Learning Types for Binaries

Zhiwu Xu[1], Cheng Wen[1], and Shengchao Qin[2,1]

[1] College of Computer Science and Software Engineering, Shenzhen University, China
[2] School of Computing, Teesside University, UK
xuzhiwu@szu.edu.cn, 2150230509@email.szu.edu.cn, shengchao.qin@gmail.com

Abstract. Type inference for Binary codes is a challenging problem due partly to the fact that much type-related information has been lost during the compilation from high-level source code. Most of the existing research on binary code type inference tend to resort to program analysis techniques, which can be too conservative to infer types with high accuracy or too heavy-weight to be viable in practice. In this paper, we propose a new approach to learning types for recovered variables from their related representative instructions. Our idea is motivated by "duck typing", where the type of a variable is determined by its features and properties. Our approach first learns a classifier from existing binaries with debug information and then uses this classifier to predict types for new, unseen binaries. We have implemented our approach in a tool called *BITY* and used it to conduct some experiments on a well-known benchmark *coreutils* (v8.4). The results show that our tool is more precise than the commercial tool Hey-Rays, both in terms of correct types and compatible types.

1 Introduction

Binary code type inference aims to infer a high-level typed variables from executables, which is required for, or significantly benefits, many applications such as decompilation, binary code rewriting, vulnerability detection and analysis, binary code reuse, protocol reverse engineering, virtual machine introspection, game hacking, hooking, malware analysis, and so on. However, unlike high-level source codes, binary code type inference is challenging because, during compilation, much program information is lost, particularly, the variables that store the data, and their types, which constrain how the data are stored, manipulated, and interpreted.

A significant amount of research has been carried out on binary code type inference, such as REWORD [1], TIE [2] , SmartDec [3], SecondWrite [4], Retypd [5] and Hex-Rays [6]. Most of them resort to program analysis techniques, which are often too conservative to infer types with high accuracy. For example, for a memory byte (*i.e.*, a variable) that is only used to store 0 and 1, most existing tools, such as SmartDec and Hex-Rays, recover the type *char* or *byte_t* (*i.e.*, a type for bytes), which is clearly either incorrect or too conservative. Moreover, some of them are too heavy-weight to use in practice, for example, in the sense that they may generate too many constraints to solve for large-scale programs.

In this paper, we propose a new approach to learning types for binaries. Our idea is motivated by "duck typing", where the type of a variable is determined by its features and properties. Our approach first learns a classifier from existing binaries with debug information and then uses this classifier to predict types for new, unseen binaries. In detail, we first recover variables from binary codes using Value-Set Analysis (VSA) [7], then extract the related representative instructions of the variables as well as some other useful information as their features. Based on binaries with debug information collected from programs that are used in teaching materials and from commonly used algorithms and real-world programs, we learn a classifier using Support Vector Machine (SVM) [8, 9]. Using this classifier, we then predict the most possible types for recovered variables.

We implement our approach as a prototype called BITY in Python. Using BITY, we conduct some experiments on a benchmark *coreutils* (v8.4). Compared with the commercial tool Hey-Rays, our tool is more precise, both in terms of correct types and compatible types. We also perform BITY on binaries of different sizes. The results show that our tool is scalable and suitable in practice.

Our main contributions are summarised as follows. We have proposed a new approach to learning types for binaries, and implemented it in a tool BITY, which is scalable and suitable in practice. Through experiments we have also demonstrated that our approach can predict types with high accuracy and has reasonable performance.

The rest of the paper is constructed as follows. Sec 2 illustrates some motivating examples. Sec 3 presents our approach to learning types for binaries, followed by the implementation in Sec 4 and experimental results in Sec 5. Related work is given in Sec 6 and Sec 7 concludes.

2 Motivation

In this section, we illustrate some examples that are not easy to recover the precise types by existing methods and explain our motivation.

Listing 1.1. C Source Code

```
int main ( )
{
  bool decode = false;
  int opt = getopt;
  switch (opt) {
      case 'd':
        decode = true;
        break;
      default:
        break;
   }
  if (decode)   do_decode;
}
```

Listing 1.2. Pseudo ASM Code

```
 mov     byte ptr [ebp-1], 0
 cmp     dword ptr [ebp-8], 64h
 jz      short loc_40101B
 jmp     short loc_40101F
loc_40101B:
 mov     byte ptr [ebp-1], 1
loc_40101F:
 movzx   eax, byte ptr [ebp-1]
 test    eax, eax
 jz      short loc_401035
 call    do_decode
loc_401035:
 retn
```

Fig. 1. Snippet Code from base64.c

The first example, shown in Figure 1, comes from an encode and decode program *base64.c* of C runtime Library. The program uses a variable *decode*

with type *bool* to record users' options. Nevertheless, after compiling, the variable *decode* is simply represented as a *byte* in stack (*i.e.*, [*ebp-1*]) and the type *bool* is lost. Due to the over-conservative program analysis they adopt, most existing tools, such as SmartDec and Hex-Rays, recover for the variable [*ebp-1*] the type *char* or *byte_t*, which is clearly either incorrect or conservative.

To make matters worse, programs with fewer instructions are more difficult to recover types correctly. Let us consider three simple assignments for three variables with different types, shown in Figure 2. SmartDec recovers the same type *int32_t* for all these three variables, while Hex-Rays infers the type *Dword** for i and f and the type *Qword** for d. Again, most of these results are either incorrect or conservative.

Listing 1.3. C Source Code

```
func1(int* i){
    *i = 10;
}
func2(float* f){
    *f = 10.0;
}
func3(double* d){
    *d = 10.0;
}
```

Listing 1.4. Pseudo ASM Code

```
mov     [i], 0Ah

movss   xmm0, ds:XX
movss   [f], xmm0

movsd   xmm0, ds:XX
movsd   [d], xmm0
```

Fig. 2. Assignments of different types

One may note that the variables of different types are compiled with different instructions, that is, *mov*, *movss* and *movsd*. Hence, a simple solution is to enhance the program analysis techniques with three new rules to infer these three different types corresponding to these three different instructions. However, it is pity that *mov* (*movsd* resp.) is not only used for *int* (*double* resp.). Even if it works, there are too many instructions and types to figure out the reasonable rules. For example, there are more than 30 kinds of *mov* instructions in the x86 instruction set and the source operand and the destination operand may have different meanings.

Generally, the set of the related instructions of a variable reflects how the variable is stored, manipulated, and interpreted. So our solution is to take the related instruction set as a feature of a variable, and then to learn for the variable the most possible type from the feature. This is motivated by "duck typing", where the type of a variable is determined by its features and properties instead of being explicitly defined. Let us consider *base64.c* again. The related instruction set of [*ebp-1*] is {mov _, 0; mov _, 1; movzx eax, _ }, which is most likely to be a feature of *bool*, where _ denotes the concerning variable. Accordingly, we recover *bool* as the type of [*ebp-1*]. Similarly to the variables of the second example. Note that *movsd* may be a feature of *double*, but not all of them belong to *double*.

3 Approach

In this section, we present our approach to learning the most possible type for a recovered variable.

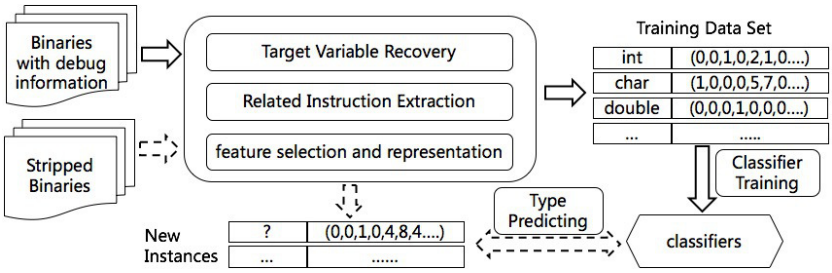

Fig. 3. Approach

```
%%  C Code
char *memchr (char *buf,int chr,int cnt) {
    while (cnt && *buf++ != chr) cnt--;
    return(cnt ? --buf : NULL);
}
%%  ASM Code Snippet
sub_401000      proc near
......
loc_401009:
07                  cmp     dword ptr [ebp+10h], 0
08                  jz      short loc_40103A
09                  mov     eax, [ebp+8]
10                  movsx   ecx, byte ptr [eax]
11                  mov     [ebp-44h], ecx
12                  mov     edx, [ebp+0Ch]
13                  mov     [ebp-48h], edx
14                  mov     eax, [ebp+8]
15                  add     eax, 1
16                  mov     [ebp+8], eax
17                  mov     ecx, [ebp-44h]
18                  cmp     ecx, [ebp-48h]
19                  jz      short loc_40103A
20                  mov     eax, [ebp+10h]
21                  sub     eax, 1
22                  mov     [ebp+10h], eax
23                  jmp     short loc_401009
loc_40103A:
24                  cmp     dword ptr [ebp+10h], 0
25                  jz      short loc_401051
26                  mov     eax, [ebp+8]
27                  sub     eax, 1
28                  mov     [ebp+8], eax
29                  mov     ecx, [ebp+8]
30                  mov     [ebp-44h], ecx
31                  jmp     short loc_401058
loc_401051:
32                  mov     dword ptr [ebp-44h], 0
loc_401058:
......
sub_401000      endp
```

Fig. 4. Snippet Code of *memchr*

As mentioned in Section 2, we try to learn the most possible type for a variable from its related instruction set. Our approach first learns a classifier from existing binaries with debug information and then uses this classifier to predict types for new, unseen binaries. Figure 3 shows the main idea of our approach. In detail, our approach consists of the following steps: (1) target variable recovery; (2) related instruction extraction; (3) feature selection and representation; (4) classifier training; (5) type predicting. In the following, we describe each of them, using another program *memchr* from C runtime Library as an illustrative example, which is shown in Figure 4.

3.1 Target Variable Recovery

During compilation, variables of the source program and their type information are not included in the resulting binary. So the first step is to identify the target variables in binaries. Indeed, variables are abstractions of memory blocks, which are accessed by data registers or specifying absolute address directly or indirectly through address expressions of the form "[*base*+*index*×*scale*+*offset*]" in binaries, where *base* and *index* are registers, and *scale* and *offset* are integer constants. Take the variables in stack frame for example. Parameters[3] are always accessed by the form "[*ebp*+*offset*]", while local variables are by "[*ebp*−*offset*]", where *ebp* is the stack base pointer register. We recover the possible variables in binaries using Value-Set Analysis (VSA) [7], which is widely used in many binary analysis platforms. Note that, due to compiler optimization, a stack location may be used to represent multiple different local variables in the same function, which is not considered here.

Considering the illustrated example *memchr* in Figure 4, the variables we recovered in stack frame are listed in Table 1. There are three parameters, which conform to the declarations in the C code. Due to the low-level instructions, there are two more local variables, which are used respectively to store the values of **buf* and *chr* temporarily.

Table 1. Target Variables in *memchr*

Variable	**Offset**	**Variable**	**Offset**	**Variable**	**Offset**
Parameter1	[ebp+8]	Parameter2	[ebp+0Ch]	Parameter3	[ebp+10h]
LocalVar1	[ebp-48h]	LocalVar2	[ebp-44h]		

3.2 Related Instruction Extraction

Next, we want to extract the related instructions for the recovered target variables, which reflect how the variables are used and will be used as a feature to learn the types.

[3] In FASTCALL convention, the first two parameters are passed in ECX and EDX.

The instructions using a variable directly are extracted for the variable. However, an instruction of a variable in high-level codes may be compiled into several instructions in low-level codes, some of which may not use the corresponding variable directly. For example, the statement *if (decode)* in *base64* is complied into two instructions in ASM codes (see Figure 1), one of which uses the corresponding variable directly (*i.e.*, "movzx eax, byte ptr [ebp-1]"), while the other does not (*i.e.*, "test eax, eax"). Clearly, the second one is more representative for *bool*. On the other hand, the data registers like *eax*, *ebx*, *ecx* and *edx* are usually used as an intermediary to store data temporarily and they may store different data (*i.e.*, have different types) in different time. Therefore, we make use of use-defined chains on the data registers to extract the indirect usage instructions: if a data register is defined by a variable, then all the uses of the register are considered as the uses of the variable as well. Consequently, the instruction "test eax, eax" belongs to the variable [ebp-1] in *base64*, since it is a use of *eax*, which is defined by [epb-1].

Let us consider the target variable [epb+8] in the *memchr* example. The instructions related with [epb+8] are shown in Figure 5. There are 10 instructions in total, 6 of which use [epb+8] directly and 4 are collected due to the use-defined chain (denoted by "use of " followed by a data register).

```
09      mov     eax, [ebp+8]            //def of eax by epb+8
10      movsx   ecx, byte ptr [eax]     //use of eax
14      mov     eax, [ebp+8]            //def of eax by epb+8
15      add     eax, 1                  //use of eax
16      mov     [ebp+8], eax
26      mov     eax, [ebp+8]            //def of eax by epb+8
27      sub     eax, 1                  //use of eax
28      mov     [ebp+8], eax
29      mov     ecx, [ebp+8]            //def of ecx by epb+8
30      mov     [ebp-44h], ecx          //use of ecx
```

Fig. 5. Related Instructions of [ebp+8] in *memchr*

3.3 Feature Selection and Representation

In this paper, we focus on the x86 instruction set on Intel platforms. The others are similar.

According to the official document of the x86 instruction set [10], different instructions have different usages. So we perform some pre-processing on these instructions based on their usages. Firstly, we note that not all the instructions are interesting for type inference. For example, *pop* and *push* are usually used by the stack, rather than variables. Secondly, as different operands may have different meanings, we differentiate between two operands in a dyadic instruction, for example, the operands of *mov* respectively represent the source and the destination, which are clearly not the same. Thirdly, some instructions need further processing, since using them in different circumstances may have different meanings. For instance, using *mov* with registers of different sizes offers us different meaningful information. Table 2 lists the typical usage patterns of *mov* we use, where _ denotes a variable, *regn* denotes a register with size n, *imm* denotes an

immediate number which is neither 0 nor 1, and *addr* denotes a memory address (*i.e.*, another variable).

Table 2. Usage Patterns of *mov*

mov _, reg32	mov reg32, _	mov _, reg16	mov reg16, _	mov _, reg8	mov reg8, _
mov _, addr	mov addr, _	mov _, 0	mov _, 1	mov _, imm	

Moreover, not all the instructions are widely used or representative. For that we do a statistical analysis on our dataset, which consists of real-world programs and source codes from some course books, using the well-known scheme Term FrequencyInverse Document Frequency (TF-IDF) weighting [11]. Based on the result, we select the N most frequently used and representative instructions as the feature indicators. Theoretically, the more instructions, the better. While in practice, we found 100 instructions are enough.

In addition, we also take into account some other useful information as features, namely, the memory size and being an argument of a function.

Finally, we represent the selected features of variables using a vector space model [12], which is an algebraic model for representing any objects as vectors of identifiers. We only concern that how many times an interesting instruction are performed on a variable, leaving the order out of consideration. So a representation of a variable is a vector consisting of the frequency of each selected instruction and the extra useful information. Formally, a variable is represented as the following vector v:

$$v = [f_1 : t_1,\ f_2 : t_2,\ \ \ldots,\ f_n : t_n]$$

where f_i is a feature term, t_i is the value of feature f_i, and n is the number of features. For example, Table 3 shows the vector of the variable [ebp+8] in the illustrated example *memchr*, where only the nonzero features are listed. Note that "mov eax, _ " and "mov ecx, _" are merged together, since both *eax* and *ecx* are registers of 32 bits. To be more precise, one can also take into account the IDF that have been computed for each selected instruction or some other correlation functions.

3.4 Classifier Training and Type Predicting

For now, we only consider the base types without type quantifiers, that is, the set L of labels we are learning in this paper are

$$L = \{bool, char, short, float, int, pointer, long\ long\ int, double, long\ double\}$$

The reason is that (1) the other types, such as structs, can be composed from the base types; (2) too many levels may make against the classifier.

We use supervised learning to train our classifier, so a labeled dataset is needed. For that, we compile a dataset of C programs with debugging and then extract type information from the compiled binaries. Generally, our training problem can be expressed as :

Table 3. Represetation of *epb+8*

Before Proceeding		After Proceeding	
Feature	**Value**	**Feature**	**Value**
size	32	size32	1
mov eax,_	3	mov reg32, _	4
movsx ecx, [_]	1	movsx reg32, [_]	1
add _, imm	1	add _, imm	1
mov _ ,eax	2	mov _ , reg32	2
sub _, imm	1	sub _, imm	1
mov ecx,_	1	Merged to mov reg32,_	
mov [ebp-44h], _	1	mov addr, _	1

Given a labeled dataset $D = \{(v_1, l_1), (v_2, l_2), \ldots, (v_m, l_m)\}$, the goal is to learn a classifier C such that $C(v_i) = l_i$ for all $i = 1, 2, \ldots, m$, where v_i is the feature vector of a variable, $l_i \in L$ is the corresponding type, m is the number of variables.

We use Support Vector Machine (SVM) [8, 9] to learn the classifier. Clearly, our training problem is a multi-class one. By using the "one-against-one" approach [13], we first reduce the multi-class problem into a *binary* classifier learning one: for every two different types, a classifier is trained from the labeled dataset. Some size information of variables may be unknown, so for simplicity, we do not distinguish between types of different sizes. That is to say, assume there are k types, we will train $k \times (k-1)/2$ binary classifiers.

As mentioned in Section 3.3, a variable is represented as a vector, namely, is regarded as a point in the feature vector space. SVM tries to construct an n-dimensional hyperplanes that optimally separates the variables into categories, which is achieved by a linear or nonlinear separating surface in the input vector space of the labeled dataset. Its main idea is to transform the original input set to a high-dimensional feature space by using a kernel function, and then achieve optimum classification in this new feature space.

After the binary classifiers are trained, we then can use them to predict the most possible type for each variable that have been recovered from new or unseen binaries. This proceeds as follows: we use each binary classifier to vote a type for a variable, and then select the one that gets the most votes as the most possible type. Let us consider the variable [epb+8] in the illustrative example again. Note that its feature instructions contain "mov reg32, _; movsx reg32, [_]" (to read from an address), "mov reg32,_; add _,imm" (to increase the address), and "mov reg32,_; sub _,imm" (to decrease the address), which are the typical usages of *pointer*. Most classifiers involved *pointers* will vote for the type *pointer* for [epb+8], and thus the most possible type we learn is *pointer*. Another example is the variable *decode* in the program *base64* presented in Section 2. According to its feature instructions (*i.e.*, "mov _, 0; mov _, 1; movzx reg32,_; test _, _"), most of the classifiers will vote for the type *bool*.

3.5 Type Lattice

Finally, we present the lattice of our types we are learning, which gives the hierarchy of types and will be used to measure the precision of our approach as TIE does [2] (see Section 5).

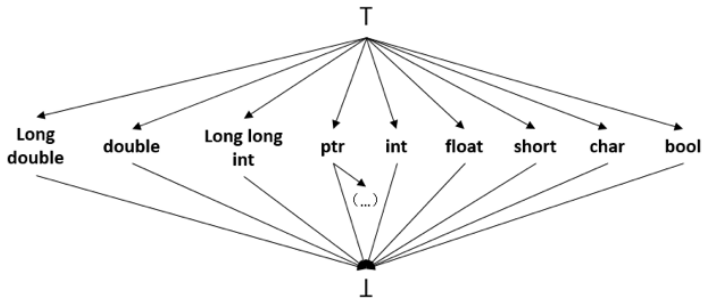

Fig. 6. Type Lattice

The lattice is given in Figure 6, where $\top$ and $\bot$ respectively denote that a variable can or cannot be any type and there is a "pointer" to the lattice itself for the type *pointer*, that is, the lattice is level-by-level. In other words, our approach handles *pointer* level-by-level, which proceeds as follows:

1. once a variable v is predicted to have type *pointer* by the classifier, our approach first tries to recover another variable that the *pointer* variable points to;
2. if such an indirect variable v' exists, the approach then extracts related features for this newly recovered variable v' and continues to learn a (next-level) type t for it;
3. finally, the type for v is a *pointer* to t if v' exists, otherwise a *pointer* (to any type).

This enables us to predict *pointer* more precise (see Section 5) and to handle multi-level *pointer*s considered in [4]. Theoretically, our approach can handle a *pointer* with any levels (and thus may not terminate). While in practice, we found only 3 levels are enough.

Let us go on with [epb+8] in the illustrative example. In Section 3.4, we have learnt that the most possible type for [epb+8] is *pointer*. So our approach carries on to recover an indirect variable, which is "byte ptr [eax]", and then to extract its feature vector [size8: 1; movsx reg32,_: 1; mov addr,_ : 1], which covers the data move with sign extension. There are two types with 8 bits, namely, *bool* and *char*. Compared with *bool*, it is more like to have type *char* according to the known binaries. Thus the final type for [epb+8] is *pointer* to *char*.

4 Implementation

We have implemented our approach as a prototype called BITY in 3k lines of Python codes. We use IDA Pro [6] as a front end to disassemble binaries, since it supports a variety of executable formats for different processors and operating systems, and use LIBSVM [14], a Library for Support Vector Machines, to implement our classifiers. Moreover, as mentioned before, we select 100 most frequently used and representative instructions as features and consider 3 levels for *pointer* types.

For a high precision, we consider a training dataset that should contain different possible usages of different types. For that, we collect binaries with debug information obtained from programs that are used in teaching materials and from commonly used algorithms and real-world programs. Programs of the first kind always cover all the types and their possible usages, in particular, they demonstrate how types and their corresponding operations are used for beginners. While programs of the second kind reflect how (often) different types or usages are used in practice, which help us to select the most possible type. In detail, our training dataset consists of the binaries obtained from the following programs:

- Source codes of the C programming language (K&R)
- Source codes of basic algorithms in C programming language [15];
- Source codes of commonly used algorithms [16];
- C Runtime Library;
- Some C programs from github.

Any other valuable data will be added into our data set in the future.

5 Experiments

In this section, we present the experiments.

5.1 Results on Benchmark *coreutils*

To evaluate our approach, we perform our tool BITY on programs from *coreutils* (v8.4), a benchmark used by several existing work [1, 2, 17]. We first compile the test programs into (stripped) binaries, and then use BITY to predict types for the recovered variables. To measure the accuracy of our approach, we compare the types that BITY predicts with the type information extracted from the binaries that are obtained by compiling the test programs with debug support. We also compare our tool BITY against Hex-Rays decompiler-v2.2.0.15[4], a plug-in of the commercial tool IDA Pro [6]. All the experiments are run on a machine with 3.30GHz i5-4590 and 8GB RAM.

[4] Hex-Rays makes use of debug information, so we perform both our tool and Hex-Rays on stripped binaries.

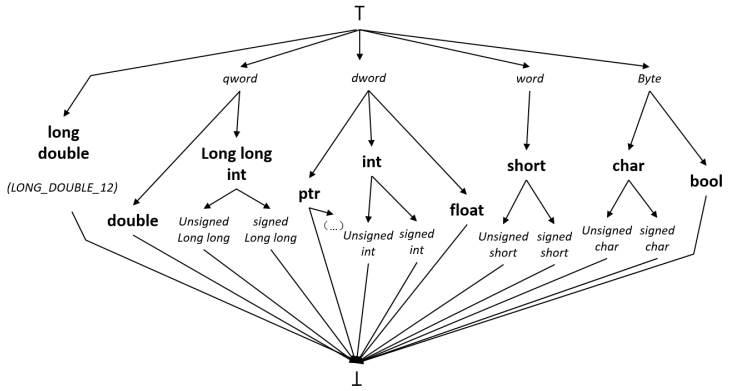

Fig. 7. Type Lattice for Hex-Rays and BITY

To measure between types, we borrow the notation *distance* from TIE [2]. For that, we extend our type lattice with the types recovered by Hex-Rays, which is shown in Figure 7, where our tool consider only the types in bold, while Hex-Rays considers all the types except $\top$ and $\bot$. We say that two types are compatible if one of them is a subtype of the other one following the top-level lattice. Given two types t_1, t_2, the distance between them, denoted as $|t_1 - t_2|$, is defined as follows: (1) at most one of them is a *pointer*: the distance $|t_1 - t_2|$ is the number of hierarchies between them in the top-level lattice if they are compatible, otherwise the maximum hierarchies' height (*i.e.*, 4); (2) both of them are *pointer*, namely, *pointer* to t'_1 and t'_2 respectively: the distance $|t_1 - t_2|$ is the half[5] of the maximum hierarchies height (*i.e.*, 2) multiplied by 0, 0.5 and 1, according to whether t'_1 and t'_2 are the same, compatible or incompatible respectively. For example, $|int - dword| = 1$ and $|*int - *dword| = 1$, while $|int - *dword| = 4$.

The selected results of our tool BITY and Hex-Rays on the benchmark *coreutils* are given in Table 4, where **Vars** denotes the number of recovered variables in stack, **Corr** denotes the number of variables, whose types are recovered correctly, **Comp** denotes the number of variables, whose types are recovered compatibly, **Fail** denotes the number of variables, whose types are recovered incorrectly, and **Dist** denotes the average distance of each program.

The results show that our tool predicts correct types for 58.15% (1356) variables and compatible types for 31.22% (728) variables (most are due to the lack of the type quantifier *signed* and *unsigned*), in total proper types for 89.37% (2084) variables; while Hex-Rays recovers correct types for 54.80% (1278) variables and compatible types for 25.26% (589) variables (most are due to the consideration of the conservative types), in total proper types for 80.06% (1876) variables. This indicates that our tool is (11.63%) more precise than Hex-Rays, both in terms of correct types and compatible types.

[5] Theoretically, we can use the radio of the number of common levels among the number of maximum levels between t_1 and t_2 [4]. Since we consider 3 levels in practice, we use the half here.

Table 4. Selected Results of BITY and Hex-Rays on *coreutils* (v8.4)

Program	**Vars**	**BITY**				**Hex-Rays**			
		Corr	**Comp**	**Fail**	**Dist**	**Corr**	**Comp**	**Fail**	**Dist**
base64	41	20	19	2	0.66	29	7	5	0.80
basename	22	17	4	1	0.55	12	4	6	1.32
cat	50	29	19	2	0.60	18	19	13	1.52
chcon	55	39	8	8	0.73	32	7	16	1.36
chgrp	31	21	4	6	0.90	17	4	10	1.42
chmod	42	19	20	3	0.71	20	13	19	1.23
chown	42	19	4	3	0.94	5	6	5	1.65
chroot	23	12	9	2	0.74	18	4	10	0.39
cksum	14	7	6	1	0.71	7	6	1	0.79
comm	20	10	4	6	1.40	11	1	8	1.65
copy	135	69	48	18	0.92	53	42	40	1.60
cp	78	46	26	6	0.78	45	23	10	0.94
csplit	66	27	32	7	1.02	26	25	15	1.38
cut	47	32	14	1	0.47	31	15	1	0.64
date	30	18	8	4	0.77	15	8	7	1.37
dd	128	81	35	12	0.72	78	30	20	0.88
df	92	51	32	9	0.89	45	25	22	1.27
dircolors	55	31	23	1	0.62	26	23	6	0.98
du	68	27	28	13	1.19	26	15	27	1.90
echo	11	8	3	0	0.55	5	6	0	0.82
expand	25	16	8	1	0.48	16	9	0	0.48
expr	85	29	39	17	1.36	28	35	22	1.47
factor	30	20	9	1	0.43	22	4	4	0.73
fmt	62	40	15	7	0.71	40	7	15	1.12
fold	25	17	8	0	0.36	20	5	0	0.28
getlimits	20	17	3	0	0.15	17	2	1	0.30
groups	9	5	4	0	0.44	5	4	0	0.55
head	111	63	41	7	0.65	52	42	17	1.14
id	20	13	5	2	0.65	12	5	3	0.90
join	106	48	52	6	0.79	54	24	28	1.33
kill	27	18	6	3	0.70	15	9	3	0.89
ln	29	23	3	3	0.62	21	5	3	0.76
ls	352	189	105	58	1.04	186	73	93	1.32
mkdir	22	15	4	3	0.91	10	5	7	1.55
mkfifo	10	7	2	1	1.11	7	0	3	1.78
mktemp	35	23	9	3	0.60	16	15	4	1.09
mv	35	20	8	7	1.14	15	8	12	1.74
nice	16	15	1	0	0.13	15	1	0	0.13
nl	18	11	4	3	1.06	12	3	3	0.94
nohup	22	20	1	1	0.27	19	1	2	0.41
od	120	88	23	9	0.53	86	25	9	0.58
operand2sig	13	11	0	2	0.62	9	2	2	0.77
paste	35	26	7	2	0.54	24	9	2	0.71
pathchk	19	15	3	1	0.37	14	4	1	0.53
pinky	62	34	22	6	0.74	44	9	9	0.71
Total	2332	1356	728	248	-	1278	589	465	-
Avg	-	-	-	-	0.72	-	-	-	1.02
pointers	1021	443	398	180	-	391	222	408	-

Moreover, we found 43.8% (1021 among 2332) of the recovered variables are *pointer* ones. For these *pointer* types of the variables, our tool can recover 43.39% (443) correct types and 38.98% (398) compatible types, in total 82.37% (841) proper types; while Hex-Rays recovers 38.30% (391) correct types and 21.74% (222) compatible types, in total 60.03% (613) proper types. Consequently, our tool is also (37.21%) more precise than Hex-Rays in terms of *pointer* types.

Concerning the failures, most of them are due to *pointer*: 72.58% (180 among 248) for our tool and 87.74% (408 among 465) for Hex-Rays. We perform manual analysis on some failure cases. There are two main reasons: (1) there are too few representative instructions to predict the right types for some variables, especially for *pointer* variables; (2) some variables are of composed types such as *struct* and *array*, which are not considered by our tool yet.

Finally, let us consider the distance. For most programs, our tool predict types with a shorter distance than Hex-Rays. While in several other cases (*e.g.*, *chroot* and *pinky*), Hex-Rays recovers types better. One main reason is that Hex-Rays can reconstruct some *pointer* to *struct* such as FILE*, FTSENT* and FTS*. On average, our tool predicts more precise types.

5.2 Performance

To evaluate the scalability of our tool, we conduct experiments on binaries of different sizes. Table 5 shows the experimental results, where **ALOC** denotes the lines of the assemble codes, **Vars** denotes the number of recovered variables in stack, **ProT** denotes the preprocessing time excluding the disassembling time by IDA Pro and **PreT** denotes the predicting time. The results show that (1) the preprocessing time accounts for a great proportion and is linear on LOC and variable numbers; (2) the predicting time does not cost too much and is linear on variable numbers; (3) our tool predicts types for binaries of sizes ranging from 7KB to 1341.44MB in just a few seconds, which indicates our tool is scalable and viable in practice.

Table 5. Results on Different Sizes of Binaries

Program	Size	ALOC	Vars	ProT	PreT
strcat	7 KB	508	8	0.187	0.011
Notepad++ 7.3.3_Installer.exe	2.80 MB	12032	113	0.807	0.229
SmartPPTSetup_1.11.0.7.exe	4.76 MB	128381	166	1.156	0.365
DoroPDFWriter_2.0.9.exe	16.30 MB	25910	71	0.692	0.068
QuickTime_51.1052.0.0.exe	18.30 MB	61240	247	2.132	0.607
Firefox Portable.exe	110.79 MB	12068	113	0.906	0.254
VMware workstation v12.0.exe	282.00 MB	39857	352	3.739	0.911
opencv-2.4.9.exe	348.00 MB	61636	287	4.130	0.722
VSX6_pro_TBYB.exe	1341.44 MB	129803	450	4.762	1.921

6 Related Work

There have been many works about type inference on binaries. In this section we briefly discuss a number of more recent related work. Interested readers can refer to [18] for more details.

TIE [2] is a static tool for inferring primitive types for variables in a binary, where the inferred type is limited to integer and pointer type. Moreover, the output of TIE is the upper bound or the lower bound rather than the specific type, which may not be accurate enough for it to be useful for a binary engineer. PointerScope [19] uses type inference on binary execution to detect the pointer misuses induced by an exploit. Aiming for scalability, SecondWrite [4] combines a best-effort VSA variant for points-to analysis with a unification-based type inference engine. But accurate types depend on high-quality points-to data. The work of Robbins *et al.* [17] reduces the type recovery to a rational-tree constraint problem and solve it using an SMT solver. Yan and McCamant [20] propose a graph-based algorithm to determine whether each integer variable is declared as signed or unsigned. Retypd [5] is a novel static type-inference algorithm for machine code that supports recursive types, polymorphism, and subtyping. Hex-Rays [6] is a popular commercial tool for binary code analysis and its exact algorithm is proprietary. However, these tools resort to static program analysis approaches, which are either too conservative to infer types with high accuracy or too heavy-weight for practical use.

REWARDS [1] and Howard [21] both take a dynamic approach, generating type constraints from execution traces, to detect data structures. ARTISTE [22] is another tool to detect data structures using a dynamic approach. ARTISTE generates hybrid signatures that minimize false positives during scanning by combining points-to relationships, value invariants, and cycle invariants. While MemPick [23] is a tool that detects and classifies high-level data structures such as singly- or doubly-linked lists, many types of trees (*e.g.*, AVL, red-black trees, B-trees), and graphs. But as dynamic analysis-based approaches, they cannot achieve full coverage of variables defined in a program.

Some tools focus on recovering object oriented features from C++ binaries [3, 24–26]. Most of them adopt program analysis, while the work of Katz *et al.* [26] uses object tracelets to capture potential runtime behaviors of objects and use the behaviors to generate a ranking of their most likely types. Similar to Katz et. al.'s work, we use the instructions set, leaving the order out of consideration, to capture potential behaviours of variables. Thus our solution is simper.

In addition, Raychev *et al.* [27] present a new approach for predicting program properties, including types, from big code based on conditional random fields. Their approach leverages program structures to create dependencies and constraints used for probabilistic reasoning. Their approach works well at high-level source code since lots of program structures are easy to discover. While for stripped binaries, less program structures can be recovered.

7 Conclusion

Recovering type information from binaries is valuable for binary analysis. In this paper, we have proposed a new approach to predicting the most possible types for recovered variables. Different with existing work, our approach bases on classifiers, without resorting to program analysis like constraint solving techniques. To demonstrate the viability of the approach, we have implemented our approach in a prototype tool and carried out some interesting experiments. The results show that our tool is more precise than the commercial tool Hey-Rays.

As for future work, we may consider the binary classifiers of different types of the same size to improve the approach or try other classifiers. We can perform a points-to analysis to improve our analysis on multi-level *pointer*s. We can take type quantifiers (*e.g.*, signed) or the composite types (*e.g.*, *struct*) into account. We can also conduct more experiments on more real world programs to compare BITY with other tools.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308, 61373033 and 61772347, Science and Technology Foundation of Shenzhen City under Grant No. JCYJ20170302153712968.

References

1. Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium*, 2010.
2. Jong Hyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, 2011.
3. Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: Approaching c++ decompilation. In *Reverse Engineering*, pages 347–356, 2011.
4. Khaled Elwazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *ACM Sigplan Conference on Programming Language Design and Implementation*, pages 51–60, 2013.
5. Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *ACM Sigplan Conference on Programming Language Design and Implementation*, pages 27–41, 2016.
6. *The IDA Pro and Hex-Rays*. http://www.hex-rays.com/idapro/.
7. Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 binary executables. *University of Wisconsin-Madison Department of Computer Sciences*, 2012.

8. Christopher J. C Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.
9. A. J. Smola and B. Schlkopf. On a kernel-based method for pattern recognition, regression, approximation, and operator inversion. *Algorithmica*, 22(1):211–231, 1998.
10. IntelCorporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. , December, 2016.
11. Josipa Crnic. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
12. G. Salton. A vector space model for automatic indexing. *Communications of the Acm*, 18(11):613–620, 1975.
13. Seokho Kang, Sungzoon Cho, and Pilsung Kang. Constructing a multi-class classifier using one-against-one approach with different binary classifiers. *Neurocomputing*, 149(PB):677–682, 2015.
14. *LIBSVM*. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.
15. *178 algorithm C language source code*. http://www.codeforge.com/article/220463.
16. Xu Shiliang. *Commonly Used Algorithm Assembly (C Language Description)*. Tsinghua University Press, 2004. In Chinese.
17. Ed Robbins, Jacob M Howe, and Andy King. Theory propagation and rational-trees. In *Symposium on Principles and Practice of Declarative Programming*, pages 193–204, 2013.
18. Juan Caballero and Zhiqiang Lin. Type inference on executables. *Acm Computing Surveys*, 48(4):65, 2016.
19. Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. *Proceedings of the Western Pharmacology Society*, 47(47):46–49, 2013.
20. Yan Qiuchen and McCamant Stephen. Conservative signed/unsigned type inference for binaries using minimum cut. *Technical Report. University of Minnesota*, 2014.
21. Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, 2011.
22. Khaled Elwazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Artiste: Automatic generation of hybrid data structure signatures from binary code executions. *Technical Report TRIMDEA-SW-2012-001. IMDEA Software Institute*, 2012.
23. Istvan Haller, Asia Slowinska, and Herbert Bos. Mempick: High-level data structure detection in c/c++ binaries. In *Reverse Engineering*, pages 32–41, 2013.
24. Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *ACM Sigplan on Program Protection and Reverse Engineering Workshop*, page 1, 2014.
25. Kyungjin Yoo and Rajeev Barua. Recovery of object oriented features from c++ binaries. In *Asia-Pacific Software Engineering Conference*, pages 231–238, 2014.
26. Omer Katz, Ran El-Yaniv, and Eran Yahav. Estimating types in binaries using predictive modeling. In *ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, pages 313–326, 2016.
27. Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *The ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, pages 111–124, 2015.