# Neural Reverse Engineering of Stripped Binaries

**Anonymous authors**
Paper under double-blind review

## Abstract

We address the problem of reverse engineering of stripped executables which contain no debug information. This is a challenging problem because of the low amount of syntactic information available in stripped executables, and due to the diverse assembly code patterns arising from compiler optimizations. We present a novel approach for predicting procedure names in stripped executables. Our approach combines static analysis with encoder-decoder-based models. The main idea is to use static analysis to obtain enriched representations of *API call sites*; encode a *set of sequences* of these call sites by traversing the Control-Flow Graph; and finally, attend to the encoded sequences while decoding the target name. Our evaluation shows that our model performs predictions that are difficult and time consuming for humans, while improving on the state-of-the-art by 20%.

## 1 Introduction

Reverse Engineering (RE) of compiled binary executables has a variety of applications. Furthermore, it is crucial for analyzing malware and finding vulnerabilities. Unfortunately, it is a tedious and time consuming task, which is usually performed *manually*. A human reverse-engineer has to *guess* the procedures to begin with; follow the flow in these procedures; find connections between procedures; and finally piece all these together to develop a global understanding of the purpose and usage of the inspected executable.

Recently, there has been a lot of work on analysis of source code using learned models (Raychev et al., 2015; Bielik et al., 2016; Allamanis et al., 2018; Alon et al., 2019a; Murali et al., 2017; Brockschmidt et al., 2019; Yin and Neubig, 2017; Chen et al., 2018). However, all of these address high-level and syntactically rich programming languages such as Java, C# and Python, and none of them address the unique challenges residing in executables. He et al. (2018) proposed a non-neural model for reasoning about binaries, but the model suffered from inherent sparsity.

We present a novel approach for reasoning about *compiled assembly code*. Specifically, we use static analysis to locate and analyze external API calls; we traverse the Control-Flow Graph (CFG) to approximate the dynamic runtime order of the calls; and finally, we decode the caller procedure name while attending to potential runtime sequences. Our approach provides an interesting and powerful balance between the program analysis effort required to obtain the representation from executables and the effectiveness of the learning model. To the best of our knowledge, this is the first work to leverage deep learning for recovering procedure names in binary code.

We compare our approach empirically with previous work which used shallow static analysis with non-neural models. We show that training Neural Machine Translation (NMT) baselines on the flat assembly code performs poorly – this necessitates leveraging semantic analysis to learn from executables. We ablate our model to demonstrate the importance of the different components of our analysis to the quality of the representation. We show that our combined approach of static analysis to enrich neural representations yields more accurate results than previous work and presents a major step in the field of neural reverse engineering.

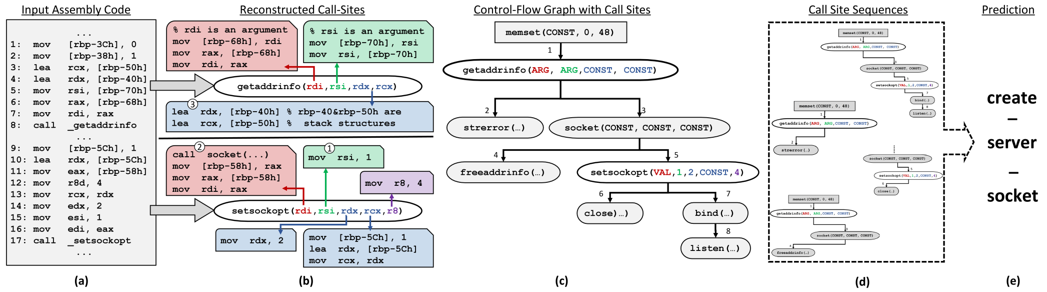


Figure 1: (a) Given assembly code of an executable, (b) we reconstruct calls to external APIs and deduce their argument kinds; (c) these external calls are placed in the reconstructed Control-Flow Graph which approximates their runtime call order; (d) our model learns all potential runtime call orders; and (e) the decoder generates the target name while attending to all potential call orders.

## 2 Enriched Representations for Binary Call Sites

### 2.1 Task: Reverse Engineering as Generation of Binary Procedure Names

Assembly code from a stripped (containing no debug information) executable is a sequence of instructions lacking variable names (see, for example, the disassembled block in Figure 1(a)). Learning such a low-level stripped representation is a challenging task. A naïve approach where the sequence of instructions is fed into a seq2seq architecture (Luong et al., 2015; Sutskever et al., 2014; Cho et al., 2014) yields hopelessly imprecise results (as we show in Section 5).

Given a nameless assembly procedure $\mathcal{X}$ residing in a stripped executable, our goal is to predict a likely and descriptive name $\mathcal{Y} = y_1 ..., y_m$, where $y_1 ..., y_m$ are the subtokens composing $\mathcal{Y}$. Thus, our goal is to model $P(\mathcal{Y} \mid \mathcal{X})$. For example, for the name $\mathcal{Y}$ = `create_server_socket`, the subtokens $y_1 ..., y_m$ that we aim to predict are `create`, `server` and `socket`, respectively (Figure 1(e)).

### 2.2 Overview

We draw our initial intuition from the way a human reverse engineer skims the code of an unknown procedure $P$, stripped from debug symbols. Disassembling the code of $P$ from hexadecimal values into assembly instructions results in a flow of instructions such as the ones shown in Figure 1(a). To understand what an assembly code snippet does, the most informative pieces of information are *calls to procedures* that were dynamically linked to the examined executable, e.g., `call getaddrinfo` (line 8) and `call setsockopt` (line 17) in Figure 1(a). The API names `getaddrinfo` and `setsockopt` *cannot be stripped* without breaking the loading process; stripping them would leave the executable unusable. While some malware may obfuscate the API *names*, the *values* for the arguments passed when calling these external procedures must remain intact. Similarly, if the examined executable is a library, procedures *exported* by the library cannot be stripped either.

To reason about compiled binaries, we need to: (i) encode each API call while capturing as much semantic information as available; and (ii) learn all API calls in a way that reflects the relationship between them. To encode API calls ((i)), we perform a static analysis that reconstructs information regarding the values, types and origin of the argument passed to the called API (Figure 1(b)). To learn the relationship between multiple encoded API calls ((ii)), we reconstruct and traverse the Control-Flow Graph (CFG) of the procedure (Figure 1(c)). Reconstruction of CFG allows to observe the potential *chronological order* in which the APIs are used, rather than the random order in which they appear in the executable. By observing the API calls *in the order they appear in the CFG*, we approximate the order in which the APIs may be used in runtime, without actually running the procedure. The alternative of observing the calls in the order they appear in the assembly loses their functional order, because the assembly order is arbitrary. Finally, our decoder generates the target procedure name while attending to all potential runtime call orders (Figure 1(d)). In this example, our model predicted the name `create server socket` (Figure 1(e)).

### 2.3 Extracting Call Sites from Binaries

***Reconstructing Call Sites*** First, we analyze each external `call` instruction to gather all available semantic information. We examine debug symbols for libraries used by the given executable to retrieve the number of arguments passed to each procedure. We map each argument to the register used to pass it, and construct a call site that is very similar to a call site in higher-level languages, as shown in Fig. 1(b).

***Using Pointer-Aware Slicing to Determine Kinds*** While the reconstructed call site structure is useful, we strive to gather information about the *value* each register holds when the call is made. Obtaining this information requires analyzing the calls in the context of the entire procedure. To do so, we compute a static *slice* of each register at the call location in the procedure. A slice of a program (Weiser, 1984) at a specific location for a specific value is a subset of the instructions in the program necessary to create the value at this specific location. As some arguments are pointers, we perform a *pointer-aware* static program slice, adapting the definition of Lyle and Binkley (1993) to assembly instructions. We generate slice information according to the specifications of the CPU manufacturer, e.g., Intel x64.

Each register of a call site in Figure 1(b) is connected by an arrow to the slice of $P$ used to create its value: for example, the value of `rsi` for the `setsockopt` call ① was created by an assignment of a constant value of 1. In other cases, the value of a specific register is unknown. One such case is the value of `rdi` passed to `setsockopt` ②, which is the result of call to the `socket` procedure. Another such case is that of the values of `rdx` and `rcx` used in `getaddrinfo` ③ which are just pointers to structures allocated on the stack.

We divide these cases into one of the following categories: value received as argument (`ARG`), locally created value (`VAL`), global value (`GLOBAL`), and unknown constant value (`CONST`). When a specific value can be extracted, e.g., the number "`1`", it is used as-is without a category. Generally we call these argument *kinds*.

***Example*** Marking argument kinds for the two procedure calls in Figure 1(b) results in `getaddrinfo(ARG,ARG,CONST,CONST)` and `setsockopt(VAL,1,2,CONST,4)`.

This representation makes it easier to reason about the procedure for both the model and a human reverse engineer. As kinds provide the model with more information about how an API is used, as shown in Section 5.3, they improve the results of our model by 4% relative over the alternative of using only the name of the external API. Moreover, as shown in Table 1, kinds allows the model to make predictions even when API names are obfuscated.

## 3 Representing Binary Procedures as Sets of Call Site Sequences

A key observation in this work is that focusing on call sites is useful for representing binary procedures. However, using the arbitrary order of calls in which they appear in the binary fails to capture their regularity. After reconstructing call sites, we examine the *order* these calls are used. Figure 1(c) shows the CFG containing only the call sites. This CFG contains four top-down sequences, with edges marked as: $(1,2)$, $(1,3,4)$, $(1,3,5,6)$, and $(1,3,5,7,8)$. Figure 1(d) shows these sequences separately.

***Analyzing the CFG of the procedure*** Given a binary procedure, $P$, we construct its CFG, we denote $G_P$. $G_P$ is a directed graph comprised of nodes which correspond to the basic blocks in $P$. These nodes are connected by edges according to control-flow instructions, i.e., jumps between basic blocks. To simplify, we: (i) add an entry node $Entry$ and connect it to the original entry block; and, (ii) connect all exit nodes to a sink node $Sink$.

We wish to represent $P$ as a set of all potential runtime call sequences, such as the ones shown in Figure 1(d). We take all sequences of instructions along simple paths from $Entry$ to $Sink$ and denote them as $Paths_{Entry \to Sink}$. For each path $p \in Paths_{Entry \to Sink}$, we use $instructions(p)$ to denote the sequence of instructions executed along $p$:

$$[P] = \{instructions(p) \mid p \in Paths_{Entry \to Sink}\}$$

```
6: call listen
   call gai_strerror
3: call socket
   call freeaddrinfo
   call errno_location
   call strerror
2: call getaddrinfo
   call close
5: call bind
   call close
4: call setsocketopt
   call errno_location
   call strerror
1: call memset
```

Figure 2: Typical call instructions for starting a server in order of appearance in the binary code.

```
1: call memset
2: call getaddrinfo
3: call socket
4: call setsocketopt
5: call bind
6: call listen
```

Figure 3: Typical call instructions for starting a server, in their correct *call order*, automatically filtered from error handling calls. This order and filtering could be obtained only by analyzing call paths the Control-Flow Graph.

We map each sequence of instructions $instSeq \in [P]$ to a sequence of call sites: we take only the "call" instructions in *instSeq*, e.g., `call getaddrinfo`, and reconstruct the argument kinds for each such call as explained in Section 2.3: `getaddrinfo(ARG,ARG,CONST,CONST)`.

***Example*** Consider the "call" instructions of Figure 2. These calls are listed in the arbitrary order they were written by the compiler, which does not reflect any logical or chronological order. The path $\pi$ =memset→getaddrinfo→socket →setsockopt→bind→listen is interleaved with error handling calls such as `close` and `strerror`. Additionally, the calls of this path themselves are randomly shuffled in the assembly, i.e., `listen` appears before `bind`. By analyzing the CFG and extracting only possible runtime paths, we approximate all potential call sequences. Figure 3 shows how the path $\pi$ is easily reordered and filtered from other calls thanks to the graph representation. This analysis detects also paths that end in error handling calls, i.e., memset→stderror, as these might also be executed at runtime (if `memset` failed).

***Combined Example*** Correctly ordered reconstructed call sites creates a powerful building block for representing binary procedures. Consider the longest path of Figure 1(d) – with edges marked as $(1, 3, 5, 7, 8)$: (i) `memset(CONST,0,48)` initializes a 48-byte memory space with zeroes; (ii) `getaddrinfo(ARG,ARG,CONST,CONST)` uses two of $P's$ arguments to search for a specific interface to be used later. (iii) `socket(CONST, CONST,CONST)` and `setsocketopt(VAL,1,2,CONST,4)` create a socket and configure it to be a TCP socket by passing the value `1`; and (iv) `bind` and `listen` determine that this procedure is part of a server listening to incoming connections. The rest of the calls handle errors (`strerror`) and free created resources (`close` and `freeaddrinfo`).

While not all the information regarding the connections between these calls is represented explicitly, the argument kinds capture important parts of it, e.g., /Applications/that the socket was created *inside this procedure*, as marked by kind "VAL", contributing to the model choosing the subtoken "create" in the decoding steps.

Finally, we represent $P$ in the learning model as the set of all potential reconstructed call site sequences, including paths that end in calls to error handling procedures.

## 4 Model

The key idea in this work is to represent a binary procedure as a set of call site sequences. We follow the general encoder-decoder paradigm (Cho et al., 2014; Sutskever et al., 2014) with attention (Luong et al., 2015; Bahdanau et al., 2014) for sequence-to-sequence (seq2seq) models, with the difference that the input is not the standard single sequence of symbols, but a *set* of call site sequences. We learn a call site sequence as a sequence of encoded call sites; finally, we decode the target procedure name word-by-word while considering a dynamic weighted average of call site vectors at each step. We note that the main focus of our work is the novel synergy between program analysis of binaries and neural models, rather than the specific neural architecture. To demonstrate this approach, our model is a simple extension of attention encoder-decoder models (Luong et al., 2015; Bahdanau et al., 2014) that encodes a set of input sequences, but the same approach can be used with more expressive architectures.

***Overview*** Encoder-decoder attention models map a sequence of input symbols $\boldsymbol{x} = (x_1, ..., x_n)$ to a sequence of latent vector representations $\boldsymbol{z} = (z_1, ..., z_n)$. Given $\boldsymbol{z}$, the decoder predicts a sequence of output symbols $\boldsymbol{y} = (y_1, ..., y_n)$, thus modeling the conditional probability: $p(y_1, ..., y_m \mid x_1, ..., x_n)$. At each decoding time step, auto-regressive models predict the next symbol conditioned on the previously predicted symbol, hence the probability of the target sequence can be factorized as:

$$p(y_1, ..., y_m \mid x_1, ..., x_n) = \prod_{j=1}^{m} p\left(y_j \mid y_{<j}, z_1, ..., z_n\right)$$

We employ a similar architecture to the standard attention encoder-decoder, with the following differences: (i) each vector in $\boldsymbol{z} = (z_1, ..., z_n)$ is an encoded call site sequence with its arguments; (ii) the encoder learns a set of call site sequences, rather than a single input sequence; and (iii) there is no positional relation between the encoded sequences $\boldsymbol{z} = (z_1, ..., z_n) = \{z_1, ..., z_n\}$. We thus refer to the representation as a *set of sequences*.

***Call site sequence encoder*** We define a vocabulary of learned embeddings $E^{names}$. This vocabulary assigns a vector for every *sub*token of API name which was observed in the training corpus. For example, if the training corpus contains a call to `open_file`, each of `open` and `file` is assigned a vector in $E^{names}$. Additionally, we define a learned embedding for each argument kind, e.g., ARG, VAL, CONST or GLOBAL (Section 2), and for every actual value (e.g., the number "`1`") that occurred in training data. We denote the matrix containing these vectors as $E^{kinds}$. We represent a call site by summing the embeddings of its API subtokens, and concatenating with up to $k_{args}$ of argument kind embeddings:

$$encode\_callsite\left(w_1...w_{k_s}, kind_1, ..., kind_{k_{args}}\right) = \left[\left(\sum_{i}^{k_s} E_{w_i}^{names}\right) ; E_{kind_1}^{kinds} ; ... ; E_{kind_{k_{args}}}^{kinds}\right]$$

We pad the remaining kind slots with an additional `no-arg` symbol. Next, we learn a call site sequence using a bidirectional LSTM. We represent each call site sequence by concatenating the last states of forward and backward LSTMs:

$$h_1, ..., h_l = LSTM\left(callsite_1, ..., callsite_l\right)$$
$$z = \left[h_l^{\rightarrow}; h_l^{\leftarrow}\right]$$

Where $l$ is the maximal length of a call site sequence. In our experiments, we used $l = 60$. Finally, given a set of call site sequences, we represent the entire procedure as a set of its encoded call site sequences: $\{z_1, z_2, ..., z_n\}$

***Decoder*** Our decoder operates much like decoders of contemporary auto-regressive attention models such as Luong et al. (2015). Given a set of encoded call site sequences $\boldsymbol{z} = (z_1, ..., z_n)$, the decoder predicts the next output symbol, i.e., procedure name subtoken, while attending over $\boldsymbol{z}$. By attending over the call site sequences $\boldsymbol{z}$, the decoder selects the relevant call site sequences for each decoding step. As the initial state of the decoder, we average the encoded call site sequences: $h_0^{dec} = \frac{1}{n}\sum_{i=1}^{n} z_i$

## 5 Evaluation

We implemented our approach in a model called `Nero`, for *NEural Reverse engineering Of stripped binaries*.

### 5.1 Experimental Setup

***Dataset*** We collected a dataset of code packages from the GNU code repository containing a variety of applications such as networking, administration tools and libraries. We focus our evaluation on Intel 64-bit executables running on Linux, but the same process can be applied to other architectures and operating systems. The compiled executables, amounts to 13,826 procedures.

We extract procedure names to use as labels, and then create two datasets by: (i) stripping, and (ii) stripping and obfuscating API names for each executable. Stripping is performed to conform to the way executables are usually distributed, and API calls obfuscation is sometimes done in malware.

We split both datasets into the same training-validation-test sets using a $(8:1:1)$ ratio. Each dataset contains 2.49 (±0.01) target symbols per example. There are **829.38** (±13.28) *assembly code tokens* per procedure, which our analysis reduces to 10.05 (±0.08) *call sites* and 12.6 (±0.23) paths per procedure[1]; the average path is 7.5 (±0.01) call sites long. We make this dataset publicly available.

To avoid dealing with mixed naming schemes, we removed all packages containing a mix of programming languages, e.g., a Python package containing partial C implementations. We filtered out wrapper procedures because they are usually very easy to both reverse-engineer and predict, thus falsely improve the scores.

***Avoiding Duplicates*** Following Lopes et al. (2017) and Allamanis (2018) who pointed out the existence of code duplication in open-source datasets and its adverse effects, we created the train, validation, and test sets from completely separate projects and packages. Additionally, we put a lot of effort, both manual and automatic, into filtering duplicates from our dataset. To filter duplicates, we filtered out the following:

1. *Different versions of the same package* – for example, "wget-1.7" and "wget-1.20".
2. *C++ code* – C++ code regularly contains overloaded procedures; further, class methods start with the class name as a prefix. To avoid duplication and name leakage, we filtered out all C++ executables entirely.
3. *Tests* – all executables suspected as being tests or examples were filtered out.
4. *Static linking* – we took only packages that could compile without static linking. This ensures that dependencies are not compiled into the dependent executable.

***Training*** We trained our model using a single Tesla V100 GPU. We used embeddings of size 128 for target subtokens and API subtokens; to encode call site sequences we use bidirectional LSTMs with 128 units each; the decoder LSTM had 512 units. We used dropout (Srivastava et al., 2014) of 0.5 on the API embeddings and the LSTMs. We used the Adam (Kingma and Ba, 2014) optimization algorithm. We trained the network end-to-end using the cross-entropy loss. We tuned hyperparameters on the validation set, and evaluated the final model on the test set.

***Metrics*** At test and validation time, we adopted the measure used by previous work (Allamanis et al., 2016; Alon et al., 2019a; Fernandes et al., 2019), and measured precision, recall and F1 score over the target *sub*tokens, case insensitive and ignoring non-alphabetical characters. For example, for a true reference of `open file`: a prediction of `open` is given full precision and 50% recall; and a prediction of `open input file` is given 67% precision and full recall.

***Baselines*** We compare our model to Debin (He et al., 2018), by training and testing their model on our dataset[2]. This is a non-neural baseline based on Conditional Random Fields (CRFs). As far as we are aware, this is the only other work attempting to perform a similar task to ours. We note that Debin was designed for a slightly different task of predicting names for both local variables and procedure names; nevertheless, we focus on the more difficult task of predicting procedure names and use only these to compute their score. Other straightforward baselines are *Transformer-text* and *LSTM-text* in which we do not perform *any* program analysis, and instead just apply standard NMT architectures directly on the assembly code: one is the Transformer (Vaswani et al., 2017), and the other has two bidirectional LSTMs as the encoder, two decoder layers and attention.

To provide further insight into our approach we implemented our approach in two models: *`Nero-LSTM`* encodes the control-flow sequences using bidirectional LSTMs and decodes with another LSTM; and *`Nero-Transformer`* encodes these sequences and decodes using a Transformer (Vaswani et al., 2017).

To evaluate the main novelties of our approach, which are learning from enriched APIs and learning Control-Flow paths – we perform a thorough ablation study, as detailed in Section 5.3.

[1]The average number of paths (12.6) is greater than the average number of call sites (10.05) because each call site may participate in multiple possible call sequences in the CFG.

[2]The dataset of He et al. (2018) is not publicly available. We make our dataset public.

| | Stripped | | | *Stripped & Obfuscated API calls* | | |
|---|---|---|---|---|---|---|
| Model | Prec | Rec | F1 | Prec | Rec | F1 |
| Debin (He et al., 2018) | 34.86 | 32.54 | 33.66 | 32.10 | 28.76 | 30.09 |
| LSTM-text | 22.32 | 21.16 | 21.72 | 15.46 | 14.00 | 14.70 |
| Transformer-text | 25.45 | 15.97 | 19.64 | 18.41 | 12.24 | 14.70 |
| `Nero-LSTM` | **45.82** | 36.40 | **40.57** | **39.12** | 31.40 | **34.83** |
| `Nero-Transformer` | 41.54 | **38.64** | 40.04 | 36.50 | **32.25** | 34.24 |

Table 1: Our model outperforms previous work by a relative improvement of 20%.

### 5.2 Results

The left side of Table 1 shows the results of the comparison to He et al. (2018), *LSTM-text*, and *Transformer-text* on the stripped dataset. Overall, our models show a 20% relative improvement over the model of He et al. (2018) and 86% over *LSTM-text*. `Nero-Transformer` performs similarly to `Nero-LSTM`, scoring an F1 score of 40.04, outperforming `Nero-LSTM` on recall but trailing in the precision and F1 score. This demonstrates the usefulness of our representation used with different learning architectures.

The right side of Table 1 shows the same models on the stripped and API-obfuscated dataset. Obfuscation degrades the results of all models, yet still our models perform significantly better than the model of He et al. (2018) and the textual baselines. This result depicts the importance of kinds in our representation. We note that overall, in both datasets, our models perform best on both precision and recall.

***Comparison to He et al. (2018)*** Conceptually, our model is much more powerful because it is able to decode out-of-vocabulary procedure names from subtokens, while the CRF of He et al. (2018) uses a closed vocabulary that can only predict already-seen procedure names. At the binary code side, since our model is neural, at test time it can utilize unseen call site sequences while their CRF can only use observed relationships between elements. Furthermore, their representation performs a shallow translation from binary instruction to connections between symbols, while our representation is based on a deeper data-flow-based analysis to find values of registers arguments of imported procedures.

***Comparison to LSTM-text and Transformer-text*** The comparison to the NMT baselines shows that learning directly from the assembly code performs significantly worse than leveraging semantic knowledge and static analysis of binaries. We hypothesize that the reasons are the high variability in the assembly data, which results in a low signal-to-noise ratio. This comparison necessitates the need of an informative static analysis to represent and learn from executables.

***Examples*** Table 3 shows a few examples for predictions made by the different models. Additional examples can be found in Appendix A.

### 5.3 Ablation study

To evaluate the contribution of our representation, we compare our model in several the following configurations:

***`Nero-LSTM` no kinds*** - uses only the CFG analysis with the called API *names*, without argument kinds.

***`Nero Transformer`→`LSTM`*** - uses a Transformer to encode the sets of control-flow sequences and an LSTM to decode the prediction.

***BiLSTM call sites*** - uses the same enriched call sites representation as our model including argument kinds, with the main difference that the order of the call sites is *their order in the assembly code*: there is no analysis of the CFG.

***BiLSTM calls*** - does not use CFG analysis neither argument kinds. Instead, it uses two layers of bidirectional LSTMs with attention to encode `call` instructions with only the name of the called procedure, in the order they appear in the executable.

| Model | Prec | Rec | F1 |
|---|---|---|---|
| BiLSTM calls | 35.95 | 30.36 | 32.92 |
| BiLSTM call sites | 36.05 | 31.77 | 33.77 |
| `Nero-LSTM` no-kinds | 43.62 | 35.25 | 38.99 |
| `Nero Transformer→LSTM` | 39.93 | **38.88** | 39.40 |
| `Nero-LSTM` | **45.82** | 36.40 | **40.57** |
| `Nero-Transformer` | 41.54 | 38.64 | 40.04 |

Table 2: Variations on our model, ablating components of our analysis.

| Model | Prediction | | | |
|---|---|---|---|---|
| Gold | read file | check new watcher | get user groups | install signal handlers |
| He et al. (2018) | bt open | read index | display | **signal** setup |
| *LSTM-text* | <unk> | **check** opt | close stdin | <unk> |
| *Transformer-text* | ipmi disable coredump | <unk> | config file ipmi | regfree |
| `Nero` (this work) | vfs **read file** | **check** file | **get** ip **groups** | **install handlers** |

Table 3: Examples from our test set and predictions made by the different models. More examples can be found in Appendix A.

***Results*** Table 2 shows the performance of the different configurations. `Nero-LSTM` achieves relatively 4% higher score than *`Nero-LSTM` no-kinds*. This shows the contribution of the information stored in argument kinds and its importance to prediction. *BiLSTM call sites* and *BiLSTM calls* relatively trail 16% and 19% behind `Nero-LSTM` and `Nero-Transformer`. These show the importance of our data-flow-based observation of the data. `Nero Transformer→LSTM` achieved slightly lower precision than `Nero-LSTM` and `Nero-Transformer`, but the highest recall.

As we discuss in Section 3, our data-flow-based analysis helps filtering and reordering calls in their approximate chronological runtime order, rather than the arbitrary order of calls as they appear in the assembly code. *BiLSTM call sites* performs slightly better than *BiLSTM calls* due to the use of argument kinds instead of plain `call` instructions.

## 6 Related Work

***Machine learning for source code*** Several works have investigated machine learning approaches for predicting names in high-level languages. Most works focused on variable names (Alon et al., 2018; Bavishi et al., 2018), method names (Allamanis et al., 2016; Alon et al., 2019b; Allamanis et al., 2015) or general properties of code (Raychev et al., 2016b; 2014). Another interesting application is measuring the likelihood of existing names to detect naming bugs (Pradel and Sen, 2018; Rice et al., 2017). Most work in this field used either syntax only (Bielik et al., 2016; Raychev et al., 2016a; Maddison and Tarlow, 2014), semantic analysis (Allamanis et al., 2018) or both (Raychev et al., 2015; Iyer et al., 2018). Leveraging syntax *only* may be useful in languages such as Java and JavaScript that have a rich syntax, which is not available in our difficult scenario of RE of binaries. In contrast with syntactic-only work such as Alon et al. (2019a;b), working with binaries requires a deeper semantic analysis in the spirit of Allamanis et al. (2018), which recovers sufficient information for training the model using semantic analysis.

Allamanis et al. (2018), Brockschmidt et al. (2019) and Fernandes et al. (2019) further leveraged semantic analysis with Graph Neural Networks, where edges in the graph were relations found using syntactic and semantic analysis. Another work (DeFreez et al., 2018) learned embeddings for C functions based on the CFG. We also use the CFG, but in the more difficult domain of stripped *compiled* binaries rather than C code.

***Static analysis models for RE*** He et al. (2018) used static analysis with CRFs to predict various properties in binaries. As we show in Section 5, our model gains 20% higher scores due to their sparse model and our deeper data-flow analysis . Katz et al. (2018) showed an approach to infer subclass-superclass relations in stripped binaries. Lee et al. (2011) used static and *dynamic* analysis to recover high-level types. In contrast, our approach is purely *static*. Shin et al. (2015) used RNNs to identify procedure boundaries inside a stripped binary. David et al. (2017) and Pewny et al. (2015) addressed the problem of finding similar procedures to a given procedure or executable, which is useful to detect vulnerabilities.

## 7 Conclusion

We present a novel approach for predicting procedure names in stripped binaries. The core idea is to leverage static analysis of binaries to encode rich representations of API call sites; traverse the Control-Flow Graph to approximate the chronological runtime order of the call sites; and encode these sequences using two different set-of-seq-to-seq architectures (LSTM-based and Transformer-based).

We evaluated our framework over real-world stripped procedures. Our model achieves a 20% relative gain over existing non-neural approaches, and over 86% relative gain over the naïve textual baselines ("LSTM-text" and "Transformer-text"). Our ablation study shows the importance of analyzing argument kinds and learning from the CFG. To the best of our knowledge, this is the first work to leverage deep learning for reverse engineering procedure names in binary code.

We believe that the principles presented in this paper can serve as a basis for a wide range of tasks that involve learning models and RE, such as malware and ransomware detection, executable search, and neural decompilation. To this end, we make our dataset and trained models publicly available.

## References

M. Allamanis. The adverse effects of code duplication in machine learning models of code. *arXiv preprint arXiv:1812.06469*, 2018.

M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL `http://doi.acm.org/10.1145/2786805.2786849`.

M. Allamanis, H. Peng, and C. A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2091–2100, 2016. URL `http://jmlr.org/proceedings/papers/v48/allamanis16.html`.

M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192412. URL `http://doi.acm.org/10.1145/3192366.3192412`.

U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL `https://openreview.net/forum?id=H1gKYo09tX`.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, Jan. 2019b. ISSN 2475-1421. doi: 10.1145/3290353. URL `http://doi.acm.org/10.1145/3290353`.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

R. Bavishi, M. Pradel, and K. Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

P. Bielik, V. Raychev, and M. T. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2933–2942, 2016. URL http://jmlr.org/proceedings/papers/v48/bielik16.html.

M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

X. Chen, C. Liu, and D. Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, pages 2547–2557, 2018.

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

Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 79–94, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062387. URL http://doi.acm.org/10.1145/3062341.3062387.

D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 423–433, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236059. URL http://doi.acm.org/10.1145/3236024.3236059.

P. Fernandes, M. Allamanis, and M. Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1ersoRqtm.

J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243866. URL http://doi.acm.org/10.1145/3243734.3243866.

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, 2018.

O. Katz, N. Rinetzky, and E. Yahav. Statistical reconstruction of class hierarchies in binaries. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 363–376, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173202. URL http://doi.acm.org/10.1145/3173162.3173202.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 84, 2017.

T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015. URL http://aclweb.org/anthology/D/D15/D15-1166.pdf.

J. R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Proceedings of the 1993 Software Engineering Research Forum*, pages 255–260. Citeseer, 1993.

C. Maddison and D. Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657, 2014.

V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017. URL `http://arxiv.org/abs/1703.05698`.

J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 709–724, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.49. URL `https://doi.org/10.1109/SP.2015.49`.

M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, Oct. 2018. ISSN 2475-1421. doi: 10.1145/3276517. URL `http://doi.acm.org/10.1145/3276517`.

V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321. URL `http://doi.acm.org/10.1145/2594291.2594321`.

V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009. URL `http://doi.acm.org/10.1145/2676726.2677009`.

V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 731–747, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984041. URL `http://doi.acm.org/10.1145/2983990.2984041`.

V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA, 2016b. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671. URL `http://doi.acm.org/10.1145/2837614.2837671`.

A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 104, 2017.

E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, pages 611–626, 2015.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1): 1929–1958, 2014.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, jul 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010248. URL `http://ieeexplore.ieee.org/document/5010248/`.

P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, 2017.

# A Additional Examples

Table 4 contains more examples from our test set, along with the predictions made by our model and each of the baselines.

| Gold | He et al. (2018) | *LSTM-text* | *Transformer-text* | BiLSTM call-sites | `Nero-LSTM` (this work) |
|---|---|---|---|---|---|
| mktime from utc | nettle pss ... | get boundary | <unk> | str file | **mktime** |
| read buffer | concat | fopen safer | mh print fmtspec | net **read** | filter **read** |
| get widech | **get** byte | user | mh decode rcpt flag | <unk> | do tolower |
| get user groups | display | close stdin | config file ipmi | **get user groups** | **get** ip **groups** |
| ftp parse winnt ls | uuconf iv ... | mktime | print status | send to file | **parse** form |
| write init pos | allocate pic buf | open int | <unk> | print type | cfg **init** |
| wait for proc | **wait** subprocess | start open | mh print fmtspec | <unk> | strip |
| read string | cmp | error | check command | process | io **read** |
| get user groups | mh alias enumerate | hol free | fi hostlist string | is group **groups** | **get user groups** |
| find env | **find env** pos | proper name utf | close stream | read token | **find env** |
| write calc jacob | usage msg | update pattern | print one paragraph | <unk> | **write** |
| write calc outputs | fsquery show | debug section | cwd advance fd | <unk> | **write** |
| get script line | **get line** | make dir hier | <unk> | read ps **line** | jconfig **get** |
| getuser readline | stdin read **readline** | rushdb print | mh decode rcpt flag | write line | **readline** read |
| set max db age | do link | **set** owner | make dir hier | sparse copy | **set** |
| write calc deriv | orthodox hdy | ds symbol | close stream | fprint entry | **write** type |
| read file | bt open | <unk> | ... disable coredump | <unk> | vfs **read file** |
| parse options | **parse options** | finish | mh print fmtspec | get **options** | **parse** args |
| url free | hash rehash | hostname destroy | setupvariables | hol **free** | **free** dfa content |
| check new watcher | read index | **check** opt | <unk> | open source | **check** file |
| open input file | get options | query in | ck rename | set | delete **input** |
| install signal handlers | **signal** setup | <unk> | regfree | <unk> | **install handlers** |
| write calc jacob | put in fp table | save game var | hostname destroy | <unk> | **write** |
| filename pattern free | add char segment | **free** dfa content | hostname destroy | glob cleanup | **free** exclude segment |
| read line | tartime | init all | close stdout | parse args | **read** |
| locate unset | var is **unset** | url get arg | regerror | var is | var is **unset** |
| ftp parse unix ls | serv select fn | canonicalize | <unk> | <unk> | **parse** syntax option |
| free netrc | gea compile | hostname destroy | hostname destroy | **free** ent | hol **free** |
| string to bool | **string to bool** | setnonblock | mh decode rcpt flag | **string to bool** | parse check line or field |

Table 4: Examples from our test set and predictions made by the different models.