DEBIN: Predicting Debug Information in Stripped Binaries

Jingxuan He
ETH Zurich, Switzerland
hej@student.ethz.ch

Pesho Ivanov
ETH Zurich, Switzerland
pesho@inf.ethz.ch

Petar Tsankov
ETH Zurich, Switzerland
ptsankov@inf.ethz.ch

Veselin Raychev
DeepCode AG, Switzerland
veselin@deepcode.ai

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT

We present a novel approach for predicting debug information in stripped binaries. Using machine learning, we first train probabilistic models on thousands of non-stripped binaries and then use these models to predict properties of meaningful elements in unseen stripped binaries. Our focus is on recovering symbol names, types and locations, which are critical source-level information wiped off during compilation and stripping.

Our learning approach is able to distinguish and extract key elements such as register-allocated and memory-allocated variables usually not evident in the stripped binary. To predict names and types of extracted elements, we use scalable structured prediction algorithms in probabilistic graphical models with an extensive set of features which capture key characteristics of binary code.

Based on this approach, we implemented an automated tool, called DEBIN, which handles ELF binaries on three of the most popular architectures: x86, x64 and ARM. Given a stripped binary, DEBIN outputs a binary augmented with the predicted debug information. Our experimental results indicate that DEBIN is practically useful: for x64, it predicts symbol names and types with 68.8% precision and 68.3% recall. We also show that DEBIN is helpful for the task of inspecting real-world malware – it revealed suspicious library usage and behaviors such as DNS resolver reader.

CCS CONCEPTS

• **Security and privacy → Systems security**; • **Software and its engineering → Software reverse engineering**;

KEYWORDS

Binary Code; Security; Debug Information; Machine Learning

ACM Reference Format:
Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. DEBIN: Predicting Debug Information in Stripped Binaries. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243866

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243866

1 INTRODUCTION

Compilers generate valuable debug information utilized by various tools in order to support debugging, inspection and security analysis of binaries, including decompilation [3, 18, 29] and bug finding [24, 40]. A stripped binary, however, only contains low-level information such as instructions and register uses which is problematic when trying to inspect the behavior of the binary. Unfortunately, debug information of commercial off-the-shelf (COTS) binaries is often stripped for optimization purposes (e.g., size reduction). More severely, vulnerable and malicious binaries are often intentionally stripped to resist security analysis.

Current techniques aiming to recover stripped information such as types (e.g., [38]) or variables (e.g., [23]) are often limited to custom, manually created rules based on domain specific knowledge of the toolchain that generated the binary. To avoid reliance on a potentially brittle set of manually created rules and automate the process, recent years have witnessed an increased interest in new methods and tools that leverage machine learning models trained on large, freely available code repositories (e.g., GitHub). Examples include programming language translation [32], statistical program synthesis [42, 43], identifier name prediction for Android and JavaScript [10, 15, 44, 53], and more recently, function signature prediction and similarity in binaries [20, 22, 55]. The initial success of these statistical approaches motivates the following basic question: can machine learning models trained on large codebases successfully recover stripped binary debug information?

Difficulties in predicting binary debug information. Creating machine learning models to recover binary debug information with sufficient accuracy is more challenging than predicting facts at the source-level or at the level of a typed intermediate representation (e.g., as done by [15, 44]). A key reason is that it is hard to find insightful features and suitable probabilistic models that capture important behaviors of the stripped binary yet work well across a range of hardware architectures and compiler options typically used to generate real-world binaries. Indeed, as debug information associating high-level information (e.g., program variables) with low-level elements (e.g., registers) is stripped, it is not immediately obvious how to recover this structured mapping.

DEBIN: predicting debug information. We address the challenge of predicting debug information, focusing on symbol names, types and locations in stripped binaries. Concretely, we built a prediction system, called DEBIN[1], which takes as input a stripped

[1] DEBIN is publicly available at https://debin.ai.

```
int sub_80534BA() {
  ...
  if ( dword_8063320 <= 0 ) {
    v0 = sub_8053DB1("/etc/resolv.conf", 'r');
    if ( v0 || (v0 =
          sub_8053DB1("/etc/config/resolv.conf", 'r')) )
      { ... }
    ...
```

(a) Decompiled code from the original malware

```
int rfc1035_init_resolv() {
  ...
  if ( num_entries <= 0 ) {
    v1 = fopen64("/etc/resolv.conf", 'r');
    if ( v1 || (v1 =
          fopen64("/etc/config/resolv.conf", 'r')) )
      { ... } // code to read and manipulate DNS settings
    ...
```

(b) Decompiled code using debug information predicted by DEBIN.

Figure 1: DEBIN predicts meaningful names for a DNS resolver reader function from a real-world malware.

binary and outputs a new binary with the rebuilt debug information. DEBIN supports ELF binaries on three popular architectures (x86, x64 and ARM) and can predict debug information with high precision and recall across these architectures. To the best of our knowledge, DEBIN is the first system to successfully handle such comprehensive debug information, especially symbol names which can range over tens of thousands of labels and are hard to recover. We provide a detailed comparison of DEBIN with existing approaches in term of capabilities in Section 6.

The key technical idea behind DEBIN is to learn and combine two complementary probabilistic models: (i) an Extremely randomized Tree (ET) classification model [28] used to recover and extract program variables. Finding these variables works by making a series of *independent* predictions for which a high accuracy algorithm such as ET is desired, and (ii) a linear probabilistic graphical model that makes joint predictions on the properties (e.g., names, types) of the extracted variables and other program elements. In contrast to ET, here, once the variables are discovered, predicting their names or types should be done *jointly* so to produce consistent final results.

Security Applications of DEBIN. DEBIN can be used to enhance practical security-related tasks such as decompilation and malware inspection. As an example, in Figure 1(a), we show a decompiled code snippet of malware[2] downloaded from VirusShare [9], generated by the popular Hex-Rays decompiler in IDA Pro [3]. Given the binary of the malware as input, DEBIN produces the corresponding code (decompiled here) in Figure 1(b) as output. We observe that DEBIN suggests the name `fopen64` for the function `sub_8053DB1`, indicating that the malware opens and reads file `"/etc/resolv.conf"`. This behavior is suspicious because the file stores DNS resolver configurations. Further, DEBIN predicts the name `rfc1035_init_resolv`, meaning that this function initializes DNS resolvers. The usefulness of these name predictions can be assessed by inspecting the implementation of the function body. Indeed, as we further investigated the malware, we found that it can intercept DNS queries and perform DNS hijacking according to remote commands from a Command and Control (C&C) server.

We also found another malware[3] where DEBIN reveals the same behavior. In general, we find that the predicted names make malicious behavior more explicit and identifiable. In Section 5.4, we elaborate on our experiments involving the use of DEBIN for analyzing malware and show another example where DEBIN correctly predicts the names of statically linked functions (helpful to identify dangerous I/O operations such as opening sensitive files and sending packets over the network). Beyond malware, a number of other analysis tasks can also benefit from the debug information predicted by DEBIN. Such tasks include bug detection [24, 40], code clone search [39, 47] and programmer de-anonymization [19].

Main Contributions. The main contributions of our work are:

- A new machine learning approach for inferring debug information of stripped binaries based on a combination of a decision-tree-based classification algorithm (Section 3.1) and structured prediction with probabilistic graphical models (Section 3.2).
- An extensive set of feature functions (Section 4.4) capturing key semantic relationships between binary code elements (Section 4.2). These are used for instantiating the probabilistic models and improving the overall prediction accuracy.
- DEBIN, a system that uses the above probabilistic models trained on a large dataset of non-stripped binaries in order to recover high-level program variables and predict debug information of new, unseen stripped binaries (Section 4).
- A thorough evaluation of DEBIN on a wide range of tasks and architectures. We show that DEBIN consistently achieves accurate results on the x86, x64 and ARM platforms (Section 5.3). We also demonstrate that DEBIN is helpful for the task of malware inspection (Section 5.4).

2 OVERVIEW

In this section, we provide an informal overview of our method on an illustrative example. Figure 2(a) shows a snippet of assembly code from a stripped binary which computes the sum of the first `n` natural numbers. Only low-level information such as instructions and register usage is present in the binary code.

Given the stripped binary in Figure 2(a), DEBIN outputs a new binary augmented with the predicted debug information, shown in Figure 2(f). There, we can see that high-level information is recovered, such as names and types of register-allocated and memory-allocated variables. Note that DEBIN also recovers the mapping from program variables to their locations in the assembly code as registers and memory offsets, marked with different colors. This predicted debug information makes it much easier to inspect and understand how the binary works. We now illustrate the key steps of DEBIN on the task of predicting debug information for the code in Figure 2(a).

[2] SHA1: `64bd5ba88d7e7104dc1a5586171e83825815362d`.
[3] SHA1: `5ab78c427e1901adf38ade73b8414340f86b6227`.

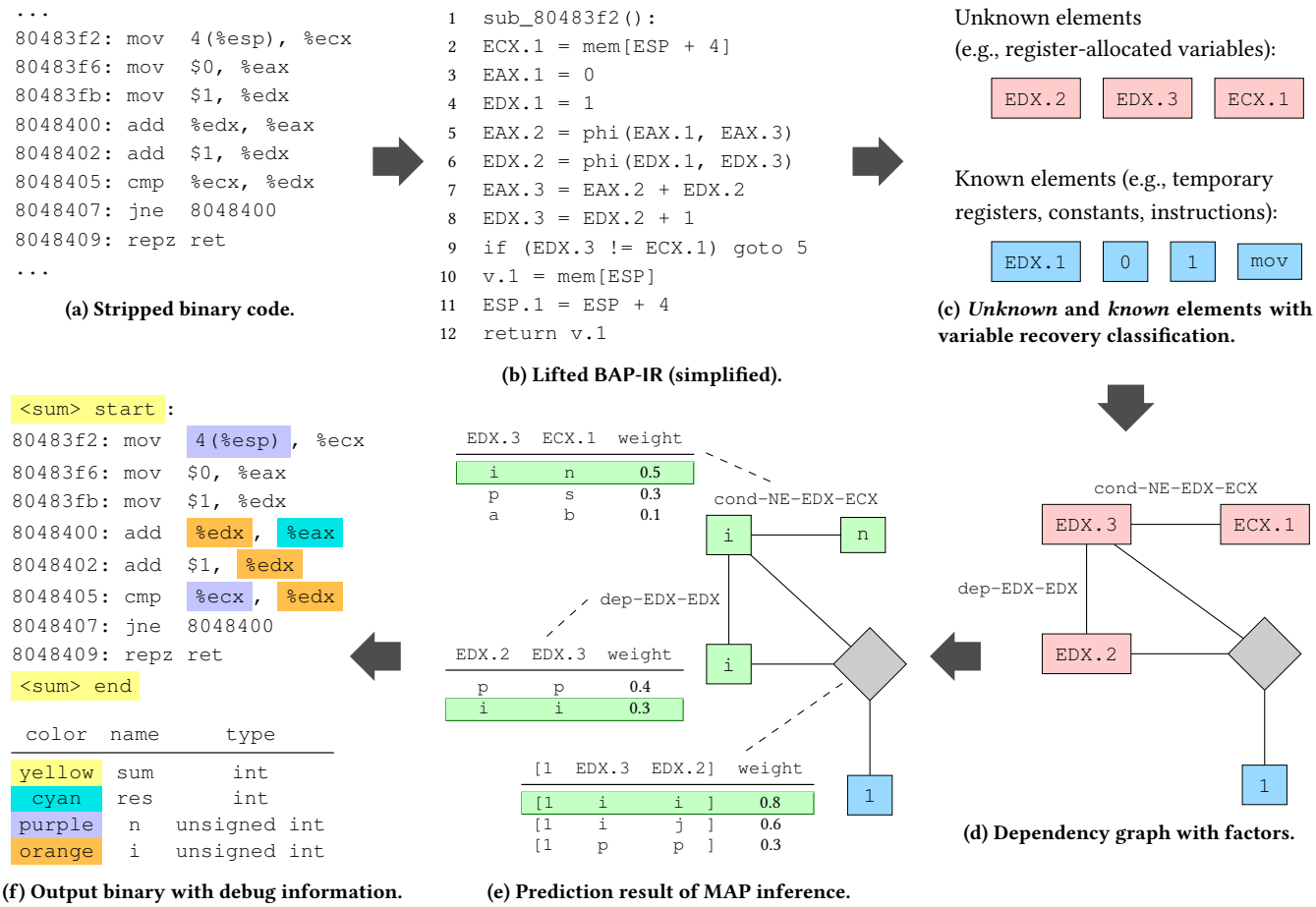

Figure 2: An overview of the steps for rebuilding debug information of the binary in Figure (a). Figure (f) shows the prediction results. The bottom table lists predicted names and types. The colors signify the mapping from a variable or a function to its locations in assembly code.

Lift Assembly into BAP-IR. DEBIN first lifts the assembly code to BAP-IR, the intermediate representation of the Binary Analysis Platform (BAP) [17]. The BAP-IR for our example is shown in Figure 2(b). BAP-IR captures semantics of instructions in a higher-level and uniform syntax across different architectures, which provides crucial insights for later steps. For example, it recovers control flow instructions as the `if` statement at line 9 of Figure 2(b). More details on BAP-IR as used in DEBIN are described in Section 4.1.

Extract Unknown and Known Elements with Variable Recovery Classification. Next, DEBIN analyzes the obtained BAP-IR, extracting two sets of program elements. The first set of program elements are *unknown* elements, marked by red color in Figure 2(c). DEBIN needs to predict properties of *unknown* elements whose information is lost during stripping. Registers that store variable values are one example of *unknown* nodes. The second set of program elements are those whose property is already present in the binary code and DEBIN does not need to infer this information. These elements are marked with blue color. Examples of *known* nodes are constants and instructions.

For certain kinds of elements, it is easy to determine whether they are *unknown* or *known* with fixed rules. For instance, the constants `0` and `1`, and the instruction `mov`, are *known* and are marked with blue in Figure 2(c). However, such rules cannot always determine if a registers or a memory offset stores a variable. In general, variables in source code can be mapped to registers or memory offsets in binary code, but not every register and memory offset corresponds to a variable in the source as memory and registers may be allocated by the compiler only for temporary use. For example, the memory offset `mem[ESP]` at line 10 of Figure 2(b) temporarily holds the return address of the function, but `mem[ESP+4]` at line 2 stores the variable `n`. Compilers leverage sophisticated techniques for allocating machine resources, a process that is generally irreversible and often differs among different compilers and versions. Therefore, manually constructing rules for finding register-allocated and memory-allocated variables is difficult.

DEBIN infers properties of registers and memory offsets allocated for variables since they capture crucial debug information such as variable names. For temporarily allocated registers and memory offsets, DEBIN treats these as *known* nodes and does not predict their name or type. We formalize the challenge of recovering variables as a binary classification problem. DEBIN trains a classifier from millions of registers and memory offsets that are labeled whether they represent variables. Then, this classifier is used to recover source-level variables from low-level registers and memory offsets. In our example, `EDX.2`, `EDX.3` and `EAX.3` are recognized as register-allocated variables and their nodes are marked with *unknown*. Unlike these, `EDX.1` is not recovered as a variable so it is marked as *known*. The variable recovery classification model is based on Extremely randomized Trees [28], a variant of Random Forests [16]. We discuss this model in more detail in Section 3.1.

Build Dependency Graph. In the next step, DEBIN builds an undirected dependency graph where nodes are extracted code elements in BAP-IR and edges represent the relationships between these code elements. In Figure 2(d), we depict a partial dependency graph for our example.

Two kinds of relationships are built upon program elements extracted in prior steps. The first are pairwise relationships which link two nodes. Note that there could be multiple relationships between two nodes with different labels. For instance, `EDX.3` and `EDX.2` are connected by the edge `dep-EDX-EDX` to formally express their dependency relationship by the statement `EDX.3=EDX.2+1`. Similarly, the `cond-NE-EDX-ECX` relationship between `EDX.3` and `ECX.1` indicates their values are compared by a "not equal" conditional expression. Note that these two relationships are both encoded with the register location (`EDX` and `ECX`) of the connected nodes in order to capture how compilers allocate and manipulate registers.

The second kind of relationship is a factor relationship. Factor relationships enhance the expressiveness of our model because they can connect multiple nodes. For instance, the grey diamond in Figure 2(d) involves the constant node `1` as well as two register variables `EDX.3` and `EDX.2` (because they are all operands of the statement `EDX.3=EDX.2+1`). In fact, this BAP-IR statement corresponds to the `++i` statement in the original C source code. Factor relationship are helpful for capturing patterns such as the one above.

In general, relationships in dependency graphs originate from the semantic behaviors of the binary code elements extracted by our analysis. Extracted elements and relationships are formally defined in Section 4. The dependency graph represents a Conditional Random Field (CRF) [37], a probabilistic graphical model that captures relationships between nodes and enables joint predictions over them. We elaborate on the details of this model in Section 3.2.

Probabilistic Prediction. In this step, DEBIN assigns a property value to every *unknown* node. DEBIN performs what is known as Maximum a Posterior (MAP) inference over the dependency graph – it jointly predicts a globally optimal assignment for all *unknown* nodes, based on the structure of the graph. Next, we illustrate how the prediction works on our example. In Figure 2(e), we show the same graph as in Figure 2(d), but with the predicted names for all *unknown* nodes and with three additional tables associated with the three relationships. Each table is a mapping from a possible assignment of the nodes connected by the relationship to a weight value. The weight values are learned a-priori during the training phase (discussed in Section 3.2) and intuitively represent the likelihood of an assignment. The goal of the MAP inference algorithm is to find a globally optimal assignment that maximizes the summation of weights.

For the top-most table, the assignment of `i` and `n` results in a score of 0.5, which is the highest score locally in this table. Similarly, a locally optimal assignment of `i` and `i` is achieved for the bottom-most table. However, for the table in the middle, DEBIN does not select the most likely assignment because the only feasible choice is `i` and `i` according to the other two assignments. Hence, the overall MAP assignment will have the (highest) score of 1.6. Note that if the assignment of `p` and `p` was selected from the middle table, the selection from the other two tables will result in a total score of only 1.0. The final MAP inference results for all names in our example are shown in Figure 2(e).

Output Debug Information. DEBIN encodes all predicted properties along with the recorded location information in the format of DWARF [8] debug information (a standard adopted by many compilers, debuggers and security analysis tools) and outputs a binary augmented with debug information. For simplicity, we visualize the recovered debug information for our example in Figure 2(f). First, DEBIN predicts names and types of functions and variables, as listed in the table. Second, the boundary of function `sum` is recovered as shown by yellow color, thanks to the ByteWeight [14] plugin of BAP. Finally, the location of every variable in the assembly code is also rebuilt, as indicated by the one-to-many colored mapping from variables to registers and memory offsets. For example, variable `n`, colored purple, has type `int` and is located in memory offset `4(%esp)` at address `80483f2` and in register `%ecx` at address `8048405`.

Scope and Limitations. We focus on inferring names and types of variables and functions. Additionally, DEBIN reconstructs program variables based on the predicted names and maps them to low-level locations in registers and memory offsets. We do not consider debug information such as line numbers as it is only applicable when source code is actually available. In terms of finer prediction granularity, while we handle 17 different types, an interesting item for future work would be predicting the contents of `struct` and `union` types.

We note that DEBIN learns patterns of compiler behavior from training samples to make predictions. When input binaries are not entirely compiler-generated, e.g., the binary is obfuscated or generated from human-written assembly, the accuracy of DEBIN may be lower. For example, Popov et al. [41] introduce a method that modifies program control flow and inserts traps to impair disassembly, which can affect an important preliminary step of DEBIN (and other binary analyzers). Those issues are beyond the scope of this paper. An interesting future direction would be to train and test DEBIN with obfuscated samples and study how DEBIN generalizes upon these. Finally, while our tool already supports ELF binaries

on x86, x64 and ARM, it can be extended to other binary formats and operating systems such as the Windows PE format.

3 PROBABILISTIC MODELS

In this section, we introduce the probabilistic models used by DEBIN. We first discuss our variable recovery techniques based on Extremely randomized Trees classification and then present the Conditional Random Field model used for structured prediction of binary code properties.

3.1 Variable Recovery

We consider the problem of predicting whether a low-level register or a memory offset maps to a variable at the source-level. This variable can be either a local variable, a global variable or a function argument (we do not classify the particular kind). We formulate this prediction problem as a binary classification task for registers and memory offsets. Every register or memory offset can be represented by a feature vector where every feature is extracted from the stripped binary. The binary classification model takes the feature vector as input and outputs a boolean value that signifies whether the input register or memory offset is a variable. We will later discuss how the feature vector is constructed in Section 4.3. We leverage Extremely randomized Trees (ET), an ensemble of Decision Trees, as our binary classification model. In the following, we first introduce Decision Trees, followed by the ET model.

Decision Tree. A *Decision Tree* is a tree-based model that takes a feature vector as input and outputs a classification label. Each non-leaf node of the tree contains a test on one feature value from the feature vector and splits further the testing space according to the test result. Each leaf node represents a label. The input propagates from the root to a leaf node of the tree, where the final classification decision is made. Typical decision tree training algorithms build the tree from the root to the leaves by selecting features and values that split the data into subtrees such that a given measure (e.g., information gain [28]) is maximized.

Extremely Randomized Trees. The *Extremely randomized Trees* (ET) is a tree ensemble learning model proposed by Geurts et al. [28]. Here, at training time, a large number of decision trees are constructed. At testing time, the ET model outputs the most commonly predicted label from the decision trees constructed at training time.

Next, we outline several technical details of the ET learning algorithm. At every step of splitting the training data into subtrees, a feature from a random subset of S features in the feature vector is selected (this step is similar to the popular Random Forests learning algorithm [16]). ET differs from Random Forests mainly in two aspects. First, ET learning utilizes the complete training set instead of sub-samples to build each decision tree. Second and most importantly, for each of the S features, a split value is selected randomly by the ET algorithm. Then, the split maximizing the information gain is selected among the S random splits from the prior step. These randomization and ensemble techniques make ET a powerful tool for classification. Further, randomized selection of the split value reduces computation time and model variance, making classification fast and stable.

The outcome of this step is a mapping where every register and memory offset is assigned an *unknown* label if the classifier predicted that the element is mapped to a variable, and a *known* label otherwise.

3.2 Binary Code Properties Prediction

Given the variable mapping computed above, our goal now is to assign the most likely name or type to each *unknown* node. This is achieved through structured prediction using a model known as Conditional Random Fields (CRFs).

We formalize the relevant elements in a binary as a vector of random variables $V = (v_1, \ldots, v_{|V|})$. The assignment of every random variable in V ranges over a set *Labels* that contains all possible names and types seen during training. Without loss of generality, we further define the vectors of *unknown* and *known* elements as $U = (v_1, \ldots, v_{|U|})$ and $K = (v_{|U|+1}, \ldots, v_{|V|})$, respectively. Then, to find the most likely properties U_{opt} for the *unknown* nodes, we perform a *Maximum a Posteriori* (MAP) query on the CRF probabilistic model P over the random variables U (discussed later) as follows:

$$U_{opt} = \underset{U' \in \Omega}{\operatorname{argmax}}\ P(U = U' \mid K = K')$$

where $\Omega \subseteq Labels^{|U|}$ denotes the set of possible assignments of properties to *unknown* elements, U' is an assignment to the *unknown* elements and $K' \in Labels^{|K|}$ is an assignment to *known* elements, which is fixed. Note that in this conditional probability, the *unknown* elements are predicted jointly, which means that they may influence each other. After the query, the values of *unknown* elements are changed according to the most likely assignment while the *known* elements remain fixed. To perform the MAP query, a trained CRF model is required. We discuss the details of CRFs, MAP Inference and the training algorithms in the remainder of this section.

Dependency Graph. A dependency graph for the input program is an undirected factor graph $G = (V, E, F)$ where V is the vector of random variables that represent program elements, $E \subseteq V \times V \times Rels$ is the set of edges where an edge $e = (v_m, v_n, rel)$ denotes a particular pairwise relationship named rel between two program elements v_m and v_n (we assume $m < n$), and F is a set of factors where each factor connects one or multiple program elements. Figure 2(d) shows an example of a dependency graph.

Feature Functions. In our work, there are two kinds of feature functions: pairwise feature functions and factor feature functions.

For pairwise feature functions, we first define helper functions $\psi_i : Labels \times Labels \times Rels \to \mathbb{R}$. After assigning two values to the two nodes linked by edge $e = (v_m, v_n, rel)$, we can apply ψ_i on this edge, obtain a real number. The returned value can be viewed as a strength measure on the presence of certain nodes-relationship triplets. We define the ψ_i function templates with respect to different pairwise relationships defined later in Section 4.4. Then, in the training phase, our model can instantiate the templates with observed edges to obtain multiple different ψ_i functions.

Given a dependency factor graph $G = (V, E, F)$ and an assignment to all program elements (U', K'), the *pairwise feature function*

f_i is defined as follows (via the i^{th} helper function ψ_i):

$$f_i(U',\ K') = \sum_{(v_m,\ v_n,\ rel)\in E} \psi_i((U',\ K')_m,\ (U',\ K')_n,\ rel)$$

where $(U',\ K')_m$ means the m^{th} element of vector $(U',\ K')$. The pairwise feature function f_i applies function ψ_i on all edges of the graph G and computes a summation of the return values. Intuitively, it can be viewed as examining the overall effect of the function ψ_i on the entire graph.

We apply the concept of a factor feature function from Factor Graphs [27] in our CRF model. For factor feature functions, we define other helper functions $\varphi_j : Labels^+ \rightarrow \mathbb{R}$. Intuitively, the function φ_j takes as input a set of assigned values to nodes connected by a factor and returns a real value that captures how good the assignment is. Further, given the dependency graph $G = (V,\ E,\ F)$ and its assignment, we define a *factor feature function* f_j:

$$f_j(U', K') = \sum_{I\in F} \varphi_j((U',\ K')_I)$$

where I is a set containing the indices of program elements of a factor and $(U',\ K')_I$ returns the ordered list of values in $(U',\ K')$ indexed by I. The factor feature function iterates through all the factors in the graph and also computes an overall influence.

Score Functions. Every feature function is associated with a weight w_l, which is computed by the learning algorithm. Based on the weights and feature functions defined above, we can score a predicted assignment by:

$$\text{score}(U', K') = \sum_{l}^{n} w_l f_l(U',\ K')$$

where f_l is either a pairwise feature function or a factor feature function and n is the total number of features.

Conditional Random Field. A *Conditional Random Field* (CRF) is a probabilistic graphical model for representing a conditional probability distribution. In our case, the definition of the distribution is as follows:

$$\begin{aligned} P(U = U' \mid K = K') &= \frac{1}{Z(K')} \exp(\text{score}(U',\ K')) \\ &= \frac{1}{Z(K')} \exp(\sum_{l}^{n} w_l f_l(U',\ K')) \end{aligned}$$

where $Z(K') = \sum_{U''\in\Omega} \exp(\text{score}(U'',\ K'))$ is a constant which ensures the probabilities of all assignments sum to 1.

MAP Inference. As mentioned earlier, to find the most likely assignment to program elements, we maximize the function:

$$U_{opt} = \underset{U'\in\ \Omega}{\text{argmax}}\ P(U = U' \mid K = K')$$

In the setting of CRF, this function can be rewritten as:

$$U_{opt} = \underset{U'\in\ \Omega}{\text{argmax}}\ \frac{1}{Z(K')} \exp(\text{score}(U',\ K'))$$

Because $Z(K')$ is a constant, we can omit it and work with the following function instead:

$$U_{opt} = \underset{U'\in\ \Omega}{\text{argmax}}\ \text{score}(U',\ K')$$

Indeed, for a MAP inference query, our goal is to find the optimal U_{opt} that leads to the highest score computed according to the definition above.

In practice, computing this query with exact search is NP-hard and thus computationally prohibitive. To solve this problem, we leverage a scalable greedy algorithm, available in the open source Nice2Predict framework [5]. This algorithm works by selecting a candidate assignment for every node from a *top-K* beam and iteratively changing the assignments until the score stops improving.

Learning CRF Model. Let $D = \{(U^{(i)},\ K^{(i)})\}_{i=1}^{t}$ be a set of t programs used for training, in which the ground truth assignments $U^{(i)}$ and $K^{(i)}$ are given. For Debin, D is a set of non-stripped binaries. During learning, our goal is to automatically compute the optimal weights $\mathbf{w} = \{w_l\}_{l=1}^{n}$ for feature functions from the patterns in the training set. For training, we aim to compute the optimal weights w_{opt} that produce the highest likelihood considering all training binaries (mathematically via the product below) specified as:

$$\mathbf{w}_{opt} = \underset{\mathbf{w}}{\text{argmax}} \prod_{i}^{t} P(U = U^{(i)} \mid K = K^{(i)})$$

Intuitively, maximizing this likelihood is to find the best weights w_{opt} that lead to correct prediction for all training samples.

However, computing the exact likelihood as defined above is very expensive because for each program we need to compute the expensive Z constant which in turn requires iterating over all possible joint assignments of all candidate labels (this set is generally large as we need to consider all possible names and types). Instead, we employ pseudo likelihood estimation to approximate the precise likelihood:

$$P(U = U^{(i)} \mid K = K^{(i)})$$

with

$$\prod_{j} P(U_j = U_j^{(i)} \mid neighb(U_j), K^{(i)})$$

where U_j is the jth element of vector U and $neighb(U_j)$ represents the neighbors of U_j.

The pseudo likelihood decomposes the expensive computation of the exact likelihood into multiplication of easier-to-compute probability for every local assignment. The main benefit stems from the fact that we now consider one node at a time, where for each node we compute the Z constant (for that node) by iterating only over the possible labels of that node. To further speed up the learning phase, we adopt a parallelized implementation of the algorithm [5]. For more details on pseudo likelihood estimation, please see [51]. We remark that while the (approximate) computation of the Z constant is needed for this training method, as mentioned earlier, it is not required for MAP inference.

4 STRUCTURED PREDICTION WITH DEBIN

In this section, we describe how Debin instantiates the CRF model described in Section 3 in order to perform structured prediction for binaries.

To build a CRF from a program, we follow several steps: we first transform the binary code by lifting it into the BAP-IR intermediate representation, then we extract the relevant program elements, determine *known* and *unknown* elements (using the predictions of

the ET model), and lastly we relate the program elements through feature functions.

For a given binary, we first construct *one* dependency graph and then perform MAP inference over that entire graph. The MAP inference predicts *jointly* the names and the types in this graph.

4.1 Intermediate Representation

The first step of DEBIN is to analyze the input binary and lift it into BAP-IR, the intermediate representation of the BAP binary analysis platform. A simplified example of BAP-IR was already shown in Figure 2(b). There are several advantages to employing BAP and its IR:

(1) BAP-IR provides a uniform syntax for binary code across various architectures, making DEBIN a cross-platform tool (currently supports x86, x64 and ARM) and easily extensible to other architectures supported by BAP.
(2) During the lifting process, BAP can recognize function boundaries of stripped binaries via its ByteWeight component [14]. This component is important for the follow-up analysis steps as well as obtaining various code elements (defined in Section 4.2).
(3) While preserving semantics of raw instructions, BAP-IR explicitly shows operations on machine states, which are more meaningful and higher-level. This is particularly useful in providing more insightful relationships than raw instructions.
(4) Registers, flags and other machine variables are transformed into a Static Single Assignment (SSA) form, where each variable is assigned once and has version numbers. It enables DEBIN to determine which register accesses touch the same variable. This is useful as registers accessed with the same version number can be merged into a single node, saving time and space during prediction.
(5) Kim et al. [35] investigated a wide range of IRs for binary analysis. BAP-IR stands out as the most suitable choice for being explicit, self-contained and robust in lifting binary instructions. It also helps in making DEBIN robust under different compiler options and hardware architectures.

4.2 Binary Code Elements

DEBIN extracts different kinds of program elements from BAP-IR in order to build a dependency graph. Each of the following program elements becomes a node in the graph:

Functions. There are two types of functions in binaries: library and user-defined. For example, `sum` in Figure 2 is user-defined while `printf` is a function from the C language standard library. For every function, we introduce a node for representing its name.

Register Variables. A register in assembly code may correspond to a variable in the source code. In BAP-IR with SSA format, every register exists as a tuple (register name, SSA version). We introduce a node for every register tuple that can be mapped to a variable to represent its variable name. The `EDX.2` node in Figure 2(d) is an example of such a node. Note that (as discussed earlier in Section 2) there are also registers that do not correspond to a variable. We discuss this case later in Section 4.3.

Memory Offset Variables. BAP-IR explicitly captures memory accesses (e.g., `mem[ESP+4]` at line 2 of Figure 2(b)). As with registers, memory offsets may also correspond to variables. We extract memory offset variables in BAP-IR and introduce nodes for these in order to capture variable names. The handling of the case when a memory access is not a variable is discussed in Section 4.3.

Types. We also introduce a type node for each *unknown* function and variable. In DEBIN, we define 17 possible types that the prediction can work with (in C language style): `struct`, `union`, `enum`, `array`, `pointer`, `void`, `bool`, `char`, `short`, `int`, `long` and `long long` (both `signed` and `unsigned` for the last five).

Flags. Flags represent machine status that indicates overflow, effect of arithmetic carry, or the result of a conditional operation. BAP-IR explicitly represents flags as variables whose name denotes the functionality (e.g., carry flag is named `CF`). We introduce a node for every flag involved in operations of BAP-IR.

Instructions. We also introduce a node for every instruction in the binary. Examples include `mov`, `add` and `jne` in Figure 2(a).

Constants. We introduce nodes for integer and string constants. String constants are extracted from `.rodata` section in binaries.

Locations. Location nodes record the locations of variables in a register (e.g., `ECX`) or a memory offset (e.g., `mem[ESP+4]`). Together with variable nodes, they reveal how compilers allocate machine resources for variables.

Unary Operators. We introduce nodes for different unary operators. Those operations on registers and memory offsets are especially helpful for predicting types (e.g., `unsigned` and `signed` cast operators contribute to predicting a variable's signedness).

4.3 Known and Unknown Elements

To determine which of the above elements are *known* (should not be predicted) or *unknown* (to be predicted), we rely on both fixed rules and the learning-based approach (ET model). We assign fixed values for the *known* nodes while the values of *unknown* nodes are to be predicted by the MAP inference:

- Dynamically linked library function nodes are *known* and are assigned their names because calls on them are done on the basis of their names, which are present even in a stripped binary. User-defined and statically linked function nodes are marked as *unknown* because their names do not exist after stripping.
- Flag, instruction, unary operator, constant and location nodes do not carry high-level information and are thus *known*. We assign a flag name, an instruction name and a unary operator name to the three kinds of nodes respectively, integer or string values to constant nodes, and names of registers or memory offsets to locations nodes.
- Register and memory offset nodes are *known* and assigned a special value `(?)`, if they are dummy variables created by BAP or used for function prologue, function epilogue and argument passing. Those nodes correspond to temporary use and do not carry meaningful debug information, hence we do not aim to make predictions involving these.

Relationship	Template	Condition for adding an edge
	Function Relationships	
Element used in Function	(f, v, `func-`**`loc`**(v))	variable v is accessed inside the scope of function f
	(f, a, `arg-`**`loc`**(a))	variable a is an argument of function f by calling conventions
	(f, c, `func-str`)	string constant c is accessed inside the scope of function f
	(f, s, `func-stack`)	stack location s is allocated for function f
Function Call	(f_1, f_2, `call`)	function f_2 is called by function f_1
	Variable Relationships	
Instruction	(v, *insn*, `insn-`**`loc`**(v))	there is an instruction *insn* (e.g., `add`) that operates on variable v
Location	(v, l, `locates-at`)	variable v locates at location l (e.g., memory offset `mem[RSP+16]`)
Locality	(v_1, v_2, `local-`**`loc`**(v_1))	variable v_1 and v_2 are locally allocated (e.g., `EDX.2` and `EDX.3`)
Dependency	(v_1, v_2, `dep-`**`loc`**(v_1)`-`**`loc`**(v_2))	variable v_1 is dependent on variable v_2
Operation	(v, *op*, `unary-`**`loc`**(v))	unary operation *op* (e.g. unsigned and low cast) on variable v
	(n_1, n_2, *op*`-`**`loc`**(n_1)`-`**`loc`**(n_2))	binary operation *op* (e.g., +, left shift « and etc.) on node n_1 and n_2
	(v_1, v_2, `phi-`**`loc`**(v_1))	there is a ϕ expression in BAP-IR: $v_1 = \phi$ (... v_2, ...)
Conditional	(v, *op*, `cond-unary`)	there is a conditional expression *op*(v) (e.g., `not(EAX.2)`)
	(n_1, n_2, `cond-`*op*`-`**`loc`**(n_1)`-`**`loc`**(n_2))	there is a conditional expression n_1 *op* n_2 (e.g. `EDX.3!=ECX.1`)
Argument	(f, a, `call-arg-`**`loc`**(`a`))	there is a call f (..., a, ...) with argument a
	Type Relationships	
Operation	(t, *op*, `t-unary-`**`loc`**(t))	unary operation *op* on type t
	(t_1, t_2, `t-`*op*`-`**`loc`**(t_1)`-`**`loc`**(t_2))	binary operation *op* on type t_1 and t_2
	(t_1, t_2, `t-phi-`**`loc`**(t_1))	there is a ϕ expression: $t_1 = \phi$ (... t_2, ...)
Conditional	(t, *op*, `t-cond-unary`)	there is a unary conditional expression *op*(t)
	(t_1, t_2, `t-cond-`*op*`-`**`loc`**(t_1)`-`**`loc`**(t_2)))	there is a binary conditional expression t_1 *op* t_2
Argument	(f, t, `t-call-arg-`**`loc`**(t))	call f (..., t, ...) with an argument of type t
Name & Type	(v, t, `type-`**`loc`**(v))	variable v is of type t
	(f, t, `func-type`)	function f is of type t

Table 1: Pairwise relationships for linking code elements in DEBIN. The first column provides a short description of each relationship and the second column defines the relationship template. Helper function `loc` is used to encode location information in relationships. Given an input node, the function returns its location (e.g., the register or the memory offset, or whether it is a constant). We add a relationship edge to the dependency graph if the condition defined in the third column holds.

- For other register and memory offset nodes, we leverage our ET model discussed in Section 3.1 to determine whether they are *known* or *unknown*. To invoke that algorithm, for every target node, we encode features similar to those defined later in Table 1 (treated as strings) and use a one-hot encoding of these strings to construct a feature vector for the node. We provide this vector as input to the algorithm which then classifies the node as *known* or *unknown*. For *unknown* nodes, their values will be predicted during MAP inference. For *known* nodes, we assign the value `(?)`.
- Type nodes of *unknown* functions are *unknown*. Type nodes of *unknown* variable registers and memory offsets are *unknown* and type nodes of non-variables are *known* and are assigned `(?)`.

4.4 Relationships between Elements

We now describe the relationships that connect program elements and enable structured prediction on the resulting dependency graph. We define all pairwise relationships in Table 1. The pairwise relationships are of the form (`a`, `b`, `rel`) specified by the second column, meaning that node `a` is connected to node `b` by the relationship named `rel`. A relationship is added to the dependency graph as an edge when the condition defined in the third column holds. For example, the edge (`foo`, `bar`, `call`) is added only when function `foo` calls function `bar`. To encode location information of variables in relationships, we define a function `loc`, which takes a node as input and outputs the location of the node (e.g., the register or the memory offset, or whether it is a constant).

Function Relationships. This set of relationships captures how functions interact with other binary code elements. First, a function node is connected to register-allocated and memory-allocated variables that appear inside its scope. This relationship encodes the way functions manipulate registers and memory offsets to modify variables. For the example in Figure 2, the function node representing

`sum` should be connected to nodes `EDX.2`, `ECX.1` and etc. Second, we relate function nodes and their register-based or offset-based arguments, which are determined by the calling convention of the specific platform. Third, we link function nodes with two categories of *known* elements, string constants and stack locations. These two relationships are helpful in recovering the patterns for functions to deal with strings and allocate stack resources. Finally, we also incorporate function call relationships.

Variable Relationships. We leverage a comprehensive list of relationships for register and memory offset variables. Those relationships provide useful syntactic and semantic patterns of how variables are allocated and manipulated.

Instruction relationships capture how instructions operate with variables. For instance, a relationship (`ECX.1`, `cmp`, `insn-ECX`) can be built for our example in Figure 2. Moreover, locality of relationships is captured through our analysis. Two register variables are locally allocated when they have the same register name and the difference between their SSA versions is 1. The idea here is that locally allocated registers are possibly from neighboring instructions and thus allocated for the same variable. Two memory offset variables are locally allocated when they can be determined to be aligned next to each other. We link those memory offsets also because they may be allocated for the same variable, especially for variables of a larger structure. For example, apart from the `dep-EDX-EDX` relationship, we also add the `loc-EDX` edge between nodes `EDX.2` and `EDX.3` in Figure 2(d) (not graphically shown in the figure). In addition, dependency, operation and conditional relationships formalize behaviors of various BAP-IR statements. Finally, we determine arguments of a function call through calling conventions of each architecture and connect the called function and its arguments. This relationship is helpful for predicting both function names and variable names.

Type Relationships. Type relationships are helpful in discovering how binaries handle types. First, operation, conditional and argument relationships are also introduced for types because these three sets of relationships are effective for both variable names and types. For example, variables accessed in conditional expressions are likely to be of type `bool` and the type of function arguments should comply with signatures of the called function. Furthermore, type nodes are also connected to their corresponding name nodes.

Factor Relationships. Apart from pairwise relationships, we also define three factor relationships. The first one connects all nodes that appear in the same ϕ expression of BAP-IR. The second one further explores the behavior of function calls, linking function node of a call and its arguments. Third, we relate together elements that are accessed in the same statement. These factor relationships can link more than two elements and thus capture semantic behaviors of binary code that are not expressible with only pairwise relationships.

4.5 Feature Functions

Recall that in Section 3.2, we defined feature functions in the general sense. We now provide the feature functions used by Debin and illustrate how these feature functions are obtained from the templates during training.

Let $D = \{(V^{(i)},\ E^{(i)},\ F^{(i)})\}_{i=1}^{t}$ be a set of graphs extracted from non-stripped binaries. Nodes in these graphs are filled with ground truth values. For each edge $(v_m,\ v_n,\ rel) \in \bigcup_{i=1}^{t} E^{(i)}$, we generate pairwise feature functions ψ_i as follows:

$$\psi_i(A,\ A',\ Rel) = \begin{cases} 1 & \text{if } A = a_m,\ A' = a_n,\ Rel = rel \\ 0 & \text{otherwise} \end{cases}$$

where a_m and a_n are assignments to v_m and v_n, respectively. Furthermore, for each factor $L_k \in \bigcup_{i=1}^{t} F^{(i)}$, we define a factor feature function as follows:

$$\varphi_j(L) = \begin{cases} 1 & \text{if } L = L_k \\ 0 & \text{otherwise} \end{cases}$$

where $L \in Labels^{+}$ is the ordered list of values (names/types) assigned to the nodes connected by a factor.

Intuitively, the feature functions defined above serve as indicators for the relationship and node assignment patterns discovered from the training set. We instantiate the feature templates with ground truth assignments, obtaining a large number of feature functions. The weights associated with every feature function are then learned in the training phase. The combination of feature indicator functions and weights is then used by the MAP inference.

5 IMPLEMENTATION AND EVALUATION

In this section, we present the implementation details of Debin [4]. Then we discuss our extensive evaluation of the system: we evaluate Debin's accuracy on predicting debug information and illustrate use cases where Debin can be leveraged for practical security analysis.

5.1 Implementation

Debin extracts dependency graphs (defined in Section 4) from BAP-IR. We developed a plugin for the BAP platform in order to obtain BAP-IR for the input binaries. Variable recovery classification is implemented using the machine learning package scikit-learn [7]. To annotate the correct values for the training graphs, we parse DWARF [8] debug information from non-stripped binaries using the pyelftools package [6]. After obtaining the MAP inference result using the Nice2Predict [5] framework, program variables are reconstructed according to the predicted names and are associated via a one-to-many mapping to their locations in registers or memory offsets. We format all of the rebuilt information according to the DWARF standard and utilize the ELFIO library [2] to produce the final output binary. All our experiments were conducted on a server with 512GB of memory and 2 AMD EPYC 7601 CPUs (64 cores in total, running at 2.2 GHz).

5.2 Evaluation Setup

Now we describe our dataset and metrics for evaluating Debin's prediction accuracy.

Dataset. Currently, Debin supports ELF binaries on x86, x64 and ARM architectures. For each architecture, we collected 3000 non-stripped binary executables and shared libraries as our dataset. The 9000 binaries (3000 for each of the three architectures) are from

[4] Available at https://debin.ai

Arch	Instructions	Functions	Variables	Known
x86	9363	80	744	87
x64	7796	72	851	79
ARM	10416	86	787	94
Average	9192	79	794	87

Table 2: Statistics on averaged number of instructions, *unknown* function nodes, *unknown* variable nodes, and *known* name nodes for our dataset. The values of the *known* name nodes are extracted from the `.dynsym` section. There is no *known* type node in the dependency graphs.

Arch		Precision	Recall	F_1
x86	Name	62.6	62.5	62.5
	Type	63.7	63.7	63.7
	Overall	63.1	63.1	63.1
x64	Name	63.5	63.1	63.3
	Type	74.1	73.4	73.8
	Overall	68.8	68.3	68.6
ARM	Name	61.6	61.3	61.5
	Type	66.8	68.0	67.4
	Overall	64.2	64.7	64.5

Table 3: Evaluation of DEBIN using structured prediction.

830 Linux Debug Symbol Packages [1], which include popular ones such as coreutils, dpkg and gcc. The source code of these binaries is written in the C language and the binaries are built with the default compilation settings for each package. Therefore, multiple optimization levels (e.g., `-O0` to `-O3`) and other different compiler options can be involved, which leads to a rich variety in the dataset. Statistics of the dataset are shown in Table 2. Further, to learn the feature functions and weights described in Section 4, we randomly select 2700 binaries as the training set for each architecture. The remaining 300 binaries are left as a benchmark for testing the prediction accuracy of DEBIN.

Metrics. DEBIN consists of two prediction phases. First, DEBIN uses a binary classifier based on the ET model to recover variables in registers and memory offsets. We measure this step using accuracy:

$$\text{Accuracy} = \frac{|TP| + |TN|}{|P| + |N|}$$

where *TP* is the true positives (i.e., registers and memory offsets that are predicted to be variables and can actually be mapped to variables), *TN* is the true negatives (i.e., registers and memory offsets that are predicted not to be variables and actually do not represent variables), *P* is the positive samples (i.e., registers and memory offsets that can be mapped to variables), and *N* is the negative samples (i.e., non-variable registers and memory offsets).

Second, structured prediction with the CRF model is employed to predict properties (i.e., names and types) for recovered variables and other elements. To measure the prediction quality of DEBIN after this step, we track the following sets:

- *Given Nodes (GN)*: the set of elements with debug information (name or type) before stripping it. The set *GN* is formed of the set *P* defined above in addition to elements which are functions (these are not included in the set *P*).
- *Nodes with Predictions (NP)*: the set of *unknown* elements determined by our fixed rules and variable recovery classification (via the ET algorithm). These are nodes for which the CRF model makes predictions.
- *Correct Predictions (CP)*: the set of elements for which the CRF model predicted the correct value (name or type). Here, correct means that the predicted value is *exactly equal* to the values given in the debug information. Note that assigning a value to a non-variable will be counted as an incorrect prediction (defined below).

With a perfect classifier, the set *NP* would ideally be the same as the set *GN*. However, because the classifier is not perfect and may introduce both false negatives and false positives, meaning the set *NP* may end up being non-comparable to (or even larger than) *GN*. The set *CP* is a subset of both *NP* and *GN*. To capture the quality of the prediction, we use the following measures:

$$\text{Precision} = \frac{|CP|}{|NP|}; \quad \text{Recall} = \frac{|CP|}{|GN|}; \quad \text{F}_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Intuitively, precision is the ratio of cases where the predicted value is equal to the given value, among all of the predicted nodes (marked as *unknown* by our rules and ET classifier). Recall refers to the proportion of correct predictions over the set of nodes that originally had debug information (i.e., the set *GN*). F_1 score is a harmonic average of precision and recall, examining the overall prediction quality of DEBIN.

5.3 Evaluation on Prediction Accuracy

To objectively measure the accuracy of our probabilistic models, we assume function scope information is given for every binary. However, in general, DEBIN can also leverage the built-in ByteWeight component in BAP whose accuracy for recovering function boundaries is around 93% [14].

Evaluation on Variable Recovery. First, we briefly discuss the accuracy of variable recovery by the ET algorithm. For x86, the accuracy is 87.1%, for x64 - 88.9% and for ARM - 90.6%. The high accuracy in this step ensures that DEBIN can effectively recover register-allocated and memory-allocated variables for later property prediction. It also filters out temporarily allocated registers and memory offsets and thus reduces noise.

Evaluation on Structured Prediction. Table 3 summarizes the evaluation results of DEBIN after structured prediction. We report results for name prediction, type prediction and overall (name+type) prediction, measured by precision, recall and F_1. Overall, the results show that DEBIN predicts a considerable amount (recall over 63%) of debug information with high precision (over 63%) across three architectures and achieves a good trade-off between precision and recall. These results indicate that our feature functions generalize well over x86, x64 and ARM architectures.

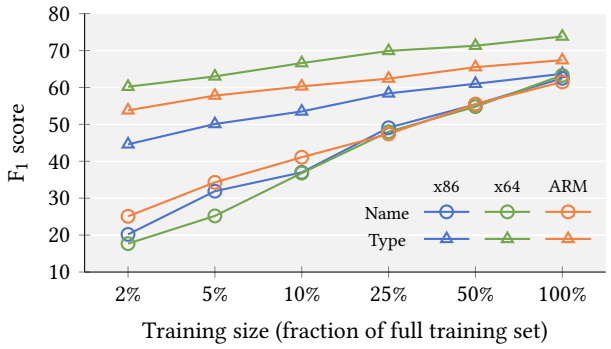

Figure 3: F_1 score of name and type prediction with different fractions of training set used for learning the CRF model. For every fraction from 2% to 50%, we repeated 5 times downsampling of the training set and running experiments to obtain scores. We finally report the averaged results.

For name prediction, DEBIN consistently achieves high accuracy (F_1 is 62.4% on average). This result shows that DEBIN often predicts names identical to their original values. Indeed, programmers typically use similar names for variables (e.g., variables that represent loop counters) or reuse functions (e.g., that open a file). For this reason, the set of names observed in our training set contains most of the names that appear in the testing set. We recall that the names in our training set are used to instantiate the feature functions, and this enables DEBIN to often recover the original names of variables and functions (since DEBIN makes predictions according to these feature functions). Later in this section, we provide examples of name prediction outputs and illustrate how those names can be employed for inspecting binary behaviors.

Further, DEBIN can also infer types accurately. The accuracy for type prediction on x64 (F_1 is 73.8%) is higher than on x86 (F_1 is 63.7%) and ARM (F_1 is 67.4%). This is likely because types on x64 (a 64-bit architecture) are generally more distinguishable in terms of their bitwise sizes than on the other two 32-bit architectures. For instance, on x64, the sizes of `pointer` type and `int` type are 64 bits and 32 bits, respectively, while on x86 and ARM they are both 32 bits. Our feature functions for types capture size differences and thus achieve a higher accuracy on x64. On x86, the type prediction accuracy is lower. A possible explanation is that x86 has fewer register-allocated variables whose types can be effectively inferred by operation relationships (e.g., x64 and ARM use registers for argument passing while x86 uses the stack).

Here we remark that our measurement is a lower bound on the capability of DEBIN since we test for unambiguously exact equivalence. First, assigning a name or a type to a non-variable is always counted as a mis-classification in our metric. However, the usefulness of the predictions may not be affected if the assigned names and types comply with program semantics. Moreover, predicting a different name for a register-allocated or memory-allocated variable is also treated as an incorrect prediction in our measurements. However, it is possible the predicted names are semantically close to the real names. For instance, in our x64 testing set, we found four cases where variable named `buf` is assigned the value `buffer`.

Name	Ratio	Precision	Recall	F_1
`i`	2.42	52.7	77.1	62.6
`s`	1.72	65.1	66.1	65.6
`p`	1.24	47.6	63.6	54.5
`self`	0.92	65.2	55.3	60.0
`cp`	0.79	69.5	77.4	73.2

Table 4: Statistics on 5 of the most frequent names in the test set. Column 2 shows a distribution ratio of every name among all names in the test set.

Name	Times	Precision	Recall	F_1
`ip`	697	58.3	71.7	64.4
`device`	124	52.4	61.3	56.5
`passwd`	81	70.0	51.9	59.6
`socket`	69	91.5	62.3	74.1
`encrypted`	5	50.0	60.0	54.5

Table 5: Statistics on 5 of the most sensitive names in the test set. Column 2 shows the appearance times of each.

Finally, certain types have more proximity (e.g., `signed` types and their `unsigned` counterparts) than others (e.g., `struct` with `int`).

Another intriguing question is how *training set size* for the CRF model affects inference accuracy. To investigate this point, we kept the ET model fixed and randomly sampled 2%, 5%, 10%, 25% and 50% of binaries from the full training set to train the CRF model for each architecture. Then, we evaluated the trained models on the same test set and report the F_1 scores. The results are plotted in Figure 3. We can see that as the size of the training set increases, the accuracy also increases for both name and type prediction among the three architectures. Since name prediction is more difficult than type prediction, it is more data-hungry as its F_1 scores grow more rapidly with increasing training samples.

Name Prediction Outputs. Now we present more details concerning the name prediction task. Table 4 shows five of the most frequent names over three architectures. DEBIN predicts upon these with an average F_1 score of 63.2%. Even though these names are frequent, they are useful when finding common programming pattern of binary code. For example, `i` is often used as a loop index and with it, reverse engineers can quickly identify loops and their conditionals. Also, some of the simple names (e.g., `s` and `p`) may suggest variable types and thus operation patterns upon them. For example, variable with a predicted name `s` is likely to be of string type and involved in string operations. This is helpful for understanding semantics of binaries.

Apart from frequent names, we list five representative sensitive names in Table 5. Variables predicted with these names typically store critical security-related information such as IP addresses (`ip`), device information (`device`), encryption (`encrypted`) and internet connections (`socket`). For real-world applications, we can search

```
\\ snippet 1
if ( sub_806D9F0 (args) >= 0) {
  ...
  sub_80522B0 (args);
  ...
}

\\ snippet 2
...
v2 = sub_818BFF1 ("/proc/net/tcp", 0, v45, a1);
if ( v2 == -1 ) return 0;
...
```

(a) Decompiled snippets for original malwares

```
\\ snippet 1
if ( setsockopt (args) >= 0 ) {
  ...
  sendto (args);
  ...
}

\\ snippet 2
...
v2 = open ("/proc/net/tcp", 0, v23, a1);
if ( v2 == -1 ) return 0;
...
```

(b) Decompiled snippets for outputs of DEBIN.

Figure 4: DEBIN predicts the statically linked library functions `setsockopt`, `sendto` and `open`. Identifying such functions helps in finding potentially unsecure I/O operations.

for such sensitive names in the output of DEBIN and quickly decide where security issues may be located. We demonstrate the usefulness of searching for sensitive names in Section 5.4.

Training and Prediction Speed. The training phase for name and type prediction lasts about five hours for each architecture: around four hours to train two ET classification models (one for registers and another for memory offsets), and one hour to train the CRF model with the pseudo likelihood learning algorithm. We used 60 threads to run the learning algorithms. The average prediction time for every binary (with one thread) is about two minutes where around 80% is spent on variable recovery, and 20% is spent on running our analysis to construct the dependency graph and perform MAP inference over this graph. A possible future work item would be to increase the efficiency of the variable recovery module.

5.4 Malware Inspection

We now discuss how DEBIN can be helpful for the task of inspecting behaviors of malicious binaries. Using VirusShare [9], we then search for around 180 categories of Linux malwares defined by Symantec [52] and tried to download the latest 10 fully stripped ELF x86 malwares for each category. Since there are usually less than 10 qualified malwares for most categories, our test dataset consists of 35 binaries over 13 categories ranging from 5.0KB to 1.7MB. We provide the malware binaries as input to DEBIN, whose models are trained on benign binaries as already discussed above, and search for security-related names in the outputs. Note that we cannot report DEBIN's accuracy on malwares because they are all stripped and contain no debug information. The original malwares and output binaries of DEBIN are further decompiled into Pseudo C code with the popular Hex-Rays decompiler in IDA Pro [3]. We then manually insepct the decompiled outputs. An example for how DEBIN reveals a DNS resolver reader was already shown in Section 1. We next report another use case where DEBIN can be helpful in identifying suspicious statically linked library functions.

Identifying Suspicious Statically Linked Library Uses. Malicious binaries are often compiled statically and stripped to hide library function uses. In our malware dataset, 26 out of 35 samples are statically built. While library usages are a crucial part for identifying malicious behaviors, it would be tedious and time-consuming for analysts to manually inspect assembly code and rename them. During our inspection, we found that DEBIN can be helpful with finding potentially harmful library uses (as it automatically renames them). In Figure 4, we show code snippets where DEBIN renames suspicious library calls for two malware examples (one[5] of category `Linux.Xorddos` and another[6] of category `Linux.Mirai`). In the first snippet, DEBIN recovers `setsockopt` and `sendto`, which indicates potential leakage of sensitive data. In the second snippet, DEBIN reveals sensitive behavior of opening the file `"/proc/net/tcp"`, which stores active IPv4 TCP connections.

Apart from our examples, DEBIN can also recover other library calls, such as string manipulations. Library function renaming can greatly reduce efforts for security analysts to examine malware. DEBIN is able to rename these library functions likely due to their existence (either statically or dynamically linked) in the training set.

Limitations. With our probabilistic models, DEBIN learns patterns of binary code from our training binaries, which we assume are benign. However, malicious binaries often exhibit more complex behaviors than benign ones. For example, we found samples in our malware dataset that use customized string encoding. Obfuscation techniques such as control flow flattening may also be performed to hinder analysis. DEBIN may make less accurate predictions on obfuscated binaries. Security analysts could combine de-obfuscation methods with DEBIN to tackle these issues.

6 RELATED WORK

We survey some of the works that are most closely related to ours.

Comparison with Existing Approaches. In Table 6, we compare DEBIN with several existing approaches in terms of capabilities. For variable recovery, DIVINE [13], TIE [38] and SecondWrite [23] employ a static analysis algorithm called Value Set Analysis [12], while DEBIN, to our best knowledge, is the first to use a learning-based classifier. For type recovery, TIE and SecondWrite adopt constraint-based type inference. EKLAVYA learns a Recurrent Neural Network to predict 7 types for function arguments, while DEBIN

[5] SHA1: `3f1f4ba2434d0ad07838ebc694ad4a4cf8c9641a`.
[6] SHA1: `5ab78c427e1901adf38ade73b8414340f86b6227`.

Approach	Variable Recovery	Type Recovery	Name Prediction	Learning Based	Architectures Supported
DIVINE [13]	✓	✗	✗	✗	x86
TIE [38]	✓	✓	✗	✗	x86
SecondWrite [23]	✓	✓	✗	✗	x86
Eklavya [20]	✗	partial[1]	✗	✓	x86, x64
Debin	✓	✓	✓	✓	x86, x64, ARM

[1] Eklavya only predicts 7 types for function arguments.

Table 6: Comparison of Debin against existing approaches for recovering variable or type information from binaries.

is capable of predicting 17 types for both function arguments and variables. Moreover, Debin is also the first system capable of recovering names, a desirable functionality for decompilation and important for malware inspection. We remark that it is difficult to compare Debin and those works quantitatively because they have different prediction granularity, benchmarks and measurements.

Binary Analysis. To perform binary analysis, usually the first step is to choose an analysis platform that translates assembly code into a corresponding IR (thereby abstracting the semantics of various instruction sets). Potential open source candidates for performing this task are BAP [17], Mcsema [4] and angr.io [50]. They lift binary code into BAP-IR, LLVM-IR and VEX-IR, respectively. A recent framework, rev.ng [25], can recover control flow graphs and function boundaries for binaries over multiple architectures. Kim et al. [35] tested a wide range of binary analysis frameworks on different tasks. For Debin, we currently choose BAP-IR for the advantages discussed in Section 4.1. An interesting future work item is to experiment with our learning-based techniques using other frameworks.

There is a wide range of literature on extracting semantic information from binaries. Apart from the ones discussed above, IDA FLIRT [3] and UNSTRIP [30] identify library functions by generating fingerprints for them. However, their approaches are not capable of suggesting names for other functions.

Machine Learning for Binaries. Machine learning methods have been adopted for different binary analysis tasks. Several works focus on function identification [14, 49, 54], which is the basis for many further analysis steps. We adopt ByteWeight [14] in our system since it is publicly available in BAP [17]. Chua et al. [20] train a Recurrent Neural Network to predict function argument counts and types. Statistical language models [33, 34] have been employed to predict types in binaries, but with rather different focus on object types and class hierarchy reconstruction.

Apart from recovering source-level information from binaries, there are also several works that predict other valuable characteristics. Rosenblum et al. [46] focus on toolchain provenance. Their work predicts compiler family and version, optimization level and programming language with high accuracy. Caliskan et al. [19] build an effective set of features and utilizes a random forest classifier to predict programmer identity.

There are other works that utilize machine learning to classify malware [11, 36, 48] and research that adopts statistical methods to calculate binary similarity [21, 22, 55]. Those tasks are complementary to the problem addressed in this work and can benefit from the recovered names and types.

Probabilistic Models for Code. In recent years, the development of large codebases has triggered studies of probabilistic models for software related tasks such as code summarization [26], method and class name suggestions [10], code completion [45], program synthesis [42, 43] and programming language translation [32].

From this set of works, the ones most related to us are [44] and [15]. Their tools also leverage structured prediction algorithms with Nice2Predict [5] so to infer types and names for Javascript and Android programs, respectively. Our work differs from these approaches in that it works on lower-level binary code, which is inherently a more difficult task. Therefore, we need to first perform variable recovery that classifies the program elements used for structured prediction, while these works use a fixed set of rules. We also support more expressive feature functions, i.e., not only pairwise kinds but also richer factors relating multiple elements. Further, these methods only work on a single high-level language (e.g., Java, JavaScript). In the context of binary analysis however, our probabilistic models work on low-level code across multiple architectures and compilation settings, in turn dictating a more general and richer classes of features. Finally, our work utilizes one model to *jointly* predict names and types while [44] uses two separate models (one to predict names and another to predict types). The work of [15] only uses one model to predict names for Android.

A Statistical Machine Translation model is employed to suggest identifier names for Javascript [53] and decompiled code [31]. While the latter is close to our work, its model only achieved around 25% accuracy. Further, their method relies on decompilation, which already suffers from great information loss. Our prediction directly works on binary code, achieves higher accuracy and shows successful cases on decompilation tasks.

7 CONCLUSION

We presented a novel approach for predicting debug information in stripped binaries. The key idea is to formalize the problem as a machine learning task and to leverage a combination of two complementary probabilistic models: an Extremely Randomized Trees classifier and structured prediction with Conditional Random Fields. To instantiate the idea, we introduced a comprehensive set of features suitable for the task of predicting binary debug information, and used this set to train our probabilistic models on a large number of non-stripped binaries.

The resulting system, called Debin, uses these probabilistic models to recover debug information of new, unseen binaries. Our extensive experimental evaluation of Debin indicates the approach is accurate enough to correctly infer large portions of stripped debug information and is helpful for practical security analysis.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback and Rong Jian for the helpful discussions on malware inspection.

REFERENCES

[1] 2018. Debug Symbol Packages. https://wiki.ubuntu.com/Debug%20Symbol%20Packages

[2] 2018. ELFIO. http://elfio.sourceforge.net/.

[3] 2018. IDA Pro. https://www.hex-rays.com/.

[4] 2018. Mcsema. https://github.com/trailofbits/mcsema.

[5] 2018. Nice2Predict. http://nice2predict.org/.

[6] 2018. pyelftools. https://github.com/eliben/pyelftools.

[7] 2018. scikit-learn. http://scikit-learn.org.

[8] 2018. The DWARF Debugging Standard. http://dwarfstd.org/.

[9] 2018. VirusShare. https://virusshare.com/.

[10] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of Foundations of Software Engineering (ESEC/FSE)*. pages 38–49.

[11] Ben Athiwaratkun and Jack W. Stokes. 2017. Malware classification with LSTM and GRU language models and a character-level CNN. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. pages 2482–2486.

[12] Gogul Balakrishnan and Thomas W. Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Proceedings of Compiler Construction (CC)*. pages 5–23.

[13] Gogul Balakrishnan and Thomas W. Reps. 2007. DIVINE: DIscovering Variables IN Executables. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*. pages 1–28.

[14] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of USENIX Security Symposium*. pages 845–860.

[15] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of Computer and Communications Security (CCS)*. pages 343–355.

[16] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of Computer Aided Verification (CAV)*. pages 463–469.

[18] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of USENIX Security Symposium*. pages 353–368.

[19] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[20] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *Proceedings of USENIX Security Symposium*. pages 99–116.

[21] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In *Proceedings of Programming Language Design and Implementation (PLDI)*. pages 266–280.

[22] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of Programming Language Design and Implementation (PLDI)*. pages 79–94.

[23] Khaled Elwazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Proceedings of Programming Language Design and Implementation (PLDI)*. pages 51–60.

[24] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[25] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of Compiler Construction (CC)*. pages 131–141.

[26] Jaroslav M. Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles A. Sutton. 2017. Autofolding for Source Code Summarization. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1095–1109.

[27] Brendan Frey, Frank Kschischang, Hans-Andrea Loeliger, and Niclas Wiberg. 1997. Factor graphs and algorithms. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*. pages 666–680.

[28] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63, 1 (2006), 3–42.

[29] Ilfak Guilfanov. 2008. Decompilers and beyond. In *Black Hat USA*.

[30] Emily R. Jacobson, Nathan E. Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries. In *Proceedings of workshop on Program analysis for software tools (PASTE)*. pages 1–8.

[31] Alan Jaffe. 2017. Suggesting meaningful variable names for decompiled code: a machin translation approach. In *Proceedings of Foundations of Software Engineering (ESEC/FSE)*. pages 1050–1052.

[32] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. pages 173–184.

[33] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating types in binaries using predictive modeling. In *Proceedings of Principles of Programming Languages (POPL)*. pages 313–326.

[34] Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical Reconstruction of Class Hierarchies in Binaries. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. pages 363–376.

[35] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *Proceedings of Automated Software Engineering (ASE)*. pages 353–364.

[36] Bojan Kolosnjaji, Ghadir Eraisha, George D. Webster, Apostolis Zarras, and Claudia Eckert. 2017. Empowering convolutional networks for malware classification and analysis. In *Proceedings of International Joint Conference on Neural Networks (IJCNN)*. pages 3838–3845.

[37] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of International Conference on Machine Learning (ICML)*. pages 282–289.

[38] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[39] Beng Heng Ng and Atul Prakash. 2013. Expose: Discovering Potential Binary Code Re-use. In *Proceedings of Computer Software and Applications Conference (COMPSAC)*. pages 492–501.

[40] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*. pages 406–415.

[41] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. 2007. Binary Obfuscation Using Signals. In *Proceedings of USENIX Security Symposium*.

[42] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2017. Program Synthesis for Character Level Language Modeling. In *Procceedings of International Conference on Learning Representations (ICLR)*.

[43] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of Principles of Programming Languages (POPL)*. pages 761–774.

[44] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of Principles of Programming Languages (POPL)*. pages 111–124.

[45] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of Programming Language Design and Implementation (PLDI)*. pages 419–428.

[46] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. pages 100–110.

[47] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. pages 117–128.

[48] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. pages 38–49.

[49] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of USENIX Security Symposium*. pages 611–626.

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. pages 138–157.

[51] Charles Sutton and Andrew McCallum. 2012. An Introduction to Conditional Random Fields. *Foundations and Trends in Machine Learning* 4, 4 (2012), 267–373.

[52] Symantec Coporation. 2018. A-Z Listing of Threats & Risks. https://www.symantec.com/security-center/a-z.

[53] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of Foundations of Software Engineering (ESEC/FSE)*. pages 683–693.

[54] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. pages 388–398.

[55] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of Computer and Communications Security (CCS)*. pages 363–376.