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
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.

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, 50]), 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.

---

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

---

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.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>2</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>3</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>4</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>5</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>6</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>7</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>8</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>9</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>10</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>11</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>12</td>
<td>[esp]</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>13</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>14</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>15</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>16</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
<tr>
<td>17</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
<td>esp</td>
<td>(⊥, [0, 0])</td>
</tr>
</tbody>
</table>

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 softwares 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, 57]). 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, 49]). While they are less intrusive and informative for assisting software debugging, such a lightweight approach cannot be used to fully restore the

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

(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.
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 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 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. 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.

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, or 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 [0x8C8]. 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 illustrated in Table [16], 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] (⊥, [−0xC, -0xC], ∅). Here, [esp+0x8] indicates the name of the stack memory cell, and (⊥, [−0xC, -0xC], ∅) 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 [15], VSA specifies its value set as a 3-tuple (global $\rightarrow$ ⊥, stack $\rightarrow$ [−0x14, −0x14], heap $\rightarrow$ ⊥), for brevity (⊥, [−0x14, −0x14], ⊥). In this set, ⊥ is a symbol denoting the empty set of offsets (i.e., ∅). 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 [16] 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 aliases of each other. In addition, we can observe [eax]

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

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.
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 to their addresses, i.e., (⊥, [−0x18, 0x18], ⊥) for [esp] and [ebp+0x8]: (⊥, ⊥, [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 1a 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 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 1b. 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 con-
ventional 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 in-
structions, 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 one of the two designs shown in Figure 2.
As is depicted in the figure, the two designs take the input dif-
frently, 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 ⊥, ⊥, ⊥, [X, X] where [X, X] denotes an unknown address on
the heap. With this, VSA could further update the value sets
for corresponding a-loc. We show the updated value sets in Table 1 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 re-
search typically utilized three types of recurrent neural net-
works (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), . . . , 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
\( g(t) \):

\[
\begin{align*}
    h(t) &= g(t)(x(t), x(t-1), x(t-2), \ldots, x(2), x(1)), \\
    &= f(h(t-1), x(t); \theta).
\end{align*}
\]

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

\[
\begin{align*}
    h(3) &= f(h(2); \theta), \\
    &= f(f(h(1); \theta); \theta).
\end{align*}
\]

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.

\[
\begin{align*}
a(t) &= Wh(t-1) + Ux(t) + b, \\
h(t) &= \tanh(a(t)), \\
o(t) &= Vh(t) + c, \\
\hat{y}(t) &= \text{softmax}(o(t)).
\end{align*}
\]
Here, bias vectors $b$ and $c$ are parameters. $\tanh(W_{h}(t-1) + Ux(t) + b)$ is the detailed form of the recurrent neuron $f$ in which $\tanh$ is an activation function [44]. 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

$$L^{(t)} = L(x^{(1)}, x^{(2)}, \ldots, x^{(t)}, y^{(1)}, y^{(2)}, \ldots, y^{(t)})$$

$$= - \sum \log p_{\text{model}}(y^{(t)}|x^{(1)}, x^{(2)}, \ldots, x^{(t)}),$$

where $p_{\text{model}}(y^{(t)}|x^{(1)}, \ldots, x^{(t)})$ is the probability from the prediction vector $y^{(t)}$ 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

$$s^{(t)} = f^{(t)} \odot s^{(t-1)} + g^{(t)} \odot \sigma(W_{h}s^{(t-1)} + Ux^{(t)} + b),$$

$$h^{(t)} = q^{(t)} \odot \tanh(s^{(t)}).$$

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

$$g^{(t)} = \sigma(W_{g}s^{(t-1)} + U_{g}x^{(t)} + b_{g}),$$

$$f^{(t)} = \sigma(W_{f}s^{(t-1)} + U_{f}x^{(t)} + b_{f}),$$

$$q^{(t)} = \sigma(W_{q}s^{(t-1)} + U_{q}x^{(t)} + b_{q}),$$

where $\{b_{g}, b_{f}, b_{q}\}$, $\{U_{g}, U_{f}, U_{q}\}$ and $\{W_{g}, W_{f}, W_{q}\}$ are respectively: biases, input weights, and recurrent weights for the forget, external input, and output gates. Similar to $b$, $U$ and $W$, 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].
As described in previous research [13], 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 \( g \) 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:

\[
\begin{align*}
    r^{(t)} &= \sigma(W_r h^{(t-1)} + U_r x^{(t)} + b_r), \\
    u^{(t)} &= \sigma(W_u h^{(t-1)} + U_u x^{(t)} + b_u), \\
    \tilde{h}^{(t)} &= u^{(t)} \odot \tanh(W(r^{(t)} \odot h^{(t-1)}) + U x^{(t)} + b), \\
    h^{(t)} &= (1 - u^{(t)}) \odot h^{(t-1)} + u^{(t)} \odot \tilde{h}^{(t)}.
\end{align*}
\]

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

### 3.3 Our Neural Network Architecture

As 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][13]), 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.
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.

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

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

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

Here, $E_i$ and $E_i$ 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 $E_i = [E_i, E_i]$. 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

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

---

Note that we can also use GRU as an alternative to LSTM for the encoding network and the sequence to sequence network.
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 $(\top, \top, \top)$, 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 $([\mathbf{X}, \mathbf{Y}], \perp, \perp)$, $(\perp, [\mathbf{X}, \mathbf{Y}], \perp)$ or $(\perp, \perp, [\mathbf{X}, \mathbf{Y}])$, denoting a memory access in that instruction could refer to a particular memory area ranging from $\mathbf{X}$ to $\mathbf{Y}$ 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 $\mathbf{X}$ and $\mathbf{Y}$. 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 $(\perp, \perp, [\mathbf{X}, \mathbf{Y}])$ and VSA updates $\mathbf{X}$ and $\mathbf{Y}$ with $\mathbf{X}=\mathbf{Y}$ indicating the register eax refers to the memory address $\mathbf{X}$ 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. 1. Does our problem require a deep learning model or could it be resolved with conventional machine learning techniques? 2. Can our proposed technique correctly link memory regions to instructions or, more precisely, identify the memory regions that instructions dereference? 3. 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? 4. 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 [1] 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, inutils 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-
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\(^3\) 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\(^4\). 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

\(^3\)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.

\(^4\)Note that we present the corresponding CVE/EDB-IDs as well as the length of each crashing trace in Appendix.
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 memory regions that conventional VSA typically separates, while 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.

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 $\text{precision} = \frac{M \cap P}{M}$, $\text{recall} = \frac{M \cap P}{P}$ and $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$ to compute precision, recall and F1 score, respectively. Here, $PM$ represents the set of memory accesses predicted to refer to memory region $M$ where $M \in \{\text{stack, heap, global, other}\}$. $TM$ denotes the set of memory accesses truly referencing memory region $M$.

To explore the answer to our last question (3), 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

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 (1, 2 and 3) 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.

7It should be noted that all the neural networks shown in the table are bi-directional. This is because previous research 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% ∼ 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 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% ∼ 36.73% non-alias pairs compared with 57.49% ∼ 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 re-
ferences that are truly aliased to each other. Even though
our deep learning model mistakenly predicts regions for in-
structions, 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 speci-
fying the facilitation of alias analysis, Table 2 also illustrates
how the analysis of memory alias benefits backward taint anal-
ysis 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 perform-
ance 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 stor-
age 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 anal-
ysis fails to restore, sys_read intervenes the propagation of
data flow, making the output of DEEPVSA less informative to
fail root cause 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 his-
tory of research about analyzing memory alias in binary code.
As pioneering research works, Debray et al. [21] and Ci-
fuentes 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, Brum-
ley 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, 5, 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 non-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 prac-
tical debugging system REPT. Technically, it first ignores
memory alias in data flow analysis and then utilizes an er-
ror correction mechanism to rectify the mistakes caused by
memory alias. This approach has demonstrated its effective-
ness and efficiency in dealing with some real world crashes.
However, as is stated in [19], it inevitably introduces inaccu-
rate 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 iden-
tifying the function boundary in the binary level. For ex-
ample, Rosenblum et al. utilize conditional random fields to
formulate function boundary identification [43] and dem-
strate decent performance in terms of pinpointing function
entry points. In a recent research work, Bao et al. propose
ByteWeight [8] which significantly improves the perfor-
ance for function boundary identification by using weighted
prefix trees.

Regarding the research works adopting deep learning techniques, their research focus includes identifying function boundary [43], 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 [43]. 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, CNS-1239054 and CNS-1239166.

References


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.

<table>
<thead>
<tr>
<th>Index</th>
<th>CVE/EDB</th>
<th>Trace Len.</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2013-0221</td>
<td>228</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2013-0222</td>
<td>285</td>
<td>155</td>
</tr>
<tr>
<td>3</td>
<td>2013-0223</td>
<td>341</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>2013-2028</td>
<td>348</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>2002-1496</td>
<td>136</td>
<td>141</td>
</tr>
<tr>
<td>6</td>
<td>2004-1271</td>
<td>3391</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>2009-2288</td>
<td>28387</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>2004-1297</td>
<td>110</td>
<td>341</td>
</tr>
<tr>
<td>9</td>
<td>NA-30142</td>
<td>419</td>
<td>357</td>
</tr>
<tr>
<td>10</td>
<td>2016-7445</td>
<td>236</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>NA-35616</td>
<td>680</td>
<td>62</td>
</tr>
<tr>
<td>12</td>
<td>2008-2950</td>
<td>672</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>2009-3050</td>
<td>704</td>
<td>151</td>
</tr>
<tr>
<td>14</td>
<td>2005-3862</td>
<td>54203</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>NA-890</td>
<td>2966</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>2004-0597</td>
<td>4107</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>2005-4807</td>
<td>15953</td>
<td>180</td>
</tr>
<tr>
<td>18</td>
<td>2015-5895</td>
<td>1446</td>
<td>27</td>
</tr>
<tr>
<td>19</td>
<td>2004-1289</td>
<td>13264</td>
<td>807</td>
</tr>
<tr>
<td>20</td>
<td>2004-2167</td>
<td>1720</td>
<td>150</td>
</tr>
<tr>
<td>21</td>
<td>2009-5018</td>
<td>76603</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>2004-1257</td>
<td>56018</td>
<td>862</td>
</tr>
<tr>
<td>23</td>
<td>2004-1288</td>
<td>69184</td>
<td>1189</td>
</tr>
<tr>
<td>24</td>
<td>NA-23523</td>
<td>1544</td>
<td>95</td>
</tr>
<tr>
<td>25</td>
<td>2017-5854</td>
<td>571</td>
<td>90</td>
</tr>
<tr>
<td>26</td>
<td>2004-1287</td>
<td>1212</td>
<td>660</td>
</tr>
<tr>
<td>27</td>
<td>2007-4060</td>
<td>4124</td>
<td>55</td>
</tr>
<tr>
<td>28</td>
<td>2009-3586</td>
<td>8612</td>
<td>80</td>
</tr>
<tr>
<td>29</td>
<td>NA-17611</td>
<td>3384</td>
<td>2724</td>
</tr>
<tr>
<td>30</td>
<td>2004-1120</td>
<td>2011</td>
<td>23</td>
</tr>
<tr>
<td>31</td>
<td>NA-33251</td>
<td>16672</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>2004-1275</td>
<td>41275</td>
<td>25</td>
</tr>
<tr>
<td>33</td>
<td>2012-4409</td>
<td>321</td>
<td>29</td>
</tr>
<tr>
<td>34</td>
<td>2016-2563</td>
<td>3586</td>
<td>1035</td>
</tr>
<tr>
<td>35</td>
<td>2006-2465</td>
<td>31806</td>
<td>10</td>
</tr>
<tr>
<td>36</td>
<td>2010-2891</td>
<td>7611</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>2004-1279</td>
<td>29371</td>
<td>9</td>
</tr>
<tr>
<td>38</td>
<td>2014-8322</td>
<td>329</td>
<td>84</td>
</tr>
<tr>
<td>39</td>
<td>2008-5314</td>
<td>200000</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>2006-2971</td>
<td>200000</td>
<td>37208</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Precision</th>
<th>HMM</th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-RNN</td>
<td>95.59%</td>
<td>47.26%</td>
<td>71.38%</td>
<td>86.28%</td>
<td></td>
</tr>
<tr>
<td>Bi-GRU</td>
<td>91.97%</td>
<td>74.21%</td>
<td>94.62%</td>
<td>97.44%</td>
<td></td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>94.23%</td>
<td>77.12%</td>
<td>94.84%</td>
<td>98.57%</td>
<td></td>
</tr>
<tr>
<td>Our Model</td>
<td>98.30%</td>
<td>94.91%</td>
<td>98.83%</td>
<td>99.40%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recall</th>
<th>HMM</th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-RNN</td>
<td>97.75%</td>
<td>83.56%</td>
<td>96.39%</td>
<td>95.07%</td>
<td></td>
</tr>
<tr>
<td>Bi-GRU</td>
<td>89.89%</td>
<td>81.28%</td>
<td>96.93%</td>
<td>95.22%</td>
<td></td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>88.79%</td>
<td>88.20%</td>
<td>97.56%</td>
<td>95.58%</td>
<td></td>
</tr>
<tr>
<td>Our Model</td>
<td>98.64%</td>
<td>95.71%</td>
<td>98.65%</td>
<td>99.53%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1 Score</th>
<th>HMM</th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-RNN</td>
<td>86.93%</td>
<td>77.86%</td>
<td>95.27%</td>
<td>96.35%</td>
<td></td>
</tr>
<tr>
<td>Bi-GRU</td>
<td>90.92%</td>
<td>77.59%</td>
<td>95.76%</td>
<td>96.31%</td>
<td></td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>91.43%</td>
<td>82.29%</td>
<td>96.18%</td>
<td>97.06%</td>
<td></td>
</tr>
<tr>
<td>Our Model</td>
<td>93.22%</td>
<td>95.31%</td>
<td>98.74%</td>
<td>99.46%</td>
<td></td>
</tr>
</tbody>
</table>

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.

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.