What Exactly Determines the Type? Inferring Types with Context

Ligeng Chen*
*National Key Laboratory for Novel Software Technology, Nanjing University, Nanajing 210023, China
chenlg@smail.nju.edu.cn

***Abstract*—Closed-source programs lack crucial information vital for code analysis because that information is stripped on compilation to achieve smaller executable size. Variable type information is fundamental in this process. In this paper, we implement a system called CATI (Context-Assisted Type Inference), 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.**

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

I. Introduction

Binary analysis is always a hot topic in the security field. 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.

Variable takes a vital part of program semantics, and the type suggests the functionality of themselves. Recently, some works (e.g., [1]) leverage dynamic analysis to collect run-time information, while others (e.g., [2]) employ static analysis. Besides excessively searching for endless rules, all the previous works are appealing to much expert knowledge. With diverse compiler configurations, 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.

To overcome the difficulties above, a considerable number of new methods and tools [3], [4] 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 [3] traces the data flow of the instructions using BAP, then builds a dependency graph, finally uses a probabilistic graphical model to infer the variable type and its name. It relies heavily on data dependency analysis to build the graph. TypeMiner [4] also builds data object trace via data dependency. 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. Meanwhile, Zeng [5] shows that even 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*.

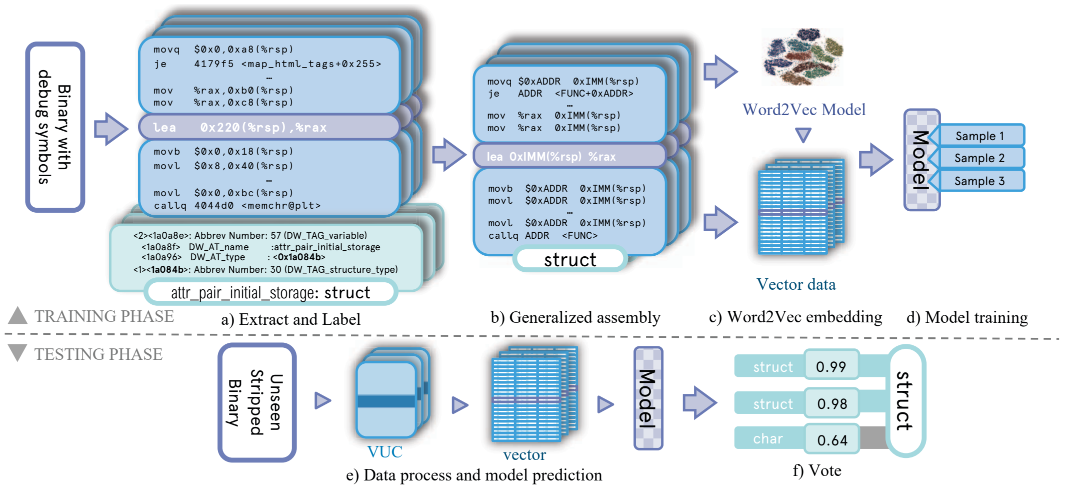
Given all these difficulties, we need to employ richer features to infer the variable type. Fortunately, although the dependency of our target instruction is sometimes not easy to build, we can take the *instruction context* into account. In our work, we focus on memory access instructions and dereference 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. In each VUC, it contains the target variable instruction with 10 instructions forward and 10 instructions backward. Moreover, if we can find several instructions that operate the same variable, we will utilize a voting mechanism to make the final decision. 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 19 types of variables.

II. Motivation and Method

Emprical Study. To validate our consideration, we did a thorough investigation of the real-world programs. Unfortunately, we survey all the variables in our data set extracted by IDA Pro — a total of 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 [3]. On the other hand, two cases with the same instruction(s) but

978-1-7281-7260-6/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN-S50200.2020.00038



```
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)
```

```
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>
```

Fig. 1. 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).

different types are *uncertain samples*. In our study, *uncertain samples* take up over 97% of *orphan variables*. That is, about 34 % of variables cannot be distinguished without extending the features.

To verify our hypothesis of *Same Type Variable Clustering Phenomenon*, we carry out statistics on over 107 thousand variables (VUC). 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. The result is impressive and it can support the following works.

Workflow. As shown in Figure 1, CATI is guided by the following process. CATI firstly locates the target instructions corresponding to variables with the help of IDA Pro in stripped binaries. Next, CATI generalizes the operands of each instruction, such as actual address, actual function name, and immediate value. To vectorize the assembly code, CATI employs *Word2Vec* model to represent each VUC, which is often used in natural language processing. Then, we train a multi-stage classifier with a convolutional neural network (CNN). Using the 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. Lastly, we add confidence results of each type for a confidence-based voting mechanism.

III. Result and Discussion

We set up a comprehensive training set consist of 2141 binaries from popular and well-written projects. For the diversity of training data, we build each project with different optimization levels (-O0 to -O3), but all with the same compiler—GCC. To validate the tool, we test it on total different 12 applications (*e.g.*, binutils, inetutils).

CATI achieves 68% of accuracy over 1,023,432 VUCs. After voting, it reaches 71% of accuracy over 159,774 variables. We also set a fair experiment with SOTA – DEBIN [3] on 17 types, and we got 11% higher on accuracy (0.84 V.S. 0.73).

To confirm that our prototype can work on other mainstream compilers, we apply our tool on Clang as an additional experiment. With the same data set and the same experiment set, CATI achieves 82% of accuracy over all variables.

However, CATI is still a black box for us. So we raise the following questions.

- RQ1: How do the different compiler options (e.g. compiler, version, optimization level, etc.) influence the appearance of different variables?
- RQ2: Could we parse the result of machine learning and extract it into rules which may help to understand the behavior of compiler on variables?
- RQ3: Could we set up a multi-modal model that gathers information from static and dynamic to better infer the semantics from stripped binaries?

Our additional experiment indicates that different compilers' behavior is distinguishable on the granularity of assembly code, while different optimization levels behave closer but still have differences. Further more, we want to investigate how a small change in compiler finally influence the binaries it produces, and how these binaries relate to its source code in a reverse direction.

References

[1] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." in *NDSS*, 2011.
[2] 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.
[3] 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.
[4] 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.
[5] 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.