CATI: Context-Assisted Type Inference from Stripped Binaries

Ligeng Chen*, Zhongling He* and Bing Mao*
*National Key Laboratory for Novel Software Technology, Nanjing University, Nanajing 210023, China
{chenlg, zhe}@smail.nju.edu.cn, maobing@nju.edu.cn

***Abstract*—Code analysis is a powerful way to eliminate vulnerabilities. Closed-source programs lack crucial information vital for code analysis because that information is stripped on compilation to achieve smaller executable size. Restoration has always been a challenge for experts. Variable type information is fundamental in this process because it helps to provide a perspective on program semantic. In this paper, we present an efficient approach for inferring types, and we overcome the challenge of scattered information provided by static analysis on stripped binaries. We discover that neighboring instructions are likely to operate the same type of variables, which are leveraged to enrich the features that we rely on. Therefore, we implement a system called CATI, which locates variables from stripped binaries and infers 19 types from variables. Experiments show that it infers variable type with 71.2% accuracy on unseen binaries. Meanwhile, it takes approximately 6 seconds to process a typical binary.**

***Index Terms*—Stripped Binary; Variable Type; Static Analysis**

I. Introduction

Binary analysis is always a hot topic in the security field. Experts need to rebuild the information from stripped binaries to provide vital information for the following tasks: decompilation [1], code hardening [2]–[4], bug-finding [5], [6], clone detection [7]–[10], etc. However, when high-level language is transformed into binary code (e.g., after compilation), it cannot preserve all the source-level information. Recovering the semantic information of stripped binaries helps understand the whole program.

A typical program stores 2 kinds of information in memory: code and data. When the program is running, it transfers value between memory and registers and does the calculation on values stored in registers. A storage location, either register or memory, that stores a value, is called a variable. If debug information is present, variables are paired with a symbolic name and type information, which are defined by the programmer in high-level source code. That is, every available memory unit and every register can be a variable. One important task for reverse engineering is to figure out the functionality of each variable, which even can help improve fuzzing [11].

Variable type suggests the functionality of the variable. For example, an array of chars is likely to be an input buffer. Therefore, inferring the variables' type partially recovers their functionality. The only way to tell the difference between variables is to observe the instructions that operate them. Hence, the chain of instructions could be used as a vital clue. But since compilation optimizes the binary code, much high-level information isn't preserved. In our task, that is, similar instructions might operate the variable of a different type. We call these variables *uncertain samples*. Therefore, there is little information left to distinguish them.

Currently, commercial tools widely used in industry, such as IDA Pro [12], have an acceptable performance on type inference but they depend on principled and heuristic-based rules. Recently, several works [2], [13]–[15] do a thorough static analysis to improve the accuracy of type inference. Others [16], [17] use dynamic analysis to collect runtime information. But dynamic analysis cannot cover all the binary code and we need to set up environments for each binary to run. Additionally, none of them avoids the reliance on a set of manually crafted rules based on expert knowledge. With the development of code optimization technology, previous rules may fail to infer types from the binary build from recent compilers. Thus summarizing and updating these clues are time-consuming.

To free from excessively searching for endless rules or appealing to expert knowledge, a considerable number of new methods and tools [1], [18], [19] are adopting machine learning models trained on massive data to do binary analysis, including inferring types from stripped binaries. Nevertheless, in these approaches, some expert knowledge is still essential. For instance, in the aforementioned work, He [1] traces the data flow of the instructions using BAP [20], then builds a dependency graph, finally uses a probabilistic graphical model called conditional random field [21] to infer the variable type and its name. It relies heavily on data dependency analysis to build the graph. TypeMiner [22] also builds data object trace via data dependency. For base type, they extract definition-use chains. For structured types, they analyze dereference instructions on its members. They also build a dependency graph and infer using a graphical model. However, they state that a smaller size of the dependency graph negatively affects their results. For half of the variables they extract, they cannot find enough instruction dependency and they ignore these variables because they are not able to predict them well. Control flow analysis might help to build data dependency between basic blocks. However, Zeng [23] shows that control flow cannot be accurately recovered without debug information. All the evidence above suggests that rebuilt information from stripped binary is always scattered, thus makes accurate type inference difficult. In our work, we aim to solve this problem. We call these variables that we cannot build rich data dependency as *orphan variables*. Since there's not much information about *orphan variables*, they are often *uncertain samples*.

Given all these difficulties, we need to employ richer features to infer the variable type. Fortunately, although the dependency of our target instruction is sometimes not easy to build, we can take the *instruction context* into account. In our work, we focus on memory access instructions and dereference instructions as these instructions operate one variable at a time. These instructions are called *target instructions*. *Instruction context* is some instructions before and after the target instruction. According to our survey, the variables that the *instruction context* operates are likely to be the same type as the target operates. We call it *same variable type clustering phenomenon*.

Thus, we leverage the *instruction context* of *target instruction* to infer the variable type. We define a new feature for a variable called *Variable Usage Context (VUC)* as the aggregation of *instruction context* and *target instruction*. Inferring variable type is transformed into VUC classification, or in general, a text classification task. Moreover, if we can build data dependency, in other words, we can find several instructions that operate the same variable, we will utilize a voting mechanism to make the final decision. More concretely, we implement a system, called CATI, which takes a stripped binary as input and outputs located variables with the predicted type information. To the best of our knowledge, CATI is the first to accomplish the most amount of types inference in stripped binaries compiled from C code with few expert knowledge, and the first system leveraging context information to infer the variable type. At the end of our paper, we provide a detailed evaluation of CATI and compare with state-of-art approaches in term of performance in Section VII.

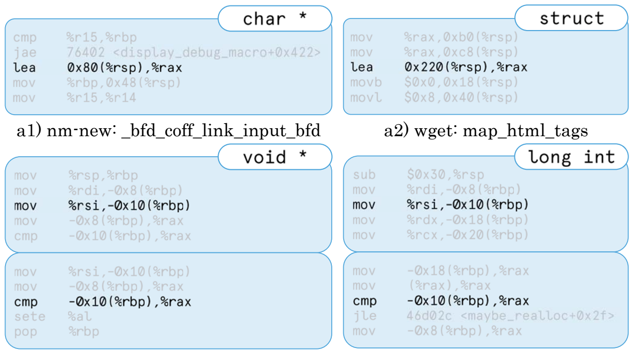
The main contributions of our work are as follows:

1. We discover that the problem of *orphan variables* and *uncertain samples* cannot be well solved by previous work, thus we leverage context information to enrich the features to infer the variable type.

2. We define a new feature called *Variable Usage Context (VUC)*, which consists of the *target instruction* and its *instruction context*. VUC captures rich behavior information from the *instruction context*, and we successfully transfer the problem into a text classification task.

3. We accomplish a thorough evaluation of a wide range oapplications, and the result shows a great performance from different aspects.

The rest of our paper is organized as follows: motivation and overview of CATI are given in Section II and III. Section IV to V introduces the individual process of our system in detail. To evaluate our method, we present our implementation part and the empirical evaluation result in Section VI and VII. We hold a discussion part in Section VIII and related works are classified as discussed in Section IX. Section X takes the end of our paper to conclude our work.



```
cmp     %r15,%rbp
jae     76402 <display_debug_macro+0x422>
lea     0x80(%rsp),%rax
mov     %rbp,0x48(%rsp)
mov     %r15,%r14
```

a1) nm-new: _bfd_coff_link_input_bfd

```
mov     %rax,0xb0(%rsp)
mov     %rax,0xc8(%rsp)
lea     0x220(%rsp),%rax
movb    $0x0,0x18(%rsp)
movl    $0x8,0x40(%rsp)
```

a2) wget: map_html_tags

```
mov     %rsp,%rbp
mov     %rdi,-0x8(%rbp)
mov     %rsi,-0x10(%rbp)
mov     -0x8(%rbp),%rax
cmp     -0x10(%rbp),%rax

mov     %rsi,-0x10(%rbp)
mov     -0x8(%rbp),%rax
cmp     -0x10(%rbp),%rax
sete    %al
pop     %rbp
```

b1) grep: raw_comparator

```
sub     $0x30,%rsp
mov     %rdi,-0x8(%rbp)
mov     %rsi,-0x10(%rbp)
mov     %rdx,-0x18(%rbp)
mov     %rcx,-0x20(%rbp)

mov     -0x18(%rbp),%rax
mov     (%rax),%rax
cmp     -0x10(%rbp),%rax
jle     46d02c <maybe_realloc+0x2f>
mov     -0x8(%rbp),%rax
```

b2) sed: maybe_realloc

Fig. 1. Examples of *uncertain samples* and *orphan variables*.

II. Feasibility and Challenge

A. Problem Definition

Debug information of commercial off-the-shelf (COTS) binaries is often stripped for several purposes, such as size reduction, information hiding, etc. It is a challenging task to learn the mapping relationship between assembly variable instruction and the corresponding variable type.

Given a piece of stripped binary B, our goal is to automatically extract VUCs from B, where $VUC_1, VUC_2, ..., VUC_k \subseteq B$. Each VUC contains the target variable instruction with 10 instructions forward and 10 instructions backward, which can be presented as,

$$VUC_i = \{Ins_{i-w}, .., Ins_{i-1}, Ins_i, Ins_{i+1}, ..., Ins_{i+w}\}$$

Ins_i denotes the central instruction which is correlated to the target variable, w denotes the windows size of VUC, and Ins_{i+j} denotes the instruction which is j instruction(s) away from central instruction. Towards to empirical experiment, we choose window size w to be 10. For traceable Variable V_t, we aim to make the final decision for it relying on the voting mechanism by the prediction result of VUCs whose central instruction is related to V_t.

B. Motivation

We illustrate some motivating cases in this part which are hard for previous work to solve and inspire our main idea.

Orphan Variable. Previously, researchers strongly believe that the trace of the variable may disclose some clues of type information. So most of the former works [1], [16], [22] coincidentally trace the variable whether based on control flow or data flow. Unfortunately, we survey all the variables in our data set extracted by IDA Pro — a totally 3 million variables, and only about 65% of variables have more than 3 related instructions. That is, 35% of variables only have 1 or 2 instructions, which we call it *orphan variable* mentioned before.

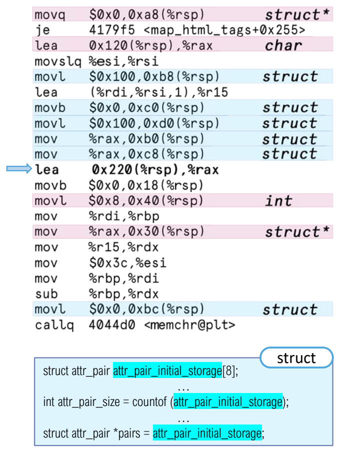
The difficulty of analyzing orphan variables lies in two aspects. On the one hand, 1 or 2 instructions cannot provide enough structural information to infer the information of type for the previous works [1], [22]. On the other hand, two cases with the same instruction(s) but different types are *uncertain samples*, which we defined before. For instance, as shown in Figure 1 a1) and a2), the two cases appear to have the same instruction (offset will be generalized in the pre-processing procedure) but indicate different variable types, and both of the target variables have only one corresponding instruction. As shown in Figure 1 b1) and b2), the two variables from different binaries share the same instructions, which suffers from a similar situation. Previous works are not able to use the limited information to find the mapping relationship with variable and instruction(s) for the cases above.

	Training Set	Testing Set
Variables	3,952,246	167,223
VUCs	22,476,210	1,074,227
Variables with 1 VUC	220,250	9,372
Uncertain Samples-1	216,909	8,687
Variables with 2 VUCs	1,189,466	39,839
Uncertain Samples-2	1,159,307	34,178

TABLE I
STATISTICAL RESULT OF *orphan variables* AND *uncertain samples* IN TRAINING SET AND TESTING SET.

To consolidate our discovery, we survey on both training set and testing set and show the result in Table I. Row 2 and row 3 show the number of variables and related VUCs. Note that each variable corresponds to different amounts of VUCs. Row 4 and row 6 show the number of *orphan variables* — variables are related to 1 or 2 instructions, which we have defined before. Row 5 and row 7 show the number of variables that have the same instruction(s) with others, but they have different types, which we call it *uncertain samples*. We discover this kind of *uncertain samples* cannot be distinguished well by previous works. It is obvious that *uncertain samples* take up over 97% of *orphan variables*, and *orphan variables* take up over 35% of all variable in the data set. As shown in Figure 1, the two cases are the typical *uncertain samples*. If we want to distinguish the *uncertain samples*, we need to extend the features of them.

Same Type Variable Clustering Phenomenon. If two entities do arithmetical operations, they need to belong to the same variable type for the reason of aligning and register storage length. Hence, we do a statistical analysis to verify our hypothesis that the context of the target variable containing a considerable amount of instructions related to the same variable type. Once the hypothesis is verified, we can leverage the likely pattern of the same type variables to infer the target variable. Fortunately, the result is convincing. We carry out statistics on over 107 thousand variables, and we take the 10 forward instructions and 10 backward instructions into consideration. We find about 540 thousand variable instructions within 107 thousand VUCs, and in each VUC, over 53% of instructions completely share the same variable type as the target variable. As shown in Figure 2, it is a variable with type *struct* in row 11 (*lea 0x220(%rsp),%rax*), and we found 10 variables around the instruction location, of which 60 percent are the same variable type — *struct*. So here we conclude that the distribution of variable type related to the instructions has spatial locality. We define it as *same type variable clustering phenomenon*. The analysis result above inspires us to leverage instruction context information as new features.


```
movq    $0x0,0xa8(%rsp)           struct*
je      4179f5 <map_html_tags+0x255>
lea     0x120(%rsp),%rax          char
movslq  %esi,%rsi
movl    $0x100,0xb8(%rsp)         struct
lea     (%rdi,%rsi,1),%r15
movb    $0x0,0xc0(%rsp)           struct
movl    $0x100,0xd0(%rsp)         struct
mov     %rax,0xb0(%rsp)           struct
mov     %rax,0xc8(%rsp)           struct
lea     0x220(%rsp),%rax
movb    $0x0,0x18(%rsp)
movl    $0x8,0x40(%rsp)           int
mov     %rdi,%rbp
mov     %rax,0x30(%rsp)           struct*
mov     %r15,%rdx
mov     $0x3c,%esi
mov     %rbp,%rdi
sub     %rbp,%rdx
movl    $0x0,0xbc(%rsp)           struct
callq   4044d0 <memchr@plt>
```

```
struct attr_pair attr_pair_initial_storage[8];
...
int attr_pair_size = countof (attr_pair_initial_storage);
...
struct attr_pair *pairs = attr_pair_initial_storage;
```
Fig. 2. Same type variable clustering.

Usage of instruction Context. To overcome the challenge of *uncertain samples*, we need to employ more related features to distinguish the variable type. Intuitively, the neighboring instructions can play a significant role to infer the type of target variable. Taking the control flow and data flow into consideration, the neighboring instructions may directly or indirectly influence the behavior of the target variable.

To describe the relationship between the target variable and neighboring variables more precisely, we take the case in Figure 2 as an example. The target variable stays in row 11. Instructions in row 7 to 10 are referring to the same variable– *attr_pair_initial_storage* and the instruction in row 15 is a pointer variable pointing to the target variable. To enrich the features of variables, we bring in *instruction context* as new features, which we define as *variable usage context*.

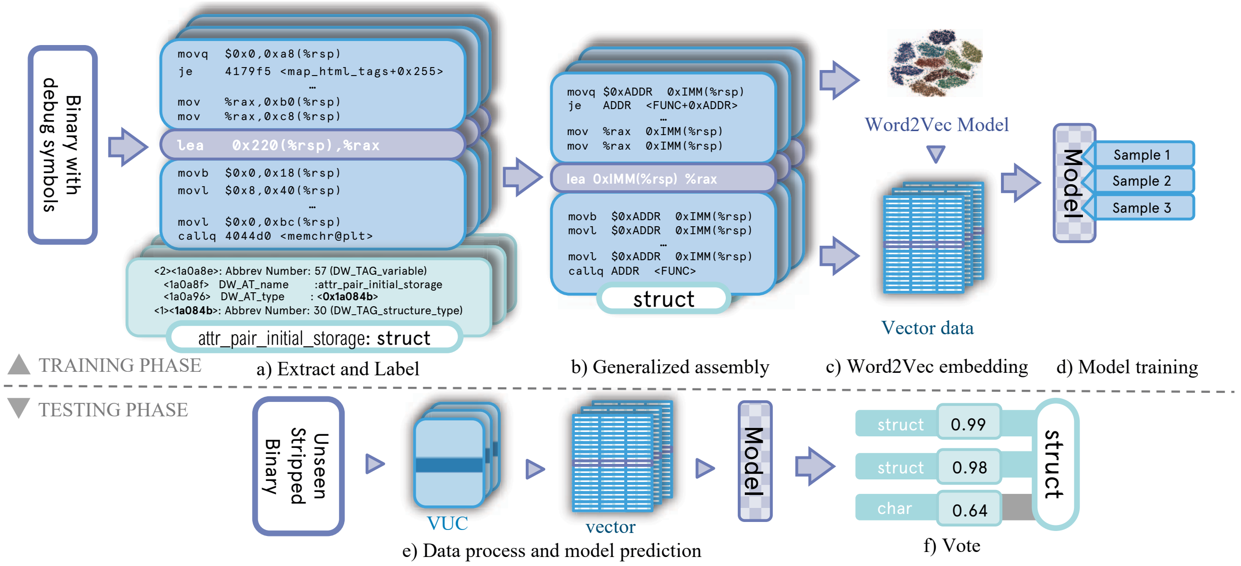
In the next section, we will introduce the process to infer variable type with neighboring instructions in detail.

III. SYSTEM WORKFLOW

In this section, we provide an informal overview of our method on an illustrative sample. Figure 3 shows how to locate variables and infer their types from a snippet of assembly code from a stripped binary. Assemblies represent the meta operations of code at a lower level.

Given the piece of binary, CATI will output the prediction result of each located variable instructions. In the following paragraphs, we illustrate the main procedures of the system on the problem of inferring the variable types.

Extract and Label Samples. CATI firstly locates the target instructions corresponding to variables in stripped binaries. At the same time, we extract their instruction contexts as 10 forward instructions and 10 backward instructions—21 instructions in total. Their aggregation is what we defined as VUC. By leveraging debug information, we can easily label VUC with the target variable's type as shown in figure 3a). For example, IDA pro helps us to find the corresponding stack frame for VUC and its *ground truth*. For the limitation of static

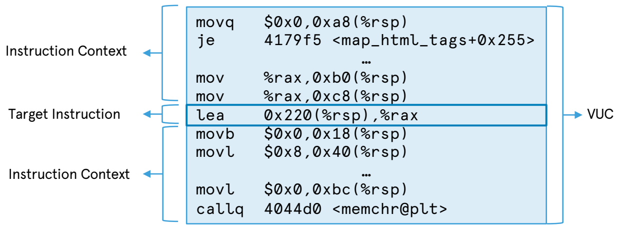


```
movq   $0x0,0xa8(%rsp)
je     4179f5 <map_html_tags+0x255>
             …
mov    %rax,0xb0(%rsp)
mov    %rax,0xc8(%rsp)
lea    0x220(%rsp),%rax
movb   $0x0,0x18(%rsp)
movl   $0x8,0x40(%rsp)
             …
movl   $0x0,0xbc(%rsp)
callq  4044d0 <memchr@plt>
```

```
<2><1a0a8e>: Abbrev Number: 57 (DW_TAG_variable)
  <1a0a8f>   DW_AT_name     :attr_pair_initial_storage
  <1a0a96>   DW_AT_type     : <0x1a084b>
<1><1a084b>: Abbrev Number: 30 (DW_TAG_structure_type)
attr_pair_initial_storage: struct
```

```
movq $0xADDR  0xIMM(%rsp)
je   ADDR   <FUNC+0xADDR>
             …
mov  %rax   0xIMM(%rsp)
mov  %rax   0xIMM(%rsp)
lea 0xIMM(%rsp) %rax
movb  $0xADDR   0xIMM(%rsp)
movl  $0xADDR   0xIMM(%rsp)
             …
movl  $0xADDR   0xIMM(%rsp)
callq ADDR   <FUNC>
struct
```

Fig. 3. An overview of the steps for rebuilding type information of the binary in Figure (a) to (d) (training phase). Figure (e) to (f) shows the final judgment of a variable (inferring phase).



```
movq   $0x0,0xa8(%rsp)
je     4179f5 <map_html_tags+0x255>
             …
mov    %rax,0xb0(%rsp)
mov    %rax,0xc8(%rsp)
lea    0x220(%rsp),%rax
movb   $0x0,0x18(%rsp)
movl   $0x8,0x40(%rsp)
             …
movl   $0x0,0xbc(%rsp)
callq  4044d0 <memchr@plt>
```

Fig. 4. Structure of VUC.

analysis, the mapping result of some VUCs may go wrong with tools. But the result is still convincible.

Generalize Operands. Next, CATI generalizes the operands of each instruction. All the VUCs related to the target variables show how the variables are stored, interpreted and manipulated, which are the only and strong features to infer the variable type. A naive way to represent the instructions is to train a huge model so that each mnemonic and operand has a vector representation. However, operands may consist of a function address or a combination of register and offset. Offset and function address might be different in each program. To reduce the possibility of these values, we make the following substitution rules: (i) the actual address is replaced with *ADDR*; (ii) the actual function name is replaced with *FUNC*; (iii) the immediate value is replaced with *IMM*.

Assembly Code Embedding. In this procedure, CATI employs *Word2Vec* [24] model to represent each VUC, which is often used in natural language processing. With the help of *Word2Vec*, we are able to vectorize the assembly code containing the semantics of neighboring instructions. This inspires us to utilize *Word2Vec* to embed the instructions to a vector representation. We embed each mnemonic and operand into a vector with a length of 32 and embed each instruction into a vector with a length of 96 (filling instructions with less than two operands with a *BLANK* operator for padding). Given a sequence of $VUC_i = \{Ins_{i-10}, .., Ins_{i-1}, Ins_i, Ins_{i+1}, ..., Ins_{i+10}\}$, the algorithm transforms it into a [21,96] matrix as shown in figure 3(c).

Train a Multi-Stage Classifier. To identify the type of each variable, we train a multi-stage classifier with a convolutional neural network (CNN), which is able to encode the hidden features. With the data set consists of enormous pairs of VUC and their types, we set up a tree-like multi-stage classifier which we will introduce in detail in Section V.

Predict Result for Each VUC. With stripped binaries input, CATI disassembles the binary code and locates the variables to extract VUCs with the assistance of IDA Pro. Using the already trained models, we can output the most likely type of each VUC and the confidence of each type which is prepared for the next stage.

Voting. In the testing phase, to make the final decision of the target variable, we leverage the prediction confidence of each VUC and a set of VUC extracted according to data dependency from the former stages. We add confidence results of each type for confidence-based voting mechanism. It avoids letting the borderline result control the decision, but almost completely utilizes all the prediction result. We will explain it later.

IV. Variable Analysis

A. Extract and Label

In this paper, we restrict our study to the x86_64 architecture for a more precise evaluation of our work. We also believe that the techniques can be extended naturally to other architectures.

Firstly, we need to recover the variables from binaries. Our work is focused on the problem of type inference, and previous works [1], [25] have achieved a good result (over 90 % on average) of variable recovery. Thus, we assume that this task can be done accurately enough by existing work and we can directly use them in our pipeline. In our system, we choose IDA Pro.

Then, we extract the data flow of these variables in stripped binaries using IDA Pro and pair each variable with type information extracted from DWARF [26] debug information. DWARF contains detailed information on each variable, such as its name, parent function, offset to stack frame and type. Afterward, we resolve the type of each variable. If we found that the type has been redefined by *typedef, we would recursively find its base type.* Finally, we find the *instruction context* of target instruction with *objdump*.

As a result, we are now able to get the VUCs and their type. For each variable, we name all VUCs on its data flow uniquely, so that we know they belong to the same variable while voting.

B. Generalization

Each VUC consists of 21 instructions and each instruction consists of 1 mnemonic and no more than 2 operands. Our system is implemented on x86_64 platform so that the mnemonic is within x86_64 architecture's instruction set. However, operands have far more possibilities. It could be an immediate number, a register, an offset to register or an effective address determined by a base pointer, an offset pointer and a scale factor. We hope to use some unified operands to represent similar operations. For example, we consider that the immediate value often doesn't play a significant role in representing semantic information. Different function names or different jump addresses do not contain much information about variable behavior. Therefore, we choose to neglect this binary-specific information and replace them with unified elements. To pad the length of each VUC, we fill *BLANK* to assembly code with less than 2 operands, such as $jump$ operation shown in Table II row 4.

Original assembly	Generalized assembly
add -0xD0,%rax	add -0xIMM,%rax
lea -0x300(%rbp, %r9, 4), %rax	lea -0xIMM(%rbp, %r9, 4), %rax
jmp 3bc59	jmp ADDR BLANK
callq 3bc59 <bfd_zalloc>	callq ADDR <FUNC>

TABLE II
EXAMPLES OF GENERALIZATION.

As shown in Table II, we use regular expression to generalize immediate numbers, function names and addresses. The first two examples show how we treat immediate numbers. Immediate numbers are either offset to a base pointer or a value for arithmetic calculation. We replace these immediate numbers with *0xIMM*. Note that we don't touch the scale factor of effective address since it is related to variable length. The remaining 2 examples show how we treat jump and call instruction. We replace target addresses with *ADDR* since they only appear in the *instruction context* and don't impact how we interpret the usage of the variables. Unconditional jump often suggests loop, conditional jump often suggests branch and call suggests function call. If *objdump* cannot find function name, its position is filled with a *BLANK*.

In our experiment, our generalization method would cover over 99% of the instructions for newly come samples.

C. Embedding

To learn the representation of each VUC in the lower dimension, and retain the relationship with neighboring instructions at the same time, we choose Word2Vec as our embedding technique. The objective function of the method is as follows:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m, j\neq 0} log_2 P(Ins_{t+j}|Ins_t) \qquad (1)$$

where T is the number of VUC, Ins_t denotes the target instruction (with 1 operation and no more than 2 operands), Ins_{t+j} denotes the neighboring instructions, $P(Ins_{t+j}|Ins_t)$ represents the appearance possibility of Ins_{t+j} when the target instruction is Ins_t. We let maximum distance m to be 5.

We directly use the `gensim` [27] library to train the Word2Vec model, and it translates each input VUC into a vector of fixed length which is 32 for CATI.

V. PREDICTION MODELS

A. Multi-Stage Classifier

As we mentioned before, we develop a system called CATI that can recover 19 types of variables from stripped binaries. This covers all base types defined in C99 standard. We don't classify pointer to these different base types because it's hard to statically trace pointers. We also add some widely used types according to our statistics. Nowadays, deep learning techniques are powerful enough to distinguish 19 classes within one model. But to make the model more interpretable and accelerate the training phase, we train a multi-stage classifier containing six different classifiers. To employ the *instruction context* as new features, we choose the convolutional neural network (CNN) as our prediction model after several empirical attempts. Each separate stage uses a different CNN models with a similar structure but different parameters as a classifier to infer a specific cluster of variable types. We employ a common 2-layer CNN model (32-64) with a fully connected layer (1024) to run the result.

As shown in Figure 5, we illustrate the multi-stage classifier as a tree-based structure. In the first stage (Stage 1), the binary classifier identifies whether the VUC belongs to a `pointer` variable or not. VUCs tagged as pointer variables are processed at Stage 2-1 and tagged as `pointer to void`, `pointer to struct`, `pointer to arithmetic`. At Stage 2-2, VUCs tagged as non-pointer variables will be processed to be tagged as one of `struct`, `bool`, `char`, `float`, `int`. Whether a variable has been tagged as `char`, `float`, `int`, will finally be classified more detailed in Stage 3-1, Stage 3-2, Stage 3-3. Each stage is trained separately with 2-layered CNN with slightly different parameters.

In this paper, we have tried our best to classify as many types as possible. We recover over 40 kinds of types listed in C programming language. Here come some reasons for

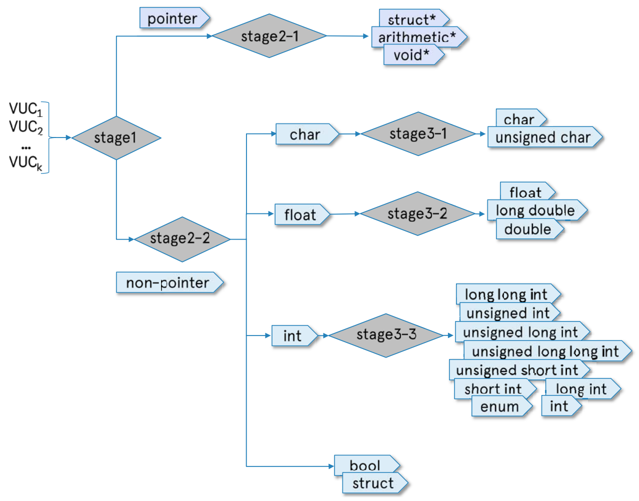


Fig. 5. Multi-stage classifier.

our decision to recover 19 of them. Firstly, we choose to classify all of the types of non-pointer variables except `union` because `union` has various behaviors that cannot be classified into one class. Secondly, we divide the type of pointer into three types: `pointer to void`, `pointer to struct`, `pointer to arithmetic`. The first two types are easy to think over. The appearance of the third type is because the static-based approach cannot capture the run time behavior of the variable, and the instant pointed by the variable cannot be fixed. Therefore, we cluster all the basic variable types to a set of the types called `pointer to arithmetic`. To the best of our knowledge, we have achieved the largest amount of variable types at present, and we cover all the types which have been recovered by the previous works.

B. Prediction and Voting

After the multi-stage classifier is trained, we can use it to predict the most possible type for each VUC extracted from unseen stripped binaries and vote for the final result.

Firstly, we take the same measures as the training set data for the binaries from the testing set: disassemble, locate variables, extract VUCs, generalization, and embedding. Here, we leverage IDA Pro to help us to disassemble the binary code and locate the variables in every binary. The embedding model is the one trained from the training set. Then, with prepared variables $V = \{VUC_1, VUC_2, ..., VUC_N\}$, where $VUC_i = \{Ins_{i-10}, .., Ins_{i-1}, Ins_i, Ins_{i+1}, ..., Ins_{i+10}\}$, we input each VUC into the classifier to predict the result.

Here, we define the classifier of each stage as function S_u, such as S_{2-1} representing the classifier of Stage 2-1. Variable V containing N VUCs is represent as $V = [VUC_1, VUC_2, ..., VUC_N]$. V is a three-dimensional tensor, with a size of $N \times L \times E$. N, L, E respectively represent the number of VUC, length of VUC, length of embedding size. Here, L and E are constants, which equal to 21 and 32.

We calculate the result of V as follows,

$$S_u(V) = Z \tag{2}$$

where Z is a matrix with size of $N \times C$. C denotes the number of classes in stage i. Z_{ij} denotes the confidence of ith VUC classified to jth class, and $\sum_{j=1}^{C} Z_{ij} = 1.0$. To increase the influence of result with high confidence, we come out a solution with the following formula:

$$Z'_{ij} = \begin{cases} 1.0 & \text{if } Z_{ij} \geq 0.9 \\ Z_{ij} & \text{if } Z_{ij} < 0.9 \end{cases} \tag{3}$$

Here, we enlarge the power of the result with high confidence to dominate the final decision, and we do several empirical experiments to set the threshold as 0.9. We make the final decision of V based on the confidence output of each VUC. The final result is calculated as follows,

$$argmax(\sum_{i=1}^{N} Z'_{i1}, \sum_{i=1}^{N} Z'_{i2}, ..., \sum_{i=1}^{N} Z'_{iC}) \tag{4}$$

where, $\sum_{i=1}^{N} Z'_{ij}$ denotes the total confidence of VUCs of V classified to class j, and the highest make the final decision.

VI. Implementation

In this section, we present the implementation of our system. CATI extracts the data flow of the target variable from stripped binaries using IDA Pro [12]. We developed a python script that uses IDA APIs to automatically achieve this. To annotate the correct type for the training VUCs, we parse DWARF [26] debug information from non-stripped binaries. After obtaining the tracing information of each variable assisted by IDA Pro, we implement a tree-like multi-stage classifier using the machine learning package Keras [28]. Finally, we utilize a python script to vote for the final result for each variable. For the evaluation part, we use machine learning library scikit-learn [29] which calculates the metrics for each stage.

All our experiments were conducted on a PC with 16GB memory, 1 Intel i7-6700k CPU(4.0 GHz) and 1 NVIDIA GTX 1070 GPU with 8GB graphics memory.

VII. Evaluation

A. Setup

Here, we describe our dataset and metrics for evaluation of the performance of CATI in each aspect.

Data Set. We create a comprehensive training data set from several open-source software projects. Here we enumerate some projects of different categories: OS tools (GCC, coreutils, binutils, etc), network programs (php, nginx, etc), computationally intensive programs (xpdf, zlib, etc) and projects like R and Python which integrate packages of different categories. In total, 2141 binaries are used to train the models. We choose popular and well-written projects so that it reviles the distribution of types in reality. For the diversity of training data, we build each project with different optimization levels (-O0 to -O3), but all with the same compiler—GCC. The reason for controlling the same compiler to compile the applications

		bash	bison	cflow	gawk	grep	gzip	inetutils	less	nano	R	sed	wget
Stage1	P	0.93	0.89	0.88	0.88	0.89	0.94	0.89	0.86	0.87	0.89	0.91	0.89
	R	0.93	0.89	0.89	0.88	0.89	0.93	0.89	0.86	0.87	0.89	0.91	0.89
	F1	0.93	0.89	0.88	0.88	0.89	0.93	0.89	0.86	0.87	0.89	0.91	0.89
Stage2-1	P	0.76	0.76	0.79	0.77	0.86	0.71	0.70	0.79	0.79	0.69	0.89	0.76
	R	0.68	0.76	0.79	0.78	0.87	0.70	0.71	0.70	0.79	0.70	0.89	0.73
	F1	0.68	0.75	0.78	0.77	0.86	0.70	0.70	0.71	0.78	0.68	0.89	0.73
Stage2-2	P	0.91	0.89	0.81	0.82	0.88	0.91	0.77	0.93	0.87	0.87	0.89	0.83
	R	0.91	0.88	0.82	0.82	0.88	0.91	0.76	0.92	0.86	0.88	0.88	0.84
	F1	0.91	0.88	0.81	0.81	0.88	0.91	0.74	0.92	0.86	0.88	0.88	0.83
Stage3-1	P	0.88	0.92	0.92	0.81	0.81	0.93	0.88	0.94	0.79	0.92	0.84	0.81
	R	0.80	0.91	0.60	0.73	0.81	0.81	0.83	0.86	0.79	0.88	0.80	0.81
	F1	0.83	0.91	0.68	0.76	0.81	0.84	0.85	0.88	0.79	0.89	0.81	0.81
Stage3-2	P	1.00	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
	R	0.33	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
	F1	0.50	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
Stage3-3	P	0.91	0.84	0.87	0.73	0.75	0.77	0.76	0.94	0.71	0.84	0.75	0.74
	R	0.85	0.78	0.80	0.72	0.74	0.75	0.74	0.81	0.70	0.84	0.72	0.72
	F1	0.88	0.81	0.83	0.72	0.74	0.74	0.75	0.86	0.70	0.84	0.72	0.72

TABLE III
VUC PREDICTION RESULT OF 12 APPLICATIONS IN 6 STAGES MEASURED BY PRECISION(P), RECALL(R) AND F-1 SCORE(F1).

		bash	bison	cflow	gawk	grep	gzip	inetutils	less	nano	R	sed	wget
Stage1	P	0.95	0.92	0.92	0.92	0.91	0.96	0.94	0.88	0.89	0.92	0.93	0.92
	R	0.95	0.92	0.92	0.91	0.91	0.96	0.94	0.87	0.87	0.92	0.93	0.92
	F1	0.95	0.92	0.92	0.91	0.91	0.96	0.94	0.87	0.87	0.92	0.93	0.92
Stage2-1	P	0.76	0.74	0.79	0.71	0.87	0.76	0.73	0.79	0.79	0.74	0.89	0.75
	R	0.64	0.71	0.76	0.72	0.85	0.75	0.70	0.65	0.77	0.70	0.88	0.67
	F1	0.63	0.70	0.75	0.69	0.85	0.74	0.69	0.66	0.75	0.68	0.88	0.66
Stage2-2	P	0.92	0.92	0.92	0.87	0.92	0.96	0.89	0.96	0.90	0.89	0.93	0.89
	R	0.92	0.90	0.92	0.87	0.92	0.96	0.89	0.95	0.89	0.90	0.93	0.89
	F1	0.92	0.90	0.92	0.86	0.92	0.96	0.89	0.95	0.89	0.90	0.93	0.89
Stage3-1	P	0.92	0.90	0.89	0.81	0.82	0.93	0.90	0.97	0.80	0.95	0.86	0.81
	R	0.87	0.86	0.67	0.76	0.81	0.91	0.87	0.88	0.82	0.94	0.81	0.81
	F1	0.89	0.87	0.71	0.77	0.81	0.92	0.89	0.91	0.80	0.95	0.81	0.80
Stage3-2	P	0.00	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
	R	0.00	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
	F1	0.00	1.00	1.00	1.00	1.00	–	1.00	1.00	–	0.99	–	1.00
Stage3-3	P	0.93	0.84	0.90	0.79	0.83	0.82	0.80	0.94	0.79	0.83	0.83	0.80
	R	0.91	0.83	0.86	0.81	0.82	0.80	0.80	0.86	0.78	0.87	0.81	0.78
	F1	0.92	0.83	0.88	0.79	0.82	0.79	0.80	0.89	0.77	0.84	0.81	0.77

TABLE IV
VARIABLE PREDICTION RESULT OF 12 APPLICATIONS IN 6 STAGES AFTER VOTING MEASURED BY PRECISION(P), RECALL(R) AND F-1 SCORE(F1).

is that we want to focus on studying one compiler's behavior. Meanwhile, we believe that our prototype can transfer easily to other compilers. To validate our idea, we do the additional experiments on Clang which will be discussed detailly in SectionVIII. With the help of IDA pro [12] and DWARF [26], we successfully disassemble the binary program, leverage the debug information to label the ground truth of each VUC, and group VUCs that belong to the same variable. Furthermore, to test the prediction accuracy of CATI and compare with former works, we carefully select some applications as a benchmark which are different from the training set to prove the general performance of our method.

Metrics. CATI is a machine learning-based method, so we use three performance metrics commonly used to evaluate machine learning classifiers: precision (P), recall (R) and F1 score. Formally, they are defined as follows:

$$P = \frac{TP}{TP+FP}, R = \frac{TP}{TP+FN}, F1 = \frac{2*P*R}{P+R}$$

where TP is the true positives, FP is the false positives, FN is the false negatives. Precision is the ratio of cases where the predicted value is equal to the given value, which is the closeness of the measurements to each class(i.e., the accuracy ratio of discovered variables). Recall leads to the proportion of correct predictions over the set of their class. F1 score is a balance measurement that is calculated by precision and recall. All three metrics are in the range of 0 to 1.

B. Evaluation

To objectively measure the performance of our type inference method, we assume the variable location of assembly code is given for every binary. However, in general, we can leverage the $Variable\ Recovery$ part of DEBIN [1] to locate the variable operations in assembly code whose accuracy can achieve about 90%.

Evaluation on Test Set. Firstly, we discuss the performance of the multi-stage classifier on predicting the most likely type of each VUC. The result of 12 different applications has shown

Type	S1-R	S2-R	S3-R	ACC	Support	cnt-same	cnt-all	c-rate
bool	1.00	0.76	1.00	0.76	1400	1.38	5.37	25.68%
struct	0.91	0.61	1.00	0.58	6874	3.60	5.46	65.85%
char	0.99	0.50	0.93	0.45	2697	1.51	4.21	35.77%
unsigned char	1.00	0.76	0.65	0.56	404	2.40	5.11	47.05%
float	1.00	0.88	0.88	0.88	8	1.22	4.44	27.50%
double	0.99	0.91	1.00	0.91	13972	4.25	6.24	68.02%
long double	1.00	0.98	0.57	0.57	153	2.90	6.30	45.97%
enum	0.99	0.99	0.11	0.11	2654	1.18	4.92	24.03%
int	0.99	0.98	0.95	0.93	38591	3.10	5.81	53.36%
short int	1.00	0.78	0.15	0.13	46	0.81	5.22	15.45%
long int	0.71	0.97	0.72	0.57	5051	3.17	5.97	53.14%
long long int	0.57	1.00	0.00	0.00	21	1.23	3.00	41.03%
unsigned int	0.99	0.97	0.28	0.27	1808	4.72	6.72	70.16%
short unsigned int	1.00	0.80	0.74	0.67	70	1.02	4.17	24.54%
long unsigned int	0.61	0.96	0.85	0.50	6188	2.43	5.89	41.29%
long long unsigned int	0.71	0.81	0.00	0.00	21	2.57	7.86	32.73%
void*	0.91	0.18	1.00	0.18	2820	3.06	4.88	62.57%
struct*	0.95	0.92	1.00	0.88	36876	2.01	5.02	40.09%

TABLE V
EVALUATION RESULT OF EACH VARIABLE TYPE.

in Table III. For Stage 1 to Stage 3-3, the table shows the performance of classifying pointer and non-pointer, subclass of pointers, subclass of non-pointer, subclass of type `char`, subclass of type `float`, and subclass of type `int`. P, R, $F1$ separately represents precision, recall ,and F1-score which are usual metrics in the machine learning field. Stage 1 shows the best performance overall applications. It is easy to come out that the features to distinguish pointers and non-pointers are obvious. The pattern of operations and operands used by pointer and non-pointers can be widely divergent, which has employed by previous work to extract rules. Stage 2-1 doesn't perform well on these tasks, where the average measure indexes are about 0.75. Especially, application *R* behaves the worst on this stage, which has the most amount of VUCs tagged as a `pointer`. The reasonable excuse of the performance results in stage 2-1 is that the behavior of pointer variables is too uncertain to capture which cannot be easily inferred by the *instruction context*. While in stage 3, stage 3-1 and stage 3-3 seem to do well in classifying the types, for only two types (`char` and `unsigned char`) making up stage 3-1 and types in `int` family behaving differently on the usage of registers. The registers storage different lengths of `int` family variables which are very different, let alone whether the variables are signed or unsigned. It is interesting to notice that stage 3-2 has a different result with other stages. Applications *gzip*, *nano*, and *sed* do not have `float` family variables. Besides application *R*, the rest of 8 applications occupy less than 200 VUCs belongs to `float`. Even some of the applications only have 1 VUC tagged as a `float`. Fortunately, even the data set is unbalanced in stage 3-2, application *R* still achieves a considerable performance to classify over 10,000 `float` family variables which are owed to the usage of *instruction context*.

To evaluate the performance of CATI in the real world, we present the result after the voting procedure in Table IV. Each data in Table IV is one to one corresponding to the data in Table III. After voting, the wrong case may be adjusted to the right class by the rule of the minority obeying the majority. In Stage 1, Stage 2-2, Stage 3-1 and Stage 3-3, the performance of all the applications has a great improvement compared to Table III. However, the result of Stage 2-1 and 3-2 has a decrease. In Stage 2-1, each VUC is classified to most likely type, but the diverse behavior of `pointer` family variables puzzling the voting mechanism to make an uncertain decision. In Stage 3-2, application *bash* only has 1 related variable which consists of 3 VUCs, of which only 1 VUC is correctly classified. Therefore, the result comes to zero.

	VUC		Variable	
	Accuracy	Support	Accuracy	Support
bash	0.70	79767	0.73	12820
bison	0.69	20863	0.69	4095
cflow	0.69	9133	0.72	1547
gawk	0.66	53605	0.68	8017
grep	0.72	23254	0.76	3563
gzip	0.67	4088	0.75	725
inetutils	0.62	130130	0.68	21183
less	0.66	7633	0.67	1563
nano	0.65	21845	0.67	3703
R	0.70	616297	0.72	93495
sed	0.74	18246	0.78	2637
wget	0.65	38571	0.66	6426
Total	0.68	1023432	0.71	159774

TABLE VI
EVALUATION RESULT ON EACH APPLICATIONS ON THE GRANULARITY OF VUC AND VARIABLE.

To thoroughly evaluate the system on the pipeline, we test the pipeline result of each application. Table VI shows the result of all applications in the test set. Weighted accuracy of all applications on the granularity of VUC is 0.68, which is test on over more than 1,000,000 VUCs. And Weighted accuracy of all applications on the granularity of variable is 0.71, which is test on over more than 150,000 variables. Here, our system CATI achieves a considerable result of variable classification of 19 types, and the result after voting mechanism increases by about 0.03 on the metric of accuracy. Application *sed* achieves the highest accuracy (0.78) of all applications. And even the worst case of *wget* achieves an accuracy of 0.66.

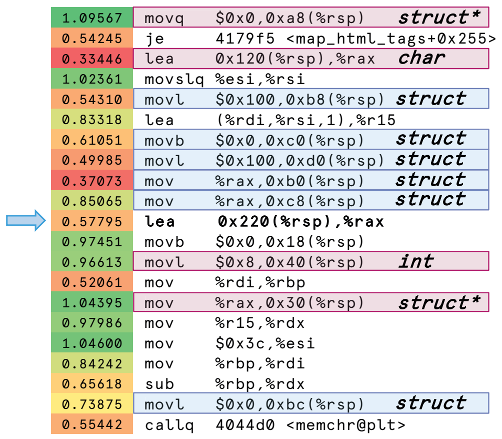
Comparison with DEBIN. We set up another experiment on CATI to compare with the state-of-art method DEBIN [1]. To make the competition as enough fair as possible, we set the preparation work as similar as possible with DEBIN. We randomly select 300 binaries from 830 Linux Debug Symbol Packages (data source of DEBIN), compiled in x86_64 architecture, to fulfill the same type inference mission in DEBIN – classify 17 different types: `struct`, `union`, `enum`, `array`, `pointer`, `void`, `bool`, `char`, `short`, `int`, `long` and `long long` (both `signed` and `unsigned` for the last five). Our system CATI has a better performance than DEBIN in a similar situation to finish the same task, in which we achieve about 11% higher on accuracy (0.84 V.S. 0.73). As expected, CATI performs better in the new settings, because we enrich the features of each variable which overcome the problem of *uncertain samples*, and we make the final decision by voting for traceable instructions.

Understanding the Clustering Phenomenon. To validate the discovery of the *Same type variable clustering phenomenon*, we study the result of each variable which is shown in Table V. Column 2 to 5 separately represents the recall of Stage 1, Stage 2-2, Stage 3 and weighted average final result. *Cnt-same*, *cnt-all* and *c-rate* in column 7 to 9 separately represents the average number of variable instructions in one VUC which have the same variable type with the target variable, the average number of variable instructions in one VUC, and the ratio which is calculated by the former two columns. Type `double`, `int`, `unsigned int` perform well in all stages, which can be explained by the high ratio of same type variable clustering shown in Column 9. It is worth to notice that, type `bool` has good performance with a low clustering ratio, while type `struct` has poor performance with a high clustering ratio. Our explanation is that variables with `bool` type have comparatively simple usage pattern and their instructions are not complex. To contrast, variables with `struct` type consist of many members, therefore their usages are really diverse. Unfortunately, both type `long long` and `long long unsigned int` do not get over Stage 3-3, even though the clustering rate is considerable, of which the reason maybe these kinds of variables are few. Last but not least, the total recall of each variable is roughly positively correlated with the clustering ratio.

To find the decisive factor of prediction result, we bring in a new measurement index ϵ, which is calculated as follows:

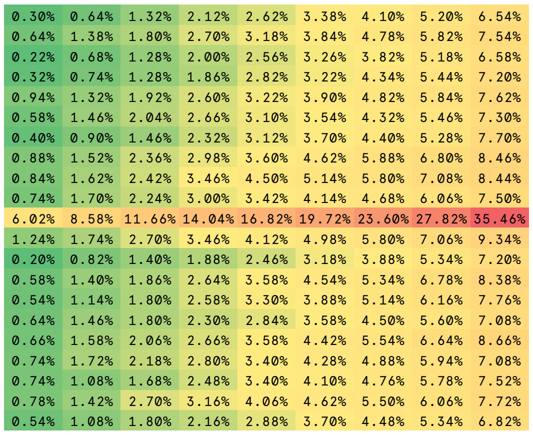
$$\epsilon_k = \frac{S_u(R(VUC_j, k))}{S_u(VUC_j)}, k \in [1, 21], j \in T \qquad (5)$$

where $S_u(VUC_j)$ denotes the confidence of VUC_j predicted in Stage u, k denotes the kth position of the instruction in VUC_j, and j denotes the jth VUC in the data set T. Function R occludes one specific instruction. As shown in formula (5), R occludes the kth instruction in VUC_j with $BLANK$. The result of $S_u(R(VUC_j, k))$ indicates the confidence of VUC_j in Stage i without the information of kth



```
1.09567  movq    $0x0,0xa8(%rsp)       struct*
0.54245  je      4179f5 <map_html_tags+0x255>
0.33446  lea     0x120(%rsp),%rax      char
1.02361  movslq  %esi,%rsi
0.54310  movl    $0x100,0xb8(%rsp)     struct
0.83318  lea     (%rdi,%rsi,1),%r15
0.61051  movb    $0x0,0xc0(%rsp)       struct
0.49985  movl    $0x100,0xd0(%rsp)     struct
0.37073  mov     %rax,0xb0(%rsp)       struct
0.85065  mov     %rax,0xc8(%rsp)       struct
0.57795  lea     0x220(%rsp),%rax
0.97451  movb    $0x0,0x18(%rsp)
0.96613  movl    $0x8,0x40(%rsp)       int
0.52061  mov     %rdi,%rbp
1.04395  mov     %rax,0x30(%rsp)       struct*
0.97986  mov     %r15,%rdx
1.04600  mov     $0x3c,%esi
0.84242  mov     %rbp,%rdi
0.65618  sub     %rbp,%rdx
0.73875  movl    $0x0,0xbc(%rsp)       struct
0.55442  callq   4044d0 <memchr@plt>
```

a) Importance Visualization



0.30%	0.64%	1.32%	2.12%	2.62%	3.38%	4.10%	5.20%	6.54%
0.64%	1.38%	1.80%	2.70%	3.18%	3.84%	4.78%	5.82%	7.54%
0.22%	0.68%	1.28%	2.00%	2.56%	3.26%	3.82%	5.18%	6.58%
0.32%	0.74%	1.28%	1.86%	2.82%	3.22%	4.34%	5.44%	7.20%
0.94%	1.32%	1.92%	2.60%	3.22%	3.90%	4.82%	5.84%	7.62%
0.58%	1.46%	2.04%	2.66%	3.10%	3.54%	4.32%	5.46%	7.30%
0.40%	0.90%	1.46%	2.32%	3.12%	3.70%	4.40%	5.28%	7.70%
0.88%	1.52%	2.36%	2.98%	3.60%	4.62%	5.88%	6.80%	8.46%
0.84%	1.62%	2.42%	3.46%	4.50%	5.14%	5.80%	7.08%	8.44%
0.74%	1.70%	2.24%	3.00%	3.42%	4.14%	4.68%	6.06%	7.50%
6.02%	8.58%	11.66%	14.04%	16.82%	19.72%	23.60%	27.82%	35.46%
1.24%	1.74%	2.70%	3.46%	4.12%	4.98%	5.80%	7.06%	9.34%
0.20%	0.82%	1.40%	1.88%	2.46%	3.18%	3.88%	5.34%	7.20%
0.58%	1.40%	1.86%	2.64%	3.58%	4.54%	5.34%	6.78%	8.38%
0.54%	1.14%	1.80%	2.58%	3.30%	3.88%	5.14%	6.16%	7.76%
0.64%	1.46%	1.80%	2.30%	2.84%	3.58%	4.50%	5.60%	7.08%
0.66%	1.58%	2.06%	2.66%	3.58%	4.42%	5.54%	6.64%	8.66%
0.74%	1.72%	2.18%	2.80%	3.40%	4.28%	4.88%	5.94%	7.08%
0.74%	1.08%	1.68%	2.48%	3.40%	4.10%	4.76%	5.78%	7.52%
0.78%	1.42%	2.70%	3.16%	4.06%	4.62%	5.50%	6.06%	7.72%
0.54%	1.08%	1.80%	2.16%	2.88%	3.70%	4.48%	5.34%	6.82%

b) Importance Distribution of Test Data

Fig. 6. Importance visualization and distribution of example case.

instruction. ϵ_k is an index ranged in $(0, +\infty)$ to measure the importance of kth instruction. A smaller value of ϵ_k indicates more significance of the instruction.

As shown in Figure 6 a), the left side is the result of ϵ of each instruction. It is obvious that the variable instructions with the same variable type play a more important role in type inference. However, the instruction in column 3 operates a variable with type `char`, which seems to take the lead of prediction with a small ϵ. The reasonable explanation of this situation is that the representation of column 3 is the same as the central instruction (column 11). It is easy to guess that the model regards these two instructions as the same variable's operation. As shown in Figure 6 b), the heat map reflects the statistical result of ϵ for each instruction among our test data. Each row represents the location of instructions in VUCs, and each column represent the rate of instructions whose ϵ is ranged in (0,1), (0.1,1), ..., (0.8,1), (0.9,1). Take the percentage in column 11, row 5 as an example, it denotes that 16.82% of central instructions have ϵ ranged in (0.5,1). The tendency of the heat map indicates that the model is able to pay attention to the central instruction which operates the target variable, and the closer instructions have a more positive influence on the prediction result. To contrast, we find the next-door neighboring instructions already vary a lot from the central instruction in all rows, not to speak of the rest of the instructions. The distribution indicates that the method of feature extraction brings in noises in some aspects.

At last, we further investigate the significance of VUC to infer variable types, and we do the following additional experiments here.

Training and Inference Speed. It takes about two hours to train the classification models based on CNN for six stages, and it takes about three hours to train the Word2Vec model. The extracting phase for test data lasts about 24 minutes, and prediction time (including inference and voting) is about 5 minutes. Each binary takes about 6 seconds on extraction and prediction. Fortunately, the speed of CATI is acceptable, because the system is based on static analysis for binaries. It would be possible bringing in some dynamic analysis in our future work to increase the accuracy of CATI.

VIII. Discussion

We mainly work on the binaries compiled from GCC because it is more widely used and previous works are all based on GCC. After the thorough experiment on the data set generated from GCC to infer the variable type from stripped binaries, we still need to confirm that our prototype can work on other mainstream compilers, like Clang.

In this work, we concern about the method of locating variables and inferring the types of them from stripped binaries, but we are not sure whether the method is compiler sensitive. We want to deeply investigate whether the behavior of variables in source code compiled to a low-level representation (e.g., binary code) is highly correlated to its compiler. To validate our hypothesis that our method is transferable, we finish the following experiments. Except for the compiler which we substitute with Clang, the rest of the experimental setups are the same as the settings in section VII. We train new models with the binaries compiled from Clang and test the model with the same applications as before. The performance of 6 classifiers is shown in Table VIII.

Stage	Precision	Recall	F1-score
Stage1	0.95	0.95	0.95
Stage2-1	0.86	0.87	0.86
Stage2-2	0.94	0.95	0.94
Stage3-1	0.88	0.88	0.88
Stage3-2	0.99	0.99	0.99
Stage3-3	0.86	0.87	0.86

TABLE VII
Evaluation result of applications compiled from Clang.

According to the table, we can see that all 6 classifiers achieve a good result in 3 stages. The total accuracy of all variables in our test applications is 82.14%, which means that most of these variables compiled from Clang can be correctly classified in all stages. The transformed model can well solve the type inference problem in the field of Clang. Hence, we can conclude that the design of the prototype is transferable.

What's more, to make the system more complete, we try to set up a classifier before our tree-based variable type classifiers to identify the scatter binaries from which compiler. For different register usages between GCC and Clang, we successfully train a model with 100% accuracy which is just used VUCs exactly from previous experiments.

Here, the results indicate that the prototype is transferable between different compilers, even though we still need to identify the source compiler of stripped binaries. We also find different compiler options may influence inferring types, which will lead us to our future work.

IX. Related Work

In this part, we introduce some works closed to us.

Existing Works. Some previous works focus on variable recoveries, such as DIVINE [25]. But we leverage the technique from Hex-Rays to avoid this problem and concentrate more on type inference. For type inference, REWARDS [16] and TIE [13] really perform well in the rule-based method and get a considerable result. ELKAVYA [30] leverages machine learning to identify 7 types of functions. In recent years, some works try to distinguish variable types by machine learning methods. Xu [18] and Maier [22] have successfully identified the part of variable types. The former employs Support Vector Machine (SVM), and the latter employs N-grams. DEBIN [1] accomplished the mission of variable recovery, type recovery, and name recovery at the same time with the help of Conditional Random Field. To the best of our knowledge, CATI is the first system that takes the problem of type inference as text classification with the help of *instruction context* and overcomes the challenge of *uncertain samples*.

Approaches for Binary Analysis. There is always a trade-off between static analysis and dynamic analysis in the field of binary analysis. Under the situation of our work, some researchers employ static analysis [2], [15], [31], [32] to pursue the code coverage, while others utilizing dynamic analysis [16], [33], [34] to trace the target on the execution time. Intuitively, some works propose hybrid approaches to make the best of both static and dynamic methods. However, traditional approaches seem to reach the end, and machine learning approaches are well adapted to the security field in time. Especially, the strong mapping ability of machine learning helps experts to solve many problems in binary analysis. With the help of deep learning, we can find some connections between binaries and source code, which is usually invisible to the human. It is natural that with the development of large amounts of code, some tasks achieve a satisfying result, such as malware classification [35]–[37] and function identification [19], [30], [38].

Our work does not directly mix new approaches to reach better performance, but we utilize the advantage of machine learning to overcome the partial weakness of static analysis. We employ static analysis for the reason of its efficiency and code coverage, and we try our best to bring in extra information which may assist to do the prediction.

X. Conclusion

We present a novel approach for inferring the variable type from stripped binaries. To solve the problem of *orphan variables* and *uncertain samples*, we leverage a new feature called VUC to transform the representation of variables which is inspired by our discovery – *same type clustering phenomenon*. To validate our hypothesis, we collect a comprehensive data set to establish a multi-stage classifier that is trained by convolutional neural networks (CNN).

Our system, called CATI, uses CNN models and a voting mechanism to infer variable types from unseen stripped binaries. The experiment of CATI shows that the system is more accurate than previous works in different aspects. As a new feature, VUC really plays a crucial part in our work to improve the classification result.

Acknowledgment

We would like to thank our shepherd Nuno Neves and the anonymous reviewers for their helpful feedback. We are grateful for Yifei Huang's contribution on polishing the paper. This work was supported in part by grants from the Chinese National Natural Science Foundation (NSFC 61272078).

References

[1] 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*. ACM, 2018, pp. 1667–1680.

[2] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries." in *NDSS*, 2015.

[3] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

[4] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 337–352.

[5] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[6] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[8] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[9] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization*. IEEE, 2019, p. 0.

[10] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 667–678.

[11] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: Using input type inference to improve fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 505–517.

[12] "IDA Pro," https://www.hex-rays.com/.

[13] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." in *NDSS*, 2011.

[14] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 51–60, 2013.

[15] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[16] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 2010, p. 5.

[17] I. Haller, A. Slowinska, and H. Bos, "Mempick: High-level data structure detection in c/c++ binaries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 32–41.

[18] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *International Conference on Formal Engineering Methods*. Springer, 2017, pp. 430–446.

[19] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "{BYTEWEIGHT}: Learning to recognize functions in binary code," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 845–860.

[20] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[21] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[22] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 288–308.

[23] D. Zeng and G. Tan, "From debugging-information based binary-level type inference to cfg generation," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 2018, pp. 366–376.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[25] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 1–28.

[26] "The DWARF Debugging Standard." http://dwarfstd.org/.

[27] "gensim," https://radimrehurek.com/gensim/.

[28] "Keras," https://www.tensorflow.org/guide/keras.

[29] "scikit-learn," https://scikit-learn.org/.

[30] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium*, 2017, pp. 99–116.

[31] D. Dewey and J. T. Giffin, "Static detection of c++ vtable escape vulnerabilities in binary code." in *NDSS*, 2012.

[32] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 27–41.

[33] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "Marx: Uncovering class hierarchies in c++ programs." in *NDSS*, 2017.

[34] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos, "Dsibin: identifying dynamic data structures in c/c++ binaries," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 331–341.

[35] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 470–478.

[36] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown malcode detection using opcode representation," in *European conference on intelligence and security informatics*. Springer, 2008, pp. 204–215.

[37] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2000, pp. 38–49.

[38] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 611–626.