DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis

Wenbo Guo, Dongliang Mu, and Xinyu Xing, *The Pennsylvania State University;*
Min Du and Dawn Song, *University of California, Berkeley*

https://www.usenix.org/conference/usenixsecurity19/presentation/guo

This paper is included in the Proceedings of the
28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the
28th USENIX Security Symposium
is sponsored by USENIX.

DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis

†*Wenbo Guo*,* †*Dongliang Mu*,* †*Xinyu Xing*, ‡*Min Du*, ‡*Dawn Song*
†*College of IST, Pennsylvania State University*
‡*Department of EECS, University of California, Berkeley*

Abstract

Value set analysis (VSA) is one of the most powerful binary analysis tools, which has been broadly adopted in many use cases, ranging from verifying software properties (*e.g.,* variable range analysis) to identifying software vulnerabilities (*e.g.,* buffer overflow detection). Using it to facilitate data flow analysis in the context of postmortem program analysis, it however exhibits an insufficient capability in handling memory alias identification. Technically speaking, this is due to the fact that VSA needs to infer memory reference based on the context of a control flow, but accidental termination of a running program left behind incomplete control flow information, making memory alias analysis clueless.

To address this issue, we propose a new technical approach. At the high level, this approach first employs a layer of instruction embedding along with a bi-directional sequence-to-sequence neural network to learn the machine code pattern pertaining to memory region accesses. Then, it utilizes the network to infer the memory region that VSA fails to recognize. Since the memory references to different regions naturally indicate the non-alias relationship, the proposed neural architecture can facilitate the ability of VSA to perform better alias analysis. Different from previous research that utilizes deep learning for other binary analysis tasks, the neural network proposed in this work is fundamentally novel. Instead of simply using off-the-shelf neural networks, we introduce a new neural network architecture which could capture the data dependency between and within instructions.

In this work, we implement our deep neural architecture as DEEPVSA, a neural network assisted alias analysis tool. To demonstrate the utility of this tool, we use it to analyze software crashes corresponding to 40 memory corruption vulnerabilities archived in Offensive Security Exploit Database. We show that, DEEPVSA can significantly improve VSA with respect to its capability in analyzing memory alias and thus escalate the ability of security analysts to pinpoint the root cause of software crashes. In addition, we demonstrate that our proposed neural network outperforms state-of-the-art neural architectures broadly adopted in other binary analysis tasks. Last but not least, we show that DEEPVSA exhibits nearly no false positives when performing alias analysis.

*Equal Contribution.

1 Introduction

Despite the best efforts of developers, software inevitably contains flaws that may be leveraged as security vulnerabilities. Modern operating systems integrate various security mechanisms to prevent software faults from being exploited [18, 36, 51, 53]. To bypass these defenses and hijack program execution, an attacker therefore needs to constantly mutate an exploit and make many attempts. While in their attempts, the exploit triggers a security vulnerability and makes the running process terminate abnormally.

To analyze the unexpected termination (*i.e.,* program crash) and thus pinpoint the root cause, software developers or security analysts need to perform backward taint analysis [17, 20, 39], track down how a bad value is passed to the crashing site and thus pinpoint the statements that led to the crash. Technically speaking, this process can be significantly facilitated – and even automated – if the control and data flows pertaining to the crash are available upon its termination.

Recently, a large amount of research has demonstrated that program execution can be recorded through hardware tracing (*e.g.,* [30, 55]) in a least intrusive manner. As a result, a software developer can easily restore the control flow pertaining to a program crash. However, the recovery of data flow from the execution trace alone is still challenging, especially when source code is not available. As it has been discussed in recent research [55], this is primarily because data flow construction is highly dependent upon the capability of memory alias analysis [4, 5].

Of all the memory alias analysis techniques proposed in past research, value-set analysis (VSA) is the most *effective* and *efficient* technique and has been broadly adopted to facilitate the ability of identifying memory alias at the binary

level [6]. Applied in the context of postmortem program analysis, it however exhibits an insufficient capability in handling memory alias identification. Technically speaking, this is mainly because VSA needs to infer memory references based on the context of a control flow. However, accidental termination of a running program only leaves behind incomplete control flow information, making memory alias analysis clueless.

To address this technical issue, we introduce a deep neural network to enhance the capability of VSA in memory alias analysis, especially in the context of software failure diagnosis. More specifically, we use this neural network to learn the memory regions that each memory access refers to. The rationale behind this approach is as follows. VSA divides the address space of a process into several non-overlapping regions (*i.e.,* stack, heap, and global) and deem pairs of memory references to different regions as non-alias. With incomplete control flow information pertaining to a software crash, VSA loses the execution context of a crashing program and typically exhibits bad performance in assigning memory references to different memory regions. Using deep learning, we can learn complex execution patterns pertaining to memory region accesses, restore the memory regions that VSA fails to infer through incomplete control flow, and finally enhance the capability of alias analysis for postmortem program analysis.

Different from previous research that utilizes deep learning to tackle other binary analysis problems (*e.g.,* [15, 48, 49, 56]), the deep neural network used in this work is novel. Instead of simply applying an off-the-shelf neural architecture to our problem domain, we propose a new neural network architecture. To be specific, our proposed solution first utilizes an instruction embedding network to capture the semantic of each instruction. Then, it employs a bi-directional sequence-to-sequence neural architecture to learn the dependency between the instructions and predict the memory access for each individual instruction. With this new design practice, we could capture the dependency relationship within and between instructions and thus accurately predict the memory regions that each instruction attempts to access. As we will discuss and demonstrate in Section 3 and 4, this perfectly reflects the characteristic of binary code analysis and significantly benefits alias analysis in the context of software failure diagnosis.

We implemented our proposed technique as DEEPVSA [1], a neural network-assisted alias analysis tool for postmortem program analysis. To the best of our knowledge, DEEPVSA is the first tool that takes advantage of deep learning to improve alias analysis in the context of postmortem program analysis. We manually analyzed program crashes corresponding to 40 memory corruption vulnerabilities gathered from the Offensive Security Exploit Database Archive [47] and compared our manual analysis with the analysis conducted by DEEPVSA.

[1] The code, data and models of DEEPVSA are available at https://github.com/Henrygwb/deepvsa/.

```
1     sub     esp, 0x14
2     call    malloc
      ......
3     ret
4     mov     [eax], test
5     mov     [esp+0x8], eax
-------------------------------------------------
6     push    eax
7     call    child
8     push    ebp
9     mov     ebp, esp
10    mov     [0xC8], 0x0
11    mov     eax, [ebp+0x8]
12    mov     [eax], 0x1
13    mov     [eax+0x4], 0x2
14    mov     eax, 0
15    pop     ebp
16    ret
17    mov     eax, [esp+0xC]
18    call    [eax]         <--- crash site
```

Figure 1: An example instruction trace prior to a program crash.

We observed that DEEPVSA can accurately resolve approximately 35% of unknown memory relationships that VSA fails to identify when performing analysis on a crashing execution. In addition, we discovered that the escalation in alias analysis significantly improves the capability in tracking down the root cause of software crashes. For about 75% failure cases, DEEPVSA is capable of assisting backward taint analysis in identifying the root causes of their crashes. Compared with the broadly adopted neural networks in other binary analysis tasks, we also demonstrate that our new neural network architecture introduces no false positives in memory alias identification.

In summary, this paper makes the following contributions:

- We discover that deep neural networks are a viable approach towards addressing alias analysis issues in the context of software failure diagnosis.
- We propose a new neural network architecture which could be used to improve alias analysis for VSA and thus escalate the ability to diagnose the root cause of software crashes.
- We implement our deep learning technique as DEEPVSA– a tool for alias analysis facilitation – and demonstrate its effectiveness by using 40 distinct software crashes covering approximately 1.6 million lines of execution trace in total.

The rest of the paper is organized as follows. Section 2 provides an overview of value-set analysis and its limitations in postmortem program analysis. Section 3 presents the deep neural network we propose to improve alias analysis. Section 4 describes our implementation and evaluation, demonstrating the utility of DEEPVSA. Section 5 surveys related work. Finally, we conclude this work in Section 6.

	[eax]@4	...	[0xC8]@10	[ebp+0x8]@11	[eax]@12	[eax+4]@13	[esp+0xC]@17	[eax]@18
[eax]@4	–	...	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...
[0xC8]@10	NA	...	–	...	0	0	0	0
[ebp+0x8]@11	NA	...	0	–	0	0	0	0
[eax]@12	NA	...	?	?	–	0	0	1
[eax+0x4]@13	NA	...	?	?	?	–	0	0
[esp+0xC]@17	NA	...	0	0	?	?	–	0
[eax]@18	NA	...	?	?	?	?	?	–

(a) Alias matrix identified by VSA. '0', '1' and '?' represent non-alias, alias and may-alias relationships respectively.

Line #	Complete Trace		Incomplete Trace without DL		Incomplete Trace with DL	
	A-loc	Value-set	*A-loc*	Value-set	*A-loc*	Value-set
1	esp	(⊥, [-0x14, -0x14], ⊥)	NA	NA	NA	NA
4	[eax] (⊥, ⊥, [0, 0])	(test, ⊥, ⊥)	NA	NA	NA	NA
5	[esp+0x8] (⊥, [-0xC, -0xC], ⊥)	(⊥, ⊥, [0, 0])	NA	NA	NA	NA
6	esp	(⊥, [-0x18, -0x18], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)
	[esp] (⊥, [-0x18, -0x18], ⊥)	(⊥, ⊥, [0, 0])	[esp] (⊥, [-0x4, -0x4], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0x4, -0x4], ⊥)	(⊥, ⊥, [X, X])
7	esp	(⊥, [-0x1C, -0x1C], ⊥)	esp	(⊥, [-0x8, -0x8], ⊥)	esp	(⊥, [-0x8, -0x8], ⊥)
	[esp] (⊥, [-0x1C, -0x1C], ⊥)	([L17, L17], ⊥, ⊥)	[esp] (⊥, [-0x8, -0x8], ⊥)	([L17, L17], ⊥, ⊥)	[esp] (⊥, [-0x8, -0x8], ⊥)	([L17, L17], ⊥, ⊥)
8	esp	(⊥, [-0x20, -0x20], ⊥)	esp	(⊥, [-0xC, -0xC], ⊥)	esp	(⊥, [-0xC, -0xC], ⊥)
	[esp] (⊥, [-0x20, -0x20], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0xC, -0xC], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0xC, -0xC], ⊥)	(⊤, ⊤, ⊤)
9	ebp	(⊥, [-0x20, -0x20], ⊥)	ebp	(⊥, [-0xC, -0xC], ⊥)	ebp	(⊥, [-0xC, -0xC], ⊥)
10	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)
11	[ebp+0x8] (⊥, [-0x18, -0x18], ⊥)	(⊥, ⊥, [0, 0])	[ebp+0x8] (⊥, [-0x4, -0x4], ⊥)	(⊤, ⊤, ⊤)	[ebp+0x8] (⊥, [-0x4, -0x4], ⊥)	(⊥, ⊥, [X, X])
	eax	(⊥, ⊥, [0, 0])	eax	(⊤, ⊤, ⊤)	eax	(⊥, ⊥, [X, X])
12	[eax] (⊥, ⊥, [0, 0])	([0x1, 0x1], ⊥, ⊥)	[eax] (⊤, ⊤, ⊤)	([0x1, 0x1], ⊥, ⊥)	[eax] (⊥, ⊥, [X, X])	([0x1, 0x1], ⊥, ⊥)
13	[eax+4] (⊥, ⊥, [4, 4])	([0x2, 0x2], ⊥, ⊥)	[eax+4] (⊤, ⊤, ⊤)	([0x2, 0x2], ⊥, ⊥)	[eax+4] (⊥, ⊥, [X+0x4, X+0x4])	([0x2, 0x2], ⊥, ⊥)
14	eax	([0x0, 0x0], ⊥, ⊥)	eax	([0x0, 0x0], ⊥, ⊥)	eax	([0x0, 0x0], ⊥, ⊥)
15	ebp	(⊤, ⊤, ⊤)	ebp	(⊤, ⊤, ⊤)	ebp	(⊤, ⊤, ⊤)
	esp	(⊥, [-0x1C, -0x1C],⊥)	esp	(⊥,[-0x8,-0x8],⊥)	esp	(⊥,[-0x8,-0x8],⊥)
16	esp	(⊥, [-0x18, -0x18], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)
17	[esp+0xC] (⊥, [-0xC, -0xC], ⊥)	(⊥, ⊥, [0, 0])	[esp+0xC] (⊥, [0x8, 0x8], ⊥)	(⊤, ⊤, ⊤)	[esp+0xC] (⊥, [0x8, 0x8], ⊥)	(⊥, ⊥, [X, X])
	eax	(⊥, ⊥, [0, 0])	eax	(⊤, ⊤, ⊤)	eax	(⊥, ⊥, [X, X])
18	[eax] (⊥, ⊥, [0, 0])	([0x1, 0x1], ⊥, ⊥)	[eax] (⊤, ⊤, ⊤)	(⊤, ⊤, ⊤)	[eax] (⊥, ⊥, [X, X])	([0x1, 0x1], ⊥, ⊥)

(b) *A-locs* and value-sets corresponding to complete and incomplete traces with and without the facilitation of deep learning (DL).

Table 1: The results of value-set analysis against the instruction trace shown in Figure 1.

2 Background and Problem Scope

As is described and discussed in many recent research works (e.g. [19, 55]), new hardware components could trace program execution in a least intrusive fashion. With this capability, security analysts could easily obtain the control flow pertaining to a software crash. Using the execution trace, it is however still challenging to pinpoint the root cause of the crash (i.e., the instructions truly attributive to the crash). On the one hand, this is because a security analyst barely has the access to the source code of the crashing program. On the other hand, this is because a security analyst needs to analyze the data flow of the crashing trace which involves memory alias analysis at the binary level. To tackle this challenge, value-set analysis (VSA) can be adopted. In this section, we first introduce how software instrumentation and hardware tracing are used to record program execution. Second, we briefly describe how to perform value-set analysis on a recorded execution trace. Third, we specify how to use the derived value set to perform alias analysis and thus diagnose the root cause of a software crash. Finally, we provide a more in-depth discussion about why VSA behaves poorly in many real-world applications.

2.1 Program Tracing for Software Debugging

Software instrumentation techniques have long been used to fully record program execution and thus facilitate the root cause diagnosis for a crashing program (e.g., [38, 37]). However, such an approach imposes significant overhead to a software normal operation. In order to minimize additional overhead, some lightweight instrumentation techniques have been proposed (e.g., [41, 40]). While they are less intrusive and informative for assisting software debugging, such a lightweight approach cannot be used to fully restore the

control flow pertaining to a software crash.

Recently, the advance in hardware-assisted processor tracing significantly ameliorates this situation. With the emergence of brand new hardware components, such as Intel PT [27] and ARM ETM [2], software developers and security analysts can trace instructions executed with nearly no overhead and save them in a circular buffer. At the time of a program crash, an operating system includes the trace into a crash dump. Since this post-crash artifact contains both the state of crashing memory and the execution history (*i.e.,* the last *N* instructions executed prior to the crash), software developers not only can inspect the program state at the time of the crash, but also fully reconstruct the control flow that led to the crash.

In this work, we focus on using an enhanced value-set analysis technique to analyze such an aforementioned post-crash artifact and thus facilitate the root cause diagnosis of a crashing program. It should be noted that the aforementioned lightweight software instrumentation approach is out of the scope of this research because they cannot provide a complete instruction trace for value-set analysis to identify memory alias and thus pinpoint the root cause of the crash.

2.2 Value-set Analysis

Value-set analysis is an algorithm designed for analyzing assembly code or an instruction trace in a static fashion. Based on the observation that memory layout generally follows, VSA partitions memory into 3 disjoint memory regions – global[2], stack and heap – and assigns instructions to the regions, accordingly. For some instructions, VSA achieves region assignment by examining the semantics of the instructions. For example, from a binary code perspective, accesses to global and stack variables appear as `[absolute-address]` and `[esp-offset]`. Thus, VSA can easily link the global and stack regions to the instructions `mov edx,[0x8050684]` and `lea eax,[esp+4]`, respectively. For other instructions, VSA performs a simple forward data flow analysis to determine the regions tied to instructions in a conservative fashion[3]. Take for example the instruction trace shown in Figure 1. The instruction at `line 4` indicates a write to the target memory `[eax]`. Through a forward data flow analysis, VSA could easily pinpoint that the value of `eax` was passed through `line 3` because the library function `malloc` places its return value in the register `eax`. Given that the semantics of `malloc` is to allocate a memory region on the heap and then return its reference to the caller function, VSA could easily assign the heap region to the instruction at `line 4`.

[2] Note that the global region consists of initialized and uninitialized data segments.

[3] By 'conservative fashion', we refer to the fact that VSA does not actively infer the value held in a memory cell if the data flow propagation is blocked by an unknown memory reference.

In addition to assigning instructions to memory regions in the ways above, VSA tracks down variable-like entities referred to as *a-locs*. By convention, an *a-loc* could be a register, a memory cell on the stack, on the heap, or in the global region. Take the instruction trace shown in Figure 1 as an example. The register *a-locs* contain all the registers `esp`, `eax` and `ebp`. The global *a-locs* contain `[0xC8]`. The stack *a-locs* include `[esp]`, `[esp+0x8]`, `[esp+0xC]` and `[ebp+0x8]`. The heap *a-locs* consist of `[eax]` and `[eax+0x4]`. It should be noticed that, as is illustrated in Table 1b, VSA represents a non-register *a-loc* as a combination of the value held by a memory cell and the value set indicating the address of that memory cell. For example, the instruction `mov [esp+0x8],eax` accesses the stack memory, and VSA specifies its corresponding stack *a-loc* as `[esp+0x8]` `(`$\perp$`, [-0xC, -0xC], `$\perp$`)`. Here, `[esp+0x8]` indicates the name of the stack memory cell, and `(`$\perp$`, [-0xC, -0xC], `$\perp$`)` is the value set of the memory address or, in other words, the values that `esp+0x8` could potentially equal to at the site of that instruction.

For each *a-loc* identified, VSA computes a value set, indicating the set of values that each *a-loc* could potentially equal to. By convention, VSA represents such a value set as a *3-tuple* pertaining to the three memory regions partitioned. For each element in the tuple, VSA specifies a range of offsets which indicates the values that the *a-loc* could equal to with respect to the corresponding memory region.

To illustrate this, we take the register *a-loc* `esp` as an example. As depicted in the first row of Table 1b, VSA specifies its value set as a *3-tuple* `(global `$\mapsto \perp$`, stack `$\mapsto$` [-0x14, -0x14], heap `$\mapsto \perp$`)`, for brevity `(`$\perp$`, [-0x14, -0x14], `$\perp$`)`. In this set, $\perp$ is a symbol denoting the empty set of offsets (*i.e.,* $\varnothing$). It reflects the fact that the register `esp` is the stack pointer in x86 architecture and cannot refer to any memory cells on the heap or global region. Since the semantics of the first instruction is to offset `esp` by `0x14` from the starting point of the stack, VSA assigns the value set {`-0x14`} to the register *a-loc* `esp`, and attaches this set to the stack. It should be noticed that for specification consistency we write the value sets {`-0x14`} tied to the stack as `[-0x14, -0x14]`.

2.3 Alias Analysis and Root Cause Diagnosis

Alias Analysis. Given a control flow specified as a sequence of instructions executed prior to a program crash, VSA can track down *a-locs*, derive value sets, and perform memory alias analysis by examining the value set tied to each of the *a-locs*. To illustrate this, we again take the instruction trace depicted in Figure 1 as an example and assume they represent the entire execution trace prior to a program crash. Supposing that Table 1b indicates the value set tied to each of the *a-locs* identified from the instruction trace, we can easily observe that `[esp]` at `line 6` and `[ebp+0x8]` at `line 11` refer to the same memory region or in other words they are alias of each other. In addition, we can observe `[eax]`

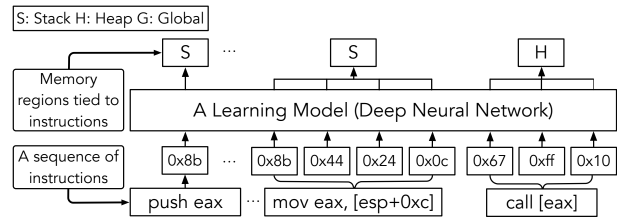
at `line 4, 12 and 18` are also alias between each other. This is simply because the *a-locs* tied to these memory regions carry the overlapping value set corresponding to their addresses, *i.e.,* `(⊥, [-0x18, -0x18], ⊥)` for `[esp]` and `[ebp+0x8]`; `(⊥, ⊥, [0,0])` for `[eax]`. To better understand the effect of VSA on alias analysis, we derive all the alias and non-alias relationships from the value sets specified in Table 1b, and depict them in the upper triangular portion of the matrix shown in Table 1a.

Root Cause Diagnosis. With the alias analysis results and the value sets in hand, it is relatively easy to perform a backward taint analysis and thus track down the root cause of a program crash. To illustrate this process, we continue the example shown in Figure 1. Given that the program crashes at `line 18` when the program performs an indirect call, we can easily discover that the bad destination `[eax]` was passed through the instruction at `line 12` in which memory `[eax]` is assigned with a constant `0x1`. As is described above, `[eax]` at `line 12 and 18` are the alias of each other. Therefore, we can safely conclude the bad destination originally comes from the instruction `mov [eax],0x1` in `line 12`. Through this backward analysis, we could deem the instruction `mov [eax],0x1` as the root cause of the crash.

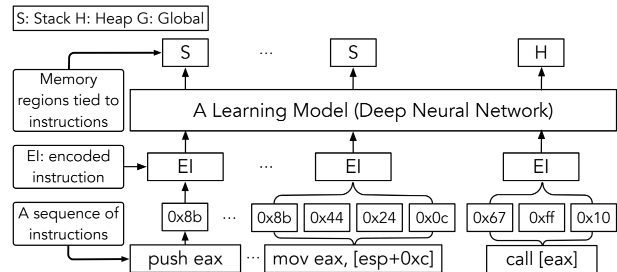
2.4 Problem Scope

As is described in the aforementioned example, VSA exhibits perfect performance in alias analysis and we could identify the root cause of the crash successfully. However, this does not imply that VSA could significantly resolve the memory alias issue and thus perfectly facilitate postmortem program analysis. To demonstrate this, we again take for example the instruction trace shown in Figure 1. However, different from the setup specified above, we assume the trace is available only starting from `line 6`. As is described in Section 2.1, hardware tracing components store a instruction trace in a circular buffer with limited size. As a result, it is commonplace that a security analyst cannot obtain a complete crashing trace but only a partial execution chronology prior to a program crash. By truncating the trace in our example, we emulate the scenario where there are only last *N* instructions recorded in a post-crash artifact.

In Table 1b, we also show the *a-locs* identified from this truncated trace. Compared with the value set derived from the full execution trace shown in the same figure, we can easily observe that nearly all the value sets tied to the *a-locs* are varied. This is because VSA performs an over-approximation in value-set construction and the missing context limits the capability of VSA with respect to reasoning memory regions or offsets within a region. Take the *a-loc* indicated by `[eax+0x4]` `(⊤, ⊤, ⊤)` as an example. Without the complete execution context of the crashing program, VSA conservatively assumes `eax` could equal to any value. Thus, memory `[eax+0x4]` could refer to any memory regions with an arbitrary offset



(a) A neural network taking machine code as its input.



(b) A neural network taking as input encoded instructions.

Figure 2: Two neural networks that identify memory region accesses pertaining to each instruction by taking as input a sequence of machine code and a sequence of encoded instructions respectively.

indicated by the symbol ⊤. As is shown in the instruction in `line 13`, the value of `[eax+0x4]` is assigned by a value from a global region. Therefore, the value set tied to this *a-loc* can be represented as `([0x2, 0x2], ⊥, ⊥)`. From the *a-locs* identified from the truncated trace along with their value set, we follow the aforementioned approach to examine value set intersection, and illustrate the alias and non-alias relationships in the lower triangular portion of the matrix shown in Table 1a. As we can easily observe, without the full execution trace, VSA over-approximates value sets tied to *a-locs*, and conservatively deems many memory pairs as may-alias relationships. Since may-alias represents uncertainty relationship, Table 1a illustrates them as the question symbol '*?*'. Using such results to derive the data flow for software crash diagnosis, it is not difficult to observe that a security analyst can barely yield any useful results or in other words pinpoint the root cause of the program crash for the simple reason that VSA has the limited capability in tracking down the memory alias.

3 Technical Approach

To address the problem above, we propose a technical approach driven by a deep neural network. In this section, we first discuss why deep learning could potentially facilitate VSA and thus improve software crash analysis. Second, we briefly describe neural network architectures commonly used in other binary analysis tasks. Third, we discuss the limita-

tion of these existing neural networks and then specify how to design a new neural architecture to better tackle our problem. Finally, we present the detail of our new neural architecture and specify how to integrate it into conventional VSA.

3.1 Overview

Recall that, when a crashing trace is incomplete, VSA exhibits an insufficient capability in alias analysis and thus fails root cause diagnosis. As is demonstrated above, this is because the missing context restricts the ability of VSA to determine the region of memory accesses for some instructions. To address this pitfall, we leverage a deep neural network to enhance VSA with the ability to infer memory region(s) for instructions. In the following, we describe the rationale behind this idea and illustrate why it could benefit the diagnosis of software crashes.

Rationale behind our idea. In many previous applications (*e.g.,* speech recognition [24] and API generation [25]), it has been demonstrated that some sequence-to-sequence neural network architectures can be used to learn patterns from a sequence of inputs, thus facilitating the determination of a label for each individual input. As a result, in order to augment conventional VSA with the ability to infer the memory region(s) that each instruction refers to, intuition suggests that we can view an execution trace as a sequence of machine code or instructions, partition memory into disjoint regions (*e.g.,* stack, heap and global), treat each region as an individual label tied to each instruction and eventually use a sequence-to-sequence deep neural network to predict that label for each instruction. For example, given the instruction `push 0x68732f2f` represented by machine code `[0x68, 0x2f, 0x2f, 0x73, 0x68]`, we could determine the stack region is tied to this instruction by using either of the two designs shown in Figure 2. As is depicted in the figure, the two designs take the input differently, one with machine code as the input directly to a deep learning model and the other with the encoded instructions as the input to a model. In Section 3.3, we compare these two designs and describe why we choose one over the other. In Section 4, we show their performance difference.

Effect upon root cause diagnosis. With the augmentation above, VSA could typically perform better alias analysis and thus benefit the diagnosis of a software crash. We illustrate this by again taking for example the instruction trace shown in Figure 1. Recall that, without the complete execution context, conventional VSA cannot determine the memory region that `eax` refers to. Therefore, it assumes `[eax]` and `[eax+0x4]` could represent any memory regions, assigns `eax` and `eax+0x4` with value-set `(⊤, ⊤, ⊤)` and eventually fails the root cause diagnosis of that crash.

Given the sequence of the instructions tied to the crashing trace, assume a deep neural network could correctly infer that, the register `eax` at `line 6` refers to a memory region at the heap. Then, VSA could assign `eax` with value-set `(⊥, ⊥, [X, X])` where `[X, X]` denotes an unknown address on the heap. With this, VSA could further update the value sets for corresponding *a-locs*. We show the updated value sets in Table 1b under the column "*Incomplete Trace with DL*". As we can observe, the memory reference `[eax]` at `line 12 and 18` are aliased to each other because they both refer to the same memory address `[X, X]` on the heap. With this alias analysis result, VSA could quickly assist backward taint in tracking down the instruction at `line 12` – the root cause of the crash – even though this crashing trace is partial and incomplete.

3.2 Existing Neural Architectures

To perform binary analysis with deep learning, previous research typically utilized three types of recurrent neural networks (RNNs) – vanilla RNN [33], long short-term memory (LSTM) [22] and gated recurrent units (GRU) [13]. Here, we briefly describe them in turn.

3.2.1 Vanilla Recurrent Neural Network

A vanilla RNN (RNN for brevity) is specialized for processing a sequence of values $x^{(1)},\ldots,x^{(t)}$. When trained to perform a prediction from the past sequence of inputs, it typically maps the sequence to a fixed length vector $h^{(t)}$ through a function $g^{(t)}$:

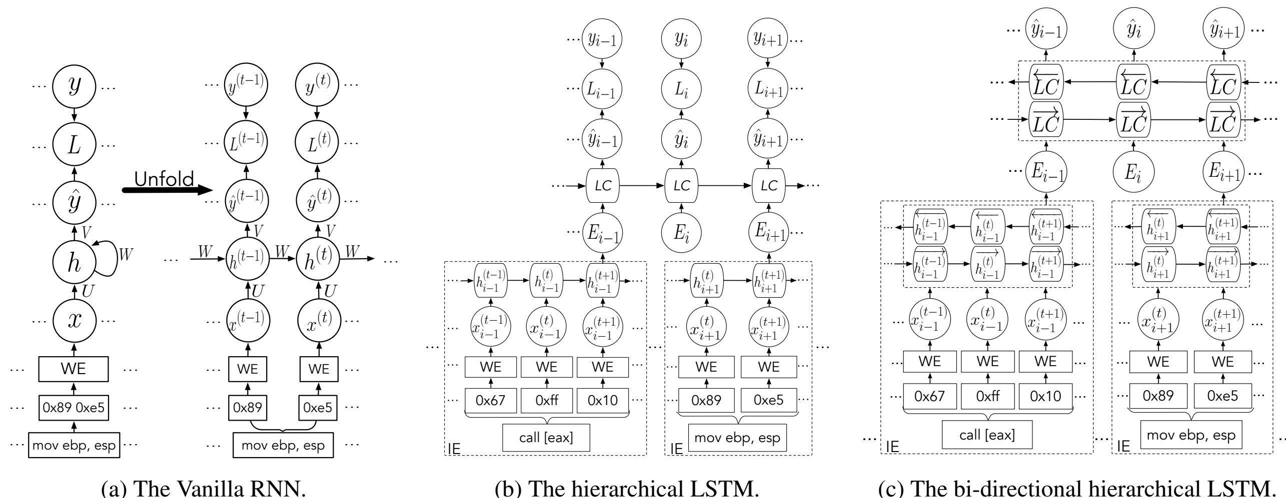
$$\begin{aligned} h^{(t)} &= g^{(t)}(x^{(t)},x^{(t-1)},x^{(t-2)},\ldots,x^{(2)},x^{(1)}), \\ &= f(h^{(t-1)},x^{(t)};\theta). \end{aligned}$$

As we can observe from this equation, the function $g^{(t)}$ takes the whole past sequence as input and produces a summary $h^{(t)}$ for that sequence. In an RNN, $h^{(t)}$ refers to a hidden state. As is illustrated in Figure 3a, an RNN can be unfolded as a chain structure where each hidden state is connected to the previous one [23]. As such, $g^{(t)}$ can be factorized into the repeated application of a function f, which controls the transition from the previous hidden state to the next one (*i.e.,* the recurrent neuron). For example, assuming the length of the chain to be 3 – indicating a finite number of hidden states – we can then obtain

$$\begin{aligned} h^{(3)} &= f(h^{(2)};\theta), \\ &= f(f(h^{(1)};\theta);\theta). \end{aligned}$$

To make predictions using the chain structure depicted in Figure 3a, an RNN follows a forward propagation in which it begins with an initial state $h^{(0)}$ and then utilizes the update equations below to compute the prediction $\hat{y}^{(t)}$ accordingly.

$$\begin{aligned} a^{(t)} &= \mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b, \\ h^{(t)} &= \tanh(a^{(t)}), \\ o^{(t)} &= \mathbf{V}h^{(t)} + c, \\ \hat{y}^{(t)} &= \text{softmax}(o^{(t)}). \end{aligned}$$



(a) The Vanilla RNN. (b) The hierarchical LSTM. (c) The bi-directional hierarchical LSTM.

Figure 3: Recurrent neural networks with various architectures serving for different purposes. Note that "LC" indicates LSTM cell, "IE" stands for the embedding for each instruction and "WE" refers to the word embedding.

Here, bias vectors b and c are parameters. $\tanh(\mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b)$ is the detailed form of the recurrent neuron f in which tanh is an activation function [54]. Softmax refers to the softmax classifier [10]. Along with the weight matrices U, V and W, pertaining to input-to-hidden, hidden-to-output, and hidden-to-hidden connections respectively, the bias vectors can be learned by minimizing the loss function described below

$$\begin{aligned} L^{(t)} &= L(x^{(1)}, x^{(2)}, ..., x^{(t)}, y^{(1)}, y^{(2)}, ..., y^{(t)}) \\ &= \sum_t L^{(t)} \\ &= -\sum_t \log p_{model}(y^{(t)} | x^{(1)}, x^{(2)}, ..., x^{(t)}), \end{aligned}$$

where $p_{model}(y^{(t)}|x^{(1)}, \ldots, x^{(t)})$ is the probability from the prediction vector $\hat{y}^{(t)}$ corresponding to the entry for the true label vector $y^{(t)}$. Similar to other neural networks commonly used (*e.g.,* multi-layer perceptron [44] and convolution neural networks [32]), the minimization of the aforementioned loss function can be achieved by using different kinds of optimization algorithms (*e.g.,* stochastic gradient descent [11], ADAM [31], RMSprop [52]) with respect to the bias parameters and weight matrices. The details of these optimization algorithms can be found in [45].

3.2.2 Long Short-Term Memory

In the cybersecurity community, recent works have demonstrated that a vanilla RNN has already demonstrated great performance when performing binary analysis (*e.g.,* [15, 48]). However, it has been noted that, as is used in other applications such as speech recognition and machine translation, such an ordinary recurrent architecture is not sufficient in processing a long sequence of inputs. This is because a vanilla RNN naturally struggles to remember information for long periods of time or, in other words, suffers from derivative vanishing and explosion problems [26]. To address this issue, other works have used a long short-term memory (LSTM) model to carry out binary analysis.

Similar to a vanilla RNN depicted in Figure 3a, LSTM also has a chain structure. However, it replaces the aforementioned hidden states with LSTM cells, and each cell carries a set of parameters and a system of gating units that controls the flow of information. In an LSTM network, each cell has a state unit $s^{(t)}$ as well as three gating units – a forget gate unit $f^{(t)}$, an external input gate unit $g^{(t)}$, and an output gate $q^{(t)}$ – which together control the output $h^{(t)}$ of the LSTM cell via the following equation

$$\begin{aligned} s^{(t)} &= f^{(t)} \odot s^{(t-1)} + g^{(t)} \odot \sigma(\mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b), \\ h^{(t)} &= q^{(t)} \odot \tanh(s^{(t)}). \end{aligned}$$

Here, $\sigma(\cdot)$ denotes a sigmoid function [29] which sets a value between 0 and 1, and $\odot$ represents the element-wise multiplication. b, U and W respectively indicate the biases, input weights, and recurrent weights into an LSTM cell. To compute the gate units, one could follow the equations below

$$\begin{aligned} g^{(t)} &= \sigma(\mathbf{W}^g h^{(t-1)} + \mathbf{U}^g x^{(t)} + b^g), \\ f^{(t)} &= \sigma(\mathbf{W}^f h^{(t-1)} + \mathbf{U}^f x^{(t)} + b^f), \\ q^{(t)} &= \sigma(\mathbf{W}^q h^{(t-1)} + \mathbf{U}^q x^{(t)} + b^q), \end{aligned}$$

where $\{b^f, b^g, b^q\}$, $\{\mathbf{U}^f, \mathbf{U}^g, \mathbf{U}^q\}$ and $\{\mathbf{W}^f, \mathbf{W}^g, \mathbf{W}^q\}$ are respectively: biases, input weights, and recurrent weights for the forget, external input, and output gates. Similar to b, $\mathbf{U}$ and $\mathbf{W}$, they are also the parameters that can be learned via the optimization algorithms mentioned above. Again, more details of parameter computation can be found at [23].

3.2.3 Gated Recurrent Units

As is described in previous research [15], gated recurrent units (GRU) can also be used for some of binary analysis tasks. GRU is an alternative LSTM which can also capture long term dependency. The main difference between GRU and LSTM is that GRU replaces the forget gate f and output gate q in LSTM with one update gate. More specifically, it integrates both forget and output gates into a single gating unit $u^{(t)}$. As a result, it reduces the parameters that a network has to learn and thus poses a lower computational cost. The following equations indicate how to compute the output $h^{(t)}$ of a GRU cell:

$$r^{(t)} = \sigma(\mathbf{W}^r h^{(t-1)} + \mathbf{U}^r x^{(t)} + b^r),$$
$$u^{(t)} = \sigma(\mathbf{W}^u h^{(t-1)} + \mathbf{U}^u x^{(t)} + b^u),$$
$$h^{(t)} = u^{(t)} \odot h^{(t-1)} + (1 - u^{(t)}) \odot \tanh(\mathbf{W}(r^{(t)} \odot h^{(t-1)}) + \mathbf{U}x^{(t)} + b).$$

Here, $r^{(t)}$ stands for a reset gate which controls the influence of the past sequences of inputs upon the current one. $\{b^r, \mathbf{U}^r, \mathbf{W}^r\}$ and $\{b^u, \mathbf{U}^u, \mathbf{W}^u\}$ are gate weights. Along with the bias b and weights $\mathbf{U}$, $\mathbf{W}$, they need to be learned through the aforementioned optimization algorithms.

3.3 Our Neural Network Architecture

As we described in Section 3.1, we could utilize two different design mechanisms to predict the memory region that each instruction refers to. For the design shown in Figure 2a, we could simply leverage any of the aforementioned recurrent neural networks to take as input the sequence of machine code, learn the pattern hidden behind the machine code sequence and predict the memory region for each instruction. As they have already demonstrated in other binary analysis tasks (*e.g.,* [48, 15]), we could expect this design could perform reasonably well in memory region identification. However, following the intuition described below, we do not utilize this design. Rather, we develop our technique by using the alternative design shown in Figure 2b.

Take for example the instruction sequence `push ebp; mov ebp, esp` indicated by the byte sequence `[0x55, 0x89, 0xe5]`. An existing neural network model could take this machine code sequence as input and make predictions for their corresponding memory accesses based on the dependency between the bytes. It is not too difficult to observe that this simple approach neglects the semantics and contexts of these instructions. As is described in Section 2, in binary analysis, the semantics and contexts of instructions could be used as indicators to infer the memory accesses tied to instructions. Therefore, intuition suggests that it could be potentially beneficial for memory region identification if we could build a neural network with the ability to capture not only the dependency between the bytes but also that between instructions.

Inspired by this, we choose the design depicted in Figure 2b and build a hierarchical LSTM architecture. We depict the structure of this learning model in Figure 3b. As we can observe, similar to existing neural networks used for other binary analysis tasks, it first maps each byte into a vector by using a word embedding mechanism [9]. Then, it groups the bytes per each instruction and utilizes an embedding network to convert each group of bytes into an instruction embedding (*i.e.,* an encoded vector). Taking the instruction embedding as the input, our neural architecture further employs a sequence-to-sequence network [50] to predict the memory region tied to each instruction.

In comparison with the aforementioned off-the-shelf recurrent architectures largely adopted by other binary analysis tasks, the proposed hierarchical LSTM architecture is composed of two networks. The embedding network models the correlation of bytes in one instruction and the sequence-to-sequence network captures the dependency between instructions. By designing the model structure in this fashion, our neural network model is able to perform memory access predictions at the instruction level and learn the dependency between and within instructions at the same time.

However, it is not difficult to note that this new recurrent architecture cannot represent a backward analysis procedure, where the memory region(s) tied to an instruction is determined by the consecutive instructions. Yet we note that this backward analysis is feasible. To illustrate this, we take the following execution trace as an example.

```
00015670 <malloc>:
 53               push  ebx
 ...
 89 44 24 04      mov   DWORD PTR [esp+0x4],eax
 e8 6d b1 fe ff call  800 <_libc_memalign@plt>
 83 c4 18         add   esp,0x18
 5b               pop   ebx
 c3               ret
```

As is illustrated above, the trace indicates the instructions and corresponding machine code executed while invoking the `malloc` function. Here, the highlighted instruction and machine code indicate the last definition of `[eax]` prior to the return of the function call. Given that the call to `malloc` places the return value in the register `eax`, indicating an address on the heap, we can reversely perform inference and conclude that the memory access tied to the highlighted instruction is within a heap region.

To enable our design with the capability of inferring memory regions in both forward and backward ways, we further upgrade our hierarchical LSTM model to a bi-directional chain structure [46]. As is shown in Figure 3c, our bi-directional chain structure is applied to both the embedding network and the sequence-to-sequence network. With respect to the embedding network, our neural architecture combines a network that moves forward, beginning from the start of the corresponding byte sequence, with another network that moves backward, starting from the end of the corresponding byte

sequence. Regarding the sequence-to-sequence network, our architecture concatenates the output of a forward embedding network with the output of a backward embedding network. Then, it takes the concatenation as input and performs memory access prediction for each individual instruction based on the sequence of instructions executed before and after that instruction.

3.4 Detail of Our Neural Architecture

Here, we describe more details of our proposed neural network architecture. More specifically, we specify how we process a crashing trace, perform corresponding computation, train the neural network and eventually utilize it to facilitate VSA.

Padding and word embedding. As is described above, our neural network utilizes a bi-directional embedding to encode each instruction prior to making predictions for their memory accesses. Before passing machine code to that embedding network, we process them as follows.

Assume we have a crashing trace containing n instructions $I_{1:n}$. For each instruction I_i, it could be represented as m bytes of machine code $b_i^{(1:m)}$. For an x86 machine, instructions do not share the same length. To design the same structure of embedding networks for instructions, we therefore pad instructions to a fixed length. To do this, we first convert each individual byte into an integer based on its value (e.g., encoding machine code `55` to its integer form `85`). Then, we pad that instruction with integer 256. In this way, we could ensure our padding does not introduce ambiguity to a target instruction. After the padding, we also utilize a word embedding to further process the padded crashing trace. In our work, our word embedding converts each byte into a one-hot vector with a dimensionality of 257. Then, the vector is multiplied with a matrix projecting the byte into a new vector (*i.e.,* $x_i^{(1:m)}$) typically with lower dimensionality.

Instruction embedding. For each instruction, we use a bi-directional LSTM model to further encode its word embedding and then generate an individual instruction embedding. Technically speaking, we achieve this by integrating the outputs of the forward and backward networks. More specifically, we utilize the following equations to compute the output of the forward network.

$$\overrightarrow{h_i^{(t)}} = \text{LSTM}(\overrightarrow{h_i^{(t-1)}}, x_i^{(t)}),$$
$$\overrightarrow{E_i} = \overrightarrow{h_i^{(m)}}.$$

Similarly, we compute the output for the backward network as follows.

$$\overleftarrow{h_i^{(t)}} = \text{LSTM}(\overleftarrow{h_i^{(t+1)}}, x_i^{(t)}),$$
$$\overleftarrow{E_i} = \overleftarrow{h_i^{(1)}}.$$

Here, $\overrightarrow{E_i}$ and $\overleftarrow{E_i}$ are the forward and backward embeddings of the instruction I_i, respectively. LSTM denotes an LSTM cell introduced above. As we can observe from the two sets of equations above, the hidden representation of the first and last bytes of the instruction, $h_i^{(m)}$ and $h_i^{(1)}$, contain the information that flows from the previous and consecutive bytes.

To combine the outputs of both forward and backward networks, we concatenate both representations in the form of $E_i = [\overrightarrow{E_i}, \overleftarrow{E_i}]$). Technically, it should be noted that we can use one single embedding network for all instructions, or employ different embedding networks for instructions. With the consideration of lowering computational overhead, our neural network architecture follows the first approach.

Sequence-to-sequence network. Given a sequence of instruction embeddings pertaining to the instructions in a crashing trace, we then use a sequence-to-sequence model mentioned above to predict the label (*i.e.,* , memory access region(s)) for each instruction. To be specific, the model takes as input the instruction embeddings $E_{1:n}$ and utilizes a bi-directional LSTM as the hidden layer of our neural architecture [4]. At the output layer of our neural network, it uses a softmax classifier to assign a corresponding label for each hidden state (*i.e.,* the hidden representation of each instruction). Different from previous deep neural network used in other binary analysis technique, which assigns a label to each byte, our new architecture gives us the ability to attach an individual prediction to each instruction.

Training strategy. Similar to the recurrent neural networks summarized in Section 3.2, we also need to leverage aforementioned optimization algorithms to estimate the parameters for our neural network. In binary analysis tasks, the training dataset is often significantly large, *e.g.,* one execution trace carries millions of lines of instructions. Using conventional gradient descent algorithms – like stochastic gradient descent – against a large data set, parameter estimation would experience significant computation overhead. To address this issue, we take advantage of mini-batch gradient descent, a variation of the gradient descent algorithm [28]. Technically speaking, this approach splits the training dataset into small batches, uses them to calculate model error through loss function and updates model parameters accordingly. Compared with other approaches, particularly stochastic gradient descent, mini-batch provides a computationally efficient process and enables parallel computations.

In addition to mini-batch gradient descent, we adopt `RMSprop` [34] to accelerate the optimization process needed for gradient descent computation. To be specific, we adjust the learning rate by dividing it by an exponentially decaying average of squared gradients. For more details, the reader could refer to an unpublished article available at Geoff Hinton's class [34]. Last but not least, we also pad the remaining sequences in the last batch with vectors where each element equals 256. In this way, we can represent each batch as a

[4] Note that we can also use GRU as an alternative to LSTM for the encoding network and the sequence to sequence network.

matrix with fixed size, making it capable of being efficiently processed at the same time.

Integration into VSA. Without the facilitation of a deep learning model and the clue of which memory region an instruction accesses, VSA initializes *a-locs* and value-set with (⊤, ⊤, ⊤), indicating the memory access in that instruction could refer to any memory regions. Using our deep neural architecture introduced above, we could have the neural network output the memory region that instruction accesses (*i.e.,* global, stack or heap). As is illustrated in Section 3.1, with this capability, we could initialize *a-locs* and value-set with ([X,Y], ⊥, ⊥), (⊥, [X,Y], ⊥) or (⊥, ⊥, [X,Y]), denoting a memory access in that instruction could refer to a particular memory area ranging from X to Y at a global, stack or heap region. Then, starting from the first instruction in the crashing trace, VSA could regularly perform forward analysis and update the value for X and Y. For example, as we have shown in Table 1b, when analyzing the instruction at `line 6`, our deep learning model initializes `eax` with value-set (⊥, ⊥, [X,Y]) and VSA updates X and Y with X=Y indicating the register `eax` refers to the memory address X at the heap region.

4 Evaluation

In this section, we describe our implementation, the dataset we utilized, set up our experiment, and summarize our experimental results. Through this evaluation, we seek to answer the following questions. ❶ Does our problem require a deep learning model or could it be resolved with conventional machine learning techniques? ❷ Can our proposed technique correctly link memory regions to instructions or, more precisely, identify the memory regions that instructions dereference? ❸ Compared with commonly adopted recurrent neural architectures that take as input the raw machine code, does the proposed neural network architecture (taking encoded instructions as the input to a neural network) exhibit better performance in terms of memory region identification? ❹ Can the memory regions identified improve the ability of VSA with respect to memory alias analysis and thus bring the positive impact upon the capability in software crash diagnosis?

4.1 Implementation

To answer the questions above, we must first train many deep neural network architectures. This requires a large training data set containing various instruction traces as well as the memory reference tied to each instruction. To facilitate the collection of the instruction traces as well as the corresponding memory accesses, we first implemented a tracing system which provides us with the ability to not only record the instructions that a target program executes but also the memory region each instruction refers to. While both Intel PT and ARM ETM could trace program execution, in this work, we utilize Intel Pin [35] to complete the implementation of our tracing system. This is because, in order to train a neural network, we have to obtain the ground truth of which memory regions instructions access but both hardware components do not provide us with such a capability (*i.e.,* recording memory regions referred by instructions).

In addition to the tracing system, we customized a VSA system which implemented an instruction parser using `libdisasm` and 84 distinct instruction handlers to perform value-set calculation. Going beyond alias analysis, the implementation of our customized VSA system also contains a backward taint component which takes the results of alias analysis and performs the root cause diagnosis for a crashing program. In total, our VSA implementation contains about 9,500 lines of C code. It should be noticed that the value-set calculation for instructions with similar semantics (*e.g.,* `ja`, `jb`, `jc`) were taken care of by a unique handler.

Recall that our ultimate goal is to use a deep neural network to facilitate VSA with respect to alias analysis and thus improve the effectiveness of software crash diagnosis. Last but not least, we therefore prototyped a neural network assisted VSA system and named it after `DEEPVSA`. In our implementation, `DEEPVSA` first utilizes a pre-trained deep neural network to predict memory accesses for each instruction. Then, it determines non-aliasing relationships based on the prediction by following the approach introduced in Section 3. Combining the results of the conventional value-set analysis with this non-aliasing analysis, our `DEEPVSA` finally performs backward taint analysis and thus pinpoints the root cause of a program crash. In this work, we ran all the aforementioned systems on a 32-bit Linux system with Linux kernel 4.4.0 running on an Intel i7-6600 quad-core processor with 16 GB RAM. We trained all the deep neural networks in this work on 2 Nvidia Tesla K40 GPUs and 4 Nvidia GTX 1080Ti GPUs using the `Keras` package [14] and with `Tensorflow` [1] as backend, amounting to about 2,000 lines of `Python` code. Upon the acceptance of this submission, we will release all of our systems along with our data set described below.

4.2 Data Set

As is mentioned above, we need to train many deep neural networks with various execution traces along with their corresponding memory accesses. In this work, we construct our training data set by using 78 unique programs in a package of GNU software – `coreutils`, `ineutils` and `binutils`. More specifically, we ran these programs by following their documentation and running examples. Using the aforementioned tracing system, we then gathered their execution traces along with their memory accesses. In total, these 78 programs generate a training data set with 96 distinct execution traces covering 49,193,919 lines of instructions.

To test our neural network and demonstrate the effective-

Index	Program	Non-alias (400)		Non-alias (800)		Non-alias (3200)		Non-alias (6400)		Non-alias (12800)		Root Cause	
		VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA
1	coreutils-8.4	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	✓	✓
2	coreutils-8.4	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	✓	✓
3	coreutils-8.4	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	✓	✓
4	nginx-1.4.0	68%	99%	68%	99%	68%	99%	68%	99%	68%	99%	✓	✓
5	nullhttpd-0.5.0	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	✓	✓
6	DXFScope-0.2	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	✓	✓
7	tiff-3.8.2	0.58%	30.50%	0.58%	30.51%	0.58%	30.51%	0.58%	30.51%	0.58%	30.51%	✓	✓
8	unrtf-0.19.3	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	✓	✓
9	gdb-6.6	23.03%	84.64%	23.34%	84.70%	41.13%	91.83%	41.13%	91.83%	41.13%	91.83%	✓	✓
10	openjpeg-2.1.1	11.93%	14.38%	11.93%	14.38%	11.93%	14.38%	92.67%	95.13%	92.67%	95.13%	✓	✓
11	python-2.7	65.00%	91.26%	71.14%	97.49%	71.14%	97.49%	71.14%	97.49%	71.14%	97.49%	✓	✓
12	poppler-0.8.4	0.00%	39.43%	0.00%	39.43%	0.00%	39.43%	60.60%	98.56%	60.60%	98.56%	✓	✓
13	htmldoc-1.8.27	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	✓	✓
14	unalz-0.52	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	✓	✓
15	psutils-p17	0.17%	45.20%	0.17%	45.20%	48.84%	90.44%	48.84%	90.44%	48.84%	90.44%	✗	✓
16	libpng-1.2.5	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	✓	✓
17	gas-2.12	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	✓	✓
18	SQLite-3.8.6	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	✗	✓
19	pcal-4.7.1	22.74%	85.42%	22.74%	85.42%	22.75%	85.43%	22.75%	85.43%	22.75%	85.43%	✗	✓
20	LaTeX2rtf-1.9	9.93%	10.55%	9.93%	10.55%	19.68%	30.16%	19.68%	30.16%	19.68%	30.16%	✗	✓
21	gif2png-2.5.2	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	✗	✓
22	abc2mtex-1.6.1	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	✓	✓
23	O3read-0.0.3	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	✗	✓
24	gdb-7.5.1	0.02%	55.30%	0.02%	55.30%	42.09%	93.56%	42.09%	93.56%	42.09%	93.56%	✗	✓
25	podofo-0.9.4	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	✓	✓
26	nasm-0.98.38	0.35%	44.78%	0.35%	44.78%	0.35%	44.78%	57.34%	99.24%	57.34%	99.24%	✓	✓
27	corehttp-0.5.3a	0.00%	40.98%	0.00%	40.98%	0.00%	40.98%	58.48%	94.40%	58.48%	94.40%	✓	✓
28	corehttp-0.5.3.1	0.00%	41.21%	0.00%	41.21%	0.00%	41.21%	58.08%	95.22%	58.08%	95.22%	✓	✓
29	unrar-3.9.3	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	✗	✓
30	prozilla-1.3.6	4.98%	56.53%	4.98%	56.53%	4.98%	56.53%	32.06%	77.97%	32.06%	77.97%	✗	✓
31	python-2.7.5	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	✓	✓
32	html2hdml-1.0.3	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	✓	✓
33	mcrypt-2.5.8	14.95%	53.02%	22.83%	59.84%	63.36%	100%	63.36%	100%	63.36%	100%	✗	✓
34	putty-0.66	5.06%	24.67%	5.06%	24.67%	5.06%	24.67%	18.58%	54.09%	18.58%	54.09%	✓	✓
35	mp3info-0.8.5a	1.9%	55.58%	1.9%	55.58%	3.82%	55.92%	3.82%	55.92%	3.82%	55.92%	✓	✓
36	LibSMI-0.4.8	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	✓	✓
37	JPegToAvi-1.5	0.00%	55.20%	0.00%	55.20%	0.00%	55.20%	0.00%	55.20%	13.81%	67.81%	✓	✓
38	aireplay-ng-1.2	4.46%	50.80%	4.96%	51.30%	49.17%	88.57%	49.17%	88.57%	49.17%	88.57%	✗	✗
39	ClamAV-0.93.3	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✗	✗
40	0verkill-0.16	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✗	✗
Total	-	-	-	-	-	-	-	-	-	-	-	27(✓)	37(✓)
Average	-	21.23%	57.49%	21.62%	57.84%	27.01%	62.79%	36.37%	72.07%	36.73%	72.40%	-	-

Table 2: The list of program crashes corresponding to memory corruption vulnerabilities. "Root cause" specifies whether the result of alias analysis successfully facilitate the root cause identification of software crashes. The percentages under VSA and DEEPVSA represent the amount of non-alias memory pairs identified. The number shown along with "non-alias" indicates the length of the trace prior to the site of the root cause instruction.

ness of DEEPVSA in alias analysis and root cause diagnosis, we exhaustively searched the Exploit Database Archive [47] and randomly selected 40 distinct vulnerability reports corresponding to 38 unique versions of software running on Linux. Following the description of each report, we compiled vulnerable programs[5], configured the underlying systems and ran the PoC programs tied to corresponding vulnerabilities. In this way, we triggered software failures, recorded their crashing traces and treated these traces as our testing data set. Using these crashing traces, we benchmarked DEEPVSA and examined the effectiveness of our proposed technique. Recall that the execution trace is stored in a circular buffer with a limited size (4KB) and that buffer is shared by multiple running processes. Since different lengths of an instruction trace stored in that shared buffer might influence memory alias identification, we retained different lengths of instructions for each of our test cases. This gives us the ability to identify the optimal memory size needed for a running process.

In Table 2, we present all the crashing programs selected[6]. From the table, we have the following observations. First of all, we can observe that the programs listed in the table has less overlaps with the programs in our training data set. This implies the dissimilarity between our training and testing data sets and thus avoids the possibility of using the same or similar data for model training and testing. Considering programs could invoke functions in the same shared library (e.g., `glibc`), and too many of such invocations could potentially

[5] In other binary analysis research works using deep learning, the binary is typically compiled with various optimization options. In this work, we compiled programs mostly with O2 option because many vulnerabilities cannot be reproduced if compiled with other options. Note that this does not influence the generalization of our approach because O2 is the default compilation options for most software.

[6] Note that we present the corresponding CVE/EDB-IDs as well as the length of each crashing trace in Appendix.

		Global	Heap	Stack	Other
Precision	HMM	67.99%	53.99%	74.47%	86.56%
	CRF	15.93%	12.31%	62.10%	71.82%
	Bi-RNN	98.63%	72.74%	95.30%	97.39%
	Bi-GRU	90.71%	78.44%	95.11%	98.40%
	Bi-LSTM	89.75%	78.47%	94.98%	97.92%
	Our Model	**96.98%**	**94.62%**	**98.92%**	**99.32%**
Recall	HMM	77.52%	39.31%	81.40%	85.28%
	CRF	10.23%	17.85%	51.95%	88.71%
	Bi-RNN	84.17%	83.72%	95.96%	95.43%
	Bi-GRU	88.04%	87.46%	97.59%	95.84%
	Bi-LSTM	91.71%	86.53%	97.16%	95.37%
	Our Model	**86.67%**	**95.99%**	**98.59%**	**99.45%**
F1 Score	HMM	72.44%	45.50%	77.78%	85.92%
	CRF	12.46%	14.57%	56.57%	79.38%
	Bi-RNN	90.83%	77.85%	95.63%	96.40%
	Bi-GRU	89.35%	82.71%	96.33%	97.10%
	Bi-LSTM	90.72%	82.30%	96.06%	96.63%
	Our Model	**91.54%**	**95.30%**	**98.75%**	**99.39%**

Table 3: The overall performance of different machine learning models.

introduce the risk of using the same data for training and testing, we further examine the instruction traces in the testing data set with those in the training. We discover that there are 14.02% of overlapping functions, appearing both in our test cases and the cases in our training set. In order to ensure our training and testing data sets do not share instructions, we eliminate the commonly shared instruction sequences from the training data set. This further avoids the situation where we perform alias analysis against a target crashing trace by using the model trained with itself.

Second, we can observe, the programs in the table cover a wide spectrum, ranging from sophisticated software like `gdb-7.5.1` with over `1.6M` lines of code to lightweight software such as `o3read-0.0.3` and `corehttp-0.5.3.1` with less than `1K` lines of code. To some extent, this diversity of our test cases imposes different levels of difficulty upon alias analysis and root cause diagnosis. Last but not least, we manually examine the memory access behaviors and observe that our test corpus encloses a variety of memory access behaviors, manifested as different amounts of memory dereferences across four disjoint memory regions (see Table 4 in Appendix). It should be noted that apart from the three memory regions that conventional VSA typically separates, we introduce '*other*' which represents the memory region pertaining to the text and global sections tied to dynamic libraries. This is an useful addition because the involvement of this region could allow us to extend conventional VSA to consider the following two memory access practices. ❶ An instruction dereferences a memory cell which held a piece of read-only data in the text section. ❷ A running process and dynamic library do not share the same global section and an instruction of the process accesses a memory cell indicating the global section of the dynamic library.

4.3 Experimental Setup

Using the systems mentioned in Section 4.1 as well as the data sets described in Section 4.2, we set up a series of experiments to evaluate our proposed technique and thus answer the four questions presented above.

To answer the first three questions (❶, ❷ and ❸) mentioned at the beginning of this section, we first trained 6 different machine learning models by using the training data set mentioned above. As is specified in Table 3, two of them are conventional machine learning models – Hidden Markov Model (HMM) as well as Conditional Random Field (CRF). While there are other machine learning approaches, such as decision tree or logistic regression, which might also work for our task, we select HMM and CRF as our baseline approaches and compare them with our proposed deep learning technique. This is because, by design, the approaches of our choice could take a sequence of input and yield a sequence of predictions, whereas other traditional machine learning approaches need to involve sophisticated feature engineering efforts in order to process a sequence of data input. In addition to conventional learning models, Table 3 depicts our proposed neural network architecture that takes instruction embedding as the input to a neural network as well as three aforementioned neural architectures that take as input the raw machine code. In this work, we compare the performance of these different neural architectures and examine whether the design of feeding instructions to a neural network outperforms that of taking raw machine code[7].

To obtain the performance measure of each machine learning models mentioned above, we applied the learning models to the aforementioned testing data set, used them to predict the memory region each instruction refers to and compare their prediction with the true labels (i.e., the memory regions a corresponding instruction truly refers to). For each memory access in the execution traces of the testing data set, we define a prediction as a correct identification if and only if the predicted memory regions aligns the true memory regions that the corresponding instruction refers to. With this definition, we further computed the precision, recall and F1 score for each machine learning model. To be more specific, we use the equations $\frac{P_M \bigcap T_M}{P_M}$, $\frac{P_M \bigcap T_M}{T_M}$ and $2 \cdot \frac{precision \cdot recall}{precision + recall}$ to compute precision, recall and F1 score, respectively. Here, P_M represents the set of memory accesses predicted to refer to memory region M where $M \in \{stack, heap, global, other\}$. T_M denotes the set of memory accesses truly referencing memory region M.

To explore the answer to our last question (❹), we further set up our experiment as follows. For each trace in our testing data set, we first applied our proposed neural network model to predict the memory regions tied to corresponding

[7]It should be noted that all the neural networks shown in the table are bi-directional. This is because previous research [48] indicates the bi-directional structure outperforms those designed with a single-directional chain particularly when using deep learning to performing binary analysis.

instructions. With these prediction results, we then utilized DEEPVSA. As is mentioned above, DEEPVSA is an extension of VSA. It is built with the additional ability to take the region prediction and determine non-alias relationships that the conventional VSA originally fails to identify. In addition, it leverages the results of alias analysis to perform backward taint analysis and thus pinpoint the root cause of the corresponding crash. Using these capabilities, our experiment compares the non-alias pairs that DEEPVSA and conventional VSA identified. Then, using the alias analysis results that DEEPVSA and conventional VSA derive, our experiment further examines their corresponding capability in facilitating the root cause diagnosis. When conducting our experiments, we also investigate the impact of the instruction trace length upon the non-alias identification. To be specific, we preserve different lengths of instructions prior to the root cause site (*i.e.,* 200, 400, 800, 1600, 3200, 6400, 12800 and 19600) and measure how different lengths impact alias identification. It should be noted we utilize 4KB of execution trace for our study if hardware cannot enclose the root cause site in its circular buffer.

4.4 Experimental Results

Performance of machine learning models. Table 3 shows the precision, recall and F1 score of various machine learning models, which demonstrate their capability of assigning correct memory regions to instructions. As we can easily observe, all deep neural network models significantly outperform traditional machine learning models. This is because a crashing trace is relatively long and deep learning approaches naturally have stronger capability than HMM and CRF in learning the patterns hidden in a long sequence. Of all the neural network models, we can also observe that our proposed neural network model (specified as 'our model') exhibits the highest classification performance (*i.e.,* with the highest F1 score). This indicates that, in comparison with the model taking as input the raw machine code, a learning model that takes instruction embedding as the input to a neural network could better capture the dependency hidden between instructions.

From Table 3, we also find that, in comparison with other deep learning models, our model typically demonstrates the performance improvement with only about $1\% \sim 12\%$. However, this does not imply that the utility of our model is only slightly better than those of other neural network models. In our binary analysis task, the crashing traces are relatively long. Using a neural network with even only 0.1% of improvement in precision, for example, we could reduce the amount of false positives or negatives by thousands. Given a long crashing trace containing hundreds of thousands of instructions, our performance improvement indicates a significant reduction in the memory regions mistakenly assigned by neural networks.

Performance of memory alias analysis. In addition to showing the superior performance of our model when conducting

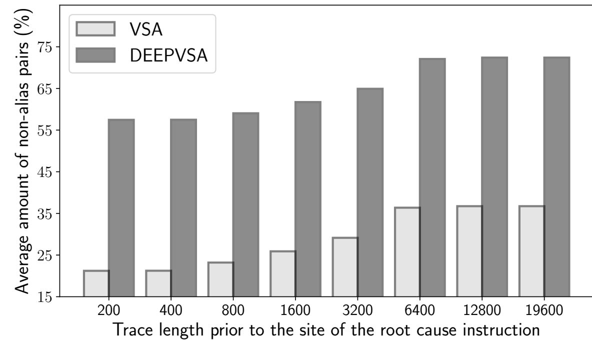


Figure 4: The average amount of non-alias memory pairs vs. the length of instructions retained.

memory region identification, we demonstrate the performance of our model in terms of its ability to facilitate VSA with respect to memory alias analysis. In Table 2, we specify the percentage of non-alias pairs that VSA and DEEPVSA track down when given different lengths of crashing traces (400, 800, 3200, 6400, 12800). As we can observe, on average, conventional VSA tracks down about $21.23\% \sim 36.73\%$ non-alias pairs compared with $57.49\% \sim 72.40\%$ of non-alias pairs identified by DEEPVSA. This is more than a 35% increase in non-alias memory reference determination. These results perfectly reflect how conventional VSA generally fails to accurately identify memory regions when execution traces are incomplete. With the assistance of a deep neural network, VSA's ability to perform memory region identification can be enhanced resulting in a significant benefit for memory alias analysis.

In Table 2 and Figure 4, we further specify the impact of the execution trace length upon the ability to perform alias analysis. We observe that, for some crashing programs (*e.g.,* `poppler-0.8.4` and `JPegToAvi-1.5`), the length of the execution trace stored in the circular buffer influences the capability of VSA and DEEPVSA upon determining memory alias relationships. With the increase in the length of an execution trace, we discover that both VSA and DEEPVSA demonstrate the improvement in their ability to analyze memory alias. This is because both techniques rely upon an execution context to perform alias analysis and a longer execution trace provides them with more abundant contexts. In addition, we observe that the capability of performing alias analysis converges when the length of the instructions (prior to the root cause instruction site) exceeds 12,800. This indicates that, even though DEEPVSA significantly improves VSA's capabilities for alias analysis, it does not completely address alias identification issues for a crashing trace. We believe there is still a room for future exploration in this space, particularly because VSA utilizes both memory regions and offsets to perform alias analysis while DEEPVSA only simply extends VSA with the consideration of coarse-grained memory region differences.

Recall that our DEEPVSA performs alias analysis by using a deep learning approach which cannot predict a memory region access with 100% of accuracy. As a result, along with the influence of a trace length upon alias analysis, we also investigate if inaccurate prediction actually causes DEEPVSA to incorrectly – or mistakenly – track down a non-memory alias pair and thus fail root cause diagnosis. We discover that, similar to conventional VSA, DEEPVSA exhibits zero error rate across all test cases shown in the table. This implies that DEEPVSA does not introduce unsoundness to alias analysis while our proposed deep neural network might mistakenly assign an incorrect region to an instruction. We believe the reason behind this surprising observation is as follows. Given a crashing trace, there is only a tiny portion of memory references that are truly aliased to each other. Even though our deep learning model mistakenly predicts regions for instructions, and DEEPVSA takes that inaccurate prediction as a strong indicator for determining non-alias relationships, the possibility of propagating that error to alias analysis is still extremely low.

Performance of root cause diagnosis. Going beyond specifying the facilitation of alias analysis, Table 2 also illustrates how the analysis of memory alias benefits backward taint analysis and thus the root cause identification. As we can observe from the table, compared with VSA – with which backward taint could successfully pinpoint the root cause of the crash for 27 test cases – DEEPVSA demonstrates superior performance in facilitating root cause diagnosis. We can observe that, with only 3 test cases, DEEPVSA fails to help backward taint to track down the root cause of a software crash. To understand the reasons behind the failure, we look closely into the instructions tainted. With respect to Overkill-0.16 and ClamAV-0.93.3, we note that the failure results from the nature of the hardware which has only 4KB memory storage to record all execution traces. Even if we allocate this entire storage to the crashing process, the hardware is still not able to enclose the instructions pertaining to the root cause of the crash. Regarding the test case aireplay-ng-1.2beta3, we discover the crashing program invoked the system call sys_read which writes a data chunk to a certain memory region. Since both the size of the data chunk and the address of the memory are specified in registers, which value-set analysis fails to restore, sys_read intervenes the propagation of data flow, making the output of DEEPVSA less informative to failure diagnosis.

5 Related Work

This research work mainly focuses on analyzing memory alias in the binary level. Regarding the techniques we employed and the problems we addressed, the lines of works most closely related to our own include machine learning in binary analysis and memory alias analysis for assembly. In this section, we summarize previous studies and discuss their limitation in turn.

Memory alias analysis for assembly. There is a long history of research about analyzing memory alias in binary code. As pioneering research works, Debray *et al.* [21] and Cifuentes *et al.* [16] both propose the same type of technical approaches that compute the values a set of registers can hold at each program point and then use the values held in the registers to determine alias. Considering such techniques determine only the possible values held in each register, but not reason about values across memory operations, Brumley *et al.* propose a logic-based approach which derives all possible alias relationships by finding an over-approximation of the set of values that each memory location and register can hold at each program point [12]. At the high level, this logic-based approach is similar to value set analysis [7, 6, 42] because they both perform value reasoning across memory operations. However, different from the work proposed in [12], value set analysis neither assumes that all memory cells and register locations must be of a single fixed width, nor assumes reads and writes have to be no overlapping. As such, value set analysis is more practical for real-world applications, whereas the logic-based approach [12] has been tested only against simple toy examples.

In a recent research work [19], Cui *et al.* propose a practical debugging system REPT. Technically, it first ignores memory alias in data flow analysis and then utilizes an error correction mechanism to rectify the mistakes caused by memory alias. This approach has demonstrated its effectiveness and efficiency in dealing with some real world crashes. However, as is stated in [19], it inevitably introduces inaccurate analysis results. This is because the proposed correction mechanism does not always catch the occurrence of memory alias, which could sometimes result in incorrectness in root cause diagnosis for a crashing program. In addition, similar to value set analysis, incomplete execution trace imposes the difficulty for REPT in performing alias analysis. In this work, we proposed new deep-learning-based approach which not only inherits the capability of VSA in providing high-fidelity analysis results but more importantly enhances its ability to analyze memory alias.

Machine learning in binary analysis. There is an extensive body of work leveraging machine learning to perform binary analysis. Technically speaking, they can be categorized into two types – conventional machine learning based approaches as well as deep learning based ones.

With respect to the works using conventional machine learning techniques, their research focus is mainly on identifying the function boundary in the binary level. For example, Rosenblum *et al.* utilize conditional random fields to formulate function boundary identification [43] and demonstrate decent performance in terms of pinpointing function entry points. In a recent research work, Bao *et al.* propose ByteWeight [8] which significantly improves the performance for function boundary identification by using weighted

prefix trees.

Regarding the research works adopting deep learning techniques, their research focus includes identifying function boundary [48], pinpointing function type signature [15], tracking down similar binary code [56] and performing memory forensics [49]. Using a bi-directional recurrent neural network, Shin *et al.* improve function boundary identification and achieve a nearly perfect performance with respect to function boundary recognition [48]. Going beyond simply identifying function boundary, Chua *et al.* explore recurrent neural networks with respect to its ability to track down the arguments and types of functions in binary [15]. In recent work, deep learning techniques have also been utilized for binary code similarity detection, in which Xu *et al.* employ Multi-Layer Perception (`MLP`) to encode a control flow graph and then use the encoding to pinpoint vulnerable code fragments [56]. Last but not least, Song *et al.* use a graph based deep learning approach to derive abstract representations for kernel objects so that one could recognize those objects from raw memory dumps efficiently [49].

In this work, we also use machine learning for binary analysis. Different from the aforementioned research, we however focus on leveraging deep learning to improve memory alias identification. Technically speaking, our work is also unique. Unlike the works above, which mostly use an off-the-shelf deep neural architecture, our work introduces a new recurrent neural architecture, which takes the consideration of the data dependency residing in binary code. As is shown in Section 4, our proposed neural network significantly outperforms neural networks largely adopted in other binary analysis tasks.

6 Conclusion

In this paper, we introduce a new deep neural network architecture to facilitate value-set analysis for alias analysis and thus improve the capability in software crash analysis. We show that this new neural architecture can significantly improve value-set analysis with respect to its capability in handling memory alias analysis and benefit data flow analysis in the context of postmortem program analysis. Since the design of our proposed neural network architecture takes into consideration not only the semantics of instructions but also their contexts, it can better capture the dependency within and between the instructions in a sequence of machine codes, making alias identification more effective.

We implemented our proposed technique as `DEEPVSA`– a deep neural network assisted tool for alias analysis and crash diagnosis – and demonstrated its utility using real-world software crashes covering about 1.6 million lines of instructions. We showed that `DEEPVSA` can facilitate the determination of non-alias relationships with no false positives and benefit the diagnosis of program crashes. In addition, we demonstrated that our newly designed neural network outperforms off-the-shelf neural architectures. Following these findings, we safely conclude deep learning can be used for the facilitation of memory alias analysis and root cause diagnosis at the binary level. We expect this work can inspire further advancements in alias analysis and postmortem program analysis through deep neural networks.

Acknowledgement

We would like to thank our shepherd Konrad Rieck and the anonymous reviewers for their helpful feedback. This project was supported in part by NSF grants CNS-1718459, TWC-1409915. In addition, this work was partially supported by the CLTC (Center for Long-Term Cybersecurity), and FORCES (Foundations of Resilient CybErPhysical Systems) which is supported by NSF under the grants CNS-1238959, CNS-1238962, CSN-1239054 and CSN-1239166.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 11st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[2] ARM. Embedded trace macrocell architecture specification. `http://www2.lauterbach.com/pdf/trace_arm_etm.pdf`, 2018.

[3] AUTHORS, A. The deepvsa project website. Anonymous link, 2019.

[4] BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. Codesurfer/x86a platform for analyzing x86 executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC)* (2005).

[5] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *Proceedings of the 13rd International Conference on Compiler Construction (CC)* (2004).

[6] BALAKRISHNAN, G., AND REPS, T. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems* (2010).

[7] BALAKRISHNAN, G., AND REPS, T. W. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC)* (2004).

[8] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)* (2014).

[9] BENGIO, S., AND HEIGOLD, G. Word embeddings for speech recognition. In *Proceedings of the 15th Annual Conference of the International Speech Communication Association (ISCA)* (2014).

[10] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] BOTTOU, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 15th International Conference on Computational Statistics (COMPSTAT)* (2010).

[12] BRUMLEY, D., AND NEWSOME, J. Alias analysis for assembly. In *CMU-CS-06-180* (2006).

[13] CHO, K., VAN MERRIËNBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[14] CHOLLET, F., ET AL. Keras. `https://keras.io/`, 2015.

[15] CHUA, Z. L., SHEN, S., SAXENA, P., AND LIANG, Z. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)* (2017).

[16] CIFUENTES, C., AND FRABOULET, A. Intraprocedural static slicing of binary executables. In *Proceedings of 13rd the International Conference on Software Maintenance (ICSM)* (1997).

[17] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)* (2007).

[18] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security)* (1998).

[19] CUI, W., GE, X., KASIKCI, B., NIU, B., SHARMA, U., WANG, R., AND YUN, I. REPT: Reverse debugging of failures in deployed software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[20] CUI, W., PEINADO, M., CHA, S. K., FRATANTONIO, Y., AND KEMERLIS, V. P. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (2016).

[21] DEBRAY, S., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (1998).

[22] GERS, F. A., SCHRAUDOLPH, N. N., AND SCHMIDHUBER, J. Learning precise timing with lstm recurrent networks. *Journal of machine learning research* (2002).

[23] GOODFELLOW, I., BENGIO, Y., COURVILLE, A., AND BENGIO, Y. *Deep learning*. MIT press Cambridge, 2016.

[24] GRAVES, A., MOHAMED, A.-R., AND HINTON, G. Speech recognition with deep recurrent neural networks. In *Proceedings of the 38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2013).

[25] GU, X., ZHANG, H., ZHANG, D., AND KIM, S. Deep api learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2016).

[26] HOCHREITER, S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* (1998).

[27] INTEL. Intel processor trace tools. `https://software.intel.com/en-us/node/721535`, 2013.

[28] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International conference on machine learning (ICML)* (2015).

[29] ITO, Y. Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. *Neural Networks* (1991).

[30] KASIKCI, B., CUI, W., GE, X., AND NIU, B. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[31] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representation (ICLR)* (2015).

[32] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2012).

[33] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *nature* (2015).

[34] LI, M., ZHANG, T., CHEN, Y., AND SMOLA, A. J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (2014).

[35] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., ET AL. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming language design and implementation (PLDI)* (2005).

[36] MICROSOFT. /safeseh (safe exception handlers). `http://msdn2.microsoft.com/en-us/library/9a89h429.aspx`, 2003.

[37] MICROSOFT. Time travel debugging - record a trace. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-record`, 2017.

[38] MOZILLA. rr: lightweight recording & deterministic debugging. `https://rr-project.org/`, 2019.

[39] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 11st Network and Distributed System Security Symposium (NDSS)* (2005).

[40] OHMANN, P., BROOKS, A., D'ANTONI, L., AND LIBLIT, B. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[41] OHMANN, P., AND LIBLIT, B. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013).

[42] REPS, T. W., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *Proceedings of the 17th International Conference on Compiler Construction (CC)* (2008).

[43] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)* (2008).

[44] RUCK, D. W., ROGERS, S. K., KABRISKY, M., OXLEY, M. E., AND SUTER, B. W. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks* (1990).

[45] RUDER, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[46] SCHUSTER, M., AND PALIWAL, K. K. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* (1997).

[47] SECURITY, O. Offensive security exploit database archive. `https://www.exploit-db.com/`, 2009.

[48] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)* (2015).

[49] SONG, W., YIN, H., LIU, C., AND SONG, D. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018).

[50] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2014).

[51] TEAM, P. Address space layout randomization (aslr). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[52] TIELEMAN, T., AND HINTON, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* (2012).

[53] VAN DE VEN, A., AND MOLNAR, I. Exec shield. `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[54] WIKIPEDIA CONTRIBUTORS. Hyperbolic function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Hyperbolic_function&oldid=866654186`, 2018.

[55] XU, J., MU, D., XING, X., LIU, P., CHEN, P., AND MAO, B. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)* (2017).

[56] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017).

Index	CVE/EDB	Trace Len.	Statistics			
			Global	Heap	Stack	Other
1	2013-0221	228	6	14	106	4613
2	2013-0222	285	155	468	10895	844
3	2013-0223	341	27	74	5103	4301
4	2013-2028	348	24	2318	5268	715
5	2002-1496	136	141	3071	0	8723
6	2004-1271	3391	64	0	7031	3884
7	2009-2285	28387	77	23596	4287	1786
8	2004-1297	110	341	1641	6086	2326
9	NA-30142	419	357	2379	6523	1584
10	2016-7445	236	20	195	5544	3829
11	NA-38616	680	62	11332	639	11535
12	2008-2950	672	5	1632	7196	2195
13	2009-3050	704	151	1114	6572	1924
14	2005-3862	54203	15	4612	15543	7372
15	NA-890	2966	32	88	6369	4184
16	2004-0597	4107	18	365	8368	3818
17	2005-4807	15953	180	8584	4973	2514
18	2015-5895	1446	27	1642	6776	1840
19	2004-1289	13264	807	6503	7736	3233
20	2004-2167	1720	150	1492	1729	311
21	2009-5018	76603	75	175	26354	18975
22	2004-1257	56018	862	10192	31420	3333
23	2004-1288	69184	1189	0	24430	25256
24	NA-23523	1544	95	2833	7107	343
25	2017-5854	571	90	1382	10698	753
26	2004-1287	1212	660	5593	5948	332
27	2007-4060	4124	55	596	7438	2072
28	2009-3586	8612	80	1317	9854	2375
29	NA-17611	3384	2724	0	3407	367
30	2004-1120	2011	23	5060	7345	1655
31	NA-33251	16672	1	33168	474	25
32	2004-1275	41275	25	12039	10383	683
33	2012-4409	321	29	216	4769	2116
34	2016-2563	3586	1035	2349	8450	869
35	2006-2465	31806	10	4564	16304	5227
36	2010-2891	7611	0	715	14626	1764
37	2004-1279	29371	9	266	11924	10989
38	2014-8322	329	84	138	6908	2738
39	2008-5314	200000	0	0	118271	85077
40	2006-2971	200000	37208	0	4652	158140
Total	-	883830	-	-	-	-

Table 4: The detail of crashing programs. The CVE/EDB column specifies the vulnerability identifiers. In this column, NA indicates those vulnerabilities with an EDB identifier but not a CVE Identifier). "Trace Len" describes the number of instructions from the root cause site to the crashing site. The numbers under "statistics" indicate the amount of memory dereferences across 4 disjoint memory regions.

Appendix

Detail of crashing programs and their crashing trace. As is described in Section 4, for our evaluation, we select 40 crashing traces corresponding to 38 distinct versions of vulnerable software. Table 4 describes the detail of these selected programs, including the CVE/EDB identifiers tied to these programs as well as the length of their crashing traces. In addition, the table shows the memory access behaviors of each program. They are retrieved from the execution trace which combines the trace from the root cause site to the crashing site and its 19,200 prefix instructions. We have already made all of the selected programs publicly available. They can be downloaded from our project website [3]. It should be noted that Table 2 and 4 share the same index.

		Global	Heap	Stack	Other
Precision	HMM	65.69%	47.26%	71.38%	86.28%
	CRF	15.87%	33.57%	63.30%	73.09%
	Bi-RNN	95.53%	72.88%	94.18%	97.67%
	Bi-GRU	91.97%	74.21%	94.62%	97.44%
	Bi-LSTM	94.23%	77.12%	94.84%	98.57%
	Our Model	**98.30%**	**94.91%**	**98.83%**	**99.40%**
Recall	HMM	63.25%	29.82%	83.08%	83.72%
	CRF	8.17%	13.02%	55.56%	88.20%
	Bi-RNN	79.75%	83.56%	96.39%	95.07%
	Bi-GRU	89.89%	81.28%	96.93%	95.22%
	Bi-LSTM	88.79%	88.20%	97.56%	95.58%
	Our Model	**88.64%**	**95.71%**	**98.65%**	**99.53%**
F1 Score	HMM	64.49%	36.57%	76.79%	84.98%
	CRF	10.79%	18.76%	59.18%	79.94%
	Bi-RNN	86.93%	77.86%	95.27%	96.35%
	Bi-GRU	90.92%	77.59%	95.76%	96.31%
	Bi-LSTM	91.43%	82.29%	96.18%	97.06%
	Our Model	**93.22%**	**95.31%**	**98.74%**	**99.46%**

Table 5: The overall performance of different machine learning models trained with the execution traces without the elimination of common instruction sequences.

Learning model performance without the elimination of commonly-shared data. As is specified in Section 4, in order to avoid the risk of using the same data to train and test a learning model, we eliminate – from the training data set – the instruction sequences commonly shared by both our training and testing sets, and show the performance of the learning models trained on non-overlapping data set. As a comparison, we also conduct an experiment in which we do not eliminate the 14.02% of shared data from the training set, and train all the learning models over the overlapping data set. In Table 5, we depict the model performance under this setting. As we can observe from the table, the model performance is actually comparable regardless whether we trim off the commonly shared instruction sequences. This implies that the shared data has nearly no impact upon model classification and thus memory alias analysis.