Graph Neural Nertwork based Cache Monitor for ROP Attack Detection

Asmit De
School of EECS
The Pennsylvania State University
University Park, USA
asmit@psu.edu

Saptarashmi Bandyopadhyay
School of EECS
The Pennsylvania State University
University Park, USA
sbandyo20@gmail.com

Swaroop Ghosh
School of EECS
The Pennsylvania State University
University Park, USA
szg212@psu.edu

***Abstract*— Return Oriented Programming (ROP) is a popular exploit that takes advantage of buffer overflow vulnerabilities to gain access to the system. Existing techniques to prevent such exploits, such as, shadow stack, CFI, etc. are performance hindering, or require explicit support from software or system. In this work we propose a Graph Neural Network (GNN) based cache monitor, which can detect ROP attacks based on distinct cache access features during an ongoing attack. A graph dataset is created where the vulnerable and non-vulnerable executions are the nodes and the cache features obtained for every execution are the node features. Node embeddings are generated by learning the structural properties of the graph followed by classification with GNN to predict the likelihood of attack based on L1I cache features.**

Keywords—Buffer overflow, ROP, Cache Monitor, GNN

I. Introduction

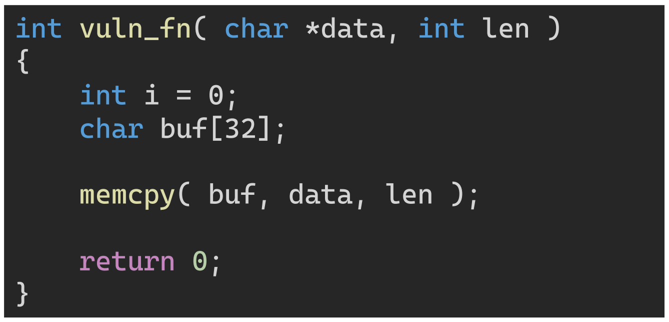
Traditional computing systems are inherently vulnerable to a wide attack surface from the topmost application level to the systems architecture level, leading to serious security and integrity concerns such as leaking private SSH keys or launching Denial-of-Service (DOS) attacks. Programming languages like C which are closer to the hardware, provide a lot of flexibility in terms of memory and IO access to facilitate system and device level programming. However, this also means that such languages often tend to have inherent security deficiencies and can lead to vulnerabilities if not used with proper and secure practices. Buffer overflow is the most commonly exploited vulnerability that can cater to a wide attack surface. In a program without bounds checking, an adversary can overload a user input with excess data that can overrun the buffer capacity and overwrite nearby memory locations with potentially malicious data, leading to several attack scenarios, such as return-oriented programming (ROP), VTable hijacking, function pointer manipulation and even violation of data flow in program.

A. *Buffer Overflow Vulnerability*

Fig 1 shows a program function vulnerable to buffer overflow. The function accepts a data stream and the length of the data as arguments to the function. The function copies the provided data into a buffer of length 32 bytes. During execution of the program, a stack frame gets created for the function. First, the return address is saved on the stack, which is the address to which the program is supposed to return to after the function has finished execution. Inside the function's stack frame, space is allocated for the variable i, and the buffer buf. Let us assume that the inputs to the function are adversary controlled. Thus, if an adversary can provide data more than 32 bytes, the buffer is overflowed, since there are no bounds checking for the buffer. The overflow causes the data beyond the buffer to get overwritten, such as the variable i, and also the saved return address beyond that. Carefully crafting the data payload can allow an adversary to divert the control flow or data flow of the program by jumping to arbitrary locations in the code.

B. *Return Oriented Programming*

Return oriented programming (ROP) is one of the most common attacks that exploit the buffer overflow vulnerability. In a return oriented programming attack, the adversary overflows a buffer with the objective of overwriting the return address in the stack to return to existing pieces of code in the program or the libc library. Fig 2 shows the steps involved in a ROP exploit execution. ROP is a type of code reuse attack where the adversary reuses existing code pieces (gadgets) from the program to execute his payload, e.g., launching a system shell with superuser privileges. Hence the adversary may often need to chain together multiple such code pieces ending in return statements and form a return chain that allows him to eventually reach the desired payload code.



```c
int vuln_fn( char *data, int len )
{
    int i = 0;
    char buf[32];

    memcpy( buf, data, len );

    return 0;
}
```

Fig. 1. Vulnerable C function.

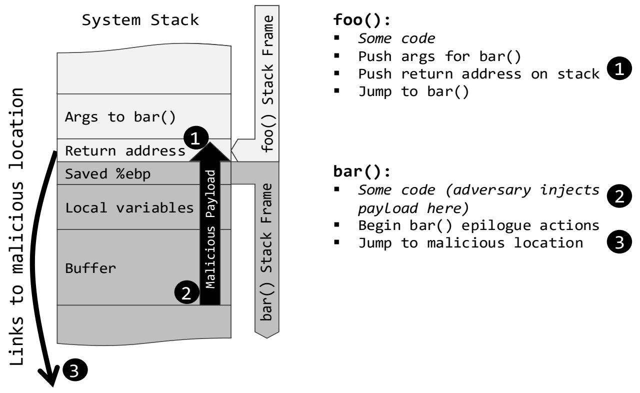


Fig. 2. Return Oriented Programming Exploit.

C. Defense Mechanisms

Stack canaries [1] are *sacrificial* words placed on the stack at stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer in order to overwrite the return address, the canary word will also be overwritten. Before returning in call stack, canary word is checked, and if modified, the return address is assumed to be compromised, and the program is halted.

Data Execution Prevention (DEP) [2] is employed to prevent an adversary from injecting malicious code onto the stack. Memory pages are marked W⊕X, meaning, a page can either be executable (code) or be writable (stack, heap), but not both. This prevents an adversary from executing malicious code from the stack. However, an adversary can return to existing code in the program or functions in the linked library using gadget chains (return-to-libc attack).

Address Space Layout Randomization (ASLR) [3] randomizes the code, stack, heap, and shared library locations on the address space, to make it difficult for the adversary to determine the specific addresses and launch attacks. However, buffer over-read and side-channel vulnerabilities can be used by an adversary to reverse engineer the randomized address.

Control Flow Integrity (CFI) [4] involves statically computing a valid control flow graph (CFG) of the program and ensuring that during runtime, the program abides by that CFG. A coarse-grained approach to ensuring control flow integrity while returning from functions is the use of a shadow stack (a separate stack residing in a secure memory location) [5]. On each function call, the return address is saved on the shadow stack alongside being put on the stack normally. While returning from a function, the return address on the stack is validated against the one on the shadow stack. On mismatch, it is assumed that the return address has been compromised and the program execution is halted. However, a shadow stack can be expensive and can hurt performance since the pages housing the shadow stack may not be present in cache and will require hundreds to thousands of cycles to bring the page onto the cache and perform the validation. Several software and compiler level systems have been proposed in literature for supporting shadow stack [6-7].

Secure hardware platforms e.g., ARM TrustZone [8] and Intel Software Guard Extensions (SGX) [9] isolate the hardware so that the access to systems assets are restricted. Hardware acceleration of security validation has been proposed to address the performance impact partially while covering a subset of security threats e.g., Intel CFI Enforcement Technology (CET) [10] to protect against control-flow hijacking. Intel Memory Protection Extensions (MPX) [10] with extended instruction set architecture is developed to prevent memory safety violations such as buffer overflow, heap overflow and pointer corruption. Intel Transactional Synchronization Extensions (TSX) [11] exposes and exploits hidden concurrency in multi-threaded applications. Intel PT [12] logs TSX events when a transaction begins, commits or aborts. It has been shown in [13] that tagging of code and data using software-defined metadata and processing the tag using custom designed processor can detect ROP, code reuse, buffer overflow, code injection, memory safety violation and pointer corruption. Although effective, this new architecture cannot be readily deployed due to lack of re-configurability, and, area, energy and performance overhead. Other hardware-assisted techniques to protect forward and backward edges in control flow are proposed in [14].

In this work, we propose a lightweight ROP attack detection methodology using a cache monitor. We analyze the cache access patterns to identify programs under normal execution and under ROP attack. We then use the cache statistics to train a machine learning classifier to detect ROP attacks. We use support vector machine, random forest, and graph neural networks for training and found graph neural networks to provide the best detection accuracy. *To the best of our knowledge, this is the first attempt at using graph neural networks in hardware security applications, specifically detecting control flow integrity violations and ROP attacks.*

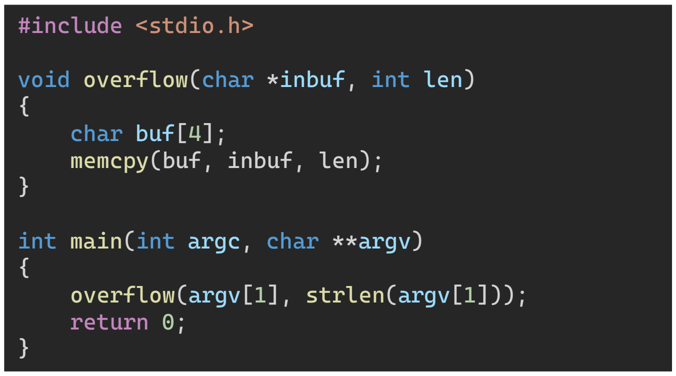
II. Cache Monitoring For ROP Attacks

A. ROP Chain Creation

In order to build a suitable detection mechanism for ROP attacks, we first look at how a ROP gadget chain is created. We use the ROPgadget [15] tool to search for gadgets in a program's binary. The tool analyzes the binary to find gadgets that can be chained together to create payload for launching a system shell. Fig. 3a shows our vulnerable program and Fig. 3b shows the result of the tool on our vulnerable program. There are three types of gadgets used for crafting the payload. In the first step, we find the write-what-where gadgets, which are used to copy data from one location to another. In the second step we find gadgets to create the system call identifier number. Next, we find gadgets to prepare the arguments for the system call. And in the last step we find a gadget for invoking the system call.

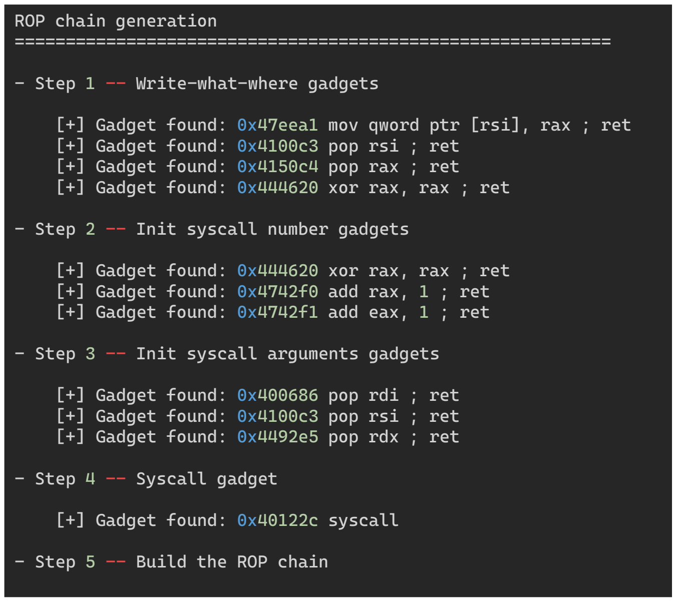
B. ROP Analysis

Fig. 3c shows the code for creating the final gadget chain. We first use the `pop rsi; ret` gadget to load the address in the data segment where we want to store the shell path. We then use



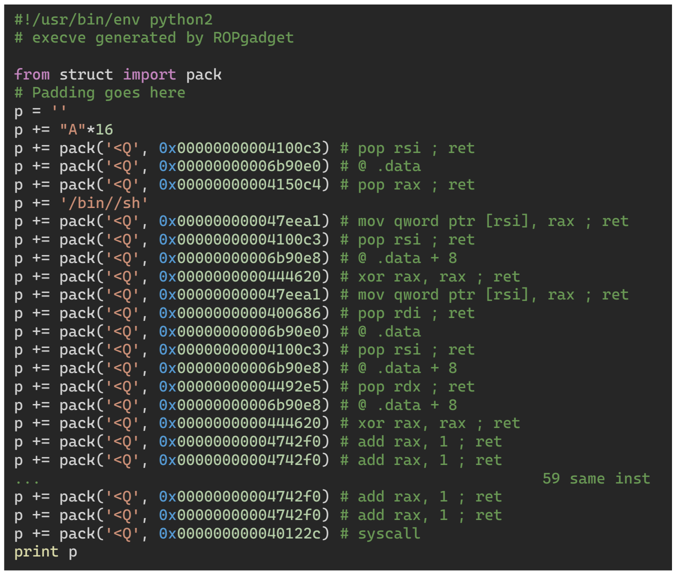
```c
#include <stdio.h>

void overflow(char *inbuf, int len)
{
    char buf[4];
    memcpy(buf, inbuf, len);
}

int main(int argc, char **argv)
{
    overflow(argv[1], strlen(argv[1]));
    return 0;
}
```

(a)



```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

    [+] Gadget found: 0x47eea1 mov qword ptr [rsi], rax ; ret
    [+] Gadget found: 0x4100c3 pop rsi ; ret
    [+] Gadget found: 0x4150c4 pop rax ; ret
    [+] Gadget found: 0x444620 xor rax, rax ; ret

- Step 2 -- Init syscall number gadgets

    [+] Gadget found: 0x444620 xor rax, rax ; ret
    [+] Gadget found: 0x4742f0 add rax, 1 ; ret
    [+] Gadget found: 0x4742f1 add eax, 1 ; ret

- Step 3 -- Init syscall arguments gadgets

    [+] Gadget found: 0x400686 pop rdi ; ret
    [+] Gadget found: 0x4100c3 pop rsi ; ret
    [+] Gadget found: 0x4492e5 pop rdx ; ret

- Step 4 -- Syscall gadget

    [+] Gadget found: 0x40122c syscall

- Step 5 -- Build the ROP chain
```

(b)



```python
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack
# Padding goes here
p = ''
p += "A"*16
p += pack('<Q', 0x00000000004100c3) # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e0) # @ .data
p += pack('<Q', 0x00000000004150c4) # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x000000000047eea1) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x00000000004100c3) # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e8) # @ .data + 8
p += pack('<Q', 0x0000000000444620) # xor rax, rax ; ret
p += pack('<Q', 0x000000000047eea1) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x0000000000400686) # pop rdi ; ret
p += pack('<Q', 0x00000000006b90e0) # @ .data
p += pack('<Q', 0x00000000004100c3) # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e8) # @ .data + 8
p += pack('<Q', 0x00000000004492e5) # pop rdx ; ret
p += pack('<Q', 0x00000000006b90e8) # @ .data + 8
p += pack('<Q', 0x0000000000444620) # xor rax, rax ; ret
p += pack('<Q', 0x00000000004742f0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000004742f0) # add rax, 1 ; ret
...                                                         59 same inst
p += pack('<Q', 0x00000000004742f0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000004742f0) # add rax, 1 ; ret
p += pack('<Q', 0x000000000040122c) # syscall
print p
```

(c)

Fig. 3. (a) Vulnerable program, (b) Gadget search using ROPgadget, (c) Gadget chain payload for ROP attack.

the `pop rax; ret` gadget to load the shell path in `rax`. We use a write-what-where gadget to copy the data from `rax` to the address pointed to by `rsi`. Similarly, we prepare the other arguments for the system call in the memory. To execute the system call `sys_execve`, we need to provide the syscall identifier 59. This is created by using 59 `add rax, 1; ret` gadgets. Finally, we execute the `syscall` gadget.

Analyzing the locations of the gadgets used in the program, we find that the locations of the gadgets are random and far from each other. For example, the gadgets `pop rax; ret` and `mov qword ptr[rsi], rax; ret` are located ~423KB apart. This indicates that during a ROP gadget chain execution the instruction pointer moves randomly in the code. However, during normal program execution, a program typically is sequential. Thus, the cache access patterns for the two executions are expected to be different.

C. *Cache Statistics*

Based on the previous analysis, it is expected that during a ROP attack, the gadget code is not likely to be in the cache, and hence is likely to incur more L1 instruction cache misses. This is because, during normal program execution, the immediate next instructions will be brought into the instruction cache due to the principle of locality. However, when the ROP gadgets need to be executed, the gadget instructions may not be present in the instruction cache since they are not part of the normal program execution flow. Furthermore, the L1 instruction cache is typically very small, around 32KB. Thus, even if the gadget instruction has been previously executed as part of the normal program flow, it is likely to be kicked out to accommodate other instructions.

We use the Valgrind-Cachegrind [16] tool to capture cache statistics for normal program execution and program under ROP attack. We collected 13 features provided by cachegrind comprsing of cache references, cache misses and cache miss rate: (i) I references, (ii) D references, (iii) LL references, (iv) I1 misses, (v) LLi misses, (vi) D1 misses, (vii) LLd misses, (viii) LL misses, (ix) I1 miss rate, (x) LLi miss rate, (xi) D1 miss rate, (xii) LLd miss rate, and (xiii) LL miss rate.

To collect the different variations in statistics, we instrumented the Nginx [17] binary to introduce a vulnerable function at 28 different positions of the code. We ran each of the 28 versions of the code for 100 iterations under normal execution, and 100 iterations under a valid ROP attack. We used the 5600 collected traces to train our machine learning classifier.

D. *Case Study: Nginx*

As an example, let us observe how the cache access patterns vary for a single execution of the Nginx binary under normal execution and ROP attack. Fig 4a shows the statistics for a single execution under normal operation, and Fig 4b shows the statistics for a single execution under ROP attack. For the normal execution the number of I references is 1208494, whereas for ROP attack, the I references increase by 3073. For the I1 misses, we observe that I1 misses for ROP attack is 54 higher than I1 misses for normal operation. Similarly, for LL references and misses, we see that the ROP attack statistics show higher values than normal operation.

III. Machine Learning based Detection Technique

A. *Motivation*

The current research work is based on the hypothesis that return-oriented programming (ROP) generates a distinctive cache signature that is different from a normal execution of the same binary. In order to identify vulnerable and non-vulnerable executions of the binary, a dataset has been created with the observed cache statistics as features and each execution of the program has been either labeled or non-vulnerable. Traditional Machine Learning techniques like Support Vector Machine (SVM) and Ensemble based classification have been used to identify the vulnerabilities in unknown execution of binaries. It has been observed that further diversity of the dataset, primarily execution of more binaries is essential to ensure unbiased and generalized learning.

Our intuition to apply Graph Neural Networks (GNN) is to understand the dependence of cache signatures when ROP based payload is introduced in different blocks of the source code to improve the classification. GNNs have a wide range of applications ranging from area like fake news detection, social media data analytics to bio-pharmaceutical activity tests, scientific publication categorization and online product recommendation. The success of the GNN has been due to the ability to capture the interdependence using the edge connectivity in the graph structured data which is not observed in typical datasets used as input for Deep Learning algorithms. A graph dataset has been created for the purpose of being used as input in the training with GNNs for the current work.

B. *Dataset and Graph Creation*

The Nginx binary in the current case study has been executed 200 times to get 5600 data points. For each of these 5600 data points, ROP based payload has been introduced in 28 positions (pos<i>) of the source code. For each (pos<i>) there are 100 data points for normal execution and 100 for vulnerable execution, which are generated by repeated execution of the Cachegrind tool. An 80/20 split has been implemented on this dataset to create the training and testing dataset that has been used for SVM and Random Forest classifier. The training dataset has equal distribution of vulnerable and non-vulnerable executions for unbiased learning.

For GNN, a 70/10/20 split has been created to generate the training, validation and testing dataset. The training and validation datasets have equal distribution of vulnerable and non-vulnerable executions for unbiased learning. A graph dataset has been created for the project where each of the 5600 data points are nodes. Each of the 200 nodes for pos<i> are connected with the 5400 data points in the remaining 27 positions. This edge connectivity has been ensured to analyze the impact of dependence on the cache signatures on the position of the ROP payload.

C. *Support Vector Machine.*

In order to establish a baseline performance for the Graph Neural Network, we train an SVM classifier on the data. A support vector machine algorithm can find a hyperplane in an N-dimensional space (N- no: features) that distinctly classifies the data points. The dataset is shuffled for improved learning and 4-fold cross-validation is used. This means that the dataset is divided into 4 slices. Every time, one of these data slices have been used as testing dataset while the other 3 data slices have been used for training dataset. A linear SVM model from the Scikit-Learn package is used as the classifier.

D. *Random Forest*

To assess the performance of ensemble methods on the dataset, we use Scikit-Learn's Random Forest Classifier implementation. Similar to the SVM approach, we shuffle the data and use 4-fold cross-validation. As shown in paper [22], Random forest is an ensemble-based classifier which is like a meta-estimator to fit a certain number of decision tree classifiers on different sub samples of the dataset. This improves classification accuracy and addresses overfitting. 1000 trees, each having a depth of 4, have been used in the random forest approach. The subsamples have been selected from the original dataset with replacement.

E. *Graph Neural Networks*

A graph can be defined as $G(V, E)$ where V is the number of vertices and E is the number of edges on the graph G. A GNN uses G as input data. A graph neural net converges to a stable equilibrium by the process of diffusion. All nodes in the neural net exchange information and update themselves accordingly until there are no more updates as shown in Fig 5.

[20] proposes graph neural network (GNN) model in 2009, that extends existing neural network methods for processing the data represented in the graph domain. They furnish a learning algorithm to estimate the parameters of the neural networks based on the backpropagation method to ensure generalization. The authors of paper [21] have presented an approach to learn representations of graphs using recurrent neural network autoencoders. They trained using sequences generated by

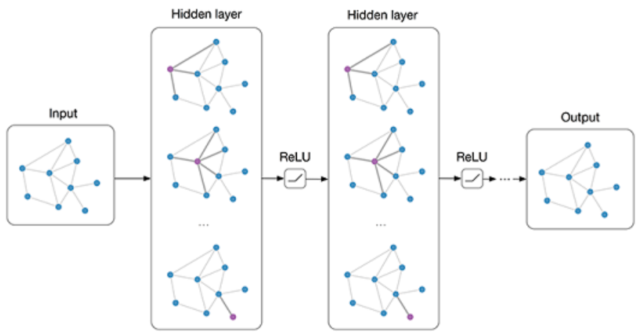


Fig. 5. A graph neural network [18]

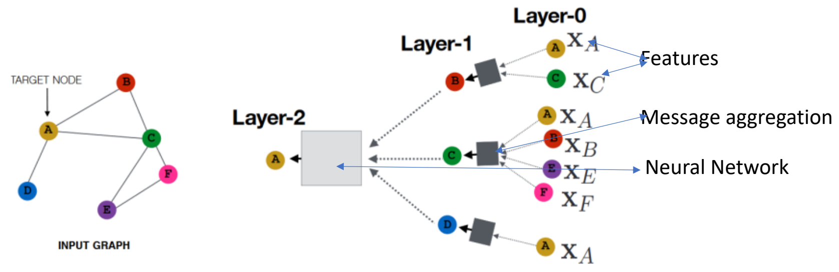


Fig. 6. Nearest neighbor approach for creating node embeddings [19]

random walks, shortest paths, and breadth-first search and demonstrate that their graph representations can increase the accuracy of graph classification tasks on both labeled and unlabeled graphs.

Learning with Graph Neural Networks can be categorized into 2 phases:

Phase 1 - Node embedding generation:

Learning is based on the structural properties of the graph. The neighborhood of nodes can be aggregated. Here we have done nearest neighbors aggregation by learning to reduce the inverse of shortest path between 2 nodes. An illustration of the nearest neighbor approach to create node embeddings is shown in in Fig. 6.

The graph dataset is provided as input with a $n \times n$ adjacency matrix where n is the number of nodes and $n \times m$ feature matrix where m is the number of features.

Embedding vectors are generated for every node in the dataset.

Phase 2: Node classification with LSTM:

The node embeddings and average binary features of the neighboring nodes are used for LSTM based classification by using the distance of learned embedding vectors.

IV. Experimental Results

A. Training

The 5600 nodes in the graph have been split in a 70/10/20 split for training, validation and testing dataset to be used for learning with GNNs. The design of the Long-Short Term Memory based Graph Recurrent Neural Network (RNN) requires setting a number of hyper-parameters as mentioned below.

The nearest neighbors approach has been used for node-embedding generation. Nearest neighbors algorithm is an unsupervised algorithm that identifies the top k neighbors of a given data point as defined by a distance metric (L1, L2, etc.) Our parameters to train the GNN for node embedding generation are:

- Nearest Neighbor based aggregation is used.
- Number of neighbors is 5.
- Dimension of dense embeddings is 100.

The Long-Short Term Memory based Recurrent Neural Network (RNN) is implemented by stacking the node embeddings, generated from graph learning to a Sequential model. Our parameters of the RNN implementation in Tensorflow 2.0 are:

- 2 layer unidirectional RNN
- 512 hidden units
- Initial Learning rate of 0.001
- Learning rate decay of 0.0001
- L1 regularization
- 0.2 drop-out rate
- Adam Optimizer
- 30 epochs

The Scikit-learn packages of SVM, Random Forest and Nearest Neighbors algorithm have been used in Python 3. Early stopping has been implemented in both node embedding generation and LSTM based classification. This means that if the validation loss is the same after 10 epochs, the training is terminated. This has been implemented to address over-fitting. The Adam optimizer has been used as it is capable to support variable learning rate decay.

B. *Results*

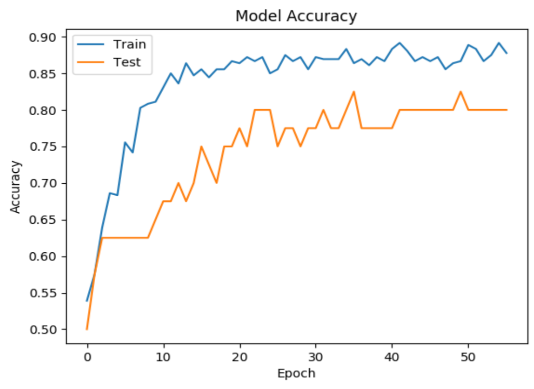
The training loss function is categorical loss entropy. It can be observed that the loss functions decrease without any evidence of overfitting in Fig. 7. The training and test accuracies are also significantly high.

The training dataset has 3900 nodes, testing dataset has 1200 nodes and the validation dataset has 500 nodes. The edge connectivity of the graph is referred to map the nodes in the graph.

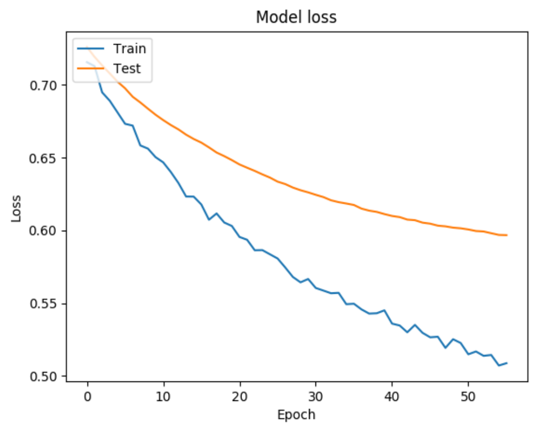
Very high accuracy implies that not only vulnerable but also non-vulnerable execution can be classified very well with these classification strategies which leads to higher precision and F measures. However, it has to be noted that the high accuracies

Table I: Accuracy of the classifier models

Classifier	Test Accuracy %
Support Vector Machine	91.25
Random Forest Classification	92.15
GNN based classification	94



(a)



(b)

Fig. 7. (a) Training accuracy, and (b) loss curve for implemented GNN

in Table 1 is due to the fact, that often features are the same across many data points when ROP payload is inserted in the early sections of the source code. Only one binary (Nginx) has been used for collecting all the data points. The execution for more binaries other than Nginx is essential for generalized learning on a diverse dataset.

V. Work Divison

Asmit De has executed the insertion of payloads to the binaries in 28 positions in the source code. Then he has statically compiled the vulnerable and original source code which he has executed to get log files of execution with cache statistics.

Saptarashmi Bandyopadhyay has implemented the SVM classifier, Random Forest classifier and Graph Neural Network based classifier. He has created the initial dataset with the data points extracted from the log files. He has also created the graph dataset with log files.

VI. Conclusions

In this work, we showed the feasibility of detecting ROP attacks at the architecture. We showed that by monitoring cache statistics during program execution we can detect malicious ROP attacks. We demonstrated a high-accuracy graph-neural network classifier that can identify vulnerable and non-vulnerable program executions. Binaries of other large applications like Wireshark can be used for generalized learning and ensure a diverse distribution of data points.

References

[1] Cowan et al. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks." In SSYM, 1998.

[2] Data Execution Prevention, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx

[3] Team, PaX. "PaX address space layout randomization (ASLR), 2003." URL: https://pax. grsecurity. net/docs/aslr.txt

[4] Abadi et al. "Control-flow integrity." In Proc. ACM CCS, 2005.

[5] Park et al. "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks." *IEEE Micro*, 2006.

[6] Nishiyama et al. "SecureC: control-flow protection against general buffer overflow attack," C*OMPSAC*, 2005.

[7] Sinnadurai et al. "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[8] Alves et al. "TrustZone: Integrated hardware and software security." ARM white paper, 2004.

[9] McKeen et al. "Innovative instructions and software model for isolated execution." In HASP@ ISCA, 2013.

[10] Intel: Control-Flow Enforcement Technology Review, 2016.

[11] Ramakesavan et al. "Intel memory protection extensions (intel mpx) enabling guide," 2015.

[12] Yoo et al. "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing." SC-Intl Conf for HPC, Networking, Storage and Analysis. 2013.

[13] Kasikci et al. "Failure sketching: a technique for automated root cause diagnosis of in-production failures." In SOSP, 2015.

[14] Dhawan et al. "Architectural support for software-defined metadata processing." SIGARCH Computer Arch News, 2015.

[15] Jonathan Salwan, ROPgadget, http://shell-storm.org/project/ROPgadget/

[16] Nethercote, Nicholas, and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." ACM Sigplan notices. Vol. 42. No. 6. ACM, 2007

[17] Nginx server, http://nginx.org/en/

[18] GNN https://towardsdatascience.com/https-medium-com-aishwaryajadhav-applications-of-graph-neural-networks-1420576be574

[19] Node Embeddings http://snap.stanford.edu/proj/embeddings-www/files/nrltutorial-part2-gnns.pdf

[20] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriel Monfardini. 2008. The graph neural network model. IEEE Transactions on Neural Networks 20, 1 (2008), 61–80

[21] Aynaz Taheri, Kevin Gimpel, and Tanya Berger-Wolf. 2018. Learning graph representations with recurrent neural network autoencoders. KDD Deep Learning Day (2018)

[22] Leo Breiman. 2001. Random forests. Machine learning 45, 1 (2001), 5–32.