Reconstruction of Data Type in Obfuscated Binary Programs

Wei Ding[ab], ZhiMin Gu[a], Feng Gao[a]

a School of Computer Science Technology, Beijing Institute of Technology ,Beijing 100081,China

b College of Information Science and Engineering, Henan University of Technology, Zhengzhou 450001,china

orangeding@163.com,dingwei@bit.edu.cn，

***Abstract*- Recently, research community has advanced in type reconstruction technology for reverse engineering, but emerging with obfuscate technology, data type reconstruction is difficult and obfuscated code is easier to be monitored and analyzed by attacker or hacker. Therefore, we present a novel approach for automatic establish data type inference rules and reconstruct type from obfuscated binary programs using machine learning algorithm.**

Keywords- Type reconstruction, Obfuscated Binary, Deobfuscation, Disassembly, Inference Rules

I. INTRODUCTION

Recent years have seen an increase in research of disassemble for reverse engineering and forensics. They offer variety techniques to lift low-level code to high level code. However, they are typically weak in reverse engineering data structures. Recover high-level program data type is a recurring step in process of reverse engineering and decompile. Source codes are translated down to operations on registers and one globally addressed memory region, a number of data types of high-level without symbol tables during compilation. In executable file there are no symbol tables and type information, therefore, reconstruction of data type depends on memory allocation access and how memory is used.

Data type reconstruction aim to transform binary code without data type symbol into meaning typing language easier to understand and recover explicit type of anonymous byte blocks in order to improve the readability and understandability and be easier to program analysis.

More and more research community is aware of importance and advances in the reconstruction of data structure have led to significant research, including well-known type inference algorithms Hindley-Milner[1]，cartesian product[2]，iterative analysis[3], abstract type inference[4]. Yet they work on source codes, we need analysis binary code.

The most common type reconstruction approaches are based on static analysis techniques in binary code, like IDA Pro[5], OllyDbg[6]. VSA[7] (Value Set Analysis) attempts to identify location of like-variable and evaluate possible value set, which use a-loc to find possible value set and track value of data object. ASI[8](Abstract Structure Identification) tries to statically partition array and variable in memory block according to memory access. It use system call and famous library function information types, the types of called parameters are known, which are marked with according types and propagate them. Then, Balakrishnan[9] combine VSA and ASI to identify simple structure, array and the nest of array and structure. But static analysis method is difficult in basic aggregation structure.

REWAEDS[10] is a dynamic analysis method based on PIN analysis technology, which infer variable type by means of function parameters, return value and type signature instruction. In other words, it marks each location with timestamp type attribute and propagates it to other memory addresses, registers with program executed data flow, yet it can't deal with control flowing limited to executed path and can't deal with obfuscated code. Howard [11] is complementary with Rewards, it is more powerful. It supplies assembler and debugger with data structure and type to relieve reverses engineering. It can reveal data structure layout according to memory access patterns and generates automatically debugger symbol. Howard can recover fields of aggregation structure, nested arrays, yet its results depend on runtime path coverage like any other dynamic analysis instruments.

TIE [12] develops a novel type inference system based on type reconstruction rules, which can be applied in static and dynamic analysis. The core of TIE is to infer type according how the codes use, for example, in arithmetic operation, SF flag is detected and it can infer two operands are signed int.

Laika[13] detects data syntactic structure through unsupervised learning during program execution, but accuracy of this technology is not enough for reverse engineer, it can identify part of obfuscated code with virus detectors, which is worthy for our reference.

Nevertheless, along with advance of data type reconstruction, a number of obfuscation techniques [14-16] are very effective against state-of-the-art disassembles, preventing a substantial fraction of a binary program from being disassembled correctly and reconstruct data structure.

In [16], Linn and Debray describe a novel obfuscation technique that can be used to thwart binary static analysis. Their techniques are independent of and complementary to previous approaches to enhance software security by making it harder for an attacker to steal intellectual property. On the other hand, obfuscation technology could also be used by virus writers to hide malicious code such as Trojan Horses from virus scanners. The reason is that if relevant program structures were incorrectly extracted and identified obfuscated data type, malicious code could be classified as benign. *test*

This paper present a novel model for automatic establish data type inference rules to reconstruct type from obfuscated binary programs using machine learning algorithm.

The remainder of this paper is organized as follows. Section II discusses background material on data type reconstruction and decision tree model. Section III discusses decision tree models for data type reconstruction rules.

II. KEY TECHNOLOGY AND DEFINITION

In this section we review background material on type inference rules, relevant definitions and decision tree model.

A. *Inference Rules Definitions*

Lattices. A lattice is a partial order among the values in a domain, which have a lowest bound element $\perp$ and a highest bound element $\top$. Lattices conclude two operations. One is the "join" operator $\cup$,which is the least upper bound, the other is the "meet" operator $\cap$,which is the greatest lower bound.

Inference Rules. General inference rules form is following:

$$\frac{P_1 \; P_2 \ldots P_n}{C}$$

The upper of inference rule formula is the premises P_1, P_2 ,…, P_n. If all the premises are satisfied, then we can conclude results C. For example, $\frac{\Gamma \mid\!- t{:}S \quad s{<:}T}{\Gamma \mid\!- t{:}T}$ (T-Sub), it shows rule relation of type and subtype, in where each element in S is all in T, if S <: T.

Typing. Every term t, whether it is a variable, value or expression, has a type T. It is denoted by $\Gamma \mid\!- t:T$, which means "t has type T under context Γ ".

Subtyping[12]. Formula $S <: T$, where is read as "Type S is a subtype of Type T". Subtyping is transitive and reflexive, such as $\frac{S{<:}U \quad U{<:}T}{S{<:}T}$.

Array subtyping relation. $\frac{S_1{<:}T_1 \quad T_1{<:}S_1}{Array S_1{<:}Array T_1}$, which indicates that subtyping S_1 and sybtying T_1 are equivalent.

B. *Decision Trees*

Decision trees are common classier algorithms, which are a technique for supervised machine learning, for example, learning an instruction from a training data set.

Algorithm Decision tree Algorithm Description

```
DECISION-TREE-LEARNING(T, default)
INPUTS: T, set of training cases
{
if T is empty return default;
else if all cases in T have the same class return the class;
else {
    test=PARTIION(T);//belong to different class and attribute
    if N-TEST(test)=1 return MAJORITY-CLASS(T);
    else{
     default= MAJORITY-CLASS(T);
     tree=a new decision tree with the root test;
     for each condition  testi  in the test{
       Ti=elements in T which satisfy test;
    Subtree= DECISION-TREE-LEARNING(Ti, default);
    Add a branch with label  testi  to the tree and the subtree;
    }
    }
    }
}
```

A collection of items S={ $S_1, \ldots, S_k$ } is given, $S_i \in S$ happens with probability p_i , the entropy of S is denoted as $\text{Entropy(S)} = -\sum_{i=1}^{n} p_i \log p_i$.

For a collection of training items S ,which along with features $F_1, \ldots, F_k$, the information obtained from a feature F_i is specified by $Gain(S, F_i) = \text{Entropy(S)} - \sum_{v \in Vi} (|S_v| / |S|) \text{Entropy}(S_v)$, where Vi means the value subset undertaken by features F_i , similarly, S_v is the subset of S, which have the value v for features F_i .

III. IDENTIFYING OF DATA TYPE OF OBFUSCATED BINARY

In this section we expand [17] algorithm to deconfuscation and use the decision tree to record the instruction information, so that we can correctly reconstruct data type.

A. *Instruction Feature Extraction*

It is necessary to identify features of the disassembled instructions so as to construct a decision tree for obfuscated binary disassembly, Different disassemble instructions are defined as different values and meaning, i.e. 'opcode, source addressing mode, destination addressing mode', which infer data type according instruction features. For each feature, the operation mnemonic and repeat prefix are used to features set with a vector of features for each operands. The features we consider for operands are shown in [12].

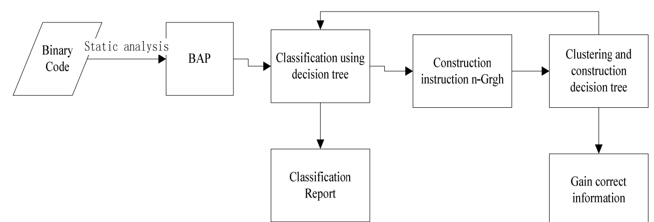

Figure 1. Deobfuscated and train feature sets

We aim to train sets and identify the sets of features and feature values of interest. As shown in Figure 1.

1) Lift BIL: At first, we translate the binary code into BIL (Binary Intermediate Language), which lift assembly by BAP[12] and can be easier to understand.

2) Training Data Sets: We can extract feature of instruction, operation, operands and other information, such as jmp, call. Operand types are register, pointer, memory address, immediate addresses, etc.

3) **Classification and Report:** Sequential report shows features of a fraction of sequence, which will help reconstruct data type.

B. Instruction n-Graph

In general, the common approach is to use adjacent instructions as construct n-Graph. However, it can not work well on transfer instruction, i.e. address that jmp or call instruction would transfer is not sequence. So we take a novel approach to set succs(P) denoted as the collections of all possible control-flow successors of an instruction P. For their possible successors are not known, we define succe(P)=ϕ. $ng_n(P)$ denotes the set of all n-Graph at beginning of instruction P. The definition is as follow.

$$ng_n(P) = \begin{cases} \{P\} & \text{if } n = 1 \text{ or } succs(P) = \phi \\ \bigcup_{J \in suss(p)} \{P \cdot P' | P' \in ng_{n-1}(J)\} & \text{otherwise} \end{cases}$$

Given a n-Graph S, $\varphi(S) = \{\varphi(\alpha)\ \alpha \in S\}$, $\varphi(S)$ means feature vector set for element of S.

C. Construct Decision Tree

In vector paces, the similarity of features can be expressed by geometric. We use the appropriate feature vectors as training input to construct a decision tree. How to identify the obfuscated instruction is a difficult problem. Given an arbitrary binary code and without any other auxiliary information, we can only judge it with instructions that are executed at different addresses.

In order to gain total assembly, rather than an executed path instructions, we use gcc-compiler to gain binaries. A mechanism is presented specifically tailored against the tool implemented by Linn and Debray [16]. Therefore, we must recognize code addresses, the targets of indirect jumps and calls. For binary P, we extract the set of addresses of all the instructions in P, denoted as InstAddrs(P) using BAP[12]. Assume binary section B span interval of addresses A. then we compute the set of all n-graphs for P at the beginning of addresses A.

$$NG_n(P) = \bigcup_{a \in A} ng_n(a)$$

Let addr(I) denote the address of an instruction I. For the set of n-Graph for P, we set flag {1, 0} to indicate whether the P is obfuscated. When flag is 1, then each instruction in the n-Graph is at an address in InstAddrs(p):

$$NG_n(P \mid flag = 1) = \{a \in NG_n(P) \mid$$
$$\forall I \in a : addr(I) \in InstAddrs(P), flag = 1\}$$

For flag=1, training inputs set is given by the set of feature vectors for good n-graphs:

$$PosInputs(P) = \varphi(NG_n(P \mid flag = 1))$$

In the same way,

$$NG_n(P \mid flag = 0) = NG_n(P) - NG_n(P \mid flag = 1)$$

$$\mathrm{Neg}Inputs(P) = \varphi(NG_n(P \mid flag = 0)) - Pos\mathrm{In}puts(P)$$

D. Constraints rules and Type Reconstruction Algorithm

The type constraints are computed using properties of assembly instructions.

Type reconstruction Algorithm

INPUTS:DECISION-TREE-LEARNING T

$T_{data} ::= T_{base} \mid T_{mem}$

$T_{base} ::= T_{reg} \mid T_{refined}$

$T_{reg} ::= reg1_t \mid reg8_t \mid reg16_t \mid reg32_t$

$T_{refined} ::= \mathrm{int} \mid long \mid char \mid float \mid double \mid$
$\quad array(0...n)\ of\ Type \mid ptr(T) \mid$
$\quad struct\{element_list\} \mid union\{element_list\}$

$T_{mem} ::= \{\forall address\ i \mid l_i : T_i\}$

$T_F ::= \{\mathrm{var}_1 : T_{1,\ldots}\ \mathrm{var}_n : T_n\}\ \{declarition_list\}$

Each type T, Generating Type Constraints;

{

$C(\mathrm{x} := \mathrm{e}) \rightarrow T_x = T_e$

$C(goto\ e) \rightarrow T_e = ptr(code_t)$

$C(if\ \ e\ then\ goto\ e_t\ else\ goto\ e_f) \rightarrow T_e = reg1_t \wedge (code_t)$

……

}

Constraint Solving

Match_type(T),OUTPUT type

IV. EXPERIMENTS

In experiments, to evaluate our approach, firstly, we use BAP[12] (Binary Analysis Platform) to gain BIL(Binary Intermediate Language). An experiment is example as obfuscated binary code including control construct. Then we will compare its result with IDA Pro and new type inference. The following section source code and obfuscated assembly.

```
push  %ebp
mov   %esp,%ebp
call  19808008
(junk)

cmp 0,%eax
Jne 804803a (L1)
Mov 0,%eax
Jmp 804803e (L2)
L1: Mov (1800000),%eax

L2: move %ebp, %eax
 pop %ebp
 ret
 nop
```

```
fun(int arg)
 {
  int a,b,c;
  c=other_fun(arg);
  if(a==0)
      b=2;
   else
     b=3;
  return b;

 }
Unsigned int other_fun(char
*buf,unsigned int *out)
{
  unsigned int c;
  c=0;
  if (buf){
   *out=strlen(buf);
  }
  if(*out){
   c=*out-1;
 }
  return c;
}
```

Figure 2. Obfuscated and Source code

For source code, we divide it into block A, the entry of B is call instruction, the entry of C is jne instruction, the entry of D is jmp instruction, and as a consequence, we construct a

decision tree. For TIE, if call instruction is obfuscated, it can not detect the call instruction, so it can not reconstruct the function type in other fun. We can take advantage of decision tree to deobfuscate and gain correct data type.

V. Conclusions

In this paper, we presents an a novel approach for automatic establish data type inference rules using machine learning algorithm, On the other hand, we can remove obfuscation during constructing the decision tree. Then we reconstruct type operating on the type lattice.

Our approach can be applied to transfer transformation obfuscation, however, it don't work well on reconstruction of obfuscated data type. In the future we can devote to research how to reconstruct splitting arrays, matrix and so on.

Acknowledgment

This work has been supported by the National Natural Science Foundation of China 61370062, the National High Technology Research and Development Program of China 2012AA101608 and the Twelfth Five-year National Key Technology Support Program 2013BAD17B04 and Commonweal Research Project for Grain Industry 201313008.

References

[1] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[2] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. *In Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, London,UK, 1995.

[3] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–164, 1990.

[4] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. *In Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA'06)*, pages 255–265, Portland, Maine, USA,2006. ACM.

[5] (2005)DataRescue. High level constructs width IDA Pro. http://www.hex-rays.com/idapro/ datastruct/datastruct.pdf,.

[6] Ollydbg. http://www.ollydbg.de/.

[7] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 binary executables. *In Proc. Conf. on Compiler Construction (CC)*, April 2004.

[8] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. *In Proceedings of the 26th ACMSIGPLAN-SIGACT symposium on Principles of Programming Languages,* 1999.

[9] Balakrishnan G, Reps T. Divine: Discovering variables in executables.*Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2007: 1-28.

[10] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. *In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, March 2010.

[11] Slowinska A, Stancescu T, Bos H. Howard: a dynamic excavator for reverse engineering data structures, *Proceedings of NDSS. 201*1.

[12] Lee J, Avgerinos T, Brumley D. TIE: principled reverse engineering of types in binary programs.*Proceedings of the 18th Network and Distributed System Security Symposium(NDSS)*.San Diego, USA: Internet society,2011.

[13] Cozzie A, Stratton F, Xue H, et al. Digging for data structures. *Symposium on Operating Systems Design and Implementation (OSDI)*. 2008.

[14] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8):735-746, August 2002.

[15] C. Collberg, C. Thomborson, and D. Low. Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[16] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. *The 10th ACM Conference on Computer and Communications Security (CCS 2003)*, 2003.

[17] Krishnamoorthy N, Debray S, Fligg K. Static detection of disassembly errors. *WCRE'09. 16th Working Conference on. IEEE*, 2009: 259-268.

Wei Ding was born in Anhui province, china. After graduated from Henan Institute of Technology in 2002, she entered into Zhengzhou University, and gained Master Degree. Then she worked into Henan Universality of Technology and became a doctorial student of Beijing Institute of Technology in 2009. Her research area of interest is binary analysis and software security.