RENN: Efficient Reverse Execution with Neural-network-assisted Alias Analysis

†‡Dongliang Mu, ‡Wenbo Guo, ‡Alejandro Cuevas, ‡Yueqi Chen, ‡Jinxuan Gai
‡Xinyu Xing, †Bing Mao, §Chengyu Song

†*National Key Laboratory for Novel Software Technology, Nanjing University, China*
‡*College of Information Sciences and Technology, The Pennsylvania State University, USA*
§*University of California, Riverside, USA*
{dzm77,wzg13}@ist.psu.edu, aledancuevas@psu.edu, {yxc431,jug273}@ist.psu.edu,
xxing@ist.psu.edu, maobing@nju.edu.cn, csong@cs.ucr.edu

***Abstract*—Reverse execution and coredump analysis have long been used to diagnose the root cause of software crashes. Each of these techniques, however, face inherent challenges, such as insufficient capability when handling memory aliases. Recent works have used hypothesis testing to address this drawback, albeit with high computational complexity, making them impractical for real world applications. To address this issue, we propose a new deep neural architecture, which could significantly improve memory alias resolution. At the high level, our approach employs a recurrent neural network (RNN) to learn the binary code pattern pertaining to memory accesses. It then infers the memory region accessed by memory references. Since memory references to different regions naturally indicate a non-alias relationship, our neural architecture can greatly reduce the burden of doing hypothesis testing to track down non-alias relation in binary code.**

Different from previous researches that have utilized 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 designed a new recurrent neural architecture that could capture the data dependency between machine code segments.

To demonstrate the utility of our deep neural architecture, we implement it as `RENN`, a neural network-assisted reverse execution system. We utilize this tool to analyze software crashes corresponding to `40` memory corruption vulnerabilities from the real world. Our experiments show that `RENN` can significantly improve the efficiency of locating the root cause for the crashes. Compared to a state-of-the-art technique, `RENN` has 36.25% faster execution time on average, detects an average of 21.35% more non-alias pairs, and successfully identified the root cause of 12.5% more cases.

***Index Terms*—Reverse Execution, Deep Learning, Memory Alias**

I. Introduction

Software defects are prevalent and inevitable. Developers are facing increasingly larger and more complex programs, while the releasing cycles are becoming shorter. When triggered, these defects can lead to runtime failures (*e.g.,* crashing or terminating abnormally) or compromise of the security and privacy of the end users. Therefore, it is important for developers to identify and fix the root cause of a runtime failure in a timely manner. Unfortunately, debugging a runtime failure, *i.e.,* understanding what statements are involved and how the bad values are propagated to the crash site is known to be difficult. [1, 2]

The most common way to pinpoint the underlying software defect of a runtime failure is core dump analysis, which looks for control- and data-flow information that could assist the analyst in piecing together the cause of the crash [1, 3, 4]. However, because core dump only captures the memory and register states at the crashing point, such information is usually insufficient to uncover the cause of the crash.

Another body of work focuses on recording the program's execution to have access to complete control- and data-flows [5–10]. Upon the crash, the analyst can use this log to replay the program's execution and study each statement carefully and reversely [11, 12]. This approach is intuitively more effective than core dump analysis as the data-flow is readily available to the developer upon the crash. However, recording a fully recoverable execution trace is too expensive for practical deployment.

To provide the best of both worlds, recent works like `POMP` [2] and REPT [13] leveraged Intel Processor Tracing (Intel-PT), a lightweight hardware tracing mechanism to record the control-flow before the point of core dump. With the help of control-flow trace, these systems showed it is possible to recover critical data-flow from a core dump to assist pinpointing the root cause of the crash. In particular, for instructions that are reversible, the state prior to the execution of such an instruction can be recovered by inverting the effects of the instruction. For instructions that are irreversible, like `xor eax, eax`, the state can usually be recovered through forward execution. The most challenging part is to handle memory accesses where the addresses are unknown. To address the memory aliasing problem (*i.e.,* whether two or more symbolic addresses actually refer to the same data), `POMP` conducts recursive hypothesis testing (HT). For each pair of memory accesses, two hypotheses are generated: one assuming they are aliased while the other assumes they are not aliased. Each hypothesis is then tested by emulating prior instructions. If the emulation result conflicts

The first two authors contributed equally to this paper.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00090

with the recorded ground truth (*i.e.,* control-flow trace and core dump), the corresponding hypothesis is rejected. Similarly, to address memory writes to unknown addresses, REPT first ignores the unknown memory write and then uses an error correcting scheme to correct these issues when the recovered states conflict with the ground truth.

As we can see, these conflict-based correction schemes are very expensive, which is the main performance bottleneck. Beyond efficiency, another fundamental limitation of these techniques is that they cannot deal with incomplete traces (*e.g.,* due to the limitation on the size of the trace). When the control-flow trace is incomplete, neither techniques are able to verify memory alias relations because they rely on prior instructions to do HT or solve access to unknown memory locations. Therefore, reverse execution can be heavily impaired (see Section II for details). To address these issues, we introduce a recurrent neural network (RNN) to enhance memory alias resolution during software failure diagnosis. Our observation is that the problem of accessing memory with unknown addresses can be modeled as a search problem—given a sequence of memory operations, find an assignment of aliasing mapping such that the emulation result will be consistent with the core dump. Note that as shown in [2], there could be multiple possible "correct" assignments that will be consistent with the core dump; however, in most cases, they will not affect the critical data-flow that caused the crash. Once we modeled the memory aliasing problem as a search problem, the reason why `POMP` is not efficient is obvious: it tries to brute force the search space with random guesses. Unfortunately, as the probability of two memory accesses being aliased is not uniformly random, the probability of being wrong with random guess is very high. As a result, the average computational complexity of `POMP`'s HT-based scheme is exponential.

The key to improving efficiency is to make "more educated guesses" thus increase the probability of guessing the right answer. As previous studies have shown that artificial neural networks are good at learning approximations to the true probability distributions of correct answers, we choose to explore the feasibility of training a neural network to predict the alias relation of two memory accesses. If the neural network can indeed "discover" some heuristics to improve the chance of guessing the right answers, we can achieve not only better performance (*i.e.,* less execution time) but also resolve alias relations and recover the program state significantly even when working with an incomplete trace.

We implemented the proposed technique as `RENN`, a neural network-assisted alias analysis tool for software failure diagnosis. We compare our technique with HT: we compare the execution time and the number of memory aliases resolved or missed by each approach. To construct our ground truth data set, we manually analyzed program crashes corresponding to 40 memory corruption vulnerabilities gathered from the Offensive Security Exploit Database Archive [14] and compared our manual analysis with the results obtained with `RENN` and HT. We observe that `RENN` can achieve 36.25% less execution time because deep learning could replace HT when identifying partial memory alias relations, saving substantial runtime. Furthermore, `RENN` can also accurately resolve 21.35% more unknown memory relationships than `POMP` when performing analyses on a crashed execution trace.

Contributions. In summary, this paper makes the following contributions:

- We discover that deep learning is effective in inferring memory alias relations.
- We propose a neural network architecture that could facilitate reverse debugging or reverse execution.
- We implement our deep learning technique as `RENN`– a tool that could help diagnose root causes of software crashes more effectively and efficiently.

The rest of the paper is organized as follows. Section II provides an overview of reverse execution and its fundamental limitations. Section III details the novel deep neural network we propose to improve memory alias analysis. Section V describes the implementation of our reverse execution system. Section V evaluates the utility of `RENN`. Section VI surveys related work. Finally, Section VII concludes this work.

II. Background and Motivations

In this section, we first use an example (Figure 1) to illustrate how `POMP`, a state-of-the-art reverse execution technique pinpoints the root cause of the crash. Then we discuss the fundamental limitations of `POMP` that motivate this work.

A. Reverse Execution

Reverse execution is a technique originally introduced in response to the large volume of information that was necessary to debug complex programs. This approach is based on the notion of reversible functions. That is, rather than recording all the executed instructions and their operands, one can leverage the reversibility of operations to restore previous program states from a single state snapshot (*e.g.,* core dump). For example, Figure 1 shows the partial core dump of a crashed program at $S15$ and the corresponding instruction trace before the crash. Based on `S15` we know that the crash happens because `eax@S15=0`, the execution of the `call eax` instruction will cause `eip=0`, and the processor cannot fetch instruction from address `0`. The last instruction `call eax` can be decoded into three steps: `sub esp,4`, `mov [esp],eip`, and `jmp eax`. Among these three steps, `sub esp,4` is reversible, so the previous `esp` can be recovered by adding 4 to it. That is, `esp@S14=esp@S15+4`. Similarly, `eax@S7=eax@S8-4`. Once the states prior to the crash are recovered, developers can then analyze the control/data flows that led to the crash. Because this approach only requires recording the final state and the instruction trace, with the help of hardware-accelerated tracing mechanisms like Intel-PT, this approach can be deployed in production systems [13]. As many crashes in real production systems are hard to reproduce, reverse execution is especially useful for developers to understand the root cause of such hard-to-reproduce crashes.

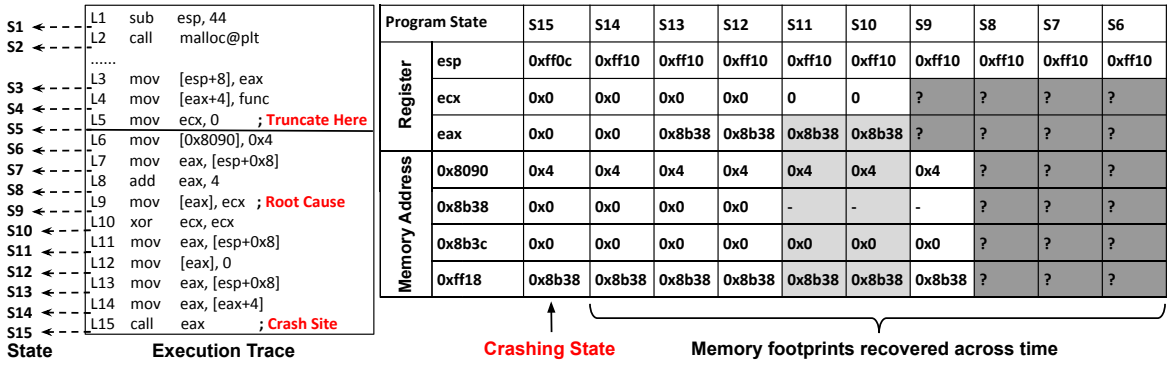


Fig. 1: Memory footprints reconstructed by reversely executing the trace. SN(*e.g.,* S15) is the execution state of the program after executing the corresponding instruction

Unfortunately, not all instructions can be reversed. For instance, `L10: xor ecx,ecx` is a common way to zero out the register. After executing this instruction, the new value is always 0 regardless of the previous value stored in the register, therefore the old value cannot be recovered using reverse execution. Thus, other pieces of information must be leveraged to recover the states. For instance, doing a forward analysis from `L5: mov ecx,0`, we can infer `ecx@S9=0`.

A more challenging problem for reverse execution is data-flow through memory. Unlike registers, whose aliasing relationships can be completely determined by the name of the register, accurate resolving memory aliasing requires address information. However, most memory addresses cannot be determined statically. Even with reverse execution, only a small portion of memory addresses can be recovered. As a result, recovering a previous memory state is very difficult. State-of-the-art systems like `POMP` [2] and REPT [13] mostly focus on solving this challenge. Take `POMP` as an example, quickly after starting the reverse execution from `L15`, we will encounter two memory read at `L14` and `L13`. Here, we want to know whether the read value is from an earlier write instruction, like `L12: mov [eax],0`. To resolve this, we would need to know whether `[esp+0x8]@L13` and `[eax]@L12` are aliasing. `POMP` uses Hypothesis Testing to resolve memory alias. Specifically, it first assumes `[eax]@L12` and `[esp+0x8]@L13` are aliased with each other, we could get the following constraint set over data objects before L14: `esp@S13+0x8=eax@S12`, `[eax@S12]=0`, and `eax@S13=[esp@S13+0x8]`. Then a forward simulation would reject this hypothesis because this will result in `eax@S14+4=4` and the program crash at L14. For the opposite hypothesis, we have another constraint set over data objects before L14: `esp@S13+0x8!=eax@S12`, `[eax@S12]=0`, and `eax@S13= [esp@S13+0x8]`. This hypothesis will not result in conflict so `POMP` will accept it. Then we could resolve program states in the light gray area of Figure 1.

B. Limitations

While `POMP` and REPT have made reverse execution much more practical, they also have limitations. The first limitation is efficiency: whenever a hypothesis testing fails, `POMP` has to backtrack and try the opposite hypothesis. As a result, the average computation complexity is exponential. The second and arguably more important limitation is their ability to handle incomplete execution traces. Specifically, although hardware-enabled execution tracing mechanisms like Intel PT [15] and ARM Embedded Trace Macrocell [16] allow software developers and security analysts to record execution trace with almost no performance overhead, keeping the whole trace for long- running programs still requires large disk storage. Therefore, similar to system logs, we can only keep the most recent N instructions. Such incomplete traces are challenging for existing techniques like `POMP` and REPT because they solely rely on conflicts to resolve memory issues. When the trace is incomplete, they may not be able to find the expected conflict, thus failed to locate the root cause of the crash.

Again, consider Figure 1 as an illustrating example. The key step to locate the root cause of this crash is to understand where `eax@S15` comes from, *i.e.,* which write instruction is aliasing to the read instruction at L14. Based on the address expression, one may be tempted to think `[eax]@L9` and `[eax+4]@L14` are aliased because they are all derived by loading from `[esp+0x8]` then add 4. However, this is only true if there is no write instruction that would be aliased `[esp+0x8]`. In the above subsection, we already showed how `POMP` uses hypothesis testing (HT) to resolve that `[eax]@S12` is not aliased with `[esp+0x8]@S13`. However, resolving whether `[eax]@S9` and `[esp+0x8]@S11` are aliased is more challenging. Note that since `esp` is not changed between L6 and L14, all `[esp+0x8]` are aliased. In particular, let us repeat the process of HT and get two constraint sets over data objects before S11:

1) `eax@S9=esp@S11+0x8`, `eax@S7=[esp@S7+0x8]`, `esp@S11=esp@S7`, `eax@S8=eax@S7+4`, and `[eax]@S9=ecx@S9`.
2) `eax@S9!=esp@S11+0x8`, `eax@S7=[esp@S7+0x8]`, `esp@S11=esp@S7`, `eax@S8=eax@S7+4`, and `[eax]@S9=ecx@S9`.

When the trace is complete, based on `L5: mov ecx,0`, we

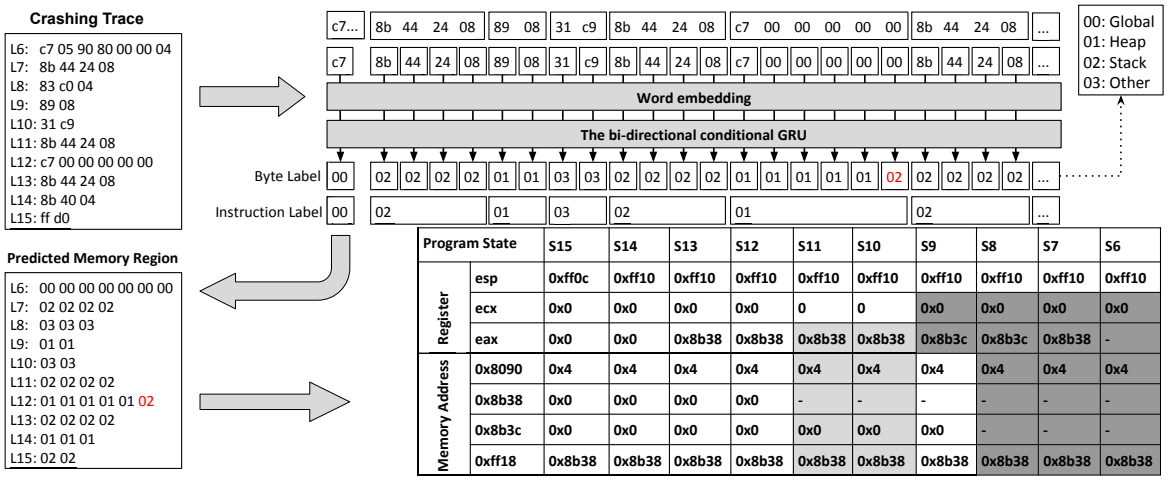


Fig. 2: Memory footprints reconstructed by reversely executing the trace with the help of deep learning.

can reject the first hypothesis because it will conflict with S11. However, in case of incomplete trace, due to `ecx@S9` is unknown, HT will not encounter any conflict when verifying both hypotheses. As a result, program states in the dark gray area of Figure 1 are kept as unknown. Finally, as POMP fails to verify this memory alias relation, it is impossible to locate the correct root cause from the crashing site.

From the above example, we can see that the lack of instructions from L1 to L5 could lead to the missing context of `ecx` register, which prevents `POMP` from resolving aliasing relationship between `[eax]@S11` and `[esp+0x8]@S9`, which eventually prevent locating the root cause. This shows that incomplete traces could greatly affect reverse execution.

III. Key Technique

To overcome the two limitations mentioned in Section II-B (*i.e.,* efficiency and incomplete trace), we need better alias analysis. Traditionally, alias analysis aims to determine whether two pointers could point to the same memory location. Unfortunately, in the existence of incomplete trace, even powerful alias analysis techniques like value-set analysis [17, 18] cannot successfully resolve the aliasing relationship of two memory accesses. In this work, we explore the feasibility of using deep learning techniques to solve this challenging problem. Specifically, we propose a novel sequence to sequence (seq2seq) recurrent neural network to predict *coarse-grained access regions*, namely, *global, heap, stack, and other* (*e.g.,* does not access memory). At a high-level, our model takes a sequence of instructions (in binary format) as inputs and outputs a sequence of labels indicating the memory region that the corresponding byte (instruction) may access. The inferred memory regions are then used to solve the memory alias problem – if two instructions access the same type of region, we assume they may be aliasing; otherwise, we assume they are non-aliasing.

The insight of using seq2seq RNN is that we want to infer a label for each byte (of an instruction) in the input and take into account the context information within the input binaries at the same time. Technically speaking, our model is different from the off-the-shelf RNN in that we relax their label independent assumption by adding an extra link. In the following, we first introduce off-the-shelf RNN architectures that have been used in binary analysis and point out their limitations. Next, we present the details of our designed network and specify how to use it to facilitate reverse execution. Last but not least, we provide more insights about why designing a deep neural network for identifying memory regions instead of other machine learning models.

A. Recurrent Neural Network

Past research in binary analysis has utilized three types of recurrent neural networks – vanilla recurrent neural network (RNN) [19], long short-term memory (LSTM) [20], and gated recurrent units (GRU) [21]. In the following, we give a brief overview of the Vanilla RNN and its variant LSTM and GRU.

Vanilla Recurrent Neural Network. A typical Vanilla RNN is composed with an input layer, a recurrent hidden layer and an output layer (Figure 3a). The input layer takes as input a sequence of values $x^{(1)}, \ldots, x^{(T)}$. At each time step t, the hidden layer outputs a representation $h^{(t)}$, which is computed based on the last hidden representation $h^{(t-1)}$ and the current input $x^{(t)}$:

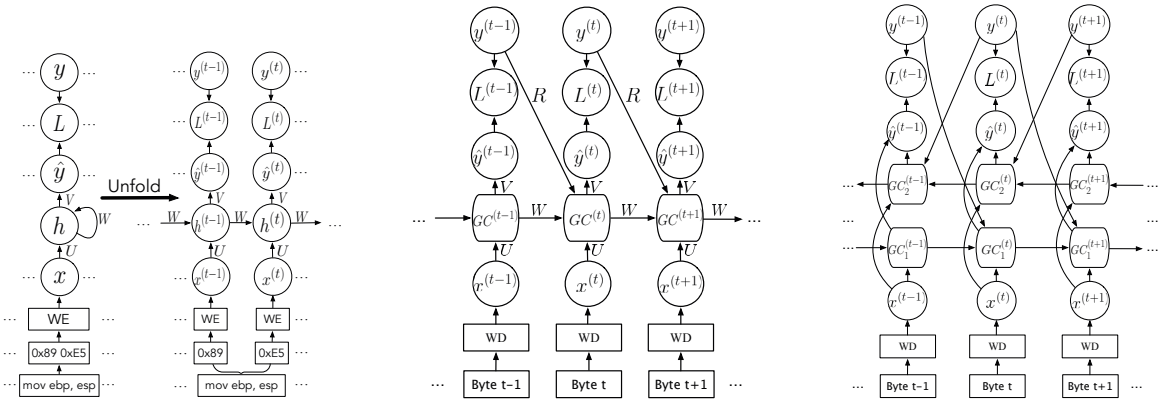
$$h^{(t)} = \text{act}(\mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b)$$

where act is the activation function [22] and the $\mathbf{W}$, $\mathbf{U}$ and b are model parameters. Then the output layer takes as input the $h^{(t)}$ and outputs a prediction of $x^{(t)}$ as $\hat{y}^{(t)}$:

$$\hat{y}^{(t)} = \text{softmax}(\mathbf{V}h^{(t)} + c)$$

where softmax refers to the softmax classifier [23] and $\mathbf{V}$, c are also model parameters.

These model parameters can be learned by minimizing a loss function, which compares the difference between $\hat{y}^{(1)}, \ldots, \hat{y}^{(T)}$ and the true label $y^{(1)}, \ldots, y^{(T)}$ (*e.g.,* root mean



(a) The ordinary RNN. (b) The conditional GRU. (c) The bi-directional conditional GRU.

Fig. 3: Recurrent neural networks with various architectures serving for different purposes. Note that "GC" indicates GRU cell and "WD" refers to word embedding.

square error [23] or cross-entropy [24]). The minimization of this loss function can be achieved by different first-order gradient based optimization algorithms (*e.g.,* stochastic gradient descent [25], ADAM [26]).

Long Short-Term Memory. Vanilla RNN suffers from gradient vanishing or explosions problem [27], which prevents it from processing a long sequence of input. In binary analysis, we usually deal with very long sequential inputs (*e.g.,* an input of 1000 bytes [28]). To avoid this defect, recent works adopted a Long Short-Term Memory (LSTM) [29] in binary analysis.

LSTM uses a memory cell to replace the hidden layer of the vanilla RNN. LSTM cell takes as input the last hidden state $s^{(t-1)}$ and current input $x^{(t)}$ as input and outputs the current hidden state $s^{(t)}$ and the current hidden output $h^{(t)}$. Within each cell, it introduces three gates to control the information flow. The insight behind designing these gates is that they can force LSTM network to forget some less important information and only memorize the key information for a longer time. These gates are functions of LSTM cell inputs (*i.e.,* $s^{(t-1)}$, $x^{(t)}$) and the LSTM outputs (*i.e.,* $s^{(t)}$, $h^{(t)}$) are functions of the cell inputs and these gates. More details about the formulation of these variables can be found in [30]. The parameters defined in the formulas can be learned with the aforementioned loss function and the optimization algorithms.

Gated Recurrent Units. GRU [31] also can be used to process long sequential inputs, but with a lower computational cost than LSTM. The reason is that GRU reduces the number of gates and variable in a cell. Recent researches [28, 32] demonstrate GRU and LSTM performs similarly when handling binary analysis tasks. The details about GRU can be also found in [30].

B. Our Proposed Technique

With the knowledge of existing recurrent neural networks in mind, we now specify the design of our proposed technique. We first describe how we design our neural network and discuss the intuition behind our design philosophy. Then, we specify the structures and the training strategy of our recurrent neural network model, followed by the description of how to facilitate reverse execution with the proposed neural network.

1) Recurrent Model Design: All of the aforementioned recurrent networks are naturally capable of learning patterns hidden behind a sequence. Therefore, we can expect that – as they have already demonstrated in other binary analysis tasks – the aforementioned recurrent networks can perform reasonably well in memory region identification. However, unlike previous work that simply using off-the-shelf models for their tasks (*e.g.,* learn function boundaries [28] and determine function arguments [32]), we design and develop a new recurrent neural network.

The intuition of our design is as follows. Recall that the goal of the classifier is to associate a region label with every instruction. Take for example the instruction `push [esp]` indicated by the byte sequence `0xff 0x34 0x24`. The label (*stack*) tied to byte `0x24` depends on the previous bytes `0xff 0x34`. The recurrent neural architectures described above (Section III-A) could learn data dependency hidden within this instruction. However, existing architectures assume that the labels tied to bytes are conditionally independent from each other given the sequence of bytes indicating the instructions. In this example, we observe that the memory region (label) attached to one byte is actually conditionally dependent upon its adjacent labels (*i.e.,* given sequence of bytes `0xff 0x34`, the label tied to byte `0x24` is dependent upon the label tied to its adjacent byte `0x34`). This 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 and within instructions but also dependencies between adjacent labels.

Inspired by this, we build a conditional GRU. The structure of this learning model is depicted in Figure 3b. As we can observe, we build this model under the structure of GRU with additional connections from the previous output to the current

GRU cell. This is in part because a byte sequence in a binary is relatively long and with GRU we could capture long-term dependency with relatively low computational cost, but largely because this recurrent structure removes the conditional independent assumption, making the model capable of capturing label relationships in addition to the sequence dependency. Comparing with the aforementioned off-the-shelf recurrent architectures largely adopted by previous binary analysis tasks, the conditional GRU better captures the relationships between (and within) instructions as well as that between nearby labels.

Similar to the previous research [28], we also enable our design with the capability of inferring memory regions in both forward and backward ways. We further upgrade our conditional GRU learning model to a bi-directional chain structure [33] as is shown in Figure 3c. As we can observe from the figure, the bi-directional chain structure combines a conditional GRU that moves forward, beginning from the start of the byte sequence, with another conditional GRU that moves backward, starting from the end of the sequence. In this structure, $GC_1^{(t)}$ stands for the GRU cell of the sub-GRU moving forward through time, and $GC_2^{(t)}$ represents the GRU cell of the sub-GRU moving backward through time. With these, we could perform a label prediction for a byte pertaining to an instruction based on the sequence of instructions executed before and after that instruction. Since the bi-directional structure enables connections from adjacent labels to GRU cells, we also augment our conditional GRU model with the ability to make label prediction based on the memory region that its adjacent bytes attach to.

2) Details of the Proposed Model: In the following, we describe how to use the proposed model for memory region identification and then how it can be used to enhance reverse execution in root cause diagnosis, including pre-processing a crash trace, performing computation within a designed network, training a network, and integrating network results into reverse execution.

Processing Traces. Given a crash trace of n bytes, we first cut it into m sequences, where each sequence has the length of k bytes and the last sequence will be padded with zeroes if its length is less than k. Then, we treat each sequence as the network input (*i.e.,* $(x^{(1)}, \ldots, x^{(T)})$), in which $x^{(t)}$ represents the t-th byte in this sequence. It should be noted that instead of directly inputting the hex value into the model, we use a word embedding mechanism to map each byte into a vector and take this vector as the input at t-th time(*i.e.,* $x^{(t)}$). Details of the word embedding process can be found in [28]. We attach a label for each byte indicating the memory region(s) (*i.e.,* [global: 0, heap: 1, stack: 2]) that the corresponding byte in the input sequence accesses. For example, if we have an input sequence with [`0x55`, `0x89`, `0x04`, `0x24`] (*i.e.,* `push ebp; mov [esp], eax`). We could infer that it accesses the stack region. Then we can label it as [2, 2, 2, 2]. Using the pre-processed input and label sequences, we can train the aforementioned sequence to sequence model with the following recurrent structure as the hidden layer.

Recurrent Model Details We integrate the results of the forward and backward pass of the designed network structure as the hidden output. To be specific, the forward pass can be achieved by the following equations:

$$\begin{aligned}
\overrightarrow{r^{(t)}} &= \sigma(\mathbf{W}^r \overrightarrow{h^{(t-1)}} + \mathbf{U}^r \overrightarrow{x^{(t)}} + \mathbf{R}^r \overrightarrow{y^{(t-1)}} + b^r)\,, \\
\overrightarrow{u^{(t)}} &= \sigma(\mathbf{W}^u \overrightarrow{h^{(t-1)}} + \mathbf{U}^u \overrightarrow{x^{(t)}} + \mathbf{R}^u \overrightarrow{y^{(t-1)}} + b^u)\,, \\
\overrightarrow{a^{(t)}} &= \mathbf{W}(\overrightarrow{r^{(t)}} \odot \overrightarrow{h^{(t-1)}}) + \mathbf{U}x^{(t)} + \mathbf{R}\overrightarrow{y^{(t-1)}} + b\,, \\
\overrightarrow{h^{(t)}} &= \overrightarrow{u^{(t)}} \odot \overrightarrow{h^{(t-1)}} + (1 - \overrightarrow{u^{(t)}}) \odot \tanh(\overrightarrow{a^{(t)}})\,.
\end{aligned}$$

Where $\overrightarrow{r^{(t)}}$ and $\overrightarrow{u^{(t)}}$ are the gates and $\overrightarrow{h^{(t-1)}}$ is the hidden output of the last time step. Both $\{\mathbf{W}^r, \mathbf{W}^u, \mathbf{W}, \mathbf{U}^r, \mathbf{U}^u, \mathbf{U}, \mathbf{R}^r, \mathbf{R}^u, \mathbf{R}\}$ and $\{b^r, b^u, b\}$ are the parameters needed to learn. $\sigma(\cdot)$ and $\tanh(\cdot)$ are activation functions. Recall that we relax the independent assumption of labels by adding a link from the past label to the current hidden cell, here $\{\mathbf{R}^r, \mathbf{R}^u, \mathbf{R}\}$ realizes and controls the information flow through these links.

The backward pass applies the same equations but from the opposite direction. We use $\overleftarrow{h^{(t)}}$ to represent the output of the backward pass. Using $\overrightarrow{h^{(t)}}$ and $\overleftarrow{h^{(t)}}$, we can then compute the prediction $\hat{y}^{(t)}$ by following the equations below:

$$\begin{aligned}
o^{(t)} &= \mathbf{V}_1 \overrightarrow{h^{(t)}} + \mathbf{V}_2 \overleftarrow{h^{(t)}} + c\,, \\
\hat{y}^{(t)} &= \text{softmax}(o^{(t)})\,.
\end{aligned}$$

Here, V_1, V_2 and c are also part of parameters that need to be learned through training.

Training Strategy. When training the proposed neural network, we select cross-entropy as loss function and adopt ADAM to train the proposed neural network, which is one of the benchmark optimization algorithms used to train the neural networks. Due to the large number of training samples, we use mini-batch strategy to reduce computational cost and the technical details can be found in [28].

Facilitating Reverse Execution. With the help of our previous deep learning model, we could predict memory region information accessed by memory reference in each instruction [1]. The memory region information is set as an attribute of corresponding memory reference/access. When memory alias is acquired from reverse execution, we first compare the memory regions of the instruction pair. If the neural network predicts the two regions are different, we choose the hypothesis that they are non-aliasing; otherwise, we choose the hypothesis that the memory is aliasing. With the assumption that deep learning model could correctly determine the memory region of memory access with high probability, we could avoid backtracking in most cases and break the tie between contradictory hypotheses when no conflict is found. Take Figure 1 and Figure 2 as an

[1]It should be noted that we map the label tied to each byte back to the instruction via a majority vote mechanism. That is, we treat the label that mostly happens in the bytes of an instruction as the label for that instruction (See Figure 2).

example. We could know that the memory regions accessed by `[esp+0x8]@L11` and `[esp+0x8]@L7` are all stack region, meanwhile, `[eax]@L12` and `[eax]@L9` refer to heap region. Thus, `[eax]@L12` and `[esp+0x8]@L11` are not aliased with each other as they reference different memory regions. Compared with hypothesis testing, deep learning requires much less execution time because we avoided testing the aliasing hypothesis. More importantly, we could choose a preferred alias relation between `[esp+0x8]@L7` and `[eax]@9` (non-aliasing), which cannot be determined by HT. With this aliasing relationship resolved, the dark gray area in Figure 2 could be recovered, thus locating the root cause at `L9`.

C. Discussion

Here, we discuss more insights behind adopting deep learning for memory region identification by answering the following two questions: ❶ Why machine learning could address the aforementioned challenges instead of simple statistics? ❷ Why using deep learning rather than other machine learning techniques?

Why not Simple Statistic. The most intuitive way of inferring memory region would be collecting the frequency specific instructions access a specific memory region within the available execution traces, building a mapping between the instructions and memory regions and then using this mapping to infer the memory access regions within the new traces. However, due to the diversity of instructions, it is extremely hard to conclude all of these relationships. Instead, we can only utilize some commonly-adopted heuristics to predict memory region accesses for instructions. For example, instructions `pop` and `push` clearly indicate the memory access to the stack. We leveraged those heuristics for our problem and discovered that we could accurately predict memory region accesses only for 5% instructions. This motivates us to automatically learn such correspondent relations from data by machine learning instead of manually extracting them via plain statistics.

In many previous applications (*e.g.,* API generation [34]), it has been demonstrated that machine learning, especially sequence to sequence models can be used to learn patterns from a sequence of inputs, thus facilitating the determination of a label for each individual input. As mentioned before, we model memory region determination as a learning task that takes as input the machine code and, in turn, predicts the memory region tied to the code. We believe that learning from the previous sequence of inputs adequately reflects the process of determining memory region access. This is because the memory region that an instruction accesses can be determined by the semantics of that instruction or its context indicated by previous instructions. Taking the following code section as another example. The memory region indicated by `[ebx]` in the second instruction depends on the first instruction. Since register `esp` is a stack pointer, we can infer that `[ebx]` indicates a memory access to the stack region.

```
0:  8d 1c 24   lea    ebx,[esp]
3:  89 0b      mov    DWORD PTR [ebx],ecx
```

Why not Shallow Learning. Besides RNN, Hidden Markov Model(HMM) [35] and Conditional Random Field(CRF) [36] are the other two widely used sequence to sequence ML models [2]. However, it has been shown that RNN has much stronger learning ability than these two models in many fields (*e.g.,* speech recognition [37]) especially for complex and big data. Here, execution trace is equipped with these two properties(*i.e.,* complex patterns and large scale data), besides, past researches [28, 32] about using RNN for binary analysis has also demonstrated that RNN outperforms those shallow learning approaches. Because of those evidence, we select RNN to solve our problems.

IV. Implementation

We implemented a prototype of `RENN` for Linux system. Our prototype consists of three major systems: ❶ Trace Collection; ❷ Deep Learning Model; ❸ Root Cause Diagnosis;

First, we developed one subsystem which collects and records runtime information of program execution based on Intel Pin [38]. To be specific, the runtime information includes binary of each instruction and its corresponding memory access(es). With that information, we could automatically label each memory access with the corresponding number. Then those instruction binary and their labels will be provided as training data of deep learning model.

Second, we implemented our deep learning model leveraging the Keras package [39] and with Theano [40] as backend. We trained our deep learning model with the binary encoding of instructions and memory region information obtained from our first system. For the testing data set, this model will predict the memory region attached to one instruction.

Third, we prototyped a neural network assisted system based on `POMP`. It interpreted the output of our pre-trained model and verified the relationship between each pair of memory references. It prefers the result of deep learning model other than HT. With this strategy, we reversely execute those instructions to get the program states prior to the crash site. With those recovered data flow, we take the crash object as a taint source and perform backward taint analysis to trace back the instructions that contribute to the crash.

To be summarized, our implementation contains about 2,500 lines of C/C++ code and about 2,000 lines of `Python` code. And in this work, we ran the whole system on one 32-bit Ubuntu 14.04 with Linux kernel 4.4.0 running on an Intel i7-4700 quad-core processor with 16 GB RAM. We trained our deep neural networks on 2 Nvidia Tesla K40 GPUs using the `Keras` package and with `Theano` as backend.

V. Evaluation

In this section, we describe our evaluation procedure for `RENN`. We aim to answer the following questions: ❶ Could our

[2]The problem of predicting the label of every element within an input sequence is also called tagging problem (*e.g.,* predicting the part of speech of each word in a sentence.). Before RNN, HMM and CRF are the state-of-art techniques for tagging [23].

CVE/EDB ID	Program	Trace Len.	Execution Time		Non-Alias		Root Cause		Statistics			
			POMP	RENN(-)	POMP	RENN(+)	POMP	RENN	Global	Heap	Stack	Other
CVE-2002-1496	nullhttpd-0.5.0	1056	1.19	2.51%	73.21%	8.48%	✓	✓	26	33	0	572
CVE-2004-0597	libpng-1.2.5	4214	95	8.42%	50.44%	31.67%	✓	✓	1	47	1345	911
CVE-2004-1120	prozilla-1.3.6	2546	15.82	69.53%	82.84%	12.26%	✓	✓	12	675	1409	152
CVE-2004-1257	abc2mtex-1.6.1	61633	15060	63.75%	59.18%	10.01%	✓	✓	742	9962	26347	1728
CVE-2004-1271	DXFscope-0.2	5000	17.39	9.20%	91.21%	7.69%	✓	✓	9	0	1594	756
CVE-2004-1275	html2hdml-1.0.3	82575	22200	59.46%	1.51%	50.48%	✓	✓	25	16379	10383	683
CVE-2004-1279	jpegtoavi-1.5	133734	39000	50.77%	12.03%	12.89	✗	✓	13	266	32458	31265
CVE-2004-1287	nasm-0.98.38	4072	72	5.56%	7.16%	54.53%	✓	✓	7	2759	986	189
CVE-2004-1288	o3read-0.0.3	80000	12600	50%	45.67%	18.46%	✓	✓	1069	0	22022	22735
CVE-2004-1289	pcal-4.7.1	58291	44700	43.62%	45.68%	43.85%	✗	✓	1409	11347	13855	5794
CVE-2004-1297	unrtf-0.19.3	200	0.34	29.41%	100%	0.0%	✓	✓	1	24	43	5
CVE-2004-2167	LaTeX2RTF-1.9.15	17056	76	13.16%	82.43%	0.06%	✓	✓	11	1035	635	90
CVE-2005-3862	unalz-0.52	61999	28860	65.28%	30.90%	28.26%	✓	✓	4	1532	14313	5659
CVE-2005-4807	gas-2.12	16464	516	58.14%	17.06%	49.57%	✓	✓	19	1849	3875	466
CVE-2006-2465	MP3Info-0.8.5a	31888	19200	31.25%	23.42%	52.92%	✗	✓	6	4476	10905	1318
CVE-2006-2971	0verkill-0.16	50012	492	-	nan	nan	✗	✗	9304	0	1163	9305
CVE-2007-4060	CoreHTTP-0.5.3a	10000	309	17.44%	55.10%	37.01%	✓	✓	42	366	3098	1321
CVE-2008-2950	Poppler-0.8.4	1000	0.39	28.21%	92.13%	2.81%	✓	✓	0	1	350	149
CVE-2008-5314	ClamAV-0.93.3	99985	53612	22.38%	92.03%	7.53%	✗	✗	0	0	59138	8091
CVE-2009-2285	LibTIFF-3.8.2	50000	8796	66.53%	24.04%	32.62%	✓	✓	97	23604	5051	2255
CVE-2009-3050	HTMLDOC-1.8.27	12171	751	13.98%	34.05%	27.41%	✓	✓	17	401	4243	1093
CVE-2009-3586	CoreHTTP-0.5.3.1	10050	370	15.16%	59.01%	31.09%	✓	✓	44	63	3351	1372
CVE-2009-5018	gif2png-2.5.2	77873	9000	53.33%	73.25%	12.17%	✓	✓	9	175	20754	15935
CVE-2010-2891	LibSMI-0.4.8	50159	33600	53.57%	67.32%	18.83%	✓	✓	4	1349	25203	3796
CVE-2012-4409	mcrypt-2.5.8	1000	0.47	27.84%	80.50%	13.92%	✓	✓	6	20	359	155
CVE-2013-0221	Coreutils-8.4	5015	8.63	57.94%	94.81%	4.57%	✓	✓	6	14	106	1129
CVE-2013-0222	Coreutils-8.4	5867	37	19.54%	18.87%	39.17%	✓	✓	14	67	1854	810
CVE-2013-0223	Coreutils-8.4	4000	4.03	39.70%	82.73%	12.24%	✓	✓	34	102	2137	176
CVE-2013-2028	NGINX-1.4.0	983	1.29	38.67%	88.95%	6.41%	✓	✓	7	34	292	68
CVE-2014-8322	aireplay-ng-1.2b3	20000	721	27.74%	64.06%	30.53%	✗	✗	84	138	6926	2746
CVE-2015-5895	SQLite-3.8.6	10000	34.08	49.88%	79.41%	18.96%	✗	✓	12	549	3474	1154
CVE-2016-7445	openjpeg-2.1.1	1035	0.29	22.26%	23.88%	33.86%	✓	✓	6	0	353	158
CVE-2016-2563	PuTTY-0.66	68728	3600	58.33%	43.86%	21.62%	✓	✓	2310	7112	22573	2412
CVE-2017-5854	PoDoFo-0.9.4	4999	271	7.38%	1.41%	17.19%	✓	✓	34	201	2725	177
EDB-17611	UnRAR-3.9.3	36216	512	37.70%	93.71%	2.66%	✓	✓	4377	0	5061	367
EDB-23523	GDB-7.5.1	4000	62	17.74%	73.07%	21.59%	✗	✓	25	506	1432	151
EDB-33251	Python-2.7.5	33431	21000	57.14%	46.86%	12.86%	✓	✓	1	32949	147	25
EDB-30142	GDB-6.6	1000	0.71	53.81%	86.96%	11.36%	✗	✓	40	126	352	43
EDB-38616	Python-2.7	1000	6.98	39.98%	90.46%	8.26%	✓	✓	31	13	365	124
EDB-890	psutils-p17	3040	4.81	27.42%	58.30%	26.91%	✓	✓	17	0	996	383
Average	-	-	-	36.25%	57.63%	21.35%	31	37	-	-	-	-

TABLE I: List of program crashes corresponding to memory corruption vulnerabilities. "Trace Len" describes the number of instructions from the crash site to the root cause. For Execution Time, "POMP" is in units of seconds, and "RENN(-)" means the percentage of decreased time compared to POMP. "Non-Alias" is the percentage of identified non-alias pairs. "RENN(+)" refers to the increased percentage in relation to "POMP". "Root Cause" shows whether the root cause is identified with each tool. The data statistics under "Statistics" indicate the amount of memory references across those memory regions.

trained neural network correctly identify the memory regions which instructions access? ❷ Could the identified memory regions improve the performance of reverse execution with respect to memory alias problem? ❸ Could the identified memory regions enhance the root cause diagnosis of reverse execution?

A. Data Set

To train our deep learning model, we need to provide various execution traces and the corresponding memory region(es) each instruction accesses. To do this, we set up a tracing system which monitors the runtime execution of 78 different popular programs (*e.g.,* GNU Core Utilities). To be specific, then run these programs with the default commands and running examples provided in the manual pages. For this work, we compiled a training data set which contains 49,193,919 lines of instructions from 96 different traces.

To evaluate RENN, we prepare a testing data set comprised of a wide variety of vulnerable programs. Our resulting data set contained 40 vulnerabilities across 38 unique Linux programs. We assembled this data set by randomly selecting vulnerabilities from one vulnerability database [41]. Each of the vulnerabilities has thorough documentation on how to set up the environment (including operating system information along with any required libraries), as well as how to trigger the vulnerability and, consequently, crash the program. For each case, we followed these procedures while tracing the program's execution. Note that, optimization level is an element which affects both vulnerability reproduction [41] and deep learning [42]. To eliminate the effects of this element, we compile the training data set and testing data set with O2 and

		Global	Heap	Stack	Other
Precision	Bi-RNN	99.55%	99.33%	99.96%	99.75%
	Bi-GRU	99.55%	99.49%	99.98%	99.80%
	Bi-LSTM	99.55%	99.30%	99.97%	99.76%
	Our model	**99.99%**	**99.79%**	**99.99%**	**99.88%**
Recall	Bi-RNN	99.50%	99.47%	99.94%	99.81%
	Bi-GRU	99.54%	99.49%	99.94%	99.78%
	Bi-LSTM	99.51%	99.55%	99.94%	99.81%
	Our model	**99.88%**	**99.76%**	**99.97%**	**99.90%**

TABLE II: The overall performance of different recurrent neural network architectures.

other options. The resulting traces constitute our testing data set. It is noteworthy that those crashing traces are affected by not only vulnerable source code, optimization level, but also the operating system and libraries. Therefore it's very difficult to get the same traces with the ones in the POMP. Taking these crash traces as input, we will evaluate RENN's effectiveness and efficiency compared with POMP.

We show our examples in Table I. The software in the data set contains a wide variety of programs ranging from sophisticated software like Nginx with over 100K lines of code to lightweight software such as psutils and corehttp with less than 2K lines of code. Additionally, the vulnerabilities include not only memory corruption vulnerabilities, but also other common software bugs like null pointer dereference and stack exhaustion. In addition, we can confirm the dissimilarity between the training and testing data sets. As we observe, the programs in Table I have less overlap with the programs in our training data set, which avoids using the same or similar data in both training and testing data sets. Furthermore, we compared the instructions in the testing data set with those in the training set. We found that only 14.02% functions overlap, appearing both in the training and testing data set.

B. Experimental Setup

With previously prepared data sets, we design a series of experiments to evaluate our proposed technique. First, we trained, tested, and compared four different recurrent neural network models with 40 execution traces. To do this, we first labeled the bytes tied to the execution traces based on the memory regions that the corresponding instruction accesses. Then, we shuffled the traces and divided them into 5 disjoint groups. With the data traces partitioned, we took each group of data as our testing data corpus and utilized the remaining to train our neural network models.

In this way, we obtained 5 distinct models for each of the neural network architectures specified in Table II. By performing label predictions with the models against the corresponding testing data corpus, we computed the precision and recall, and finally took the corresponding average as the performance of each neural network architecture. It should be noticed that all the neural networks shown in Table II are bi-directional. This is because previous research [28] indicates the bi-directional structure outperforms those designed with a single-directional chain.

In addition to our deep learning experiments, we also set up another test to verify the ability of deep learning to link instructions and corresponding memory references. First, we feed the execution instructions into a deep learning model and get the output of predicted memory regions. Following, doing reverse execution, we combine the prediction of memory regions with HT. As mentioned in Section IV, RENN is based on POMP and we enhance this technique by allowing it to predict the memory regions and verify alias relationships that HT alone fails to identify. In addition, we follow the selection strategy of POMP in which we first introduced the instructions of a crashing function. If this partial trace is not enough to identify the root cause, we extend the partial trace function-by-function until the root cause is spotted. In this way, we could significantly reduce the execution time Hypothesis Testing takes to identify a conflict. Finally, we measured the execution time of POMP and RENN, the percentage of identified non-alias pairs, and whether the root cause is caught or not.

C. Experimental Results

Deep Learning Model Performance. Table II shows the precision and recall of various recurrent neural networks, regarding their capability of assigning correct memory regions to executed instructions. As we can easily observe, of all the neural network models, our proposed bi-directional conditional GRU model (specified as 'our model') exhibits the highest classification precision and recall. This indicates that our model has a better capability of capturing data dependencies hidden in the sequence of instructions.

From the table, we also find that all the neural network models have more than 99% of precision and recall. 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 logged execution traces are relatively long. Using a neural network with only 0.1% improvement in precision or recall, we could reduce the number of false positives or negatives by the thousands. As we can observe from Table II, our model generally increases classification performance by about $0.1\% \sim 0.4\%$. Given a long execution trace containing tens of thousands – or even millions – of instructions, this performance improvement indicates a significant reduction in the memory regions mistakenly assigned by neural networks.

In addition to showing the superior performance of our deep learning model, we demonstrate the utility of RENN in terms of its ability to save execution time (efficiency) and resolve non-alias pair (effectiveness).

Efficiency Improvement on Reverse Execution. Table I presents the execution time of POMP for each case and the decreased percentage of execution time caused by resolved non-alias pairs identified by RENN. From the data statistics, we can see that, compared with POMP, the execution time of RENN is decreased by more than 36% on average. And some cases (*e.g.,* prozilla, libtiff) could even save about 60% of execution time. The underlying reason is that deep learning

can verify some alias relations that hypothesis testing cannot while incurring significantly lower computational complexity. The more non-alias pairs deep learning is able to resolve, the more execution time is saved.

Effectiveness on Root Cause Diagnosis. Table I also shows that the percentage of identified non-alias pair is increased by 21.35% on average. The 37% increase of identified non-alias pair indirectly reflects that RENN could catch more alias relation of memory pair and have more possibility to catch the root cause. The internal reason is that deep learning can verify some alias relations which reverse execution alone cannot identify. They understand memory alias relationship from different perspectives. Finally, we discover that backward taint analysis does benefit from the memory alias analysis, being able to locate more root causes with the help of a deep learning model. For 9 cases that cannot be identified by POMP, RENN could successfully fix 6 of them. For the other 3 cases that RENN cannot resolve, we proceed to analyze them. For each case, we manually verify the failure and double-check each case. We find that, for Overkill-0.16 and ClamAV-0.93.3, it is extremely difficult to record all the execution traces including the root cause as the root cause is really far from the crash site, more than 10 million. Regarding aireplay-ng-1.2b3, we discover that the crashing program invoked the system call SyS_read which writes a data chunk to a certain memory region. Since memory alias relation between the buffer to read in SyS_read and the corrupted data by stack overflow is in the same memory region, but different offsets, deep learning could not help verify such memory alias relation. As a result, SyS_read intervenes the propagation of data flow, making the output of RENN less informative to failure diagnosis.

VI. Related Work

Our work focuses on complementing reverse execution at the binary level with deep learning. In this section, we present existing research in the following two domains: reverse execution and adopting machine learning for binary analysis and discuss their limitations.

Reverse Execution. Reverse execution refers to a debugging technique that allows developers to recover a previous program state. To retrieve a specific state, existing works [43–45] first try to restore the execution state from a saved point to that state by either state saving or program instrumentation. However, these approaches do not perform well due to space and time limitations during run-time. Then, many kinds of researches [1, 46] seek to perform the reverse execution using coredump information. For example, RE-tracer [1] performed backward taint analysis to triage program crashes based on semantics reconstructed from coredump and Zamfir *et al.* developed a technique named reverse execution synthesis (RES), which takes a coredump from a software crash as input and automatically computes the suffix of an execution that leads to that crash [46]. However, these techniques fail to achieve decent performance when the coredump is corrupted. To resolve this issue, recent studies [2, 13] utilize a new processor hardware feature (*i.e.,* Intel PT) together with the coredump to perform the reverse execution. To be specific, POMP conservatively assumes an unknown memory write may write to anywhere and uses hypothesis testing to resolve the memory alias. On the other hand, REPT aggressively ignores unknown memory writes, and then uses an error correction mechanism to rectify the mistakes caused by omitting previous unknown memory writes. Despite the former technique could guarantee the correctness of the reverse execution, it has a really bad performance. Compared to POMP, REPT performs relatively well, however, it will encounter correctness issues during the error-correcting phase. In this work, we propose a novel RNN to facilitate the memory alias identification in crashed traces and then use that information to perform reserves execution. As is shown in Section V, RENN not only improves the performance of reverse execution but also enhances the ability of diagnosing vulnerabilities root causes.

Machine Learning in Binary Analysis. Current works about using simple machine learning for binary analysis mainly focus on identifying binary function boundaries. For example, Rosenblum *et al.* [47] and Bao *et al.* [48] used conditional random fields and tree-based approach to solve this problem respectively and achieved better performance than non-machine learning approaches. However, Shin *et al.* [28] shows that deep learning could perform even better on this task. Besides binary function boundaries identification, deep learning also has been used to perform the following tasks: pinpointing function type signatures [32], detecting similar binaries [49] and decompiling binary codes [50]. Unlike these works that adopt off-the-shelf network architectures [3], we propose a new RNN based on the specific attributes of binaries and achieve better performances than the existing architectures.

VII. Conclusion

In this paper, we introduce RENN, a neural network-assisted reverse execution system to diagnose software crashes. Our RNN is based on a novel bi-directional conditional GRU, designed to facilitate reverse execution for alias analysis at the binary level. We conduct an evaluation of RENN on 40 manually tested memory corruption vulnerabilities across a variety of software. Our results show that this new neural architecture can significantly improve reverse execution with respect to its capability in resolving memory aliases and reducing execution time. Therefore, we show that RENN greatly benefits current data flow analyses, both in terms of efficiency (execution time) and effectiveness (correctly identifying the root cause), when facing a post-crash incomplete execution trace.

VIII. Acknowledgement

We would like to thank the anonymous reviewers for their valuable feedback. This project was supported in part by NSF Award-1718459, and by the Chinese National Natural Science Foundation 61272078.

[3]Note that we do not include the architecture proposed in [49], because their architecture works on the control flow graph instead of the raw binaries.

REFERENCES

[1] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "Retracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[2] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[3] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[4] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[5] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[6] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[7] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[8] Mozilla Corp., "Mozilla RR," http://rr-project.org.

[9] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 3, 2012.

[10] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu, "Coreracer: a practical memory race recorder for multicore x86 tso processors," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 216–225.

[11] GNU Foundation, "Gdb and reverse debugging," https://www.gnu.org/software/gdb/news/reversible.html.

[12] Microsoft Corp., "Time travel debugging," https://docs.microsoft.com/en-us/windowshardware/drivers/debugger/time-travel-debuggingoverview.

[13] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse debugging of failures in deployed software," in *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[14] O. Security, "Offensive security exploit database archive," https://www.exploit-db.com/, 2009.

[15] "Processor tracing," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[16] "Embedded trace macrocell architecture specification," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/\index.html, 2017.

[17] G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the 13th International Conference on Compiler Construction (CC)*, 2004.

[18] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems*, 2010.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, 2015.

[20] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, "Learning precise timing with lstm recurrent networks," *Journal of machine learning research*, 2002.

[21] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[22] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013.

[23] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[24] C. H. Li and C. Lee, "Minimum cross entropy thresholding," *Pattern recognition*, 1993.

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

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representation (ICLR)*, 2014.

[27] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 1998.

[28] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.

[30] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press, 2016.

[31] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.

[32] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in

Proceedings of the 26th USENIX Security Symposium (USENIX Security), 2017.

[33] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, 1997.

[34] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[35] L. R. Rabiner and B.-H. Juang, "An introduction to hidden markov models," *IEEE ASSP magazine*, 1986.

[36] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, 2001.

[37] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proceedings of the 38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, 2005.

[39] F. Chollet *et al.*, "Keras," 2015.

[40] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[41] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security)*, 2018.

[42] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security)*, 2015.

[43] T. Akgul and V. J. Mooney III, "Assembly instruction level reverse execution for debugging," *ACM Trans. Softw. Eng. Methodol.*, 2004.

[44] T. Akgul, V. J. Mooney III, and S. Pande, "A fast assembly level reverse execution method via dynamic slicing," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[45] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc, "A new method for program inversion," in *Proceedings of the 21st International Conference on Compiler Construction (CC)*, 2012.

[46] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea, "Automated debugging for arbitrarily long executions," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

[47] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code." in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, 2008.

[48] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[49] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "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.

[50] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.