like instruction sequences. Similarly, we use all (B) for the image processor (i.e., ImageMagick), and all (B)(C) for the two ftp servers (i.e., proftpd and vsftpd) to do the same job. As a summary, we consider

Program	Memory Dump Size	HTTP Data	ImageNet	arXiv PDF	Input Data Size	Generation Time	# of Gadget-Chain-Like Instruction Sequences	# of Real Gadget Chains
nginx 1.4.0	3.2 MB	✓	✓	✓	2 TB	7 hours	40,674	40,674
apache 2.4.18	3.7 MB	✓	✓	✓	2 TB	7 hours	81,127	81,127
proftpd 1.3.0a	2.4 MB	×	✓	✓	2 TB	7 hours	26,020	26,020
vsftpd 3.03	4.5 MB	×	✓	✓	2 TB	7 hours	105,057	105,057
ImageMagick 7.08	8.5 MB	×	✓	×	1.2 TB	4 hours	46,224	46,224

Table 2
Generation of Gadget-Chain-Like Instruction Sequences

all valid inputs for the target program. For example, all three datasets all typical traffic that pass web servers; image processor, however, only deals with images.

To quickly process the huge amount of data, we use a Google Cloud Platform instance with 96 vCPUs. The memory dump size, input data, generation time, and the number of generated gadget-chain-like instruction sequences are shown in Table 2. As a brief summary, we generate 26k to 105k gadget-chain-like instruction sequences for different programs, respectively.

It can be verified here that we do need a classifier in DEEPRETURN. Otherwise, if we simply do ASL-guided disassembly on input data and treat all potential gadget chains as real gadget chains, all gadget-chain-like instruction sequences generated above will become false positives. Once a deep learning model is trained for a program we intend to protect, its deployment will assume a representative server setup in practice. However, it should be noticed that the model training stage is a separate offline machine learning process, where the training data is prepared in advance. Since big data is usually involved in preparing the training data, we assume that some enterprises do not have the needed computing resources, and hence view a public cloud provider (i.e. Google Cloud) as a viable option.

As explained in Section 5, for each program, we generate the same number of real gadget chains as gadget-chain-like instruction sequences. The generation is rapid and is done on a local workstation. Now that we have five datasets for the five programs, respectively. Each of them contains two smaller datasets: the gadget-chain-like instruction sequences and the real gadgets chains, which have the same number of samples.

7.2. Tuning the hyper-parameters of DEEPRETURN

Hyper-parameter tuning is a very important aspect of deep learning applications. Different hyper-parameter sets can lead to very different results. In this subsection, we use grid search [55] to empirically find the best hyper-parameters for our CNN. Grid search exhaustively searches the hyper-parameter space to look for the combination of hyper-parameters that yields the best result. In DEEPRETURN, we have a handful of hyper-parameters and the grid search works well.

We implement our model using Keras [62] and run it on NVIDIA Tesla P100 GPU. We ran the experiment under each setting for three times and take the average value of the accuracy. This averaged accuracy is the primary factor to compare different hyper-parameter settings. However, simply rely on accuracy leads to a problem. For example, let us assume 128 convolution filters already leads to the best result. Doubling the number and make it 256 is likely to be a waste of training time – but the accuracy might see a tiny increase, say 0.05%. So we also consider how fast the hyper-parameters lead to the convergence of error rate, measured in training epochs. If hyper-parameter set a is less than 0.05% better

than set b in accuracy, but at the same time the training epochs is 20% more, we favor set b. In this way, we can not only get high accuracy, but also avoid unnecessary training time.

We use the nginx dataset to fine-tune the hyper-parameters. We split the whole dataset into two subsets: a training dataset that is 80% of the whole dataset (80% of the samples from both labels), selected at random; a test dataset that is the rest 20% of the samples. We use the training dataset to train the model and fine-tune the hyper-parameters.

The parameters we aim to tune and their corresponding candidate values are listed in Table 3. After a grid search, we find the best set of hyper-parameters, which is highlighted in Table 3 (these are also the values included in Section 6). We achieve 99.6% accuracy under this best setting.

Hyper-parameters	Candidate Values
# of filters	(32, 16, 8)[2], (48, 24, 12), **(64, 32, 16)**, (96, 48, 24), (128, 64, 32)
filter size	(3, 3, 3), (5, 3, 3), **(5, 5, 3)**, (7, 5, 3), (9, 5, 3), (9, 7, 5)
dropout rate	0.2, **0.5**, 0.8
learning rate	1, **0.1**, 0.01, 0.001

Table 3
Candidate values and the best value of hyper-parameters for DEEPRETURN

We also consider using word embedding to preprocess the data, which works very well in NLP. But it does not help in our task. It produces similar accuracy to one-hot encoding but the required training time is 30% longer. The longer training time can be explained by the fact the embedding layer itself needs to be trained along with the deep neural network. Also, instruction sequences are inherently different from language data, so the embedding is unable to improve the result.

We do not use grid search to find the best penalizing factor (introduced in Section 6.2). In fact, we not only want to minimize the false positive rate, we also want to maintain a high accuracy. However, using penalizing factor to decrease false positive rate inevitably increase false negative rate and may impact the overall accuracy, so a decision on the balance is better made manually. To illustrate this, we test an extremely large penalizing factor, 100, and the false positive rate is 0. However, the false negative rate is high and the overall accuracy drops to 92%, which is undesirable. On the other hand, if we use a trivial penalizing factor 1, the false positive is 0.1%. After manually testing several penalizing factors, we find a factor of 5 to be a good balance, where the false positive rate is 0.01% and the detection rate is 99.3%. Since we have same amount of samples for the two labels, so $accuracy = (detection\ rate + (1 - false\ positive\ rate))/2$, thus the overall accuracy is still 99.6%.

A quick estimation demonstrates what 0.01% false positive rate means in real world. Suppose a web server receives 10TB incoming traffic per day, since the number of potential gadget chains is rather small (40, 674 out of 2TB benign data), it is estimated only 20 false positives will be reported in 24 hours. Note the 10TB incoming traffic is equivalent to 1Gbps all the time during 24 hours, which is not a trivial amount, so the estimation is not under-estimating.

[2]This means the first layer has 32 filters, the second has 16, and the third has 8. The notation for filter size is interpreted in the same way

Program	CVE-ID	# of Chains for Training	Detection Rate	False Positive Rate	Training Time	Classification Speed
nginx 1.4.0	2013-2028	65,078	99.3%	0.01%	45min	27,116 chains/s
apache 2.4.18	N/A	129,803	98.7%	0.04%	94min	27,120 chains/s
proftpd 1.3.0a	2008-2950	41,632	98.3%	0.05%	31min	26,730 chains/s
vsftpd 3.03	N/A	168,091	99.5%	0.02%	118min	26,020 chains/s
ImageMagick 7.08	N/A	73,958	98.1%	0.02%	55min	27,231 chains/s

Table 4
Evaluation results on five tested programs

7.3. Can DEEPRETURN *Accurately Detect Gadget Chains in the Test Dataset?*

In this subsection, we evaluate the accuracy of DEEPRETURN against the five commonly used programs.

For every program, we select 80% of the dataset as training data the rest 20% as test data. We use the best hyper-parameters from the previous subsection, make 5-fold cross validation and the evaluation result is shown in Table 4. We reach 99.3% detection rate and 0.01% false positive rate for nginx 1.4.0, 98.7% detection rate and 0.04% false positive rate for apache 2.4.18, 98.3% detection rate and 0.05% false positive rate for proftpd 1.3.0a, 99.5% detection rate and 0.02% false positive rate for vsftpd 3.03, and 98.1% detection rate and 0.02% false positive rate for ImageMagick 7.08, respectively. As we can see from the results, DEEPRETURN has very high detection rates and very low false positive rates, and work well on different programs.

The high accuracy shows that roughly 50k-100K samples are sufficient to train an accurate classifier for potential gadget chains. In contrast, for image classification, millions of samples are required to train neural networks [60].

Besides CNN, there are also other candidate classifiers, e.g., RNN (LSTM), MLP and SVM. Our evaluation shows that the CNN is best suitable for this particular task. Due to space limitation, detailed comparisons are shown in Appendix A.

It is also noteworthy that the best hyper-parameters tuned on nginx dataset work well across different programs. This indicates although the specific instruction sequences are different in different programs (thus a dedicated CNN is needed), it has something in common across different programs.

We observe that the training time roughly increases linearly proportional to the size of the training data. However, once the neural network is trained, the test speed is fast and less sensitive to the amount of test data. In the first row, it only takes 0.6 seconds to classify all test data for nginx (20% of all samples, 16,270 in total).

We also test what would happen if the amount of training data is small. In particular, we train the same model with 5% of the nginx data (4K samples). We achieve 95.1% detection rate and 1.2% false positive rate. They are considerably inferior to the best achievable result. It validates that deep neural networks work better with larger training data since the networks can discover intricate features in large data sets instead of over-fitting small training data and missing key features.

7.4. Can DEEPRETURN *Accurately Detect Real-World ROP Exploits?*

In this subsection, we test DEEPRETURN against real-world ROP exploits that are collected in-the-wild or generated by ROP exploit generation tools (i.e., Ropper [45] and ROPC [63]).

The motivations behind these experiments are two-fold. Firstly, in the previous subsection, we show that the trained deep neural network of DEEPRETURN is an accurate classifier with a high detection rate and low false positive rate. However, despite our efforts to make the real gadget chains valid, they are not necessarily useful from an attacker's point of view. Therefore, we need a more direct evaluation here to show DEEPRETURN is able to detect real-world ROP exploits (that are not part of the training data). Secondly, since the real gadget chains are directly generated (not from ASL-guided disassembly), we also need to show the ASL-guided disassembly is capable of correctly identify the addresses of gadgets in a real gadget chain, which is the basis for our approach.

Among the five tested programs, nginx and proftpd have known vulnerabilities and we directly exploit them with ROP attack. For the rest three, we use the latest version so there is no known vulnerability. Instead, we inject a trivial stack buffer overflow vulnerability into each of them to make the exploiting possible. In fact, as long as the exploit is a ROP attack, the underlying detail of the vulnerability has nothing to do with the effectiveness of our method, so the injected vulnerabilities do not undermine our evaluation.

For each vulnerable program, we first obtain one ROP exploit that leads to a shell. For nginx, we use the attack script published in BROP [64]. For proftpd, we use the attack script published in Exploite-DB [65] but change it to launch a ROP attack. For each of the rest three, we create a working exploit that leads to a shell. After that, we manually mutate the exploit to generate 4-5 more samples for testing. For example, we can exchange the order of several ROP gadgets without changing the behavior of the exploit. We also substitute gadgets with new gadgets that have the same effects.

To further create test samples, we use Ropper [45] to generate ROP exploits that execute mprotect or execve. Ropper can generate different exploits because we can block the used gadgets and force it to generate new ones. Meanwhile, we create ROP exploits with ROPC [63], which is a ROP compiler that can compile scripts written in a special type of language (ROP Language, ROPL) into a gadget chain. Although in Section 5 we are unable to use these tools to generate a large number of real gadget chains, it successfully generate sufficient amount of samples for us to test DEEPRETURN.

To sum up, for each of the five vulnerable programs, we collect and create 20 ROP exploits. All of them are manually verified to be working. i.e., achieving their designed functionality, for example getting a shell or executing mprotect.

We then test DEEPRETURN against all of the 100 exploits. We first observe that the ASL-guided disassembly successfully extracts the gadget addresses embedded in the payload and obtains the corresponding instruction sequences. Subsequently, the DNN correctly classifies all of the potential gadget chains as real gadget chains. This demonstrates the ASL-guided disassembler and the neural network synergize well and the system works as designed. DEEPRETURN is able to detect real-world ROP exploits.

Thanks to the generalization capacity of DNN, the detection capacity of DEEPRETURN is not restricted to the ROP attacks it sees during training. In fact, it detects the blind ROP attack which is not part of the training data. Furthermore, ROP exploits generated by ROPC can have more than 100 gadgets, which are three times longer than the longest one in the training dataset. They are also correctly detected.

7.5. Could Transfer Learning Be Leveraged to Make DEEPRETURN *More Scalable?*

We conduct an experiment to test whether the technique of transfer learning could be adopted by DEEPRETURN. Essentially, we prepare two datasets using two different programs: proftpd and vsftpd,

following the methods described in Section 4 and 5. Then a model is trained using the proftpd dataset from scratch, which is called baseline model. After the baseline model is trained, another model is trained using the vsftpd dataset, but the weights are initialized using the weights in the already-trained baseline model. We keep the hyper-parameters of the baseline model to be consistent with the ones in other experiments.

We first investigate whether a baseline model trained by the proftpd dataset could be used to detect ROP payloads against vsftpd. The accuracy of such investigation is 89.63% with a false positive rate of 2.94%. On one hand, the outcome implies that directly using a model trained for program A to detect ROP payloads against program B would result in unacceptable accuracy. On the other hand, since 89.63% is substantially higher than what is expected by us, the outcome implies that the "knowledge" learned to detect ROP payloads against proftpd might be transferable when a model is learned to detect ROP payloads against vsftpd.

Then we initialize a second model borrowing the weights from the baseline model, and use the vsftpd dataset to train the second model. The performance stop improving after 4.2 minutes training (5 epochs). Note that the baseline model here can be trained until it converges in 13 minutes. Therefore, the time for transfer learning instead of training from the scratch is 67.69% less. The accuracy stops improving at 97.77%, and the false positive rate drops to 0.04%. When compared to the vsftpd results shown in Table 4, we found that 97.77% is slightly lower than 99.5%, and 0.04% is slightly lower than 0.02%.

In summary, it appears that transfer learning could provide a more desired trade-off between the amount of training time and accuracy, especially when there is a need to train deep-learning models for a large number of programs in a scalable way.

7.6. Could Incremental Learning Be Leveraged to Reduce the Cost of Retraining When a Program Is Updated?

If a protected program is updated (e.g. patched), then there could be new instruction sequences available for attackers to utilize. Thus, we design an experiment to see whether DEEPRETURN could be trained using incremental training. In particular, we want to see if incremental training could significantly reduce the cost of model retraining. It should be noticed that this evaluation question is clearly related to the practicality of the DEEPRETURN approach.

In our experiment, the data of proftpd are divided into two parts. To simulate the software update scenario, one major part of the data will hold over 90% of all data. The other minor part is then considered as the new instruction sequences from patching, which is significantly smaller. We conducted the experiment with two different splitting ratios: 90%-10% and 95%-5%, respectively.

After we train the model from the scratch using majority of the data, the weights in the convolutional layers as well as the parameters in the batch normalization layers will be fixed. Therefore, when we do incremental training later, only weights in the dense layer of the neural network are updated to speed up the training process. This is a very common technique in incremental training.

As shown in Table 6, the result shows that the performance is neither downgraded nor improved much, which implies that incremental training for software updates is feasible. In our experiment, since only dense layer(s) need to be incrementally trained, the training time for newly added data is about 88% less than the time needed for training first time from scratch. Therefore, we conclude that for minor software updates, incremental training could be an ideal and cost-effective solution.

	Accuracy	False Positive Rate	CPU Time for Training
Train from scratch before updating	99.366%	0.017%	13 minutes
Incrementally train after updating	99.396%	0.017%	1.5 minutes

Table 5
Results of incremental training on proftpd with 90%-10% split

	Accuracy	False Positive Rate	CPU Time for Training
Train from scratch before updating	99.620%	0.017%	13 minutes
Incrementally train after updating	99.727%	0.017%	1.5 minutes

Table 6
Results of incremental training on proftpd with 95%-5% split

Ratio	# of gadget-like	# of real	Accuracy	FPR	FNR	F1	ROC-AUC
0.4	19500	48750	99.75%	0.0010	0.0039	0.9975	1.000
0.6	19500	32500	99.75%	0.0008	0.0042	0.9975	1.000
0.8	19500	24375	99.67%	0.0008	0.0056	0.9968	0.999
1	19500	19500	99.52%	0.0004	0.0092	0.9951	1.000

Table 7
Results of training on unbalanced proftpd data

7.7. Can DEEPRETURN *Handle Unbalanced Data?*

Using unbalanced data to train a neural network could cause the neural network to learn features that are not expected to be learned. For example, if a binary classification dataset contains 10% label 0 and 90% label 1, then the model can just learn to predict 1 and still achieve 90% accuracy. As a result, the rule of the thumb for training a neural network classifier is to ensure the dataset is balanced. However, a unique characteristic of the problem of detecting ROP payloads is that generating benign data samples is magnitudes harder and costlier than generating malicious data samples. And this unique characteristic indicates that there is actually a trade-off between how balanced the datasets are and how practical (e.g., how costly) to apply deep learning to detect ROP payloads. To gain a deeper understanding of the trade-off, we conduct an experiment to compare the performance of a model A trained with a balanced dataset and a model B trained with an unbalanced dataset (i.e., using the same amount of benign data samples used to train model A but more malicious data samples than those used to train model A).

We use proftpd data for this experiment. Suppose we only have 19,500 (benign) gadget-like instruction sequences available and it is too costly to generate more, then we use more than 19,500 (malicious) real gadget chains when training a model. We conduct the experiment 4 times with different levels of unbalance to investigate how the model's performance is affected by levels of unbalance. The ratios of the number of gadget-like-instruction sequences, which is always 19,500, to the number of real gadget chains are set to be 0.4, 0.6, 0.8, and 1.0, respectively.

We use five essential metrics when training on unbalanced data: accuracy, false positive rate (FPR), false negative rate (FNR), F1-score, and area under receiver operating characteristic curve (ROC-AUC). A good classification model should do well in all these metrics. Note that we do not include figures of ROC curve here because the ROC-AUC is always over 0.9 for most of the cases.

The results are shown in the Table 7. The accuracy is increasing as more malicious data samples are used in training, but the FPR is in general getting worse. F1 score and ROC-AUC have the same trend as accuracy, which has increased performance when more malicious data samples are used.

The results have several implications. First, when the number of benign data samples cannot be further increased, neither using balanced data nor using unbalanced data is a clear "winner". Each strategy has pros and cons. Second, if the security officer is most concerned with the FPR, then he or she will need to be careful in using unbalanced data. In particular, the security officer should determine whether the trade-off between the FPR and the other 4 metrics is worth it. Third, if the security officer is most concerned with any of the other 4 metrics, then using unbalanced data can be a good strategy.

7.8. Throughput of DEEPRETURN

We now consider the throughput of DEEPRETURN on nginx. We use throughput, rather than latency, as the performance criteria because DEEPRETURN is an IDS and it does not block the traffic. In other words, DEEPRETURN exerts zero runtime overhead for the protected program.

We can calculate from Table 2 that the disassembler works at a speed of 665 Mb/s, which can match the traffic on a server that is not too busy. Moreover, since the ASL-guided disassembly can be parallelized very well, we can split the workload across multiple servers running DEEPRETURN, further increasing the throughput.

The CNN in DEEPRETURN can classify 27k potential gadget chains in one second. We observe the number of potential gadget chains is rather small. On average, in the 665 Mb data processed within one second, only 13 potential gadget chains are fed into the neural network for classification. Thus, the performance of the entire IDS is bounded by the speed of disassembly and the overall throughput is 665 Mb/s.

8. Discussion and Limitations

CFI mechanisms could be used to detect ROP attacks. One advantage of DEEPRETURN over CFI is on the runtime overhead. CFI typically has a considerable runtime overhead, especially fine-grained CFI, since they have a stricter policy and require more checks. Conversely, DEEPRETURN is transparent to the protected program and does not incur any runtime overhead (please refer to the next subsection). Additionally, DEEPRETURN also does not need to instrument the program or modify the compiler. One advantage of CFI over DEEPRETURN is that CFI may stop the ROP exploit in action but DEEPRETURN is not a blocker. Nevertheless, IDS is designed to detect the attacks and enable other defense reactions.

Readers may wonder how can DEEPRETURN work if the traffic is encrypted (e.g., HTTPS). Encryption can hide the addresses of ROP gadgets and hinder the ASL-guided disassembly. The solution is to deploy DEEPRETURN between the point of encryption and the protected program. For example, when a reverse proxy is deployed in front of a web-server to provide encryption, then DEEPRETURN should be deployed between the reverse proxy and the web-server. Now DEEPRETURN sees the unencrypted HTTP traffic and works in its normal way.

Our IDS can cooperate well with Address Space Layout Randomization (ASLR). The two gadget chain datasets only contain instructions, not addresses. So the CNN is not affected by different memory layout. Meanwhile, when a server program is newly launched (thus a new memory space layout is created), we update the memory snapshot used by DEEPRETURN. In this way, the disassembler always

works on the current image of the running program. So the result from the disassembler is also accurate. Given these two observations combined, we find that our IDS can work in conjunction with ASLR, further raising the bar for attackers. Note the memory space layout remains stable after the initial randomization (even in fine-grained ASLR). So the update does not happen very often and does not incur high runtime overhead (if any). Take Nginx for an example, although several work processes are created, they are `forked` from the master process and their memory layouts are the same.

Regarding whether a deep learning model should be trained for every program, we observe that in real world one program could be substantially more likely to be targeted by ROP attacks than another program. For example, on a web server, Nginx is usually much more likely to be targeted by ROP attacks than Libc. Based on this observation, we assume that the security admin will allocate the available resources to only run the deep learning models which help protect the programs that are most likely to be targeted by ROP attacks. Since the resources are limited, we do not assume that a model will be trained for every program in the real world.

Readers may notice our criteria for a potential gadget in Section 4.1 is relatively broad and it may allow non-gadgets. The reason is that we do not want to let any real ROP gadgets evade the disassembly engine and further bypass the detection. It makes our approach more robust and harder to bypass. But we showed this does not lead to too many false positives.

In case of a patch or an update, whether the neural network needs to be re-trained depends on the detailed change of the program. If a part (e.g., a function) of the program is removed and this section happens to be present in some gadget chains, then these gadget chains should be removed from the dataset. If some instructions or functions are inserted into the program, it is very likely that the original gadget chains are still valid and there is no need to regenerate the entire dataset or re-train the entire neural network. We have conducted experiment to see whether increment learning could help to avoid retraining the model for an software update. The experiment details and results are described in Section 7.6.

ROP attack is getting more and more complex and has several variants. DEEPRETURN is able to detect polymorphic ROP attacks [66] because there always have to be some un-masqueraded ROP gadgets to unpack, decrypt, or arrange the real attacking gadgets. DEEPRETURN can detect these un-masqueraded ROP gadgets, but unfortunately DEEPRETURN is not able to recognize masqueraded ROP gadgets. On the other hand, some recent variations of ROP, e.g, JIT-ROP, that leverages a JavaScript environment to launch the attack, can potentially bypass our detection. However, unless the ROP attack is planned and launched ENTIRELY from JavaScript (which is considerably complex [8]), we can still detect it. For example, if the attacker uses JavaScript to leak certain memory information or arrange the heap layout, and then launch ROP attack through the network, DEEPRETURN can still detect the ROP exploit. Meanwhile, high profile targets for ROP attacks, e.g., server programs, seldom provide a JavaScript environment.

9. Conclusion

DEEPRETURN is a novel intrusion detection system that leverages the power of deep neural networks to classify potential gadget chains produced by an ASL-guided disassembler. We show that DEEPRETURN has a high detection rate and a very low false positive rate. It also successfully detects all of the real-world ROP exploits collected or created by us. Meanwhile, it is non-intrusive and does not incur runtime overhead. We argue that DEEPRETURN is a practical widely-deployable detection method against ROP attacks.

Acknowledgments

This work was supported by the following grants: NSF CNS-1814679, ARO W911NF-15-1-0576, ARO MURI W911NF-13-1-0421, and NSF CAREER ECCS 1846706.

Appendix A DEEPRETURN V.S. LSTM, MLP or SVM

Method	Detection Rate	False Positive Rate	Training Time	Classification Speed
DEEPRETURN	99.3%	0.01%	45min	27,116 chains/s
ASL-guided & LSTM	97.8%	0.2%	143min	7,150 chains/s
ASL-guided & MLP	96.9%	22.2%	26min	85,629 chains/s
ASL-guided & SVM	77.6%	42.4%	12min	310 chains/s

Table 8
Comparison of DEEPRETURN, LSTM, MLP and SVM on nginx

In this appendix, we compare the performance of DEEPRETURN to that of combining ASL-guided disassembly with other well-known classifiers, i.e., LSTM (one type of RNN), Multiple-Layer Perceptron (MLP) and Support Vector Machine (SVM). These three competitors are also widely used and produce many good results [67–69]. Specifically, we use an LSTM with 96 hidden units, an MLP with three fully-connected layers with 32 units in each layer, and an SVM with the radial basis function (RBF) kernels, and And we use 80% of Nginx dataset for training. The results are shown in Table 8.

The combination of ASL-guided disassembly and MLP leads to 96.9% detection rate and 22.2% false positive rate. The combination of ASL-guided disassembly and SVM leads to 77.6% detection rate and 42.4% false positive rate. These two false positive rates are too high for practical application because they can easily cause denial of service. LSTM reaches a 97.8% detection rate and 0.2% false positive rate, which is still far from CNN's result. Also, LSTM takes 3 times longer to train.

The result is not surprising. In fact, RNN (or LSTM) is more suitable for temporal data [53] whereas our instruction sequence data is more spatial in nature. MLP uses fully connected layer that is known to be hard to train. Using the same amount of data, it does not achieve comparable accuracy to CNN. The SVM is not good at directly dealing with minimally preprocessed data and it requires certain feature engineering to first extract features from the input data. In summary, CNN is superior to its three competitors and more suitable to the task.

References

[1] Microsoft, A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2., 2008, http://support.microsoft.com/kb/875352.
[2] I. Molnar, Exec Shield, 2003, http://people.redhat.com/mingo/exec-shield/.
[3] T. Bletsch, X. Jiang, V.W. Freeh and Z. Liang, Jump-oriented programming: a new class of code-reuse attack, in: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM, 2011, pp. 30–40.
[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham and M. Winandy, Return-oriented programming without returns, in: *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 559–572.
[5] E.J. Schwartz, T. Avgerinos and D. Brumley, Q: Exploit Hardening Made Easy., in: *USENIX Security Symposium*, 2011, pp. 25–41.

[6] H. Shacham, The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), in: *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[7] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund and T. Walter, Breaking the memory secrecy assumption, in: *Proceedings of the Second European Workshop on System Security*, 2009, pp. 1–8.

[8] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen and A.-R. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013, pp. 574–588.

[9] J. Seibert, H. Okhravi and E. Söderström, Information leaks without memory disclosures: Remote side channel attacks on diversified code, in: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 54–65.

[10] K.Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose and M. Polychronakis, Return to the zombie gadgets: Undermining destructive code reads via code inference attacks, in: *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 954–968.

[11] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz et al., Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity., in: *NDSS*, 2017.

[12] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao and L. Xie, DROP: Detecting return-oriented programming malicious code, in: *International Conference on Information Systems Security*, Springer, 2009, pp. 163–177.

[13] Y. CHENG, Z. ZHOU, Y. MIAO, X. DING and R.H. DENG, ROPecker: A Generic and Practical Approach For Defending Against ROP Attack.(2014), in: *NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium, San Diego, February 23*, Vol. 26, pp. 1–14.

[14] V. Pappas, M. Polychronakis and A.D. Keromytis, Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, in: *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 447–462.

[15] P. Akritidis, C. Cadar, C. Raiciu, M. Costa and M. Castro, Preventing memory error exploits with WIT, in: *2008 IEEE Symposium on Security and Privacy (sp 2008)*, IEEE, 2008, pp. 263–277.

[16] M. Payer, A. Barresi and T.R. Gross, Fine-grained control-flow integrity through binary hardening, in: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2015, pp. 144–164.

[17] A.J. Mashtizadeh, A. Bittau, D. Boneh and D. Mazières, CCFI: Cryptographically Enforced Control Flow Integrity, in: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[18] Y. Tanaka and A. Goto, n-ROPdetector: Proposal of a Method to Detect the ROP Attack Code on the Network, in: *Proceedings of the 2014 Workshop on Cyber Security Analytics, Intelligence and Automation*, ACM, 2014, pp. 33–36.

[19] M. Polychronakis and A.D. Keromytis, ROP payload detection using speculative code execution, in: *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, IEEE, 2011, pp. 58–65.

[20] B. Stancill, K.Z. Snow, N. Otterness, F. Monrose, L. Davi and A.-R. Sadeghi, Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets, in: *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2013, pp. 62–81.

[21] D. Pfaff, S. Hack and C. Hammer, Learning how to prevent return-oriented programming efficiently, in: *International Symposium on Engineering Secure Software and Systems*, Springer, 2015, pp. 68–85.

[22] M. Elsabagh, D. Barbará, D. Fleck and A. Stavrou, Detecting ROP with Statistical Learning of Program Characteristics, in: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 219–226.

[23] K. Böttinger, P. Godefroid and R. Singh, Deep Reinforcement Fuzzing, *arXiv preprint arXiv:1801.04589* (2018).

[24] M. Du, F. Li, G. Zheng and V. Srikumar, Deeplog: Anomaly detection and diagnosis from system logs through deep learning, in: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1285–1298.

[25] W. Song, H. Yin, C. Liu and D. Song, DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis, in: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 606–618.

[26] J. Sacco, Crashmail 1.6 - Stack-Based Buffer Overflow (ROP), 2018.

[27] J. Sacco, PMS 0.42 - Local Stack-Based Overflow (ROP), 2018.

[28] negux, Freefloat FTP Server 1.0 - DEP Bypass with ROP, 2013.

[29] ZadYree, HT Editor 2.0.20 - Local Buffer Overflow (ROP), 2012.

[30] L.D. Long, Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028), VNSECURITY, 2013.

[31] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos and G. Portokalidis, Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard, in: *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[32] M. Abadi and M. Budiu, lfar Erlingsson, and J. Ligatti. Control-flow integrity, in: *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2005.

[33] T. Bletsch, X. Jiang and V. Freeh, Mitigating code-reuse attacks with control-flow locking, in: *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 353–362.
[34] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger and A.-R. Sadeghi, MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones., in: *NDSS*, Vol. 26, 2012, pp. 27–40.
[35] Z. Wang and X. Jiang, Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity, in: *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 380–395.
[36] M. Zhang and R. Sekar, Control Flow Integrity for COTS Binaries, in: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.
[37] N. Burow, S.A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler and M. Payer, Control-flow integrity: Precision, security, and performance, *ACM Computing Surveys (CSUR)* (2017), 1–33.
[38] N. Carlini and D. Wagner, ROP is Still Dangerous: Breaking Modern Defenses, in: *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.
[39] L. Davi, A.-R. Sadeghi, D. Lehmann and F. Monrose, Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection, in: *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.
[40] N. Carlini, A. Barresi, M. Payer, D. Wagner and T.R. Gross, Control-flow bending: On the effectiveness of control-flow integrity, in: *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.
[41] H. Shacham, The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), in: *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.
[42] N.A. Quynh, Capstone - The Ultimate Disassembler, GitHub, 2013.
[43] A. Souchet, rp++, GitHub, 2012.
[44] J. Salwan, ROPgadget, GitHub, 2015.
[45] S. Schirra, Ropper - rop gadget finder and binary information tool, GitHub, 2016.
[46] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer and E. Bodden, PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution, in: *Security and Trust Management: 12th International Workshop, STM*, Springer International Publishing, 2016, pp. 212–228.
[47] P. Sue, Ropper - rop gadget finder and binary information tool, GitHub, 2017.
[48] N.A. Quynh, Unicorn – The ultimate CPU emulator, Unicorn, 2015.
[49] J. Leek, Don't use deep learning your data isn't that big, 2017.
[50] D.C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella and J. Schmidhuber, Flexible, high performance convolutional neural networks for image classification, in: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22, 2011, p. 1237.
[51] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural computation* **9**(8) (1997), 1735–1780.
[52] S. Lai, K. Liu, S. He and J. Zhao, How to generate a good word embedding, *IEEE Intelligent Systems* **31**(6) (2016), 5–14.
[53] I. Goodfellow, Y. Bengio, A. Courville and Y. Bengio, *Deep learning*, Vol. 1, MIT press Cambridge, 2016.
[54] Y. Kim, Convolutional neural networks for sentence classification, *arXiv preprint arXiv:1408.5882* (2014).
[55] C.-W. Hsu, C.-C. Chang, C.-J. Lin et al., A practical guide to support vector classification (2003), 1396–1400.
[56] S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, *arXiv preprint arXiv:1502.03167* (2015).
[57] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting., *Journal of Machine Learning Research* **15**(1) (2014), 1929–1958.
[58] N. Qian, On the momentum term in gradient descent learning algorithms, *Neural networks* **12**(1) (1999), 145–151.
[59] X. Wang, C.-C. Pan, P. Liu and S. Zhu, Sigfree: A signature-free buffer overflow attack blocker, *IEEE transactions on dependable and secure computing* **7**(1) (2010), 65–79.
[60] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.
[61] P. Ginsparg, arXiv.org e-Print archive, Cornell University, 1991.
[62] F. Chollet et al., Keras: Deep learning library for theano and tensorflow, *URL: https://keras. io/k* **7**(8) (2015).
[63] pakt, pakt/ropc: A Turing complete ROP compiler, GitHub, 2012.
[64] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières and D. Boneh, Hacking blind, in: *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 227–242.
[65] E. Database, Exploits Database by Offensive Security, Exploit Database, 1998.
[66] K. Lu, D. Zou, W. Wen and D. Gao, Packed, printable, and polymorphic return-oriented programming, in: *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2011, pp. 101–120.
[67] G. Mountrakis, J. Im and C. Ogole, Support vector machines in remote sensing: A review, *ISPRS Journal of Photogrammetry and Remote Sensing* **66**(3) (2011), 247–259.
[68] R.M. Summers, Progress in fully automated abdominal CT interpretation, *American Journal of Roentgenology* **207**(1) (2016), 67–79.

[69] L. Yao, A. Torabi, K. Cho, N. Ballas, C. Pal, H. Larochelle and A. Courville, Describing videos by exploiting temporal structure, in: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4507–4515.