2020 IEEE Symposium on Security and Privacy

Neutaint: Efficient Dynamic Taint Analysis with Neural Networks

Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray and Suman Jana
Columbia University

***Abstract*—Dynamic taint analysis (DTA) is widely used by various applications to track information flow during runtime execution. Existing DTA techniques use rule-based taint-propagation, which is neither accurate (*i.e.*, high false positive rate) nor efficient (*i.e.*, large runtime overhead). It is hard to specify taint rules for each operation while covering all corner cases correctly. Moreover, the overtaint and undertaint errors can accumulate during the propagation of taint information across multiple operations. Finally, rule-based propagation requires each operation to be inspected before applying the appropriate rules resulting in prohibitive performance overhead on large real-world applications.**

In this work, we propose NEUTAINT, a novel end-to-end approach to track information flow using neural program embeddings. The neural program embeddings model the target's programs computations taking place between taint sources and sinks, which automatically learns the information flow by observing a diverse set of execution traces. To perform lightweight and precise information flow analysis, we utilize saliency maps to reason about most influential sources for different sinks. NEUTAINT constructs two saliency maps, a popular machine learning approach to influence analysis, to summarize both coarse-grained and fine-grained information flow in the neural program embeddings.

We compare NEUTAINT with 3 state-of-the-art dynamic taint analysis tools. The evaluation results show that NEUTAINT can achieve 68% accuracy, on average, which is 10% improvement while reducing 40× runtime overhead over the second-best taint tool Libdft on 6 real world programs. NEUTAINT also achieves 61% more edge coverage when used for taint-guided fuzzing indicating the effectiveness of the identified influential bytes. We also evaluate NEUTAINT's ability to detect real world software attacks. The results show that NEUTAINT can successfully detect different types of vulnerabilities including buffer/heap/integer overflows, division by zero, etc. Lastly, NEUTAINT can detect 98.7% **of total flows, the highest among all taint analysis tools.**

I. INTRODUCTION

Dynamic Taint Analysis (DTA) [41] is a well-known technique to track information flow between source and sink variables during a program's execution. It has been used in many security-relevant applications including guided fuzzing, automatic vulnerability discovery, run-time policy enforcement, information leak detection and malware behavior analysis. [19] [22] [31] [34] [37] [41] [51] [58] [59] [61] [64]. Most, if not all, applications of DTA require high accuracy and low run-time overhead. Unfortunately, existing DTA techniques suffer both from high false positive/negative rates and incur prohibitive performance overhead especially for large real-world programs [18].

All existing DTA techniques *propagate* taint labels from the taint source to the sinks during the target program's execution based on a set of rules for every executed statement. The final taint results are computed by propagating and composing the individual per-statement taint rules together. Essentially, the final output indicates whether a taint source influences a sink.

Unfortunately, this rule-based propagation approach has three fundamental limitations: (i) *Specifying accurate propagation rules*: Even for seemingly simple operations, accurately specifying propagation rules is often hard as there can be many different cases to consider. For instance, the correct propagation rule for `s = x*c` might vary based on different values of taint labels of `x` and constant `c`—if `c` is always 0, `s` is not influenced by `x`. Similarly, if `c` is very large and `x` is small, the influence of `x` on the output might be negligibly small. It is extremely difficult to enumerate all such possibilities exhaustively. (ii) *Accumulating errors*: Even if taint propagation rules for each operation are accurate, their composition across multiple operations can introduce large errors. For example, consider two operations `s = a + b; t = s - b`, where the rule-based propagation will conclude that both `s` and `t` are influenced by `b`. Although this is the correct analysis for each operation individually, `t` is not affected by `b`. (iii) *Large run-time overhead*: The rule-based propagation introduces prohibitive run-time overhead as each operation has to be examined to decide which rules to apply.

In this paper, we propose a novel technique, NEUTAINT that automatically learns the information flow, *i.e.*, taint, in a program by modeling its source-sink behaviors with neural program embeddings and gradient analysis. Neural program embeddings are essentially neural networks that learn to predict program behaviors from different representations of a program (e.g., graph representation, input-output pairs) [60], [46], [52], [53]. Such embeddings have shown promise in various tasks including fuzzing, program repair, program synthesis, vulnerability localization and binary similarity detection [60], [25], [14], [47], [52]. Leveraging the chain rule of calculus, gradient analysis is a more precise technique using automatic gradient computation to accurately track the influence of sources over sinks in programs. Our NEUTAINT first learns a neural program embedding of a program's run-time behaviors and then performs gradient analysis for light-weight and accurate end-to-end information flow tracking.

NEUTAINT addresses the aforementioned limitations of rule-based taint tracking. First, while rule-based DTA applies the propagation rules based on the program statements executed along a single flow, neural networks can generalize

DOI 10.1109/SP40000.2020.00022

and infer new flows based on the past program behaviors. This allows us to more accurately model different degrees of influence from different taint sources. Second, since neural networks are continuous, the gradient computation provides an efficient and precise mathematical way of deriving a source's influence on a particular sink, thus avoiding the need to manually specify propagation rules. This also minimizes the composition errors that plague existing rule-based approaches and significantly improves the accuracy of taint tracking by cutting down the false positive/negative rates. Lastly, the neural program embeddings can be trained using program traces generated offline by adding light-weight instrumentation and executing the target program with multiple inputs. Once trained, the neural program embeddings can be used to perform taint analysis without even examining the computations performed by the target program in an highly efficient manner compared to rule-based propagation.

Specifically, we train dynamic neural program embeddings using observed execution paths between sources and sinks. Once we have observed one path reaching the sink from the source during program execution, we generate a lot of other program paths in a cheap way (e.g., mutating the input) to obtain the training data. Note that any form of DTA (rule-based or learning-based) requires an input triggering a program execution path from the source to the sink to begin its analysis. Our key advantage is that we can accurately infer new information flows without inspecting each executed statement. Even with simple training data, we can track more flows than three state-of-the-art DTA tools: Libdift, Triton, and DFSan. We can potentially further improve the training data quality by using techniques such as symbolic execution. We present detailed quantitative results showing the number of detected and missed flows by NEUTAINT in Section IV-G.

After training the neural program, we use a gradient-based attribution method on the trained network to create saliency maps that accurately measure the flow of information from NN inputs (*i.e.*, taint sources) to NN outputs (*i.e.*, taint sinks). Depending on the application domain, NEUTAINT supports two types of saliency maps to track information flow: (i) a coarse-grained map that aggregates the influence of all sources over all sinks. This map contains the information flow summarized by all executed inputs over all taken paths. (ii) a fine-grained map containing separate influence analysis for each source-sink pair. Such information is useful for tasks like zero-day attack detection. A common use case for such analysis is taint-guided fuzzing where the sources are the input bytes and the sinks are the variables used in program branches. Input bytes with high saliency values have large influences on the output variables in program branches. Mutating these bytes can maximally trigger the execution of a diverse set of program branches. The number of mutations spent on each byte can be adjusted according to its corresponding aggregate influence on all program branches, *i.e.*, the higher the influence, the more mutations should be tried on the corresponding byte.

We evaluate NEUTAINT against 3 state-of-the-art dynamic taint analysis tools: Libdft, Triton, and DFSan. We train neural network models representing two sets of real-world programs. For the first set of programs, NEUTAINT can successfully find the information flow from source to sink in known CVEs. On the second set of programs, we compare NEUTAINT against other tools in regards to accuracy, overhead, and effectiveness when applied to fuzzing (an important security application). We utilize the parsing logic of programs to build ground truth of hot bytes, *i.e.*, file format bytes (influential taint sources) that trigger different program behaviors (taint sinks at branching conditions). The evaluation results show that NEUTAINT achieves on average 10% higher taint accuracy than the second-best DTA tool. To compare the runtime overhead, we measure the total amount of time needed to process all the inputs in the training dataset, and our NEUTAINT is almost 40× more efficient than the second-fastest tool, Libdft. We then validate the taint information obtained from all tools through taint-guided fuzzing. We feed the hot bytes produced by the four different tools into a common fuzzer backend that supports the same mutation algorithm in which our NEUTAINT achieves 61% more edge coverage on all the real-world programs in 24 hours.

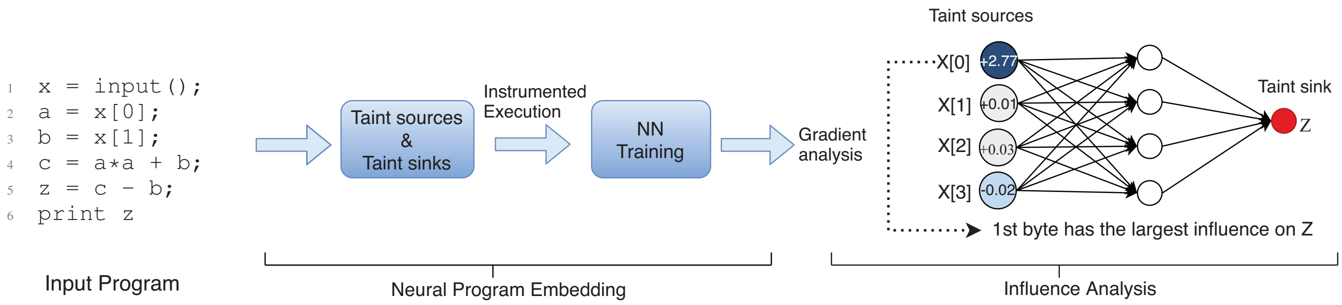
Our main contributions are as follows:

- We propose a novel information flow tracking technique based on neural program embedding and gradient analysis.
- We design and implement our technique as part of NEUTAINT and evaluate it against 3 state-of-the-art DTA tools. The evaluation shows that NEUTAINT can achieve on average 10% higher taint accuracy than the second-best tool while taking 40× less analysis time.
- We further validate the taint information obtained from 4 different tools by using a real world taint application, taint-guided fuzzing. The results show that NEUTAINT achieves 61% more edge coverage than that of the second-best DTA tool.
- We analyze and identify the key factors that allow NEUTAINT to outperform traditional DTA tools. In addition, we present quantitative results showing NEUTAINT's ability to infer new information flows and discuss different ways to further improve the training data quality.

II. BACKGROUND

This section first gives a brief overview of dynamic taint analysis. Then, we introduce existing work in program embeddings, among which dynamic program embeddings can be used to capture the runtime program behavior. Lastly, we discuss saliency maps for neural networks, which can be used to conduct information flow analysis for dynamic program embeddings.

Dynamic Taint Analysis (DTA). A dynamic taint analysis pre-defines taint sources (*e.g.,* untrusted file, network, etc.) and as a program executes tracks the effect of them on program state such as internal variables. In most cases, DTA wants to determine whether the taint sources affect some predefined target locations, commonly known as taint *sinks*. Depending on the specific application, taint sources and sinks vary.



```
x = input();
a = x[0];
b = x[1];
c = a*a + b;
z = c - b;
print z
```

Fig. 1: Simple code snippet demonstrating the workflow of NEUTAINT. NEUTAINT uses light-weight instrumentation to collect a diverse set of sources and sinks from the input program. Then, we train neural program embeddings and use gradient-based analysis to infer information flow for the programs.

For many security applications, user inputs are often used as taint sources [41]. For example, during fuzzing [49], [16], DTA explores diverse program execution behaviors and checks which input bytes affect the branches (*i.e.*, the taint sinks) of the target program. In the case of malware analysis [64], DTA monitors if program instruction registers (*i.e.*, taint sink) are manipulated by untrusted user input [41]. DTA is also applied to identifying user information leakage [21] [56], where DTA monitors a set of sensitive user data as taint source and a set of sensitive functions (*e.g.,* socket write) that leak that data to the outside world as taint sinks.

DTA is typically implemented with taint tags. There are mainly two types of taint tags used in the literature: binary tags and multiple tags. The binary tag approach marks all taint sources with a single binary value: 1 or 0 to represent tainted or untainted respectively. Binary tags are commonly found in simple tasks such as privacy leakage and detecting attacks from user-supplied inputs. However, they fail to monitor more fine-grained information flow used in malware analysis and taint-guided fuzzing because they can only track the existence of taint rather than the ownership of taint source. In contrast, multi-tag DTA, which tracks every taint source independently, tracks taint at a more detailed granularity at the cost of significantly large runtime overhead which grows quadratically with tag size. The large runtime overhead prohibits practical deployment of online DTA tasks to check security properties such as policy enforcement and intrusion detection in Android [21] [56]. Moreover, time-sensitive applications such as fuzzing ideally require taint analysis for a large number of program executions in a short amount of time for defenders to find vulnerabilities before attackers do [49]. These limitations are further detailed in Appendix D3.

Fundamental Problems of DTA. There are three fundamental problems in the design and implementation of taint: under-taint, over-taint, and large runtime overhead. Even with the heavy instrumentations that cause large runtime overheads, manually engineered rules for taint propagation still have poor accuracy at capturing information flow. These limitations of taint severely affect its applicability to real-world programs. A recent work TaintInduce [18] has proposed to learn the taint propagation rules instead of manually specifying them. This can increase the accuracy of individual rules, but the error accumulation and large overhead issues still remain, due to propagation-based design. Therefore, we choose to use end-to-end program embeddings, and we conduct influence analysis directly on the neural program to track information flow.

Program Embeddings. In general, there are two types of program embeddings, static and dynamic. Static program embeddings first generates a program representation, then use neural networks to encode the representation into embeddings. Example program representations include token sequences [25], [14], [47], [20], abstract syntax trees [39], and control and data flow graphs [63], [13]. Static program embeddings have been applied to correcting student code errors, automatic vulnerability detection, and detecting variable misuse. Since such program representations cannot capture program semantics, dynamic program embeddings learn program behavior from input-output pairs [60], [46], [53] by executing the program. Dynamic program embeddings have been used for fuzzing [52], solving symbolic constraints [53], program repair [60] and generating feedback on student code [46]. Since information flow analysis reflects the runtime behavior of programs, we use dynamic program embeddings to learn from program execution traces. Our neural program model approximates the program logic from taint source to taint sink. Then, we analyze the flow of information in the model.

Information Flow in Neural Network. A popular technique to track information flows in a neural network (NN) is a saliency map, which measures the sensitivity of the NN output to changes in the input features [54]. For example, in image classification, the saliency map can be viewed as an annotated representation of the input image, where the annotations at every pixel correspond to the gradient of the output *w.r.t.* to the corresponding original pixel value (*i.e.*, how the output category changes as the input image pixels change). Saliency maps have also been used to construct inputs with minimal perturbations as adversarial examples to an image classifier [44]. Since the saliency map indicates the most critical input features that affect final neural network output, it guides an attacker's construction of the adversarial example by localizing the changes needed on features to change the classifier output.

As a gradient-based attribution method, a saliency map has been widely used in interpreting neural networks. Compared to other gradient-based methods (*e.g.,* integrated gradient [57]), saliency maps focus on the sensitivity of neural output to every feature, *i.e.*, how the NN output changes with respect to a small change in the input. In contrast, integrated gradient tries to explain the attribution of neural output to every feature, *i.e.*, how each feature of input contributes to the final NN output. This implies that saliency values for specified input features may differ from their corresponding integrated gradient value. Since integrated gradient value is essentially $gradient * input$ and saliency value is gradient, the disparity between these two values is maximized when the input value is significantly small but the gradient value is large. In our case, since we want to infer which byte in the input affects the taint sink, *i.e.*, induce the greatest sensitivity to the neural network output, we use the saliency map method.

III. Methodology

A. Overview

In this section, we give a motivating example to show the workflow of NEUTAINT. As shown on the left side of Fig 1, we assume the taint source is `x`, taking 6 bytes from the user input, and the taint sink is variable `z`. The propagation-based dynamic taint analysis cannot derive accurate information flow in this case. Since variable `c` at line 4 is computed by `a` and `b`, the first two bytes of user input, so taint value for `c` is `a` and `b`. At line 5, `z` is computed from `c` and `b`, thus the taint value for `z` is composed from `c` and `b`. The analysis is accurate for both line 4 and line 5, but composing the propagation rules together amplifies errors. The analysis ignores the fact that at line 5 `z` actually equals `a*a` and is only affected by the first byte of user input. Composition introduces and amplifies errors and runtime overhead in the dynamic taint analysis.

On the contrary, NEUTAINT uses an end-to-end approach to build neural program embeddings for information flow analysis. Based on some training samples (*i.e.*, user input, `z`), NEUTAINT learns a neural program from dynamic execution results which preserve program context–`z` is only affected by `a`. As shown on the right side of Fig 1, given a user input `x`, NEUTAINT computes the gradient of variable `z` with respect to `x` and constructs a saliency map which indicates the sensitivity how each byte of `x` affects `z`. From the saliency map, we find that first byte is the most critical byte of input affecting `z`. Fig 1 presents a high-level overview of our approach.

Training. We first train a neural program to learn the information flow from taint source to sink. For a given program and a set of inputs, we mark these inputs and use light-weight instrumentation to collect values of sink variables. They represent the dynamic behavior of a program. Next, we train a neural network model (NN) to learn this dynamic behavior. Our NN approximates a function that maps sources to sinks. The training process minimizes the errors of learning this function, thereby improving the precision of information flow tracking.

Influence Estimation. We construct two saliency maps to infer the information flow from taint source to sink. Saliency maps analyze the sensitivity of input features for NEUTAINT. The more important a feature is, the more it influences the NN output. We first define a saliency map to summarize coarse-grained information flow for the program behavior, aggregating gradient information for all inputs and all paths. Then, we define the second saliency map to identify the most important taint sources for specific sinks, utilizing first-order partial derivatives of the NN output with respect to the input.

Since our end-to-end methodology of collecting program behavior information, training, and influence estimation is lightweight, the runtime overhead is much smaller than a traditional taint analysis tool. NEUTAINT directly performs analysis at the program semantic level by learning from dynamic program behaviors rather than on the instruction semantic level in traditional taint analysis which leads to under-taint and over-taint. Learning the end-to-end model with NEUTAINT reduces overall information tracking errors, which mitigates the issues of over-taint and under-taint. Thus, NEUTAINT achieves more accurate results than traditional taint analysis tools.

B. Program Embedding

NEUTAINT learns the information flow by observing a large set of taint source-sink pairs from program execution traces. The model predicts the values of taint sink variables given taint sources as model input. We formally define our neural network model as follows, with detailed architecture shown in Appendix A. Given a set of concrete taint sources $\boldsymbol{x}$ and the corresponding taint sinks $\boldsymbol{y}$ for a specified program P, the neural program predicts the taint sinks as $\hat{\boldsymbol{y}}$, with the following equations.

$$\boldsymbol{a} = \phi(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1) \tag{1}$$

$$\hat{\boldsymbol{y}} = \sigma(\boldsymbol{W}_2^T \boldsymbol{a} + \boldsymbol{b}_2) \tag{2}$$

We denote $\boldsymbol{W}_k, \boldsymbol{b}_k$ as trainable parameters for every layer where k represents the layer index, ϕ represents the ReLU function, and σ represents the sigmoid function. In Equation 1, $\boldsymbol{a}$ represents the output vector of the hidden layer of neural network. The NN model learns the function f that takes numerical vector of size m as input and outputs n taint sink variables. Let $\boldsymbol{\theta}$ denote the trainable weight parameters of f. Given a set of training samples $(\boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{X}$ is a set of taint sources and $\boldsymbol{Y}$ represents the correct taint sink values, the training task of the parametric function $f(\boldsymbol{x}, \boldsymbol{\theta})$ is to obtain the parameter $\hat{\boldsymbol{\theta}}$ that minimizes the multi-variable regression loss, where each variable is a taint sink.

After we train the NN model, we construct two saliency maps to analyze the flow of information in the neural program. From the neural program model, the first saliency map provides a global view of coarse-grained information flow when all sinks are considered as a whole. The second saliency map can extract the most influential taint sources for any given sink. We now explain the details of the information flow analysis.

C. Coarse-Grained Information Flow

We discuss the method to extract coarse-grained information flow from the NN model. We define the coarse-grained information flow as the influence of each source on *all* sinks. Since some dynamic taint analysis applications have a set of taint sink variable, e.g., taint-guided fuzzing, it is important to consider coarse-grained information flow to all the sinks. The aggregated information flow to a set of taint sink variables can highlight which part of the taint source has the most significant effect on them.

To extract coarse-grained information flow, we first compute the partial derivatives of the taint sink with respect to all sources. Let $f_i(\boldsymbol{\theta}, \boldsymbol{x})$ denote the output value for the i-th taint sink variable during the execution of the targeted program with taint source $\boldsymbol{x}$. We compute the derivative with respect to a given taint source $\boldsymbol{x}$, defined below, where x_j denotes the j-th byte in the taint source.

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{\theta}, \boldsymbol{x}) = \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{x})}{\partial \boldsymbol{x}} = \left[\frac{\partial f_i(\boldsymbol{\theta}, \boldsymbol{x})}{\partial x_j}\right]_{i \in 1 \ldots n, j \in 1 \ldots m} \tag{3}$$

The partial derivatives constitute a Jacobian matrix of the neural network function. Each element of the matrix represents the gradient of output neuron $f_i(\boldsymbol{\theta}, \boldsymbol{x})$ with respect to taint source byte x_j. Note that the gradient we compute has two main differences from the gradient used in a neural network trained by backpropagation. First, the target function is different. The gradient used for backpropagation is computed on a loss function which includes information about the state of the model parameters and the expected outputs. In contrast, our method computes the derivative on the output of the neural network, which includes only information about the model parameters. Since we aim to interpret how neural networks make the decision after convergence, our gradient computation does not need to consider the corresponding ground truth information. Second, we compute the gradient with respect to the input, rather than trainable parameters of neural network model. By computing the gradient directly with respect to the input, we obtain the sensitivity of NN output to all the bytes in the input.

Then, we construct a saliency map to provide the global view for coarse-grained information flow, based on partial derivatives of the neural network model. The saliency map $S(\boldsymbol{x})$ is defined as follows.

$$S(\boldsymbol{x})[j] = \sum_i \left| \frac{\partial f_i(\boldsymbol{\theta}, \boldsymbol{x})}{\partial x_j} \right| \tag{4}$$

$S(\boldsymbol{x})[j]$ is the sum of all the sink sensitivity to the j-th byte, representing the effect of the j-th byte to the overall program behavior from the current execution. Summarizing the sensitivity to all sinks includes information about all paths to these sinks. In addition, the neural program includes information about all the input data. Therefore, we can analyze the coarse-grained information flow using this saliency map.

D. Fine-Grained Information Flow

We define fine-grained information flow as the influence of each source to a *single* sink. Dynamic taint analysis applications that are interested in fine-grained information flow often set taint sink at a certain variable such as function pointer, jump target address and instruction pointer register. We refer to these applications as fine-grained information flow analysis.

To reason about how information arrives at a given sink, we follow similar steps from the coarse-grained information flow analysis as mentioned in Section III-C. First, we compute the Jacobian matrix to obtain the gradient information using equation 3. After obtaining the gradient value for every byte in the taint source, we can construct a saliency map to infer the fine-grained information flow from taint source to a particular taint sink. Since we are only interested in the sensitivity of the taint sink to every byte in the taint source, we take the absolute value of the gradient to construct the saliency map $S(\boldsymbol{x})$ defined as follows.

$$S(\boldsymbol{x})[j] = \left| \frac{\partial f_i(\boldsymbol{\theta}, \boldsymbol{x})}{\partial x_j} \right| \tag{5}$$

The bytes that causes the maximum fluctuations of NN output are considered taint source bytes that influence the sink. The set of source bytes that determines taint sink variables can be inferred by finding the top-k bytes with maximum values, defined below.

$$H_i(k) : arg(top\_k(\left| \frac{\partial f_i(\boldsymbol{\theta}, \boldsymbol{x})}{\partial \boldsymbol{x}} \right|)) \tag{6}$$

Let $H_i(k)$ denote the set of indices of K source bytes, top_k denotes the function to select k largest elements from a vector and arg denote the function to return indices of selected elements. Since our neural network learns a summary of dynamic program behavior from all training samples, the influential source bytes inferred from the neural network model contains knowledge from a large number of concrete runs of the program. On the contrary, traditional dynamic taint analysis tools have information from only the specific path taken from one execution.

E. Data Collection

In this section, we describe the general method to collect a set of training samples for our neural program training. To learn the information flow from taint source to taint sink, it is crucial to obtain a large, diverse, and representative dataset. However, unlike traditional machine learning tasks (*e.g.,* image classification, natural language processing, speech recognition), there is no standard dataset for various taint sources and sinks. There are many options to collect training datasets. A natural solution to collect such a dataset would be to randomly sample taint source-sink execution pairs for a specified programs. As an example to generate a training dataset, we can start with a common taint source, randomly flip the bytes in the taint source, and record the corresponding taint sink values. Alternatively, we can use a simple fuzzer to generate a set of taint source which trigger diverse program states and record the taint sinks values. The training data coverage affects the amount of information NEUTAINT can track. We can further improve the information flow coverage using more sophisticated techniques like coverage-guided fuzzing, symbolic execution, etc. However, in this paper we demonstrate that even with training data generated by a simple fuzzer, NEUTAINT can easily outperform existing DTA tools.

Note that taint sources are normally user input, files or user privacy strings that can be represented as byte sequences. So we can easily convert the byte sequences to bounded numerical vectors ranged in $[0, 255]$. However, the taint sink can be arbitrary variables in the program with unbounded values such as instruction pointer register, a complex socket structure, or a user defined variable in the program. These arbitrary variables are hard to model as unified representations for NN output and make it difficult for the NN to converge. To tackle this problem, we normalize these unbounded variables to bounded data for different applications. For example, in taint-guided fuzzing, we set taint sinks at a set of variables used in branch conditions and normalize the sink variables with binary data (*i.e.*, 1 represents the branch is taken, 0 represents the branch is not taken). The binary representation of the NN output can ensure the fast convergence of the model.

IV. EVALUATION

In this section, we evaluate the effectiveness and efficiency of NEUTAINT against three state-of-the-art dynamic taint analysis tools (Libdft [32], Triton [2], and DFSan [5]). We answer the following research questions about NEUTAINT in the evaluation.

1) **Hot Byte Accuracy:** Is NEUTAINT more accurate at finding hot bytes (*i.e.*, the most influential bytes that determine different program behaviors) in input for 6 real-world programs?
2) **Runtime Overhead:** What is the runtime overhead of NEUTAINT compared to state-of-the-art dynamic taint analysis tools (both with and without GPU)?
3) **Exploit Analysis:** Can NEUTAINT detect vulnerabilities in real-world programs?
4) **Application on Taint-Guided Fuzzing:** Since fuzzing is one of the most important security applications of taint, does NEUTAINT help taint-guided fuzzing achieve better edge coverage compared to other taint analysis tools?
5) **Model Choice:** How does NEUTAINT perform with different machine learning models other than neural networks?
6) **Information Loss:** What kinds of flows are missed by NEUTAINT? How does the training data quality affect such information loss and how to mitigate this loss?

To answer these questions, we will first describe our experiment setup and how we learn the neural program embeddings for the real-world programs.

A. Experiment Setup

Environment Setup. All our measurements are performed on a ubuntu 16.04 system with an Intel Xeon E5-2623 v4@2.60GHz CPU , an Nvidia GTX 1080 Ti GPU and 256 GB RAM. We implement the NEUTAINT in Keras-2.1.4 [6] with Tensorflow-1.8.0 [7] as the backend.

Next, we give a brief introduction of the three front-end tools in our evaluation, how we set up the tools, and our implementation of NEUTAINT.

1) Libdft: Libdft is a widely-used dynamic taint analysis engine based on Intel PIN framework [8]. It predefines taint propagation rules for every type of instruction via external functions using the PIN analysis API. Then, it dynamically instruments the binary code using the PIN instrumentation API at runtime. For every executed instruction, Libdft calls a corresponding external function to track the taint flow. If a particular type of instruction (*e.g.,* pop, ret) lacks any taint information, no external function will be invoked. Nevertheless, for real-world applications, most instructions contain taint information, and hence this design incurs a large runtime overhead due to the sheer number of external function calls. Another notable drawback of Libdft is that the current implementation only supports the x86 architecture.

Setup. We set up a modified version libdft with support of multiple taint tags [1]. The dependency PIN version is 2.13.

2) Triton: Triton is a platform that supports concolic execution, dynamic taint analysis, and abstract syntax tree representation. Similar to Libdft's approach for dynamic analysis, it uses Intel PIN to monitor taint flow corresponding to a set of predefined taint propagation rules. Currently, its taint analysis engine only supports the x86 architecture. Triton provides users Python bindings to the underlying PIN API so that they can write scripts to perform customized analysis tasks. However, these bindings are limited and fail to capture the full functionality of PIN. Moreover, the limited Python bindings cause imprecise dynamic taint analysis results in addition to the runtime overhead from heavy instrumentation and monitoring taint propagation in PIN.

Setup. We set up a develop fork Triton to support the multiple taint tags feature [3]. The dependency PIN version is 2.13. We write the analysis script with Triton Python binding to set corresponding taint sources and taint sinks according to different programs.

3) DFSan: DFSan (DataFlowSanitizer) is a data flow analysis framework provided by Clang [4]. It consists of a compile-time instrumentation module and a runtime dynamic library to track taint flow for the x86-64 architecture only. Users only need to define taint source and taint sink with the public DFSan API. DFSan relies on predefined taint propagation rules for LLVM IR instructions rather than architecture-specific assembly instructions. This enables DFSan to insert taint tracking functions at compile time into a program. Thus, it has a smaller runtime overhead compared to other PIN-based tools that use dynamic instrumentation. DFSan, however, fails to run on programs which depend on external shared libraries. Since the dynamically shared libraries cannot be instrumented when compiling a given program, DFSan cannot insert these taint tracking functions and fails to work on programs depending on dynamic shared libraries. This along with the exhaustion and resolution of taint tags (Section IV-C) limits the applicability of DFSan to real-world applications.

DTA Engine	Propagation Level	Dependency	Tag Type
Libdft	assembly instruction	Pin 2.13	multi-tag
Triton	assembly instruction	Pin 2.13	multi-tag
DFSan	LLVM instruction	LLVM-7.0.0	multi-tag

TABLE I: Dynamic Taint Analysis Engines.

Setup. We set up DFSan from the Clang runtime library. The underlying LLVM version is 7.0.0. We use DFSan's API to set taint source and sink for different programs.

4) NEUTAINT*:* **Model Architectures.** For each program, we train a neural program model which learns the program logic from taint sources to taint sinks. The NN model consists of 3 fully-connected layers. The hidden layer uses ReLU as activation functions with 4096 hidden units. The output layer uses Sigmoid as the activation function to predict the sink variables. Since each program has different taint sources and taint sinks, the corresponding neural program model has different number of input/output neurons. We describe taint sources and taint sinks for all programs in Table II. We use the first 6 programs to evaluate the hot byte accuracy and the taint-guided fuzzing experiments, so we set multiple taint sinks as branch variables (*i.e.*, the variables used in conditional predicates). The later 5 programs are evaluated in the exploit analysis experiment, so they only have a single taint sink at a specified variable. All 11 programs set each byte of the program input as a taint source. Thus the total number of taint sources are the total number of input bytes.

Training Data Collection. To collect the training data, we first run the AFL fuzzer with an initial seed to collect its mutation corpus. Next, we use a simple LLVM pass to add light-weight instrumentation for recording the two operands of CMP instructions during runtime. These operands of CMP instructions are branch variables (*i.e.*, our taint sinks) evaluated in conditional predicates. We run the instrumented program with the generated input and record the taint sink values. We collect around 2K input-output pairs (each input has multiple source bytes and reaches multiple sinks) on each program for training. For hot byte evaluation, we normalize the taint sink variables into binary, *i.e.*, 1 if a sink value satisfies the conditional predicate and 0 if not. We check the value of predicates by computing the difference of two operands of CMP instructions. For exploit analysis, we use the similar LLVM pass to obtain the specified taint sink values, then perform standard min-max normalization to the sink values (*i.e.*, $y_{norm} = (y - y_{min})/(y_{max} - y_{min})$). Note that the data collection and normalization can be easily done by a simple python script, no manual labeling effort is required.

Training Procedure. We adopt random weight initialization and cross-entropy loss function or mean-square-error loss depending on specific model output data type. The NN model is trained with an Adam optimizer for 100 epochs with an initial learning rate 0.01 and decay rate 0.7 per 10 epochs. We choose a mini-batch size 16. For exploit analysis evaluation, we use mean-square-error as loss function and metric to evaluate our model performance. Since our NN model is simple, the training process is very efficient and takes on average 73s across all tested programs. In Section IV-B, we test the accuracy and false positive rate of our neural program models at identifying hot bytes. In addition, we evaluate the information loss in Section IV-G.

Program	Taint Sources #	Taint Sinks #	Taint Sinks Types
readelf	7467	2122	branch variables
harfbuzz	5049	2805	branch variables
mupdf	4861	1377	branch variables
libxml	8040	1929	branch variables
libjpeg	5873	997	branch variables
zlib	8306	571	branch variables
sort	500	1	length variable
openjpeg2	298	1	denominator
libsndfile	446	1	length variable
nm	847	1	counter variable
strip	1123	1	length variable

TABLE II: We set program input bytes as taint sources, and we use different types of taint sinks. For the first 6 programs, we select sink variables at program branches to evaluate NEUTAINT's accuracy, overhead, and application on taint-guided fuzzing. For the rest 5 programs, we set taint sinks according to the vulnerability information for exploit analysis.

B. Is NEUTAINT *more accurate at finding hot bytes (*i.e.*, the the most influential bytes) in input for 6 real-world programs?*

It is extremely hard to evaluate the accuracy of dynamic taint analysis. We would like to find ground truth for taint that is relevant to its applications. DTA is often used to search for important bytes in the inputs to trigger specified program behaviors. For example, vulnerability analysis needs to find which part of untrusted input triggers malicious behaviors. Taint-guided fuzzing aims to find importance bytes which explore new program behaviors and yield new code coverage. DTA finds these important bytes by setting and propagating taint labels from taint sources (program input) to taint sinks (various variables that determines program behaviors). Therefore, we propose to use these *hot bytes* to evaluate the accuracy of DTA tools and NEUTAINT.

Hot Bytes Definition. We define importance bytes that can maximally influence the variables in program branches as *hot bytes*.

Then next question is how to obtain the ground truth hot bytes as baseline to compute the hot byte accuracy. We observe that a large number of real-world programs are parsers which take in a specified file type and check its format. Meanwhile, most of the program behaviors of these parser programs are determined by bytes at the specified locations of input (*i.e.*, the fixed locations where file format headers locate), rather than the file content. Hence, for a parser program, we can approximate that the hot bytes are mostly located at the structured format sections. Further, by analyzing the fixed structured locations of a file, we can obtain the estimated ground truth of hot bytes for a particular parsing program. These ground truth hot bytes can be used as a metric to evaluate the effectiveness and efficacy of dynamic taint analysis tools on a particular parsing program.

Programs	File Format	Hot Byte Accuracy				Hot Byte FPR				Runtime			
		NEUTAINT	Libdft	DFSan	Triton	NEUTAINT	Libdft	DFSan	Triton	NEUTAINT (GPU/CPU)	Libdft	DFSan	Triton
readelf-2.30	ELF	81%	74%	49%	44%†	1.6%	3.0%	3.5%	3.7%†	6m/25m	161m	117m	>24h
harfbuzz-1.7.6	TTF	86%	86%	n/a	13%†	0.8%	0.8%	n/a	5.2%†	5m/18m	204m	n/a	>24h
mupdf-1.12.0	PDF	80%	48%	57%	33%†	2.0%	2.9%	2.6%	3.9%†	3m/9m	224m	532m	>24h
libxml2-2.9.7	XML	65%	56%	n/a	14.2%†	2.3%	3.0%	n/a	5.8%†	6m/29m	197m	n/a	>24h
libjpeg-9c	JPEG	29%	n/a	n/a	n/a	3.9%	n/a	n/a	n/a	3m/5m	n/a	n/a	n/a
zlib-1.2.11	ZIP	66%	26%	n/a	3%†	1.8%	3.9%	n/a	5.1%†	3m/4m	20m	n/a	>24h

†indicates cases where Triton analyzed partial dataset within 24 hours

TABLE III: For each program, we measure the accuracy and false positive rate of identifying hot bytes, as well as runtime overhead for different DTA tools and NEUTAINT. NEUTAINT achieves the highest accuracy and lowest false positive rate. On average, NEUTAINT increases the accuracy by 10% and reduces the false positive rate by 0.44% compared to the second-best DTA tool Libdft. NEUTAINT is $40\times$ faster on GPU and $10\times$ faster on CPU than the second-best DTA tool Libdft.

Ground Truth. Note that all evaluated real-world programs in this sections are file parsing programs. Since these 6 parsing programs have particular structures in their file formats, we can obtain estimated ground truth of hot bytes by analyzing these file formats. A saliency map of ELF file format is shown in Appendix B. Fig 2 shows the ground truth of all 6 file formats. Similar to ELF, other formats also include the header and trailers. But some files may have unique format features. For example, ZIP file has an additional local file header after the ZIP file header at the beginning. TTF file contains unique character tables near the trailers.

Extract Hot Byte. We perform the following step to extract the hot bytes from our neural program models. 1) For each program, we feed the seed input to NN model. 2) We compute the gradient of taint sinks with respect to the taint sources (seed input) and construct the saliency map using Equation 4. The saliency value indicates the extent to which the byte in taint sources affect the all the taint sink variables. 3) We select the top 5% bytes with highest values from the input saliency map as possible hot bytes using Equation 6. The reason we set this threshold is that in practice, only a small number of hot bytes in the input determine program behaviors. After analyzing all 6 file formats, we find that the total number of ground truth hot bytes range from 250 to 500 which takes around 5% of total input bytes.

Compute Hot Byte Accuracy. We compute the hot byte accuracy by checking if the hot bytes analyzed from taint tools are consist with ground truth hot bytes. To be specific, if a hot byte identified by NEUTAINT locate in the estimated ground truth range, we consider it as a true hot byte identification (*i.e.*, true positive); otherwise, we consider it as false identification (*i.e.*, false positive). For the 3 other state-of-the-art dynamic taint analysis tools, we also evaluate their abilities to find hot bytes to compare against NEUTAINT. Since dynamic taint analysis operates on a single execution trace, we run the dynamic taint analysis tools on every input in the training data and collect the tainted bytes for every execution; we then aggregate these tainted bytes by counting the total number of times a specified byte is tainted. In this way, we construct a similar saliency map as NEUTAINT for these 3 tools. We then select the same threshold 5% of top tainted bytes as possible hot bytes and calculate the hot byte accuracy. Lastly, we measure the total runtime to obtain the final hot byte accuracy for the 4 tools. For the 3 dynamic taint analysis tools, we record the total runtime for tracking all the input samples in the dataset. For NEUTAINT, we record the total runtime cost for NN training and gradient computation. Based on the best-effort ground truth for the 6 file formats, we compute accuracy and false positive rate of identifying hot bytes in a standard way.

Results. The results for hot byte accuracy is shown in Table III. For programs with simple and straightforward parsing logic such as `readelf` and `harfbuzz`, we observe that all four tools find hot bytes with high accuracy. For programs with complex parsing and transformation logic such as `libjpeg` and `zlib`, the accuracy drops for all tools due to large taint flows inside the decompression algorithm. Nonetheless, NEUTAINT achieves the highest hot byte accuracy on 5 programs. Even for the remaining program `libjpeg` which causes significant runtime overhead due to large taint propagation flows, NEUTAINT is the only one to be able to finish the analysis within a reasonable time (minutes as opposed to hours). Fig 2 shows the detailed visualisation of hot byte accuracy using a heat map.

Traditional taint propagation can result in significant slowdowns for time-sensitive operations in large real-world applications. `libjpeg`'s decompression algorithm requires massive memory read and write operations which carry taint information. Indeed, all three traditional dynamic taint analysis tools fail to finish the analysis of `libjpeg` on the dataset within 24 hours. Moreover, the second-best tool Libdft spends more than 3 minutes on `libjpeg` to finish a single execution (more than 2 days for all 993 inputs in the dataset) to track all the taint flows inside the decompression algorithms while a vanilla execution of `libjpeg` without dynamic taint analysis takes 0.015s. The runtime overhead for intense taint tracking could be more than 10^4X times while NEUTAINT avoids such heavy instrumentation overhead through a lightweight neural network model that identifies the hot bytes.

Analysis: Hot Byte Accuracy. The average hot byte accuracy for all 4 tools across all evaluated programs are $68\%, 58\%, 53\%, 27\%$. NEUTAINT achieves 10% more accuracy improvement than second-best dynamic taint tool Libdft. As for hot byte accuracy of every program, NEUTAINT achieves $7\%, 0\%, 7\%, 23\%, 40\%$ more

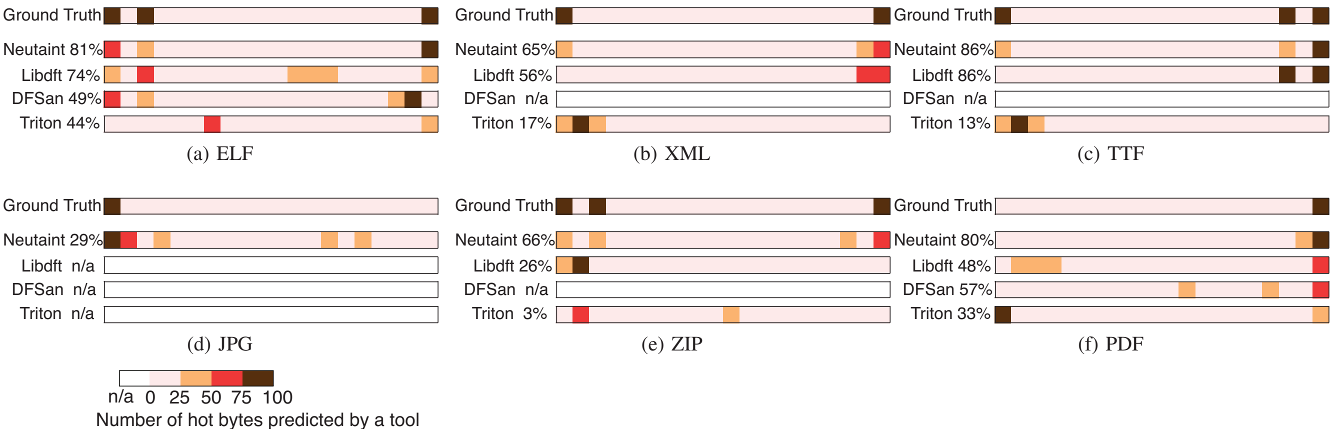


Fig. 2: The six heatmaps show how each tool identifies hot bytes for a given file format. The x-axis is broken into byte intervals, and an interval's darkness is proportional to how many hot bytes a tool predicts. Since the first row represents the ground truth, the correctness is defined by how each subsequent row aligns with the ground truth row.

improvement respectively on program `readelf`, `harfbuzz`, `mupdf`, `libxml`, `zlib`. The reason for NEUTAINT's higher accuracy is that NEUTAINT is an analysis based on program semantics by learning dynamic program logic rather than an analysis based on instruction semantics performed by traditional dynamic taint analysis tools. NEUTAINT could flexibly adapt to diverse execution context which cannot be accurately modeled by dynamic taint analysis tools through fixed, predefined taint propagation rules.

Analysis: Hot Byte False Positive Rate. For all programs, NEUTAINT achieves the lowest false positive rate at identifying hot bytes (Table III). On average, NEUTAINT has 2.07% false positive rate, that is less than half of that for Triton. Compared to the second-best DTA tool Libdft, the hot byte false positive rate of NEUTAINT is 0.44% lower. The results show that learning end-to-end program embeddings can effectively reduce the overtaint issue.

Result 1: NEUTAINT achieves the highest hot byte accuracy and lowest false positive rate in six popular file formats compared to state-of-the-art dynamic taint analysis tools. On average, NEUTAINT increases the accuracy by 10% and reduces the false positive rate by 0.44% compared to the second-best DTA tool Libdft.

C. What is the runtime overhead of NEUTAINT *compared to state-of-the-art dynamic taint analysis tools, with and without GPU?*

We measure the runtime overhead for NEUTAINT and all three dynamic taint analysis tools. We measure the total time needed to process all the inputs in our dataset, as the runtime in Table III. For NEUTAINT, the total runtime of NEUTAINT is composed of three parts, collecting program behavior data, training NN and computing the saliency map of NN.

Table III summarizes the result. Overall, NEUTAINT has the least runtime overhead for all six programs compared to other tools, since the time cost for training and computing the saliency map is negligible. To collect the training dataset, we obtain values of the sink variables in the binary through lightweight instrumentation which introduces negligible overhead. In addition, computing the saliency map is computationally efficient. Therefore, NEUTAINT enjoys the fastest runtime among all four tools evaluated. In particular, on program `mupdf`, NEUTAINT can save up to 74× runtime overhead on GPU and 24× runtime overhead on CPU than the second-best DTA tool Libdft. The average runtime overhead for the four tools are 4 mins(GPU)/15min (CPU), 161 mins, 325 mins and $>$ 24 hours. Compared to the second fastest tool Libdft, NEUTAINT on average achieves 40× and 10× smaller runtime overhead on GPU and CPU, respectively.

Among the other three tools, Triton achieves the worst result on hot byte accuracy and runtime overhead, due to PIN dynamic instrumentation and inefficient analysis routine. So we only evaluate the partial inputs from dataset on Triton within 24 hours to compute the hot byte accuracy. As for runtime of Triton, we use $>$ 24 hours to indicate the significantly large runtime overhead. DFSan achieves the second fastest analysis on program `readelf`. Its runtime overhead is smaller than Libdft because of its efficient instrumentation during compile time rather than runtime. However, DFSan fails to run on four programs because it cannot instrument external dynamically linked libraries at compile time, and it incurs large overhead for recursive tag resolution. Libdft runs faster than DFSan and Triton for all programs except `readelf`. It is also more accurate at identifying hot bytes than DFSan and Triton, in all programs except `mupdf`. However, Libdft is still 43 times slower than NEUTAINT, due to execution of heavily instrumented binary and accumulated overhead through propagation.

Result 2: The runtime overhead of NEUTAINT is 40× faster on GPU and 10× faster on CPU than the previously fastest dynamic taint analysis tool Libdft.

Ablation Studies. We break down the total runtime overhead of NEUTAINT for processing 2,000 inputs into three parts,

Program	All 2,000 Inputs Data Collection	Training GPU	 CPU	Saliency Map GPU	 CPU	Total GPU	 CPU
readelf	18s	214s	1352s	98s	119s	330s	1471s
harfbuzz	40s	115s	951s	119s	130s	274s	1080s
mupdf	30s	116s	472s	64s	82s	210s	554s
libxml	44s	216s	1652s	93s	109s	353s	1761s
libjpeg	10s	112s	251s	36s	37s	158s	288s
zlib	5s	110s	202s	26s	30s	141s	232s

TABLE IV: Runtime breakdown for NeuTaint. The table shows the total runtime of processing 2,000 inputs for data collection, training, and saliency map computation.

Program	Vulnerability Type	CVE ID
sort	buffer overflow	CVE-2013-0221
openjpeg2	integer division-by-zero	CVE-2016-9112
libsndfile	out-of-bound read	CVE-2017-14245
nm	heap overflow	CVE-2018-19931
strip	integer overflow	CVE-2018-19932

TABLE V: NeuTaint can successfully identify the information flow from source to sink in the following exploits.

Programs	File Format	Edge coverage NeuTaint	 Libdft	 DFSan	 Triton
readelf-2.30	ELF	5540	4164	2489	440†
harfbuzz-1.7.6	TTF	5395	3796	n/a	11†
mupdf-1.12.0	PDF	399	248	192	48†
libxml2-2.9.7	XML	918	428	n/a	236†
libjpeg-9c	JPEG	649	n/a	n/a	n/a
zlib-1.2.11	ZIP	200	131	n/a	54†

†indicates cases where Triton analyzed partial inputs from dataset.

TABLE VI: Edge coverage comparison of 5 taint-guided fuzzers for 24 hour time budget

data collection, training and saliency map on both GPU and CPU settings. The results are shown in Table IV. The average runtime of NeuTaint is 244s across 6 programs, 4× faster than on CPU (898s). Even on a machine with only CPU computation, the runtime overhead of NeuTaint is still 10× lower than traditional DTA tools. Since our NN model has a small number of hyperparameters, the model can be efficiently trained with and without GPU. With GPU, the training time takes around 60%. The construction of saliency map and data collections takes around 30% and 10%, respectively. With only CPU, the runtime splits into training, saliency map computation, and data collection as 88%, 9% and 3%. Using only CPU causes on average 4.5× and 0.16× slowdown in training and saliency map construction than with GPU, respectively. In general, the training time takes up the majority of total runtime. Whereas, data collection takes up the least part of runtime because the light-weight instrumentation of program only records the taint sink values during execution and introduces minimal overhead than vanilla execution.

Result 3: The total runtime for NeuTaint to process all 2,000 inputs is 244s on average for each of the six programs. Training time takes up the most part of total runtime overhead, around 60% with GPU and 88% without GPU.

D. Can NeuTaint *detect vulnerabilities in real-world programs?*

We evaluate the effectiveness of NeuTaint in the analysis of software attacks. We choose 5 known real-world vulnerabilities as listed in Table V. These vulnerabilities are all from open-sourced programs. Then we perform light-weight instrumentation on the programs and record the values for (taint source, taint sink) pairs during runtime. The 5 vulnerabilities covers various exploit types such as buffer overflow, heap overflow, integer overflow, integer division-by-zero and out-of-bound read. For each vulnerabilities, we set the taint sink at the variables which causes the vulnerability (*e.g.,* length variable used by read/write function, variable used as denominator). We also set program input as taint source for every vulnerability. To collect the training samples, we randomly flip bytes at a specified program input (the exploit) and execute the vulnerable programs with the generated input to record the corresponding taint sink values. For the 5 vulnerabilities, we generate 2K training samples for each of them. We train the neural program model which learns the mapping from taint source to taint sink. Then we feed the exploit as taint source to the neural program and construct the saliency map using Equation 5 based on the gradient of taint sink with respect to taint source. According to the saliency, we can infer which part of taint source determines the taint sink variables. The key result is that NeuTaint successfully locates the hot bytes which control the taint sink variables, and thus can reason about the influence from taint source to taint sink.

Result 4: NeuTaint can successfully find the information flow from source to sink in known CVEs.

E. Since taint-guided fuzzing is one of the most important security applications of taint, does NeuTaint *help taint-guided fuzzing achieve better edge coverage compared to other taint analysis tools?*

In this section, we compare the performance of all tools when applied to one of the most important security applications, taint-guided fuzzing.

Fuzzer Backend Implementation. Dynamic taint analysis has been used as the front end by many fuzzers to identify hot bytes for guiding further mutations. [62] [40] [27] [49] [16]. Although these fuzzers leverage dynamic taint analysis tools to find hot bytes, each of them applies different algorithms to mutate hot bytes. For example, Vuzzer [49] copies the magic number extracted from binary directly to the locations of hot bytes; and Angora [16] implements gradient descent along with other strategies to mutate these hot byte locations. To eliminate the effects of different searching strategies and different execution backends of fuzzing modules, we build a simple and efficient fuzzer backend in C (shown in Appendix C). For each dynamic taint analysis front end, we use a common backend that generates new inputs based on mutations of the hot bytes. Therefore, we have a total of four fuzzer prototypes using the same mutation algorithm as shown in Algorithm 1. The metric we use for comparison is the edge coverage achieved from the different front ends.

Edge Coverage Comparison. We run each fuzzer prototype for 24 hours on the 6 programs shown in Table VI. We have discussed the evaluation of hot byte accuracy for these programs in Section IV-B. Each fuzzer is given the same initial seed corpus for each program and assigned a single CPU. Each dynamic taint analysis front-end works with the back-end fuzzer to guide the mutation. Since Triton incurs a significantly large runtime overhead greater the our evaluation time, Triton has only analyzed partial inputs in 24 hours. The results are shown in Table VI. Neutaint achieves the highest edge coverage for all 6 program evaluated. On average, Neutaint reaches 61% more edge coverage than the second best analysis front end Libdft. Triton is the worst analysis front end due to its extremely large runtime overhead and lowest hot byte accuracy. DFSan front end could only run on two programs and achieves intermediate results. To summarize, the taint-guided fuzzing results further validate that Neutaint obtains the most accurate hot bytes among the four taint analysis tools in real-world applications. The consistency of taint-guided fuzzing results and hot byte accuracy evaluation (Section IV-B) demonstrates the efficiency and effectiveness of Neutaint.

Result 5: Neutaint achieves 61% more edge coverage than other dynamic taint analysis tools for taint-guided fuzzing, demonstrating that the taint information obtained from Neutaint is more effective.

F. How does Neutaint *perform with different machine learning models other than neural networks?*

In this section, we compare other machine learning models (*e.g.,* logistic regression and support vector machines (SVM)) against neural network for the implementation of Neutaint on the same set of real world programs. We also use the hot byte accuracy and FPR as mentioned in Section IV-B to evaluate the performance of different machine learning models.

Logistic Regression: We implement the logistic regression using the same NN architecture, but without any non-linear activation in hidden layers. We use sigmoid as final layer and train with the same setting as mentioned in Section IV-A. To extract the hot byte information, we perform the gradient analysis routine as mentioned in Section IV-B.

SVM models: We implement the SVM models (linear, polynomial kernel, and RBF Gaussian kernel) using the scikit-learn library [9]. Our dataset has a large number of output labels (*i.e.*, up to 2K sink variables for each input). The runtime overhead is large for SVM model on such dataset because it uses a simple one-vs-one scheme. Therefore, we leverage the correlations between labels and encode the large number labels into a smaller and compact ones. To obtain the importance of each input feature, we compute the dot product of all weights associated with each input feature to the final model outputs. Bigger weights mean that the output is more sensitive to the change of the corresponding feature.

Results: Table VII shows the hot byte accuracy and false positive rate from the five different ML models. Neural network model achieves the best results on all 6 programs, on average 68% hot byte accuracy. Among the four other machine learning models, SVM with polynomial kernel is the best model, achieving on average 30.5% hot byte accuracy. Logistic model achieves the second-best results on programs `libjpeg` and `zlib`. SVM with polynomial kernel is the best-performed SVM model, achieving the second-best results on programs `harfbuzz` and `readelf`. SVM with linear kernel achieves the second-best result on program `libxml` and SVM with RBF Gaussian kernel achieves the second-best result on program `mupdf`. The reason for neural network's superior performance is that neural network has a large model capacity such that it fits diverse datasets well. Moreover, unlike SVM, neural network can naturally support datasets with a large number of labels.

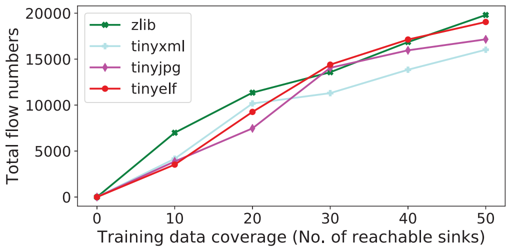


Fig. 3: Total number of flows detected by Neutaint when training dataset coverage increases. Training data with higher coverage can increase the flow coverage of Neutaint.

Result 6: The neural network model achieves on average 68% hot byte accuracy and 2.07% FPR, the best among five machine learning models.

G. What kinds of flows are missed by Neutaint*? How does the training data quality affect such information loss and how to mitigate this loss?*

Flow Definition: Before describing our techniques to measure the information loss, we formally define the flows. The goal of DTA is to detect flows based on dynamic execution. Therefore, we collect a ground truth dataset that contains the total number of flows based on *unseen* test inputs. We define one flow as a tuple of (input value, source, sink), and the total flows are collected from all test inputs. To measure the information loss, we evaluate how many flows Neutaint and dynamic taint analysis tools can detect out of the total flows from the ground truth.

Flow Dataset: Since we need the ground truth of exact number of total flows as baseline, we choose four programs (including three tiny programs [12], [10], [11], and one small real world program) where such information can be obtained reliably through static analysis and manual inspection. For each program, we choose 50 sink variables in conditional predicates, such as file magic bytes, field values and offsets. We obtain these sink values through light-weight instrumentation and normalize them into binary data as mentioned in Section IV-A. These sink variables are commonly used to perform conditional checking that determines program behaviors. If a particular input can reach 20 sinks, we count 20 flows for that input. To collect the dataset, we randomly flip bytes of a specified input using a simple fuzzer, generating

Program	Hot Byte Accuracy					Hot Byte FPR				
	NN	Logistic	SVM(linear)	SVM(poly)	SVM(rbf)	NN	Logistic	SVM(linear)	SVM(poly)	SVM(rbf)
readelf	81%	38.6%	23.8%	47.4%	27.4%	1.6%	4.2%	5.2%	3.6%	4.9%
harfbuzz	86%	19.8%	26.3%	56.7%	28.7%	0.8%	4.8%	4.4%	2.6%	4.2%
mupdf	80%	13%	14%	14.8%	17.4%	2%	5%	5%	4.9%	4.8%
libxml	65%	34.8%	47%	42.3%	5.7%	2.3%	4.3%	3.5%	3.9%	6.3%
libjpeg	29%	7.3%	5.9%	7%	1.4%	3.9%	7.3%	5.1%	5%	5%
zlib	66%	44%	7.1%	15%	10%	1.8%	3%	4.8%	4.4%	4.7%

TABLE VII: NEUTAINT performance on different ML models. The neural network model achieves on average 68% hot byte accuracy and 2.07% FPR, the best among five machine learning models.

Program	Total Flows (Ground Truth)	NEUTAINT	Libdft	Triton	DFSan
TinyELF	19,464	19,046	7,227	18,048	5,120
TinyJPG	17,188	17,160	11,439	17,184	15,510
TinyXML	17,036	16,691	15,233	7,671	14,720
Zlib	19,957	19,804	18,043	14,743	14,322

TABLE VIII: Comparison of information flow losses of different taint tracking tools on three tiny programs and one real world program. We use static analysis and manual examination to estimate the number of total flows.

6K inputs which cover all the 50 sink variables. Then we split the dataset into 5K training inputs and 1K testing inputs. Both training and testing datasets can cover all 50 sink variables. We achieve on average 99% testing accuracy. After training the neural program, we perform the gradient analysis as mentioned in Section IV-B to reason if a sink variable is tainted or not. If the gradient value for a sink variable is greater than a specified threshold, it is considered as tainted.

Information Loss: All dynamic taint analysis tools suffer from information loss, since they cannot track all flows in a program. The information loss of NEUTAINT can be categorized into two classes.

- Coverage of training dataset. When the training data do not cover all the sink variables that appear in the testing data, there could be information loss on NEUTAINT.
- Inaccuracy of machine learning model. No model is 100% accurate on unseen testing data. The information loss happens when the neural network model makes wrong predictions to unseen testing data.

To investigate the information loss caused by training data coverage (*i.e.*, the number of sink variables covered by training inputs), we downsample the 5K training inputs into five subsets with different coverage threshold. Each subset covers a different number of sink variables from 10, 20, 30, 40, to 50. Then we train NEUTAINT with each subset and evaluate the total number of flows detected on the 1K *unseen* test inputs. The result is shown in Fig 3. The total number of flows detected by NEUTAINT increases as the training data coverage increases. When training data covers all the sinks, NEUTAINT can detect the highest number of flows in the unseen testing dataset.

Furthermore, we evaluate the information loss caused by the inaccuracy of the neural network model. Specifically, we compare the number of flows detected by NEUTAINT against three state-of-the-art DTA tools on the four programs. We obtain the ground truth (*i.e.*, total number of flows) from all the testing inputs. The result is shown in Table VIII. On average, NEUTAINT detects 98.7% flows, the highest among all tools. Triton, Libdft, and DFSan detect on average 78%, 70.9% and 69% flows, respectively. The 1.3% loss of NEUTAINT is due to model inaccuracy.

Advantages of NEUTAINT: The reason for NEUTAINT's superior performance is that NEUTAINT can detect information flows passed through some complex code such as external library calls strcmp() and strncmp(). It is hard for DTA tools to propagate the taint tags through the library calls. Moreover, NEUTAINT also detects some implicit control flow dependency which is not supported by common DTA tools. We will cover more details of these cases in Section V and Appendix D. NEUTAINT achieves the best result on 3 programs (`TinyELF`, `TinyXML`, and `Zlib`) among all four tools and the second best on `TinyJPG`. For the program `TinyJPG`, Triton finds slightly more flows than NEUTAINT. `TinyJPG` does not contain any external library calls or implicit control dependency in selected taint sinks, while NEUTAINT could make some minor mistakes when approximating sink variables in unseen inputs. Lifdft performs the worst on the program `TinyELF`, because most functions in `TinyELF` pass parameters through a float point instruction MOVSD which is not supported by Libdft. Compared to other tools, DFSan finds the least number of flows. The results show that information loss is common for all tested tools.

How to improve the coverage of training data for NEUTAINT? To mitigate the information loss, we can add more training data to reach more sink variables. Using a fuzzer to generate data with coverage guidance is more helpful than only randomly flipping the input without the guidance. In addition, we can use existing techniques such as symbolic execution to generate training data with high quality. Finding a new path in any form of DTA is the hard problem. Once we find one path between source and sink, we can generate many more paths via input mutation. Though training the neural program requires at least one path between a source and a sink, we can achieve lower false positive rate and higher accuracy than rule-based traditional DTA tools.

Result 7: Training data with higher coverage increases the flow coverage of NEUTAINT. On average, NEUTAINT detects 98.7% flows which is 20% more than the second-best tool Triton.

V. Undertaint Case Study

In this section, we present a case to explain why Neutaint is more accurate than traditional dynamic taint analysis. Specifically, Neutaint tracks implicit information flows and avoids under taint in real world programs. More examples can be found in Appendix D.

Example: Implicit Information Flows. Most DTA tools ignore implicit information flows (*i.e.*, implicit control dependency and complex external library calls) and only support explicit information flow in data-dependency form. The reason is that supporting implicit information flows could cause high runtime overhead and false positive rate [32]. Lack of support to these implicit information flows often result in under-taint issue on some real world programs. We discuss the implicit control dependency in a popular tiny program `TinyXML` [12]. As shown in List 1, `p` is a input buffer which stores program input as taint source. At line 16, `ele->ClosingType()` is the sink variable determining the program branching behavior. At line 6, `ele->ClosingType()` can be modified to a constant value when there is a special character in the input buffer. Since `ele->ClosingType()` is control-dependent but not data-dependent on taint source input buffer `p`, DTA tools fail to track the flows to such sink variables. In the second example, we consider such implicit control dependency in complex external function calls. Many sink variables are the return values of complex external function calls such as strcmp() and strncmp(). The return values are state variables which are control-dependent on the taint source, but not explicit data dependent. Therefore, DTA tools would lose taint information for such sink variables.

```
 1 // tinyxml2/tinyxml2.cpp:1066
 2 while(p){
 3   ...
 4   if( *p == '/' )
 5   {
 6     /* implicit control flow dependency*/
 7     ele->ClosingType() = CLOSING;
 8     ++p;
 9   }
10 }
11
12 char* XMLNode::ParseDeep(...)
13 {
14   ...
15   /* under-taint */
16   if(ele->ClosingType() == XMLElement::CLOSING)
17   {
18     ...
19   }
20   ...
21 }
```

Listing 1: Implicit control dependency in **TinyXML**

Neutaint Solution. Neutaint avoids this problem by directly learning the mapping from taint sources to any taint sink variables (including both data-dependency and control-dependency variables). Compared to traditional dynamic taint analysis tools, Neutaint has the advantage of generalizing to various real-world programs.

VI. Related Work

Recently, many works [14], [47], [24], [46], [50] have used machine learning for different program analysis tasks such as program synthesis [45], vulnerability detection [28], [43], [39], [35], [30], [33], [17], program repair [60], [26], fuzzing [48], [23], [42], [36], [29], [52], and symbolic execution [53].

Dynamic taint analysis [41] [15] [64] [32] [38] executes programs with concrete inputs to perform the analysis. However, it incurs large overhead and suffers from overtaint and undertaint issues. To address these issues, TaintInduce [18] proposes to learn platform-specific taint propagation rules from (input, output) pairs of instructions. Their approach learns propagation rules based on a template, and uses an algorithm to reduce the task to learning different input sets and pre-conditions for propagating the taint tags. TaintInduce increases the accuracy for individual propagation rules, but they still suffer from accumulated errors and large overhead due to propagation-based design. On the contrary, Neutaint uses machine learning technique to track end-to-end information flow. We use light-weight instrumentation to build neural program embeddings, and directly analyze the flow of information captured by the neural network models. Our technique minimizes end-to-end information flow tracking errors and significantly reduces runtime overhead.

VII. Conclusion

We present a novel approach Neutaint to perform taint analysis using neural program embeddings. Our neural program learns the information flow directly from taint sources to taint sinks. We use saliency maps to analyze the information flow in the neural programs. To evaluate the accuracy, overhead, and application utility of Neutaint, we compare against three state-of-the-art dynamic taint analysis tools. The results show that Neutaint achieves on average 10% increase in accuracy and 40 times less runtime overhead over the second best dynamic taint analysis tool Libdft. Neutaint can also successfully track the information from source to sink in exploits. We further evaluate Neutaint through a popular taint application–fuzzing. The taint-guided fuzzing results demonstrate that Neutaint can achieve on average 61% more edge coverage than state-of-the-art dynamic taint analysis tools.

Acknowledgement

We thank Mingshen Sun, our shepherd Mathias Payer and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670, CNS-16-18771, CCF-16-19123, CCF-18-22965, CNS-19-46068; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; a NSF CAREER award; a Google Faculty Fellowship; and a Capital One Research Grant. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Google, or Capital One.

References

[1] Libdft with support of multiple taint. https://github.com/m000/dtracker.

[2] *Triton: A Dynamic Symbolic Execution Framework*. SSTIC, 2015.

[3] Triton with support of multiple taint tags. https://github.com/bntejn/Triton/tree/dev-tagging-taint, 2017.

[4] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2018.

[5] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html, 2019.

[6] Keras: The python deep learning library. https://keras.io/, 2019.

[7] An open source machine learning framework for everyone. https://www.tensorflow.org/, 2019.

[8] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2019.

[9] scikit-learn Machine Learning in Python. https://scikit-learn.org/stable/, 2019.

[10] TinyELF. https://github.com/TheCodeArtist/elf-parser, 2019.

[11] TinyJPG. https://github.com/cdcseacave/TinyEXIF, 2019.

[12] TinyXML2. https://github.com/leethomason/tinyxml2, 2019.

[13] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[14] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. 2016.

[15] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[16] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 711–725, 2018.

[17] M.-J. Choi, S. Jeong, H. Oh, and J. Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, 2017.

[18] Z. L. Chua, Y. Wang, T. Blu, P. Saxena, Z. Liang, and P. Su. One engine to serve'em all: Inferring taint rules without architectural semantics. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[19] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. 2007.

[20] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[21] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[22] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, May 2009.

[23] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[24] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. 2014.

[25] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, 2017.

[26] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017.

[27] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[28] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, MetriSec '12, 2012.

[29] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu. Ganfuzz: A gan-based industrial network protocol fuzzing framework. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018.

[30] X. Huo, M. Li, and Z.-H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.

[31] M. G. Kang, S. McCamant, P. Poosankam, and D. X. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[32] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[33] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[34] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204. ACM, 2013.

[35] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[36] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*, 2018.

[37] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[38] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, New York, NY, USA, 2016. ACM.

[39] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, 2016.

[40] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The BORG: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[41] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[42] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard. Faster fuzzing: Reinitialization with deep neural models. *arXiv preprint arXiv:1711.02807*, 2017.

[43] Y. Pang, X. Xue, and A. S. Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 2015.

[44] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. *2016 IEEE European Symposium on Security and Privacy*, 2016.

[45] E. Parisotto, A. rahman Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

[46] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1093–1102, 2015.

[47] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. Sk_P: A Neural Program Corrector for MOOCs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016.

[48] M. Rajpal, W. Blum, and R. Singh. Not All Bytes Are Equal: Neural Byte Sieve for Fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[49] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-Aware Evolutionary Fuzzing. In *Proceedings of the 2008 Network and Distributed Systems Security Conference*, 2017.

[50] S. E. Reed and N. de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2015.

[51] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th*

Conference on USENIX Security Symposium, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[52] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, , and S. Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019.

[53] S. Shen, S. Ramesh, S. Shinde, A. Roychoudhury, and P. Saxena. Neuro-symbolic execution: The feasibility of an inductive approach to symbolic execution. 2019.

[54] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[55] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, New York, NY, USA, 2009. ACM.

[56] M. Sun, T. Wei, and J. C. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS'16, 2016.

[57] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[58] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[59] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[60] K. Wang, R. Singh, and Z. Su. Dynamic neural program embedding for program repair, 2017.

[61] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, Washington, DC, USA, 2010. IEEE Computer Society.

[62] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2010.

[63] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[64] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS)*. ACM, 2007.

APPENDIX

A. NN Architecture

We use the neural network architecture shown in Figure 4 to learn neural program embeddings. The taint sources are NN inputs and taint sinks are NN outputs. The network has one hidden layer with ReLU activations and one output layer with sigmoid activations.

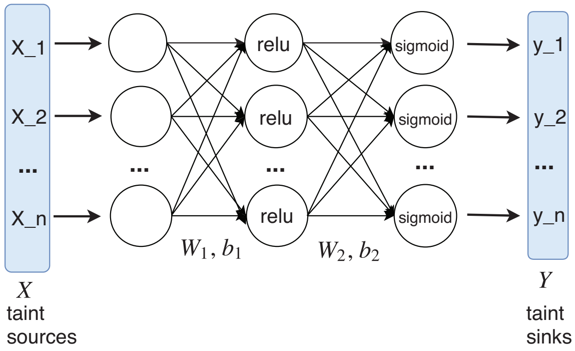


Fig. 4: The Neural Network architecture we use to generate dynamic program embeddings.

B. Saliency Map for ELF File Format

The ELF file format can be broken down into four main regions, as shown in Fig 5. Three of them are header information (ELF, Program Header Table, and Section Header Table) and the other one is for non-header information (*e.g.*,.text, .data, .bss). For a common ELF file parser (`readelf`), the main parsing logic focuses on these shaded header regions and typically ignores the non-header information. Therefore the hot bytes should locate at the these header regions.

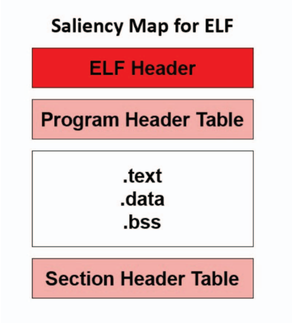


Fig. 5: Saliency Map of user input on program **`readelf`**. The darkness of the color corresponds to the influence of the region's bytes on the taint sinks. The ELF Header field has the largest saliency values.

C. Fuzzing Algorithm

Our fuzzing algorithm is shown in Algorithm 1. The algorithm takes in hot bytes locations from taint tools and perform deterministic mutations on these hot bytes. Random mutations are discarded to ensure that the fuzzer performance is only affected by the quality of hot bytes identified by the different dynamic taint analysis front ends.

Algorithm 1 Our mutation algorithm for taint-guided fuzzing that focuses on influential bytes.

```
Input:  $seed \leftarrow$ initial seed
        $iter \leftarrow$ number of iterations
        $hot\_bytes \leftarrow$ hot bytes from taint tools
for $i = 1$ to $iter$ do
    $locations \leftarrow top(hot\_bytes, (2^i))$
    for $m = 1$ to $255$ do
        for $loc \in locations$ do
            $v \leftarrow seed[loc] + m$
            $v \leftarrow clip(v, 0, 255)$
        end for
        $gen\_mutate(seed, loc, v)$
        for $loc \in locations$ do
            $v \leftarrow seed[loc] - m$
            $v \leftarrow clip(v, 0, 255)$
        end for
        $gen\_mutate(seed, loc, v)$
    end for
end for
```

D. *Case Study*

In this section, we present case studies to explain why Neutaint is more accurate and has lower runtime overhead than traditional dynamic taint analysis tools.

1) Undertaint: When a taint analysis tool fails to track all the taint labels for a specified variable, it is considered as under-taint. Under-taint is a common issue for dynamic taint analysis. Since the taint propagation rules in dynamic taint analysis tools are neither sound nor complete, some taint labels are easily missed during the analysis process [18].

Example: Pointer Taint. In the code example 2, we demonstrate the classic pointer taint dilemma in a popular XML parser library `libxml`, where we track the taint flow from program input to `NXT(len)` (line 7 and 8). From the prior execution context, variables `ctxt->cur` and `len` are all affected by program input taint source and therefore carry the taint label of the taint source. The example shows a function, `xmlXPathComPathExpr()`, that is frequently used by the library to parse the path expression of a XML element. It uses `len` as the index to check the one-character operator of a path expression at line 7 and 8, through the byte reading macro `NXT(val)` defined in line 2. After taking `len` as the offset to the current pointer location `ctxt->cur`, `NXT()` returns the byte located at the address `ctxt->cur + len`. The propagation rules state that the byte memory is only affected by a single byte read from the memory content `NXT(val)`, not by the base address `ctxt->cur` and `len`. However, these addresses determine the byte memory content, which are missed due to pointer under-taint. In practice, the taint flow from pointer to memory content is intentionally ignored by most taint analysis tools as handling them could easily cause many false positives and even a taint explosion.[55] In contrast, Neutaint can capture such information flow from pointer to program behavior. Specifically, Neutaint models the function mapping from program input to branch variables at line 7 and 8. Then, by learning the differences among program behavior triggered by various input samples at line 7 and 8, Neutaint can infer input bytes that reach line 10. The advantage of our method over traditional taint analysis is that it is based on the knowledge learned from a summary of runtime program semantics rather than fixed taint propagation rules.

```
1  // libxml2-2.9.7/xpath.c:10736
2  #define NXT(val) ctxt->cur[(val)]
3
4  static void xmlXPathCompPathExpr(...)
5  {
6    ...
7    if((NXT(len) == ’<’) || (NXT(len) == ’>’)
8    || (NXT(len) == ’=’))
9    {
10     lc = 1;
11     break;
12   }
13   ...
14 }
```

Listing 2: Pointer under-taint in **`libxml`**

Example: Incomplete Taint Source. Incomplete taint source identification can also cause severe under-taint issue in real-world applications. Dynamic taint analysis tools identify the taint source by installing some hooks to system calls `open()`, `read()`, `mmap()` and setting taint marks on corresponding memory/registers. These predefined interception procedures usually rely on developers' experience to cover some common cases. However, real-world applications have diverse and complex IO procedures. The simple system call hooks in dynamic taint analysis tools may fail to fully capture all the taint sources and lose track of taint information at the beginning of program execution. For example, a popular XML parser library `libxml` supports uses compressed IO interface `gzread()` by default for both compressed and uncompressed input, which a lot of tools are not aware of. In this case, a state-of-the-art dynamic taint tool libdft would lose track of partial taint source and result in a severe under-taint problem. To make the matter worse, many different under-taint reasons can co-occur at different parts of the program.

Neutaint Solution. Neutaint avoids this problem by not relying on any human engineered system call hook procedures. It directly uses a neural network model to learn the mapping between taint sources and taint sink variables. Compared to traditional dynamic taint analysis tools, Neutaint has the advantage of generalizing to various real-world programs.

2) Overtaint: Over-taint occurs when dynamic taint analysis marks irrelevant taint labels on specified variables. Dynamic taint analysis tools define over-approximated taint propagation rules for certain types of instructions, making it hard to track the precise taint flow. Since the taint labels are propagated at the instruction level and often ignore the semantic of the program context, the over-taint issue is inevitable. Furthermore, in a real-world program, a single over-taint label could instantly propagate to generate many over-taint labels during a repeating operation (*e.g.,* a loop or recursive function call). This over-taint issue pollutes further execution and analysis results.

```
// zlib-1.2.11/inffast.c:50
void ZLIB_INTERNAL inflate_fast(strm, start)
{
  ...
  for(...)
  {
    hold += (unsigned long)(*in++) << bits;
    bits += 8;
    hold += (unsigned long)(*in++) << bits;
    bits += 8;
    ...
    /* decoding match distance */
    if(dodist)
    {
      dist=(unsigned)hold & ((1U << op) - 1);
      if(dist > dmax) {...}
    }
    ...
    hold >>= (bits + 16); // over-taint
  }
  ...
}
```

Listing 3: Over-taint propagation in **`zlib`**

Example. Error Accumulation. As shown in code 3, `inflate_fast()` is a decoding function in a `zlib` decompression procedure, where we track the taint flow from input buffer `in` (line 7 and 9) to the distance variable `dist` (line 16). The function reads compressed data from input buffer `in` and decodes the distance variable `dist` for further deflation. Going through the for loop, every time two bytes are read from `in` buffer (lines 7-10) and stored into the bit accumulator `hold`, then the content in `hold` are dropped at line 19. After a few iterations, the condition at line 13 is satisfied and the function starts to decode the match distance variable `dist`. So `dist` is only affected by the two new bytes in `hold` that are read from the current round. The over-taint occurs at line 19 when `hold` drops bits from the previous round through a shift operation, but the taint rules did not drop the labels. Since no taint propagation rule can be general enough for different program semantic, dynamic taint analysis tools use a conservative taint rule to copy all the taint labels from source (`hold`) to destination (`hold`) for shift instructions. For every round in the loop, `hold` obtains two byte taint labels while still keeping the old taint labels from the previous round. The total taint labels of `hold` were accumulated through the loop, then spread to `dist` that determines the conditional program behavior at line 16. The over-taint propagation issue stems from the inaccurate taint propagation rules pre-defined by experts. It is impossible to design fixed taint propagation rules to accurately handle all the real-world cases.

NeuTaint Solution. NeuTaint directly models the mapping from program input to conditional program behaviors at line 16. As long as there exist samples that cover different branches at line 16, NeuTaint can easily infer the two critical bytes of program input that affect `dist`.

3) Overhead: Traditional dynamic taint analysis tools track the taint labels by instrumenting the execution of the program. Specifically, for every executed instruction, the taint analysis tool calls a corresponding taint propagation handler to process the taint labels associated with the operation. Most instructions trigger calls to these handlers to track the taint flow with very few exceptions (*e.g.,* push, pop and ret), leading to a large runtime overhead. The runtime overhead of taint propagation is also affected by the granularity of the taint tag. Marking taint tags for each input on the coarse binary taint tag granularity already incurs significant overhead, not to mention the overhead to mark each input offset. A complex dynamic taint analysis task such as fuzzing requires more fine-grained taint labels which represents different offsets of user input. Tracking numerical taint labels incurs more runtime overhead than simple binary taint labels due to the fact that a basic union operation over two tainted source tags with respectively m and n offsets is O(mn) time complexity, rather than O(1) on two binary taint labels. As a result, adopting fine-grained taint labels drastically increases the runtime overhead of taint propagation handlers.

Example. Real-world programs can have many time-critical operations that are frequently executed. If every instruction of these time-critical procedures needs to call additional taint propagation handler, then the runtime overhead would be significantly large. As shown in code 4, `decode_mcu()` is a common function used in JPEG parser library `libjpeg`, where we track taint flow from program input to DC coefficients `block`. The function uses a for loop to decode Huffman-compressed coefficients, by repeatedly calling a function macro `HUFF_DECODE` at line 17. Since `HUFF_DECODE` is a time-critical operation, it is implemented as inline-macro for better performance. However, most operations in `HUFF_DECODE` involve taint propagation. Starting from line 5, the macro reads a byte `c` from input buffer, then performs a binary OR between `get_buffer` and `c` at line 6. So `get_buffer` takes the union of taint labels for `c` and itself. At line 7, the global variable `get_buffer` is used to perform huffman decoding which requires to propagate the taint label from `get_buffer` to `result`. Repeated calls to the macro `HUFF_DECODE` all involve taint label propagation, causing extreme runtime overhead. These taint labels finally propagates into result variable `s` at line 20, which would be intensively used during later decoding procedure and introduce even more runtime overhead. Our experiments show that a state-of-the-art dynamic taint analysis tool has more than 10,000X runtime overhead than normal execution on libjepg.

NeuTaint Solution. As mentioned in last section, NeuTaint is a black-box analysis that does not need to track every instruction, enabling it to avoid large runtime overheads.

```
1  // jpeg-9c/jdhuff.c:1197
2  #define HUFF_DECODE(result, ...) \
3  {
4    ...
5    c = read_byte(); \
6    get_buffer = get_buffer | c; \
7    result = huff_decode(get_buffer, ...); \
8    ...
9  }
10
11 bool decode_mcu(j_decompress_ptr cinfo, ...)
12 {
13   ...
14   for(...)
15   {
16     ...
17     HUFF_DECODE(s, br_state, htbl, ...);
18     ...
19     /* Output the DC coefficient */
20     (*block)[0] = (JCOEF) s;
21     ...
22   }
23   ...
24 }
```

Listing 4: Extreme runtime overhead in **`libjpeg`**. In Line 17, macro `HUFF_DECODE` is repeatedly called, which involves expensive taint label propagation, causing extreme runtime overhead.