Type Learning for Binaries and its Applications

Zhiwu Xu, Cheng Wen, Shengchao Qin

***Abstract*—Binary type inference is a challenging problem due partly to the fact that during the compilation much type-related information has been lost. Most existing research work resorts to program analysis techniques, which can be either too heavy-weight to be viable in practice or too conservative to be able to infer types with high accuracy. In this work, we propose a new approach to learning types for binary code. Motivated by "duck typing", our approach learn types for recovered variables from their features and properties (*e.g.*, related representative instructions). We first use machine learning to train a classifier with basic types as its levels from binaries with debugging information. The classifier is then used to learn types for new, unseen binaries. While for composite types, such as *pointer* and *struct*, a points-to analysis is performed. Finally, several experiments are conducted to evaluate our approach. The results demonstrate that our approach is more precise, both in terms of correct types and compatible types, than the commercial tool Hey-Rays, the open source tool Snowman and a recent tool EKLAVYA using machine learning. We also show that the type information our proposed system learns is capable of helping detect malware.**

***Index Terms*—Type Recovery, Type Learning, Binary Analysis, Duck Typing, Machine Learning**

I. Introduction

Due to the significant growth of untrusted code and malware, such as viruses and worms, there is an increasing demand for tools to help security analysts and programmers analyse and understand binary code. A recurring step in many such tools is binary type inference, which aims to infer high-level typed variables from binary code. Binary type inference is required for, and would significantly benefit, many applications such as binary analysis, binary code rewriting, binary code reuse, decompilation, game hacking, hooking, protocol reverse engineering, malware analysis, virtual machine introspection, vulnerability analysis and detection, and so on. Comparing to type inference for high-level code, binary type inference is much more challenging, largely due to the fact that most program information is lost during compilation, in particular, information about variables (which store the data), and their types (which constrain how the data are stored, interpreted and manipulated). Hence, binary type inference involves two tasks: one is to identify high-level variables from the binary code (called variable recovery), and one is to give a high-level type to each recovered variable (called type recovery).

Lots of research works on binary type inference have been carried out, such as Hex-Rays [1], Retypd [2], REWORD [3], SecondWrite [4], SmartDec [5], TIE [6] , and so on. However, most of them tend to resort to program analysis techniques, which are often too conservative to infer types with high accuracy. For instance, considering a memory byte (*i.e.*, a variable) which is only used to store 0 and 1, most existing tools, such as Hex-Rays and SmartDec, recover for this memory byte the type *byte_t* (*i.e.*, a type for bytes) or *char*, which is clearly too conservative or incorrect (some further discussion is given in Section II later). Furthermore, some of them are too heavy-weight to use in practice, for instance, in the sense that for large-scale programs they may generate too many constraints to solve. For instance, "DIVINE [7] spends 2 hours while analyzing programs of the order of $55,000$ assembly instructions" [4]. This is because (i) there are much more instructions in the low-level code than in the high-level code in general; and (ii) there may be several possible constraints for a low-level instruction, such as *add* and *sub*.

This work aims to propose an approach to learning high-level types for binary code. Motivated by "duck typing", we propose to learn types for recovered variables from their features and properties (*e.g.*, related instructions). Specifically, we first identify variables from binary code by analysing memory accesses. For instance, parameters and local variables are always accessed through address expressions of the form "[*ebp* + *offset*]" and "[*ebp* - *offset*]", respectively, where *ebp* is the stack base pointer register. Then for these recovered variables we extract their related instructions and some other helpful information as their features. Next, we train a classifier with basic types as its levels via various machine learning methods, based on binaries with debugging information compiled from a dataset of C programs. After the classifier is trained, we then can use it to learn the most possible types for the recovered variables. While for composite types, such as (multi-level) *pointer* and *struct*, we resorts to a combination of machine learning and program analysis techniques: we use a points-to analysis first to identify the possible variables and then use the classifier to learn for these variables basic types, which form the result type.

We implement our approach in a prototype named BITY, wherein we use the tool IDA Pro [1] as our front end to disassemble binaries and *scikit-learn* [8] to implement various classifiers. Using BITY, a series of experiments are conducted to evaluate our approach. Firstly, we conduct some experiments to see how well various machine learning methods perform. We have found that the classifiers trained by SVM with a linear kernel and Random Forest perform better than the others. Secondly, we conduct experiments to compare with the commercial tool Hey-Rays [1] and the open source tool Snowman [9] on the benchmarks *coreutils* (v8.4), *diffutils* (v3.5) and *findutils* (v4.7). The experimental results demonstrate that our tool is more precise than Hey-Rays and

Zhiwu Xu and Cheng Wen are with College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China.

Shengchao Qin (the corresponding author) is with School of Computing, Media and the Arts, Teesside University, Borough Road, Tees Valley, TS1 3BX, United Kingdom. He is also affiliated with Shenzhen University (as a Visiting Professor). Email: S.Qin@tees.ac.uk

Snowman, both in terms of correct types and compatible types. Thirdly, we also conduct experiments to compare BITY against EKLAVYA [10], a recent tool that can learn types for function parameters via machine learning, and the results show that BITY performs better than EKLAVYA. Fourthly, to evaluate BITY further, we conduct experiments on binaries of different sizes, which indicates that our prototype BITY is scalable and suitable in practice. Finally, as an immediate application, we feed the type information we learn into malware detection and find that discovered type information is capable of helping detect malware.

The contributions of the paper are twofold:

- An approach to learning types for binary code, using a combination of machine learning and program analysis, is proposed.
- A series of experiments are conducted to evaluate our approach, which demonstrated that our approach is able to learn more precise types, with reasonable performance, and can help detect malware.

This paper extends [11] and further contains the details of the revised algorithms, a points-to analysis for *pointer* and *struct*, the generation of the type learning problem, more experiments and several recent related work. In more detail, we first revise the variable recovery algorithm and instruction extraction algorithm for global variables. Second, we extend the original points-to algorithm to collect more possible variables, which can be used to form the possible struct types. Based on the extended points-to algorithm, an algorithm to recover struct pointer is then proposed. Third, we generalise the type learning problem such that any other machine learning technique can be applied here. Finally, we conduct more experiments to evaluate our tool BITY further: (1) we conduct several cross-validation experiments to evaluate how well the classifiers trained by various machine learning methods would perform; (2) we conduct experiments to compare BITY against Hex-Rays and Snowman, which recover types via program analysis on two new dataset diffutils and findutils; (3) We also conduct experiments to compare BITY against EKLAVYA, a recent tool that can learn types for function parameters from binaries via machine learning; (4) As an immediate application, we conduct experiments to check the viability of using the type information BITY learns to help malware detection.

Organization. Section II illustrates some motivating examples whose types are not recovered correctly by most of the existing tools and explains our idea. Our type learning for binary code is introduced in Section III. Section IV gives the experiments. Section V discusses the limitations and Section VI presents related work. Section VII concludes the paper.

II. Motivating Examples

This section illustrates some motivating examples where it is difficult to recover types correctly by existing approaches and based on which we explain our main idea.

The first example, given in Figure 1, is obtained by compiling a decode and encode program *base64* from C runtime Library with MSVC or GCC into a binary and then disassemble the binary with IDA Pro. For comparison, the source code in C is given as well. In the high-level program, a variable *decode* is declared with *bool* type and is used to record users' options. While in the low-level code, a *byte* in stack, that is [*ebp-1*], is simply used to represent the variable *decode*, without any type information. In other words, after compiling, the variable and its related type is lost. Because of the over-conservative program analysis techniques they adopt, most existing tools, such as Hex-Rays and SmartDec, recover the type *byte_t* or *char* for the recovered variable [*ebp-1*] , which are clearly too conservative or incorrect.

```
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
```

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

Fig. 1. Snippet Code of the program *base64*

```
......
mov eax, dword ptr [ebp + 8]
......
mov eax, dword ptr [ebp + 8]
......
mov eax, dword ptr [ebp + 8]
......
mov eax, dword ptr [ebp + 8]
......
mov eax, dword ptr [ebp + 8]
......
mov dword ptr [ebp + 8], eax
cmp dword ptr [ebp + 8], 0
......
```

Fig. 2. Snippet Code for the program *debug_script*

Secondly, consider the program *debug_script* from GNU Diff Utilities [12], which is used to print the changed information. This program takes as a parameter a variable with type (struct) pointer, which is represented as [*ebp + 8*] in assembly codes and whose related instructions are listed in Figure 2. According to the standard typing rules for assembly codes [2], [6], the first 6 related instructions infer that the type of [*ebp + 8*] should be a type of size 32, while the last instruction suggests that the type of [*ebp + 8*] is *int*. Accordingly, both the types recovered for [*ebp + 8*] by Hex-Rays and Snowman are *int*, which is incorrect.

What's worse, programs with fewer instructions can be more difficult for their types to be recovered correctly. Let us consider the pseudo assembly code given in Figure 3, which are compiled with MSVC[1] from three simple assignments for three variables with different types. Hex-Rays recovers for both the variables *i* and *f* the type *Dword** and for the variable *d* the type *Qword**. While SmartDec infers for all these three

[1]When compiling with GCC, the instruction for *i* remains the same, while the ones for *f* and *d* will be a *fld* followed by a *fstp*.

```
mov     [i], 0Ah
movss   xmm0, ds:XX
movss   [f], xmm0
movsd   xmm0, ds:XX
movsd   [d], xmm0
```

Fig. 3. Snippet Code for the Assignments of Different Types

variables the same type *int32_t*. Again, most of these results are over-conservative or incorrect.

One may note that these assignments for the variables of different types are compiled into different instructions, namely, *mov*, *movss* and *movsd* for *int*, *float* and *double*, respectively. Accordingly, one may want to improve the program analysis approaches by adding three new rules to infer those three different types corresponding to those three different instructions. However, this will not work since the instruction *mov* (*movsd* resp.) is not only used for the type *int* (*double* resp.). Even if it works, there are too many such kinds of instructions and types, therefore it would be difficult to figure out the possible reasonable rules. For instance, in the x86 instruction set, there are more than 30 kinds of *mov* instructions.

Another challenge in binary type inference, as discussed in [13], is from equivalent instruction sequences. The equivalent instruction sequences do not share the same type information under the typing rules. So the type information obtained from an instruction sequence can be lost in another equivalent instruction sequence. For example, both of the single instruction "and 0XFFFFFFFC, [ebp]" and the instruction sequence "not [ebp], or 0X3, [ebp], not [ebp]" produce the same result, that is, to align a pointer by clearing its lowest two bits. But for the second sequence, the type of [ebp] (*i.e.*, a pointer type) is lost, after applying on it the *not* operation twice.

Generally, the related instructions of a variable reflect how this variable is stored, interpreted, and manipulated. For instance, just as shown in Figure 3, variables of different types can have different instructions. Motivated by "duck typing", in which the type of a variable is determined by its features and properties rather than being explicitly defined, we take the related instructions of a variable as its feature, and learn the most possible type from the feature for the variable. Consider the program *base64* given in Figure 1 again. The related instructions of the memory byte [*ebp-1*] is the set "{mov _, 0; mov _, 1; movzx eax, _ }", which is most likely to be a feature of the type *bool*, where _ denotes the concerning variable. Therefore, we infer for the variable [*ebp-1*] the type *bool*. Similarly to the variables of the program given in Figure 3. Note that *movsd* may be a feature of *double*, but not all of them belong to *double*. Moreover, we believe that the equivalent instruction sequences can be treated as different possible features of the same type. For example, the single instruction "and 0XFFFFFFFC, [ebp]" and the instruction sequence "not [ebp], or 0X3,[ebp], not [ebp]" could be two possible patterns of pointer types.

III. Approach

We present our approach to learning types for binary code in this section.

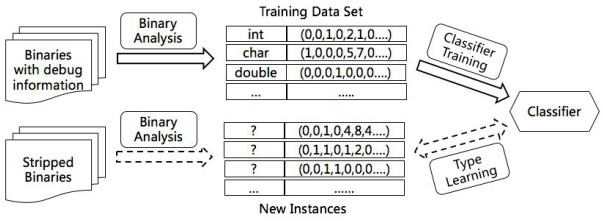


Fig. 4. Framework of Our Approach

As discussed in Section II, our approach is to learn the most possible type for the recovered variables from their related instructions. Figure 4 shows the framework of our approach, which consists of two main steps: (i) we first train a classifier with types as levels from existing binaries with some debugging information (marked by arrows with solid line), and (ii) we then use this classifier to learn the most possible types for new, unseen binaries (marked by arrows with broken line). In detail, we first perform some analysis on binaries with some debugging information to recover the possible variables and extract the related features and types for these variables, yielding a training data set. Based on the training set, we train a classifier with types as levels via machine learning methods. Next, we perform the similar analysis on stripped binaries to recover the variables and extract their features. We then learn the most possible types for these recovered variables, using the trained classifier. To conclude, our approach involves three tasks: (i) binary analysis; (ii) classifier training; (iii) type learning.

In what follows, we describe the types we used in the classifier and each task of our approach, using as an illustrative example the program *memchr*, given in Figure 5, from C runtime Library.

A. Types

The types we use in the classifier are the base types without type quantifiers, namely, the labels we are learning in the classifier are the set

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

There are two reasons behind this decision: (i) the other types can be composed from the base types, for instance, *struct*; and (ii) too many levels may work against the classifier. Figure 6 gives the lattice for the types we are learning, where $\top$ ($\bot$ resp.) denotes that a variable can (cannot resp.) be any type and there is a "pointer" starting from the type *pointer* to the lattice itself, namely, the type lattice is level-by-level (see the processing of *pointer* in Section III-D). This type lattice describes the hierarchy and the distance of types, and will be used to measure the precision of types (see Section IV). Similar to TIE [6], we focus on the sizes of types such that there are no subtype relations for types with different sizes, for instance, *short* is not a subtype of *int*.

Given a type t, we define its *level* as the number of the (outermost) levels occurring in it, that is, $level(t)$ is defined as $level(t') + 1$ if t is a *pointer* to t', otherwise 0.

```
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:
33      mov     eax, [ebp-44h]
......
sub_401000      endp
char *memchr (char *buf, int chr, int cnt) {
   while (cnt && *buf++ != chr) cnt--;
   return(cnt ? --buf : NULL);
}
```

Fig. 5. Snippet Code of the Program *memchr*

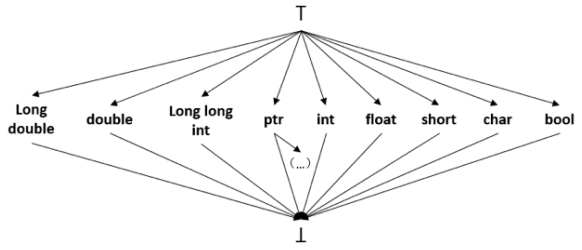


Fig. 6. Type Lattice

B. Binary Analysis

In this paper, we use the assembly code as an intermediate representation for binaries, which can be obtained by any disassembler such as IDA Pro. And we restrict our study to the x86 instruction set on Intel platforms, though the techniques presented here can be extended naturally to the others. Note that, some existing binary analysis techniques or toolkits, such as IDA pro [1], can be used here to recover variables from binaries and/or to extract direct related instructions. But the results may not suitable for our analysis for type learning, such as the indirect related instructions and instruction proceeding. So for practicability and for completeness, we presented the analysis that are targeted for our type learning.

After compiling, variables of the high-level source program and their type information are not included in the resulting binary. Therefore, the first step is to recover the target variables in the binaries. Since the types of variables are determined by their features, the next step is to extract the related features, namely, the related instructions of the recovered variables. In order to use the features in a classifier, the last step is to select the instructions that are the most representative and frequently used as the feature indicators, and represent them as a vector. In a word, our binary analysis consists of three steps: (i) target variable recovery; (ii) related instruction extraction; (iii) feature selection and representation. We present the detail of each step in the following.

1) Target Variable Recovery: As shown in [14], variables are abstractions of memory blocks, which are accessed by specifying absolute addresses directly or indirectly through address expressions of the form "[*base* + *index* × *scale* + *offset*]" in binaries, where *offset* and *scale* are integer constants, and *base* and *index* are registers. Hence, to recover the target variables in a binary is to identify the possible memory blocks in it. Similar to (the first step of) Value-Set Analysis (VSA) [14], we identify the target variables from functions, global data and heap memory. Since every function has its own stack frame to access to both function parameters and local variables, we proceed this task function by function. Function boundaries identification is an independent problem of interest, which is not covered here. Existing disassemblers or recent approaches [15]–[17] can identify the functions quite correctly. Even if not, the whole program can be treated as a single function.

Consider (the stack frame of) a general function. In CDECL and STDCALL conventions, parameters and local variables are always accessed through address expressions of the form "[*ebp* + *offset*]" and "[*ebp* - *offset*]", respectively, where *ebp* is the stack base pointer register[2]. While in FASTCALL convention, the first two parameters are passed by the registers *ecx* and *edx*, the other parameters and local variables are handled as the same as the conventions above. In all the conventions, the return values are passed by the register *eax*. So for each function, we identify variables from its stack frame and the data registers. Since data registers may be used multiply with possible *different* types in a function, we also take different uses of the same data register as *different* variables. For simplicity, we only consider the data registers here. But they are dependent. One can take some other registers into account as well, such as *ESI* and *EDI*. At last, global memory blocks and blocks in heap, which may be used by another function, are also considered. For example, global variables are always defined in the *data* or *bss* section, accessed by the address expression "ds:offset", while blocks in heap are accessed by the address expressions involving general registers, such as "[*eax* +/- *offset*]".

Algorithm 1 shows the procedure for variable recovery, which takes a function in ASM as input, and returns a set of possible variables. The algorithm starts with an empty set V (Line 1). Then it marks different uses of data registers (*i.e.*, *eax*, *ebx*, *ecx* and *edx*) in the function as Static Single Assignment Form (SSA) does (Line 2), which is given

[2]Under some higher optimization options, the *ebp* register is often used to store variables, which will be treated as "[*ebp* - 0]" (or "[*ebp*+0]" if one likes) in our setting.

in Algorithm 2. Finally, the algorithm proceeds with each instruction to collect the variables (Lines 3-16): Lines 4-6 and Lines 7-9 respectively handle the variables in stack frame and data registers used only in the target function, where v^f denotes that the target variable v only belongs to the function f, reg_n denotes the n-th definition (see below) of data register reg and $reg \in \{eax, ebx, ecx, edx\}$. While Lines 10-12 handle the other possible variables, where $varAddr$ denotes any other possible block, such as the global memory blocks accessed by the address expression "ds:offset" or the blocks in heap accessed by "[reg +/- offset]". Note that, due to compiler optimisation, multiple different local variables may be compiled into a simple stack location, which is not considered here.

Algorithm 1 Variable Recovery Algorithm $varrec(f)$

Input: a target function f
Output: the possible variable set V

```
1:  V = ∅
2:  D = regmark(f)
3:  for each instruction i ∈ f.b do
4:    if [ebp ± offset] ∈ i then
5:      V = V ∪ {[ebp ± offset]^f}
6:    end if
7:    if reg_n ∈ D(i).u ∪ D(i).d then
8:      V = V ∪ {reg_n^f}
9:    end if
10:   if varAddr ∈ i then
11:     V = V ∪ {varAddr}
12:   end if
13: end for
14: return V
```

We say an instruction i is a use (resp. a definition) of a data register r if i reads data from (resp. writes data into) r. For example, the instruction "add eax 1" reads data from eax first and then writes data back to eax, so it is a use as well as a definition of eax. In particular, a function call is a definition of eax, since the return value is always stored in it[3]. If any other register is not saved by the function, then the function call is also a definition for it.

Algorithm 2 shows the procedure for register marking, which takes a function f as input and returns a marked mapping for instructions. The algorithm first builds a control flow graph for function f (Line 1). Then it initializes each node of the graph with the empty definition set and use set (Lines $2-4$), and set the number of current definitions for each register as 0 (Line 5). After that, the algorithm starts from the entry nodes of graph with the zero definitions (Line 6), and traverses over the graph to collect the definition and use information (Lines $7-23$). That is, for each node n in the queue and each register r, the algorithm collects the definition and use information, according to n and r. In detail, if n is a use of r, then it updates the use information of n with the current definition of r (Lines $10-12$), where reg_n is the same to the one in Algorithm 1; and if n is a definition of r, then it increases the number of definitions of r and updates the current definition of r, which is added into the definition information of n (Lines $13-16$). If the information is different from the original one (Line 18), that is, there are some definitions of registers are fresh with respect to n, then the algorithm propagates the current definitions of registers to all the successors of n and enqueue them in the queue (Line $18-22$). This marking procedure is essentially a classic definition-use and use-definition analysis, wherein only the data registers are considered.

[3]It is possible that the return value is stored in the other registers, which should belong to the ones used by the exit node.

Algorithm 2 Register Marking Algorithm $regmark(f)$

Input: a target function f
Output: the marked mapping D

```
1:  let cfg be the control-flow graph of f
2:  for each node n ∈ cfg do
3:    D(n).u = ∅ and D(n).d = ∅
4:  end for
5:  num = [eax ↦ 0, ebx ↦ 0, ecx ↦ 0, edx ↦ 0]
6:  enqueue the entry node e and num into queue q
7:  while q ≠ ∅ do
8:    (n, cur) = dequeue q and (ou, od) = D(n)
9:    for each r ∈ {eax, ebx, ecx, edx} do
10:     if n is a use of r then
11:       D(n).u = D(n).u ∪ {r_cur[r]}
12:     end if
13:     if n is a definition of r and D(n).d = ∅ then
14:       num[r] + + and D(n).d = {r_num[r]}
15:       cur[r] = num[r]
16:     end if
17:   end for
18:   if (ou, od) ≠ D(n) then
19:     for each successor n' of n do
20:       enqueue (n', cur) into q
21:     end for
22:   end if
23: end while
24: return D
```

Let us consider the example *memchr* given in Figure 5. Table I lists the recovered variables in the stack frame, which consist of 3 parameters (which conform to the declarations in the C code) and 2 more local variables (which are due to the low-level instructions and are used to temporarily store the values of the variables **buf* and *chr*, respectively). Moreover, there are 9 different definitions of eaxs (*i.e.*, eax_0 - ebx_8), 4 different definitions of ecxs (*i.e.*, ecx_0 - ecx_3), and 2 different definitions of edxs (*i.e.*, edx_0 - edx_1). It seems that there are too many variables for data registers, but they can be reduced by definition-use chains (see the next section). Please note that compiling with a different compiler or with different options may generate different assembly code and thus different numbers of recovered variables.

2) Related Instruction Extraction: The next step of our binary analysis is to extract for the recovered target variables the related instructions from the binary code, which reflect how the variables are stored, interpreted and manipulated, and

TABLE I
VARIABLES IN THE STACK FRAME OF *memchr*

Variable	Expression	Variable	Expression
localVar1	[ebp-48h]	localVar2	[ebp-44h]
parameter1	[ebp+8]	parameter2	[ebp+0Ch]
parameter3	[ebp+10h]		

```
09    mov    eax, [ebp+8]
14    mov    eax, [ebp+8]
16    mov    [ebp+8], eax
26    mov    eax, [ebp+8]
28    mov    [ebp+8], eax
29    mov    ecx, [ebp+8]
```

Fig. 7. Instructions Using [ebp+8] Directly in *memchr*

will be used as a feature of the variables when learning their types.

A naive and simple solution is to extract for a variable the instructions which use it directly. Take the variable [ebp+8] of the program *memchr* for example. Figure 7 lists the instructions which use [ebp+8] directly. We can see that all these instructions are a *move* operation, which can be used by any type of variables. So these instructions are not enough for us to learn a type for it. Nevertheless, an instruction of a variable in high-level code is always compiled into several instructions in low-level code, some of which may not use the corresponding variable directly and thus are dropped by the simple solution. For instance, as shown in the program *base64* in Figure 1, the instruction "*if (decode)*" in C is complied to 2 instructions in ASM code with the variable [ebp-1], one of which uses [ebp-1] directly (*i.e.*, "movzx eax, byte ptr [ebp-1]"), while the other does not (*i.e.*, "test eax, eax"). Obviously, the second instruction is much more representative for the type *bool* and should be considered as well.

On the other hand, as mentioned above, there may be too many variables for different definitions of the data registers, namely, *eax*, *ebx*, *ecx* and *edx*. This is due to the fact that these data registers are usually used as an intermediary to temporarily store data, and that at different time they may store different data of different types. So not all of them are interesting. Moreover, according to the classic type system [18], when a variable is assigned by another variable or an arithmetical expression involving with another variable, the type of assigning variable is a subtype (denoted by $\leq$) of the one of the assigned variable. This indicates that the behaviours belonging to the assigned variable also belongs to the assigning variables. In particular, limited by our type lattice, both variables have the same type, as justified by Lemma III.1. Therefore, we collect these instructions of both variables together and learn for the assigning variable a type, which will be considered as the type for the assigned variable as well, based on the merged instructions.

Lemma III.1. *Let t, s be two types in L. Then $t \leq s \iff t = s$.*

Proof. Since any type in L is not a subtype of another type, the only solution to a subtype relation is the type itself. □

In detail, we make use of definition-use chains on these data registers to extract more interesting instructions: the uses of the data register, which is defined by a variable, are also considered as the uses of the variable. For instance, the instruction "test eax, eax" is considered as an instruction related to the target variable [ebp-1] in the program *base64* as well, since it is a use of *eax*, which is defined by [epb-1]. And we only need to learn for the variable (*e.g.*, [epb-1] in *base64*) a type, which will be considered as the type for the corresponding data register (*e.g.*, *eax* in *base64*) as well. This benefits to not only extracting more interesting instructions for a variable but also reducing the number of variables for different definitions of data registers. Indeed, only the initialised registers (*e.g.*, the parameters) and the ones used by the exit node (*e.g.*, the return variables) are interesting. Likewise, definition-use chains through function calls are also considered to extract as much interesting information as possible. Please note that the variable number for different definition of data registers can be reduced during variable recovery[4]. But in order to better explain instruction extraction, we present it here.

The procedure for instruction extraction is shown in Algorithm 3, which takes a target variable as input, and returns its related instruction set. The algorithm analyses the function f if the target variable belongs to f only, otherwise the whole program (Lines 2-6). Then for each function, the algorithm marks the definition-use chains, which is recorded in D (Line 8). At last, for each instruction, if it involves the target variable v, then it is added to the related instruction set (Lines 10-11). Moreover, if it is a definition of a data register, then the set of instructions that are related to this register is collected as well (Lines 12-13). In addition, our extraction considers the inter-procedural analysis: (i) if the current instruction calls a function f' and v is one of the arguments, then we also extract the related instructions for the corresponding parameter $v^{f'}$ of f' (Lines 16-18); and (ii) if v is (stored in) the return variable of f, then for each function f' that calls f, the related instructions for the variable (*i.e.*, eax) in f' that stores the return value from f is extracted (Lines 20-24).

Take the recovered variable [ebp+8] in the program *memchr* for example again. Its related instruction set is given in Figure 8, where the number n following a data register corresponds to the n-th definition identified during the variable recovery. Compared with the instructions listed in Figure 7, there are 4 more interesting instructions, which are collected due to the definition-use chains of these data registers (denoted by "use of " followed by a definition of a data register).

3) Feature Selection and Representation: According to the official document of the x86 instruction set [19], different instructions have different usages. Hence based on the usages, we perform some pre-processing on these collected instructions. Firstly, we notice that not all the instructions are of interest to type learning. For instance, the instructions *push* and *pop* are usually used by the stack, instead of by any variable. Secondly, different operands may have different meanings, so we distinguish between these two operands in a dyadic instruction. For instance, the two operands of

[4]BITY is implemented in this way.

Algorithm 3 Instruction Extraction Algorithm $insext(v)$

Input: a target variable v
Output: the related instruction set I

1: $I = \emptyset$
2: **if** $v \in f$ **then**
3: $F = \{f\}$
4: **else**
5: $F =$ the set of all functions
6: **end if**
7: **for** each function $f \in F$ **do**
8: $D = regmark(f)$
9: **for** each instruction $i \in f$ **do**
10: **if** $v \in i$ or $v \in D(i).u$ **then**
11: $I = I \cup \{i\}$
12: **for** $reg_n \in D(i).d$ **do**
13: $I = I \cup insext(reg_n)$
14: **end for**
15: **end if**
16: **if** i calls function f' and v is an argument **then**
17: $I = I \cup insext(v^{f'})$
18: **end if**
19: **end for**
20: **if** v is (stored in) the return variable of f **then**
21: **for** f' calling f **do**
22: $I = I \cup insext(eax_f^{f'})$
23: **end for**
24: **end if**
25: **end for**
26: **return** I

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

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

the instruction *mov* represent the source and the destination respectively, which are clearly different. Thirdly, we make abstractions on some operands, since they always lead to too many instructions: (i) we abstract the data registers with sizes, due to their interim uses; and (ii) we abstract immediate numbers into 0, 1 and $Other$, the first two of which are always used by the type *bool*. Fourthly, we also take the circumstances where the instructions are used into account. Considering the instruction *mov* again, using it with data registers of different sizes offers us different meaningful information. Its typical usage patterns we consider are given in Table II, where *Regn* denotes a data register with size n, _ denotes a concerning variable, and *varAddr* denotes a memory address (*i.e.*, another variable).

There are more than 600 instructions, but not all the instructions are representative or widely used. Therefore, using

TABLE II
TYPICAL USAGE PATTERNS OF *MOV*

mov Reg8, _	mov _, Reg8	mov Reg16, _	mov _, Reg16
mov Reg32, _	mov _, Reg32	mov _, 0	mov _, 1
mov _, Other	mov varAddr, _	mov _, varAddr	

the well-known scheme Term Frequency-Inverse Document Frequency (TF-IDF) [20], we perform a statistical analysis on the dataset, which consists of source code from textbooks and real-world programs. According to the results, we select the top N representative and frequently used instructions as the feature indicators. Theoretically, the more instructions, the better. While in practice, we found 100 instructions are enough.

In addition, some other useful information for type inference are considered as well. Firstly, the memory size is very helpful for type inference, so we also take it into account as a feature if we can identify it. Secondly, when the type of a called function is known, especially for system functions, if a variable is one of its arguments, then we can infer for the variable a type easily, that is (a subtype of) the type of the corresponding parameter, according to the rule for function calls in classic type system. Therefore, in Lines 16-18 of Algorithm 3, if the type of f' is known, we represent the related instructions of $v^{f'}$ as one feature "being an argument of t", where t is the type of $v^{f'}$. In practice, we can use another feature "being the n-th argument of f" for a system function f to avoid finding the type of f. Likewise, Lines 20-24 could be simplified as "being the return of t" or "being the return of f".

Finally, similar to information processing, we would like to encode the selected features of variables into vectors, where only the numbers of times the selected instructions are performed on the target variables are considered, leaving the orders out of consideration. That is, we represent variables as vectors, which consist of the frequencies of the selected instructions and the extra useful information. Formally, a representation of a variable is a vector of the form:

$$v = [t_1 : x_1,\ t_2 : x_2,\ \ \ldots,\ t_n : x_n]$$

where n is the number of features, t_i is a feature term, and x_i is the value of feature term t_i. Note that ignoring the instruction order can give us a simple and easy model, but it may reduce the precision of the model if there are two different types with different behaviour patterns such that these patterns share the same instruction set.

Take the variable [ebp+8] of the program *memchr* for example, whose related instructions are given in Figure 8. The representation vector of [ebp+8] is given in Table III, where the left hand side shows the vector of the specific instructions before proceeding, while the right hand side gives the vector of the abstracted instructions after proceeding, and only the nonzero features are listed. Note that "mov eax, _ " and "mov ecx, _" are merged together, since both *eax* and *ecx* are data registers of 32 bits.

TABLE III
REPRESENTATION OF [EBP+8]

Before Proceeding		After Proceeding	
Feature	**Value**	**Feature**	**Value**
add _, 1	1	add _, 1	1
mov eax,_	3	mov Reg32, _	4
mov ecx,_	1	merged to "mov Reg32,_"	
mov [ebp-44h], _	1	mov varAddr, _	1
mov _ ,eax	2	mov _ , Reg32	2
movsx ecx, [_]	1	movsx Reg32, [_]	1
sub _, 1	1	sub _, 1	1
Size	32	Size32	1

C. Classifier Training

In our approach, the classifier is trained by supervised learning. So we need a labeled dataset. To the end, we compile a dataset of C programs with debugging support and then extract the related information (*i.e.*, variables, types, and features) from the compiled binaries, yielding a training set. Let V be the feature space for all possible vectors. Our classifier training problem is expressed as: given a labeled dataset $D_0 = \{(v_1, l_1), (v_2, l_2), \ldots, (v_m, l_m)\}$, the goal is to find a classifier $C : V \rightarrow L$ that minimizes the sum of the distances of all the variables, namely,

$$argmin_C \sum_{(v,l)\in D_0} d(C(v), l) \quad (1)$$

where m is the number of variables, v_i is the feature vector of a variable, $l_i \in L$ is the corresponding type of v_i, and d is the distance function on types. Similar to TIE [6], we define the distance function d as the distance between them in our lattice:

$$d(s,t) = \begin{cases} 0 & s = t \\ 1\ (\textit{half maximum height}) & s, t\ are\ pointer \\ 2\ (\textit{maximum height}) & otherwise \end{cases}$$

Note that, similar to Elwazeer *et.al.*'s work [4], we can use the radio $min(level(s), level(t))/max(level(s), level(t))$ for pointers in theory, but in practice we consider only 3 levels for pointers, so we use the half here. Our classifier C aims to find the most possible type for a variable, so we define C as

$$\text{given a variable } v,\ C(v) = argmax_{l\in L}\ P(v, l)$$

where $P(v, l)$ is the probability that the variable v is assigned by the type l. Without loss of generality, the probability $P(v, l)$ can be encoded as follows:

$$P(v, l) = exp^{\textit{Score}(v,l)} / \sum_{l\in L} exp^{\textit{Score}(v,l)}$$

where $\textit{Score}(v, l)$ is a function that returns the score of assigning the type l for the variable v. Assignments with higher scores are more likely than assignments with lower scores. Since we learn types from the related features of variables, we express the *Score* function as a composition of a sum of n feature functions F_i associated with the related weights w_i:

$$\textit{Score}(v, l) = \sum_{i=1}^{n} w_i \times F_i(x_i, l) = \vec{w} \times \vec{F}(v, l)$$

where $v = [t_1 : x_1,\ t_2 : x_2,\ \ldots,\ t_n : x_n]$, $\vec{w}$ is a vector of weights w_i, and $\vec{F}$ is a vector of feature functions F_i. There are several solutions to the definition of $\vec{F}$, for example, Kullback-Leibler divergence. But here we reuse the TF-IDF values, computed in Section III-B3, to define the feature function as follows:

$$F_i(x_i, l) = x_i \times idf^l_{t_i}$$

where $idf^l_{t_i}$ is the IDF value of feature term t_i with respective to the type l. The remaining problem is to find a vector $\vec{w}$ of weights such that Condition (1) is satisfied, which can be solved by various algorithms of machine learning, such as decision tree, k-nearest neighbor, native bayes, random forest and support vector machine. We have tried these algorithms to solve our training problem in our implementation. We have also carried out several experiments with these algorithms and have found that the classifier trained by Support Vector Machine with a linear kernel and Random Forest with 10 gini trees perform the best (more details will be given in Section IV).

D. Type Learning

Once it is trained, the classifier C can be used to learn types for new, unseen binaries. Intuitively, the solution is to return for a given variable v the type whose probability is the highest one as its definition, that is

$$argmax_{l\in L}\ P(v, l).$$

Take the variable [ebp+8] of the program *memchr* as an example again. Its feature instruction set contains "mov Reg32, _; movsx Reg32, [_]" (to read data from an address expression), "mov Reg32, _; add _, 1" (to increase the address expression), and "mov Reg32, _; sub _,1" (to decrease the address expression), which are the typical usages of the type *pointer*. Accordingly, the probability of P([ebp+8], *pointer*) gets the highest score, and thus the most possible type the classifier learns is *pointer*. Let us consider the variable *decode* in the program *base64* given in Figure 1. As discussed in Section II, its feature instruction set (*i.e.*, "mov _, 0; mov _, 1; movzx Reg32, _; test _, _") is one of the typical usages of the type *bool*, so our classifier will learn *bool* as the most possible type.

1) Composite Types: This section presents how to handle composite types, namely *pointer* and *struct*, using a combination of machine learning and program analysis techniques.

Pointer. For a higher accuracy, we handles *pointer* level-by-level as shown in our type lattice in Figure 6. This is because we would like (i) to learn not only the pointer type itself but also the type that the pointer type points to, which may be a pointer type as well; and (ii) to handle the multi-level *pointer*s.

We say a variable is *indirect* if there exist at least one other variable of *pointer* type pointing to it.

In detail, our approach proceeds as follows:

1) Once a variable v is learnt to have type *pointer* by our classifier, our approach first tries to identify any variable that the *pointer* variable points to, using a points-to analysis, which is motivated by Brumley and Newsome's work [21] and shown in Algorithm 4.

2) If such indirect variables exist, the approach then collects the related features for these newly recovered variables and continues to learn a (next-level) type t for all these variables with the classifier. Similar to definition-use chain (*i.e.*, Lemma III.1), we argue that the variables pointed to by the same variable share the same type. One can think that the type for an indirect variable is a vote, so t is the type which gets the most votes.
3) At last, the type for the variable v is a *pointer* to t if there exists at least one indirect variable, otherwise a *pointer* (to any type).

Our approach can handle *pointer*s with any levels in theory (and thus may not terminate). But in practice, we have found that up to 3 levels are enough.

Algorithm 4 shows our points-to algorithm, which takes a target variable as input and returns the set of possible variables pointed to by the target variable. The idea is that whenever a variable is loaded or stored by the memory location addressed by the target variable, the variable is pointed to by the target one (Lines 9-12). Transitivity of variables is taken into account (Lines 13-15). The proceeding above is quite similar to the IDB predicates in [21] without the rules for expression operators, since our analysis considers the possible variables rather than the (address) values. In addition, our analysis considers the inter-procedural situations (Lines 16-24).

Algorithm 4 Points-to Algorithm $point\_to(v)$

Input: a target variable v
Output: the possible points-to variable set V

```
1: V = ∅
2: if v ∈ f then
3:    F = {f}
4: else
5:    F = the set of all functions
6: end if
7: for each function f ∈ F do
8:    for each instruction i ∈ f  do
9:       if i is v' = *v or *v = v' then
10:         V = V ∪ {*v, v'}
11:      else if  *v ∈ i or *(v + offset) ∈ i then
12:         V = V ∪ {*v}
13:      else if i is v' = v or v = v' then
14:         V = V ∪ point_to(v')
15:      end if
16:      if i calls function f' and *v is an argument then
17:         V = V ∪ {p_{*v}^{f'}}
18:      end if
19:   end for
20:   if *v is (stored in) the return variable of f then
21:      for f' calling f do
22:         V = V ∪ {eax_f^{f'}}
23:      end for
24:   end if
25: end for
26: return V
```

Compared to our conference version [11], there are two minor differences for the processing for *pointer* type: one is to take some possible offsets into account, that is, the address pattern $*(v + offset)$ in Line 11; and the other is to extend transitivity of variables from data registers to any possible variables, that is v' in Line 13 can be any possible variable. Both of them enable us to find more indirect variables. Theoretically, the more the indirect variables, the more precise the learnt type. But in our experiments, we found the results are almost the same. The reason is that (1) when there is an indirect variable with an offset, there always exists another indirect variable without an offset, and transitivity of variables are always passed by data registers; (2) a (correct) type can always be learnt from some indirect variables (*e.g.*, the ones found by the original version), since they share the similar instructions. However, this extended algorithm benefits struct recovery very well: it enables us to find more fields (see the struct recovery).

Let us carry on with the variable [ebp+8] of the program *memchr*. In Section III-D, we have learnt for the variable [ebp+8] the most possible type *pointer*. So our approach goes on to identify any possible indirect variable, yielding the variable set "{byte ptr [eax], ecx1}". Then our approach extracts the following feature vector for it:

[mov Addr, _ : 1; movsx Reg32, _: 1; Size8: 1]

which covers the data move with sign extension. There are two types with 8 bits, that is, *char* and *bool*. According to the known binaries, the feature above more likely belongs to *char* than to *bool*. Therefore, the final type for the variable [ebp+8] is a *pointer* to *char*, which is exactly the same as the one in the high-level code.

Struct. Another common composite type is *struct*, which consists of several possible different types. For example, Figure 9 shows a definition of struct *Point* in C language, which consists of two components of type int. Let us consider two simple functions that perform on the struct Point. The first one $func1$ defines a variable of type struct Point and initializes all its members, while the second one $func2$ does the similar operations, except that the variable is defined with type struct Point pointer. Figure 9 also gives the snippet assembly codes of these two functions. From the assembly codes, we can see that the struct Point variable in $func1$ is encoded into two memory blocks [ebp-8] and [ebp-4] in statck, corresponding to two components of the struct Point. In other words, this struct variable is compiled into two different variables and thus the struct information is lost. While in $func2$, the variable of type struct Point pointer is represented as a single block [ebp-8] in stack, and through this block, two other blocks [eax] and [eax+4] (*i.e.*, [[ebp-8]+0] and [[ebp-8]+4]) in heap can be accessed, which corresponds to two components of the struct Point. Compared to $func1$, it is easier to recover the struct information from $func2$.

In this paper, we only consider the structs that are accessed indirectly (*i.e.*, through a pointer), since structs that are accessed directly are always compiled into several different variables corresponding to their components, which is difficult to recover. That is, the form of *struct* type we learn is of the form **struct*. So the first problem is to distinguish struct

```
struct Point {
 int x;
 int y;
};
void func1() {
  struct Point p;
  p.x = 2;
  p.y = 4;
}
void func2() {
 Point* p =  (Point*)
       malloc(sizeof(Point));
 p->x = 2;
 p->y = 4;
}
```

```
func1 proc near
 ......
 mov dword ptr [ebp-8], 2
 mov dword ptr [ebp-4], 4
 ......
func1 endp

func2 proc near
 ......
 mov eax, [ebp-8]
 mov dword ptr [eax], 2
 mov eax, [ebp-8]
 mov dword ptr [eax+4], 4
 .......
func2 endp
```

Fig. 9. Snippet Code for simple struct programs

pointer from the other pointers, which is also a classifier problem. But, instead of the machine learning methods used above, our solution here is to take advantage of the points-to analysis above and to seek the common pattern used for **struct* in low-level code: the indirect access pattern [base + offset] with the same base (*i.e.*, variables pointed to by the same variable). *Arrays* can be handled in similar way.

Our struct recovery proceeds as follows:

1) Once a variable v is learnt to have type *pointer* by our classifier, assume that the variable set it points to is V. Then our approach tries to collect the variables whose bases are in V, yielding a variable set V'
2) If $|V'| > 1$, we infer that the type of v is a struct pointer. Then we continue to learn a type t_i for each variable v_i in V'. After that, we order the variables in V' and construct the following struct type:
   $$\begin{array}{l} struct\ \ anyname\ \ \{ \\ \quad offset_1 :\ t_1 \\ \quad ... \\ \quad offset_n :\ t_n \\ \} \end{array}$$
   where there exists a $base_i \in V$ such that $[base_i + offset_i] \in V'$.
3) If $|V'| \leq 1$, we proceed as the pointer recovery.

Consider $func2$ in Figure 9 again. The type of [ebp-8] is learnt to be pointer, according to its instructions. By applying the points-to algorithm, we get the points-to variable set $V = \{[eax_n], [eax_{n+1}]\}$, where eax_i denotes the i-th definition of eax in $func2$. Note that, without the extension of points-to algorithm, $[eax_{n+1}]$ cannot be identified. Based on V, we found two variables that share the same base: $[eax_n+0]$ and $[eax_{n+1}+4]$, and both of their types are learnt to be int. According, we obtain the following struct type:

$$\begin{array}{l} struct\ \ anyname\ \ \{ \\ \quad 0 :\ int \\ \quad 4 :\ int \\ \} \end{array}$$

IV. Experiments

We have implemented our approach in a prototype named BITY, wherein we use the tool IDA Pro [1] as our front end to disassemble binaries and *scikit-learn* [8] to implement various classifiers. Using BITY, several experiments are conducted to evaluate our approach. Firstly, several cross-validation experiments are conducted to evaluate how well the classifiers trained by various machine learning methods would perform. Secondly, to evaluate our tool BITY, we conduct experiments to compare BITY against Hex-Rays and Snowman, which recover types via program analysis. Thirdly, we also conduct experiments to compare BITY against EKLAVYA, a recent tool that can learn types for function parameters from binaries via machine learning. Fourthly, experiments on evaluating the scalability of BITY are also conducted. Finally, as an immediate application, we conduct experiments to check the viability of using the type information BITY learns to help malware detection.

The experiments were conducted on a personal computer with Intel Processor i5-4590 (3.30GHz) and 8GB memory.

A. Training Dataset

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 via MSVC and gcc:

- Source codes of the C programming language (K&R);
- Source codes of basic algorithms in C programming language [22];
- Source codes of commonly used algorithms [23];
- C Runtime Library;
- Some C programs collected from github randomly.

B. Performance of Different Classifiers

As mentioned in Section III-C, there are various machine learning algorithms to train our classifier. For that, we conduct a series of experiments to compare the performance of various classifiers here, wherein 5-*fold cross validation* is performed. First of all, the dataset is divided into 5 equal folds randomly, each of which is taken as the testing set and the others as the training set. Then based on the training set, we train the classifier using a machine learning algorithm. At last, we test the classifier on the testing set. The machine learning algorithms we use here are: decision trees (DT for short), where the metrics Gini impurity and information gain are used; k-nearest neighbour (KNN for short), where the value of k ranges in $\{1, 3, 5, 7\}$; native Bayes (NB for short), including the models Gaussian naive Bayes (GNB for short), multinomial naive Bayes (MNB for short), and Bernoulli naive Bayes (BNB for short); support vector machine (SVM for short), where the linear function kernel, the radial basis function kernel (RBF for short) and the sigmoid function kernel are used; and random forest (RF for short), which consists of 10 decision

TABLE IV
RESULTS OF DIFFERENT CLASSIFIERS

Classifier	Precision	Recall	F1
DT-Gini	0.9457	0.9447	0.9444
DT-Gain	0.9434	0.9418	0.9419
1-KNN	0.9462	0.9437	0.9432
3-KNN	0.9390	0.9368	0.9364
5-KNN	0.9337	0.9329	0.9325
7-KNN	0.9296	0.9289	0.9384
GNB	0.7661	0.5775	0.6002
MNB	0.9172	0.9072	0.9052
BNB	0.9240	0.9299	0.9264
SVM-Linear	0.9466	0.9437	0.9432
SVM-RBF	0.9440	0.9427	0.9425
SVM-Sigmoid	0.6619	0.5548	0.4304
RF-(10, Gini)	0.9461	0.9457	0.9453
RF-(10, Gain)	0.9454	0.9447	0.9444

trees. The performance measures we use to validate the results quantitatively are as follows:

$$\textit{Precision} = \frac{TP}{TP + FP}$$
$$\textit{Recall} = \frac{TP}{TP + FN}$$
$$\textit{F1} = 2 * \frac{\textit{Precision} * \textit{Recall}}{\textit{Precision} + \textit{Recall}} = \frac{2 * TP}{2 * TP + FP + FN}$$

The experimental results are shown in Table IV. From the results we can see that most of the classifiers, except for GNB and SVM with sigmoid kernel, work quite well: the precision, recall and F1-measure are all over 90%. In particular, SVM with a linear kernel obtains the best precision (94.66%), while RF with 10 Gini trees gets the best recall (94.57%) and F1-measure (94.53%). One of the main reasons for RF to outperform others is that RF is an ensemble learning method which may correct the decision tree's habit of overfitting to the training set. While SVM with a linear kernel tries to transform the original input set into a high-dimensional feature space by using a linear kernel function, and then construct an n-dimensional hyperplane that optimally separates the variables into categories. Theoretically, we can obtain a correct classifier if the dimension n is large enough. So we suggest to use the classifier trained by SVM with a linear kernel.

C. Comparison against Hex-Rays and Snowman

This section presents the experiments to compare BITY against Hex-Rays (v2.2.0.15), which is a plug-in of the commercial tool IDA Pro [1], and Snowman (v0.1.0), which is an open source C/C++ decompiler [9].

In order to quantitatively validate the result types recovered by different tools, we extend the *distance* function d given in Section III-C such that it still works on the types recovered by Hex-Rays and Snowman. For that, we extend the lattice in Figure 6 with the types recovered by these two tools. Figure 10 gives the extended lattice, where Hex-Rays and Snowman consider all the types except $\top$ and $\bot$, while BITY considers only the types in bold. Note that the types recovered by Snowman are similar to Hex-Rays but with different names,

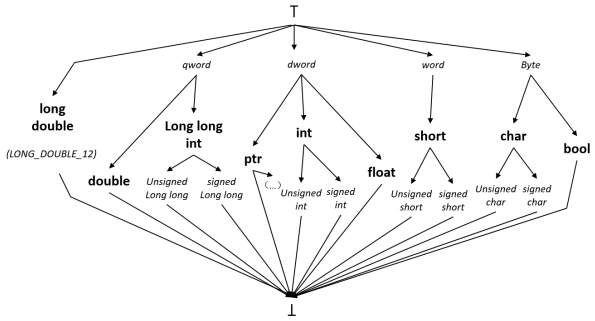


Fig. 10. Type Lattice for Hex-Rays and BITY

for example, Snowman uses *int32_t* for *int* and *uint32_t* for *unsigned int*. It seems that the distance function d can be easily and naturally extended on this new extended lattice. However, due to the types we imported for Hex-Rays and Snowman, there are more options for us to predict for a variable. For example, we can predict the types *dword*, $\top$, or *float* for a variable of type *int*. Clearly, with respect to the type *int*, the type *dword* is better than the type $\top$, and both of them are better than the type *float*. But the original distance function d does not tell us the differences among these three types. So to express the differences and validate the results more precisely, we borrow the notation *compatible types* from TIE [6]. Given two types, if one type is a subtype of the other one, we said they are *compatible*. Now the distance function d between two types t, s is extended as follows:

- if t and s are *pointer* to t' and s' respectively, then $d(t, s)$ is defined as the half of the maximum hierarchy height 2 multiplied by 1, 0.5 and 0, according to whether t' and s' are incompatible, compatible, or the same, respectively;
- otherwise, $d(t, s)$ is defined as the number of hierarchies between t and s in the top-level lattice if they are compatible, otherwise the maximum hierarchy height 4.

For instance, both $d(*dword, *int)$ and $d(dword, int)$ are 1, while the distance $d(*dword, int)$ is 4. Note that, compared with the original one, the extended distance function returns a higher value if BITY gives a wrong answer.

1) Experiments on Coreutils: The first test programs we used for comparison are from GNU Core Utilities, namely, *coreutils-v8.4*, which is a benchmark used for evaluation by several existing work [3], [6], [24]. The experiments proceed as follows: (i) The test programs are compiled into binaries with debugging support, from which the type information is extracted and used as a reference for the tools. (ii) We perform our tool BITY on the stripped binaries, obtained from the test programs by compiling without debugging support, to identify the target variables in stack and learn for them types, and compare the types learned by BITY against the reference types. (iii) We use Hex-Rays to decompile the stripped binaries, and compare the recovered types against the reference types. (iv) Similar to Hey-Rays, Snowman is used as well. But due to the different front-ends, the corresponding relation between the variables recovered by our tool and the ones by Snowman is unclear. So we have to analyse the decompiled code by Snowman to build this correspondence

manually. And it is more difficult for global variables to figure out this correspondence than local variables. Due to time limitation, we perform only on 42 programs in coreutils and consider only local variables and parameters for comparison.

As pointed out in [25], we also found that there are some duplicate functions in coreutils, so we eliminate some common functions, such as usage and emit functions. Moreover, during experiments, we only counted the functions whose related types that can be recovered by all the tools. The experimental results are given in Table V, where **Variables** is the number of the target variables, **R**, **C** and **F** are the number of variables, whose types are respectively recovered correctly, compatibly and incorrectly, and **P** is the percentage of the correct types and the compatible types, both of which together are called *proper* types, among all the types. From the results, we can see that (i) BITY can learn 80% above proper types for most of the programs (*i.e.*, 44 programs among 45 ones), while Hex-Rays and Snowman can respectively recover 26 and 19 programs with 80% above proper types; (ii) On the whole, among the types learned by our tool BITY, 1356 (58.12%) types are correct, 729 (31.25%) types are compatible, 2085 types (89.37%) in total are proper; While Hex-Rays recovers 1276 correct ones (54.69%) and 589 compatible ones (25.25%), in total 1865 proper ones (79.94%); and Snowman recovers 995 correct ones (42.65%) and 713 compatible ones (30.56%), in total 1708 proper ones (73.21%); (iii) On average, the proper accuracy rating of BITY, Hex-Rays and Snowman are 90.32%, 82.96% and 74.28%. In conclusion, BITY performs better than Hex-Rays and Snowman, all in terms of correct types, compatible types, or proper types.

According to the reference type information, we have found that there are 1021 variables that are typed by *pointer*. Table VI gives the type results recovered for these variables by our tool BITY, Hex-Rays and Snowman, where the notation is the same as the ones in Table V. Among the types learned by BITY, 444 (43.49%) ones are correct, 397 (38.88%) ones are compatible, and in total 841 (82.37%) ones are proper. While for Hex-Rays, 397 (38.88%) ones are correct, 203 (19.88%) ones are compatible, and in total 600 (58.77%) ones are proper; and for Snowman, 238 (23.31%) ones are correct, 399 (39.08%) ones are compatible, and in total 637 (62.39%) ones are proper. The main reason for BITY to learn compatible types is the lack of the type quantifiers such as *unsigned* and *signed*, while the main reason for Hex-Rays and Snowman is to use the conservative types. The results indicate that in terms of *pointer* types, BITY also performs better than Hex-Rays and Snowman.

Among the variables of type *pointer*, we have found that there are 350 of them that are typed by *struct**. Table VII shows the results for these struct pointers recovered by BITY, Hex-Rays and Snowman. For these variables, our tool can learn *pointer* as the type for 296 (84.57%) variables, among which 130 (37.14%) ones are learned correctly with the type *struct**[5]. While Hex-Rays recovers 146 (41.71%) variables with *pointer*, among which 60 (17.14%) ones are recovered correctly; and Snowman recovers 216 (61.71%) variables with

[5]For simplicity, types for fields are not considered here.

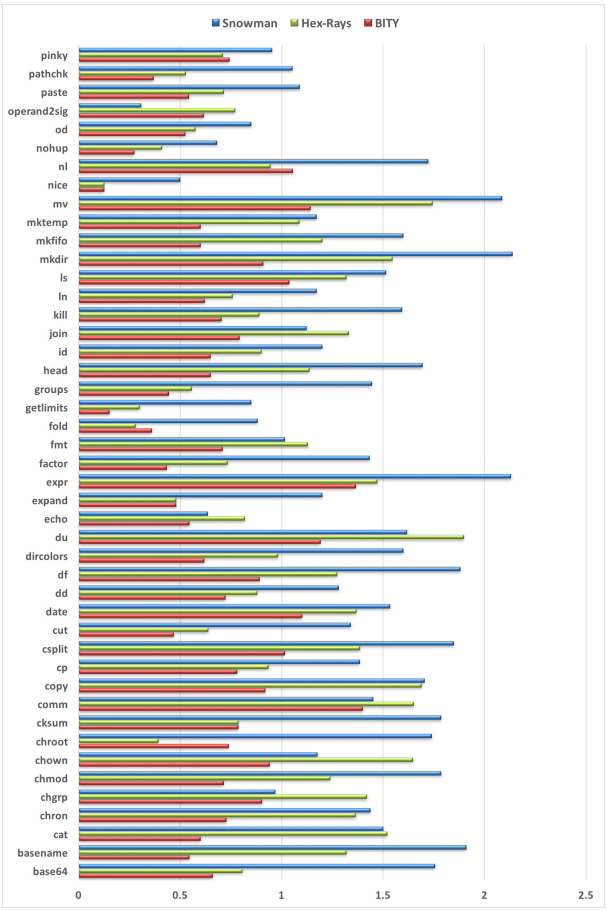


Fig. 11. Distances of BITY, Hex-Rays and Snowman on Coreutils

pointer, among which 182 (52.0%) ones are recovered correctly. This indicates that, on struct pointers, our tool performs better than Hex-Rays as well, but a litter worse than Snowman. Although Snowman can recover more struct pointers, it tends to infer the pointer into a struct pointer even a pointer of the basic type.

Moreover, we have analysed some failure cases manually, and found that there are 3 main reasons: (i) some variables have too few related instructions for us to learn the right types, particularly the variables typed by *pointer*, which contribute to most of the failures as shown in Tables V and VI (72.58%, 89.96% and 61.44% for BITY, Hex-Rays and Snowman, respectively); (ii) there are some variables typed of composed types like *array* and (direct) *struct*, which are not easy to recover and not considered by BITY yet; (iii) for a variable typed of *struct**, if only one field is accessed, then BITY would learn a pointer pointing to the type of the accessed field, rather than *struct**.

At last, concerning the distance, Figure 11 gives the average distances of each program for BITY, Hex-Rays and Snowman, from which we can see that BITY can learn types with a shorter distance for most programs than both Hex-Rays and Snowman. On average, the average distance of the types recovered by BITY, Hex-Rays and Snowman for the test programs is 0.715, 1.014 and 1.372, respectively, indicating that BITY can learn more precise types than both Hex-Rays

TABLE V
COMPARISON RESULTS OF BITY, HEX-RAYS AND SNOWMAN ON COREUTILS

Program	Variable	BITY				Hex-Rays				Snowman			
		R	C	F	P	R	C	F	P	R	C	F	P
Base64	41	20	19	2	95.12%	29	7	5	87.80%	21	4	16	60.98%
Basename	22	17	4	1	95.45%	12	4	6	72.73%	11	1	10	54.55%
Cat	50	29	19	2	96.00%	18	19	13	74.00%	18	18	14	72.00%
Chron	55	39	8	8	85.45%	32	7	16	70.91%	28	12	15	72.73%
Chgrp	31	21	4	6	80.65%	17	4	10	67.74%	17	10	4	87.10%
Chmod	42	19	20	3	92.86%	20	13	9	78.57%	13	15	14	66.67%
Chown	17	10	4	3	82.35%	6	6	5	70.59%	8	7	2	88.24%
Chroot	23	12	9	2	91.30%	18	4	1	95.65%	8	7	8	65.22%
Cksum	14	6	7	1	92.86%	7	6	1	92.86%	4	5	5	64.29%
Comm	20	10	4	6	70.00%	11	1	8	60.00%	10	4	6	70.00%
Copy	135	69	48	18	86.67%	50	42	43	68.15%	29	72	34	74.81%
Cp	78	46	26	6	92.31%	45	23	10	87.18%	21	43	14	82.05%
Csplit	66	27	32	7	89.39%	26	25	15	77.27%	24	16	26	60.61%
Cut	47	32	14	1	97.87%	31	15	1	97.87%	21	17	9	80.85%
Date	30	18	8	4	86.67%	15	8	7	76.67%	11	10	9	70.00%
Dd	128	81	35	12	90.63%	78	30	20	84.38%	63	33	32	75.00%
Df	92	51	32	9	90.22%	45	25	22	76.09%	27	35	30	67.39%
Dircolors	55	31	23	1	98.18%	26	23	6	89.09%	24	10	21	61.82%
Du	68	27	28	13	80.88%	26	15	27	60.29%	17	32	19	72.06%
Echo	11	8	3	0	100%	5	6	0	100%	7	3	1	90.91%
Expand	25	16	8	1	96.00%	16	9	0	100%	13	6	6	76.00%
Expr	85	29	39	17	80.00%	28	35	22	74.12%	23	22	40	52.94%
Factor	30	20	9	1	96.67%	22	4	4	86.67%	17	3	10	66.67%
Fmt	62	40	15	7	88.71%	40	7	15	75.81%	37	13	12	80.65%
Fold	25	17	8	0	100%	20	5	0	100%	14	8	3	88.00%
Getlimits	20	17	3	0	100%	17	2	1	95.00%	15	1	4	80.00%
Groups	9	5	4	0	100%	5	4	0	100%	3	4	2	77.78%
Head	111	63	41	7	93.69%	52	42	17	84.68%	37	36	38	65.77%
Id	20	13	5	2	90.00%	12	5	3	85.00%	7	10	3	85.00%
Join	106	48	52	6	94.34%	54	24	28	73.58%	44	46	16	84.91%
Kill	27	18	6	3	88.89%	15	9	3	88.89%	12	6	9	66.67%
Ln	29	23	3	3	89.66%	21	5	3	89.66%	11	13	5	82.76%
Ls	352	189	105	58	83.52%	186	73	93	73.58%	156	93	103	70.74%
Mkdir	22	15	4	3	86.36%	10	5	7	68.18%	6	6	10	54.55%
Mkfifo	10	7	2	1	90.00%	7	0	3	70.00%	6	0	4	60.00%
Mktemp	35	23	9	3	91.43%	16	15	4	88.57%	16	13	6	82.86%
Mv	35	20	8	7	80.00%	15	8	12	65.71%	7	14	14	60.00%
Nice	16	15	1	0	100%	15	1	0	100%	12	1	3	81.25%
Nl	18	11	4	3	83.33%	12	3	3	83.33%	8	3	7	61.11%
Nohup	22	20	1	1	95.45%	19	1	2	90.91%	13	7	2	90.91%
Od	120	88	23	9	92.50%	86	25	9	92.50%	82	18	20	83.33%
Operand2sig	13	11	0	2	84.62%	9	2	2	84.62%	9	4	0	100%
Paste	35	26	7	2	94.29%	24	9	2	94.29%	16	15	4	88.57%
Pathchk	19	15	3	1	94.74%	14	4	1	94.74%	8	8	3	84.21%
Pinky	62	34	22	6	90.32%	44	9	9	85.48%	41	9	12	80.65%
Total	2333	1356	729	248	-	1276	589	468	-	995	713	625	-
Average	-	-	-	-	90.32%	-	-	-	82.96%	-	-	-	74.28%

TABLE VI
RESULTS OF POINTERS ON COREUTILS

Num	BITY			Hex-Rays			Snowman		
	R	C	F	R	C	F	R	C	F
1021	444	397	180	397	203	421	238	399	384

TABLE VII
RESULTS OF STRUCT POINTERS ON COREUTILS

Num	BITY		Hex-Rays		Snowman	
	pointer	struct*	pointer	struct*	pointer	struct*
350	296	130	146	60	216	182

and Snowman.

2) Experiments on Diffutils and Findutils: The second benchmark we used for comparison are from GNU Diffutils and GNU Find Utilities, namely, *diffutils-v3.5* and *findutils-v4.7.0*, which is a benchmark used by EKALVYA [10]. For convenience, we collect the data of findutils and diffutils from the dataset of EKALVYA, which are obtained from gcc compiler without optimization and have eliminated the duplicated functions. Here we focus on the function parameters, as EKALVYA can only learn types for them. The experiments proceed as the same as the one on coreutils.

Table VIII shows the experimental results, where the nota-

tions are the same as the ones in Table V. The experimental results shows that (i) BITY can learn 75% above proper types for most of the programs, while Hex-Rays and Snowman can only recover 65% above proper types; (ii) On the whole, among the types learned by our tool BITY, 566 (43.81%) types are correct, 334 (25.85%) types are compatible, 900 types (69.66%) in total are proper; While Hex-Rays recovers 489 correct ones (37.85%) and 118 compatible ones (09.13%), in total 607 proper ones (46.98%); and Snowman recovers 393 correct ones (30.42%) and 322 compatible ones (24.92%), in total 715 proper ones (55.34%); (iii) On average, the proper accuracy rating of BITY, Hex-Rays and Snowman are 77.52%, 64.34% and 63.49%. To sum up, BITY performs better than Hex-Rays and Snowman, all in terms of correct types, compatible types, or proper types in the benchmarks diffutils and findutils as well.

Concerning *pointer* types, we have found that there are 992 variables that are typed by *pointer* from the reference type information. The results for these variables recovered by our tool BITY, Hex-Rays and Snowman are given in Table IX, where the notations are the same as the ones in Table VIII. Among the types learned by BITY, 301 (30.34%) ones are correct, 332 (33.47%) ones are compatible, and in total 633 (63.81%) ones are proper. While for Hex-Rays, 244 (24.60%) ones are correct, 113 (11.39%) are compatible, and in total 357 (35.99%) ones are proper; and for Snowman, 161 (16.23%) ones are correct, 317 (31.96%) are compatible, and in total 478 (48.19%) ones are proper. The results indicate that BITY also performs better than Hex-Rays and Snowman on diffutils and find utils, in terms of *pointer* types.

Let us consider *struct pointer* types. There are 420 variables that are typed by struct pointers among the variables identified in diffutils and findutils. Table X shows the results for these struct pointers recovered by BITY, Hex-Rays and Snowman. From the results, we can see that BITY can learn *pointer* as the type for 272 (64.76%) variables, among which 110 (26.19%) ones are learned correctly with the type *struct**. While Hex-Rays recovers 68 (16.19%) variables with *pointer*, among which 60 (14.29%) ones are recovered correctly; and Snowman recovers 224 (53.33%) variables with *pointer*, among which 205 (48.81%) ones are recovered correctly. This indicates that, on diffutils and findutils, BITY still performs better than Hex-Rays on struct pointers as well, but a litter worse than Snowman. Through manual analysis, we found that one of the reasons that the accuracy of our struct pointer recovery is low, is that our inter-procedural analysis is relatively weak. For example, many functions just pass the variables of type pointer to other functions or checks the pointer variables, without accessing any fields.

Finally, we compute the (average) distances of each variables in the programs for BITY, Hex-Rays and Snowman. Figure 11 gives the average distances of each program. The results shows BITY can learn types with a shorter distance for most programs than both Hex-Rays and Snowman. That is to say, BITY can learn more precise types than both Hex-Rays and Snowman on diffutils and findutils as well.

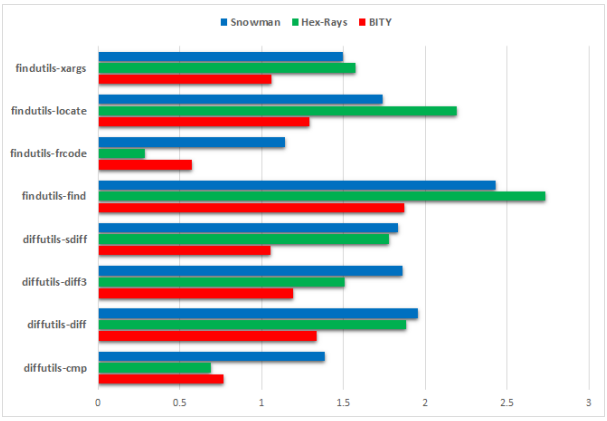


Fig. 12. Distances of BITY, Hex-Rays and Snowman on Diffutiils and Findutils

D. Comparison against EKLAVYA

To evaluate BITY further, we also compare BITY against EKLAVYA [10], a recent tool that can learn types for function parameters from binaries via machine learning. For that, we perform two experiments, wherein findutils and diffutils, collecting from the dataset of EKLAVYA, are still used as the benchmarks. In the first experiment, we respectively train BITY and EKLAVYA on their own dataset, namely, BITY is trained on the dataset presented on Section IV-A (dubbed BITY-Own), while EKLAVYA is trained on the data in EKLAVYA dataset obtained from gcc compiler (dubbed EKLAVAY-Own), except for the benchmark findutils and diffutils. Then we perform BITY and EKLAVYA on the benchmark. While in the second experiment, we train BITY and EKLAVAY on the same dataset, namely coreutils (dubbed BITY-Core and EKLAVAY-Core, respectively), and proceed on as the same as the first one.

In these experiments we focuses on function parameters, since EKLAVAY only considers these variables. And we take in account the functions whose parameter types that can be learnt by both BITY and EKLAVYA. The experimental results are shown in Table XI, where **Variables** denotes the number of the target variables, **RV** denotes the number of variables that are recovered by the tools, **CV** denotes the number of variables that are recovered correctly, **PT** denotes the number of variables, whose types are recovered properly, and **PP** is the percentage of the proper types among all the types. Note that the numbers of variables for some functions are different from the ones in Table VIII, due to that different tools use different front ends or generate different high-level codes.

First, from the results we can see that, among the 1367 variables, BITY (both BITY-Own and BITY-Core) can identify 1364 variables with a total num 1373, while EKLAVAY-Own (EKLAVAY-Core, resp.) predicts 1274 (1240 resp.) variables with a total num 1294 (1298 resp.). That is to say, BITY can identify more correct number of variables than EKLAVYA, and has a higher accuracy than EKLAVAY. This is mainly because that BITY identities parameters via pattern analysis, which enables us to recover almost all the parameters; while EKLAVAY predicts the number of parameters via machine

TABLE VIII
COMPARISON RESULTS OF BITY, HEX-RAYS AND SNOWMAN ON DIFFUTILS AND FINDUTILS

Program	Variable	BITY				Hex-Rays				Snowman			
		R	C	F	P	R	C	F	P	R	C	F	P
diffutils-cmp	13	9	2	2	84.62%	8	4	1	92.31%	6	4	3	76.92%
diffutils-diff	251	120	68	63	74.90%	108	33	110	56.18%	76	74	101	59.76%
diffutils-diff3	108	59	25	24	77.78%	57	14	37	65.74%	41	25	42	61.11%
diffutils-sdiff	54	30	14	10	81.48%	27	4	23	57.41%	21	11	22	59.26%
findutils-find	568	188	157	223	60.74%	151	43	374	34.15%	130	130	308	45.77%
findutils-frcode	7	6	0	1	85.71%	5	2	0	100.00%	5	0	2	71.43%
findutils-locate	204	106	46	52	74.51%	81	17	106	48.04%	74	58	72	64.71%
findutils-xargs	87	48	22	17	80.46%	52	1	34	60.92%	40	20	27	68.97%
Total	1292	566	334	392	-	489	118	685	-	393	322	577	-
Average	-	-	-	-	77.52%	-	-	-	64.34%	-	-	-	63.49%

TABLE IX
RESULTS OF POINTERS ON DIFFUTILS AND FINDUTILS

Num	BITY			Hex-Rays			Snowman		
	R	C	F	R	C	F	R	C	F
992	301	332	359	244	113	635	161	317	514

TABLE X
RESULTS OF STRUCT POINTERS ON DIFFUTILS AND FINDUTILS

Num	BITY		Hex-Rays		Snowman	
	pointer	struct*	pointer	struct*	pointer	struct*
420	272	110	68	33	224	205

learning, so some parameters are still lost, although the accuracy is high.

In the first experiment, we found that EKLAVYA-Own learns a little more proper types than BITY-Own, while BITY-Core performs more better than EKLAVYA-Core in term of proper types in the second experiment. The main reason is that programs in the dataset of EKLAVYA, including diffutils and findutils, are almost common utils such that they share many similar features, which make the type learning perform better. Moreover, the results also show that BITY-Core performs best in term of proper types, with an accuracy 85.74%. This is because BITY learns types for variables from their related instructions, while EKLAVYA predicts types or numbers for functions from all the instructions in the functions or all the call instructions of the functions. In conclusion, we believe that BITY performs better than EKLAVYA, as the features characterised by BITY is more representative.

Concerning *pointer* types, we have found that there are 1070 variables that are typed by *pointer* from the reference type information. As EKLAVYA can only learn the pointer type for a pointer variable without the type information the variable points to, we focus on pointer type itself here. Table XII gives the results for these variables recovered by BITY and EKLAVYA, where the numbers in table denote the numbers of variables whose types are learnt to be pointers. Due to the same reason discussed above, BITY-Own performs a litter worse than EKLAVYA-Own in terms of pointer types in the first experiment, while BITY-Core performs much better than EKLAVYA-Core in the second experiment. So when trained on the same dataset, we believe that BITY also performs better than EKLAVYA in terms of pointer types.

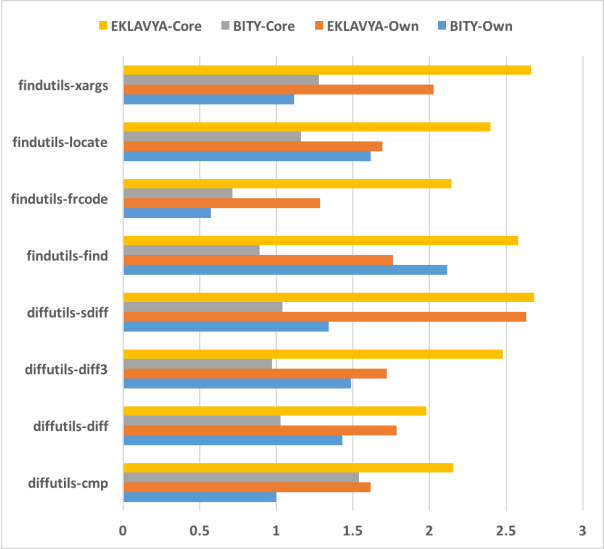


Fig. 13. Distances of BITY and EKLAVYA on Diffutiils and Findutils

Finally, let us consider the distances. Figure 13 shows the average distances of each program for BITY and EKLAVYA. From the results, we can see that BITY can learn types with a shorter distance for most programs than EKLAVYA for both experiments. It is worth noting that BITY-Own learns a little less proper types than EKLAVYA-Own in the first experiment as shown in Table XI. This is due to that BITY can learn types that are more close to the correct ones such that the distances are smaller. The results indicate that BITY can learn more precise types than EKLAVYA on diffutils and findutils.

E. Performance

This section presents the experiments to evaluate the scalability of BITY. For that, we perform BITY on binaries of different sizes. The experimental results are given in Table XIII, where **LOC** is the lines of the assembly code, **Size** is the size of the file in MB, **Var** is the number of target variables in stack, **Time-L** and **Time-P** denotes the type-learning time and the preprocessing time excluding the disassembling time by IDA Pro, respectively. From the results, we can see that (i) the preprocessing time accounts for a great proportion and is linear on LOC and variable numbers; (ii) the predicting time

TABLE XI
COMPARISON RESULTS OF BITY AND EKLAVYA ON DIFFUTILS AND FINDUTILS

Experiment	Program	Variable	BITY				EKLAVYA			
			RV	CV	PT	PP	RV	CV	PT	PP
Exper 1	diffutils-cmp	13	13	13	10	76.92%	13	13	9	69.23%
	diffutils-diff	246	246	246	190	77.24%	245	241	177	72.25%
	diffutils-diff3	103	103	103	81	78.64%	96	95	80	83.33%
	diffutils-sdiff	50	50	50	36	72.00%	46	46	26	56.52%
	findutils-find	655	655	653	366	55.88%	614	605	522	85.02%
	findutils-frcode	7	7	7	6	85.71%	7	7	6	85.71%
	findutils-locate	208	213	207	145	68.08%	195	190	166	85.13%
	findutils-xargs	85	86	85	67	77.91%	78	77	58	74.36%
	Total	1367	1373	1364	901	-	1294	1274	1044	-
	Average	-	-	-	-	74.05%	-	-	-	76.44%
Exper 2	diffutils-cmp	13	13	13	9	69.23%	13	13	7	53.85%
	diffutils-diff	246	246	246	228	92.68%	255	238	170	66.67%
	diffutils-diff3	103	103	103	100	97.09%	103	94	63	61.17%
	diffutils-sdiff	50	50	50	43	86.00%	44	44	27	61.36%
	findutils-find	655	655	653	636	97.10%	605	584	381	62.98%
	findutils-frcode	7	7	7	6	85.71%	7	7	4	57.14%
	findutils-locate	208	213	207	178	83.57%	197	188	129	65.48%
	findutils-xargs	85	86	85	65	75.58%	74	72	46	62.16%
	Total	1367	1373	1364	1265	-	1298	1240	827	-
	Average	-	-	-	-	85.87%	-	-	-	61.35%

TABLE XII
RESULTS OF POINTERS FOR BITY AND EKLAVYA

Num	Exper 1		Exper 2	
	BITY	EKLAVYA	BITY	EKLAVYA
1070	613	888	1060	726

TABLE XIII
RESULTS ON BINARIES OF DIFFERENT SIZES

Program	LOC	Size	Var	Time-P	Time-L
Strcat	508	0.007	8	0.187	0.011
Notepad	12032	2.80	113	0.807	0.229
SmartPPT	128381	4.76	166	1.156	0.365
Doro PDF	25910	16.30	71	0.692	0.068
QuickTime	61240	18.30	247	2.132	0.607
Firefox	12068	110.79	113	0.906	0.254
VMware	39857	282.00	352	3.739	0.911
OpenCV	61636	348.00	287	4.130	0.722
VSX6 pro	129803	1341.44	450	4.762	1.921

does not cost too much and is linear on variable numbers; (ii) BITY learns types in just a few seconds for binaries of sizes ranging from 7KB to 1341.44MB, which indicates that BITY is scalable and viable for practical use.

F. Application on Malware Detection

On the Internet, one of the most serious security threats is malware. Various machine learning algorithms have been used to detect malware recently. Most of them use opcode and system library as features, while few of them consider the data information such as types. In this section, as an immediate application, we conduct experiments to test the ability of the type information that BITY learns to detect malware.

The training dataset is taken from [26] and consists of $11,376$ malware samples and $8,003$ benign samples. On this dataset, we perform 10-*fold cross validation* experiments using various machine learning algorithms listed in Section IV-B, and taking opcode, system library, types and their possible combinations as features separately, wherein we just consider their data sizes for the composite types for simplicity as their encoding into vector is not straightforward. The results are shown in Table XIV, where **O**, **L** and **T** are short for opcode, library and type, respectively, **AUC** is short for the area under the Receiver Operating Characteristic curve (ROC), and the number in the table denotes the average value of classifiers trained by NB, KNN, RF and SVM.

TABLE XIV
RESULTS OF DIFFERENT FEATURES

Feature	Precision	Recall	Accuracy	AUC
O	0.9518	0.8840	0.9339	0.9790
L	0.8711	0.8485	0.8785	0.9397
T	0.8483	0.7701	0.8496	0.9427
O+T	0.9556	0.8883	0.9371	0.9818
L+T	0.9109	0.9256	0.9328	0.9735
O+L	0.9572	0.8996	0.9423	0.9856
O+L+T	0.9582	0.9008	0.9431	0.9869

From the results we can see that type information is also effective to help detect malware with the average precision 84.83%, the average accuracy 84.96%, and the average AUC 0.9427. Compared with the other two features, the performance of the type feature is quite close to system library, although it is a little worse than opcode. We believe that considering composite types fully, especially structs, could obtain a better accuracy. Moreover, we have also found that type information would improve the malware detection which does not take it into account: the average accuracy of classifiers, which consider not only opcode (resp., library and opcode plus library) but also type, is 0.34% (resp., 6.18% and 0.85%) higher than that of classifiers with only opcode (resp., library and opcode plus library).

TABLE XV
RESULTS OF CLASSIFIERS WITH AND WITHOUT TYPE

Sample	With Type	Without Type	Only Type
364	297	295	287

To see whether type information can improve the malware detection in practice, we conduct experiments to compare the performance between the classifier learned with type feature and the one without type feature on the recent malware samples collected from the DAS MALWERK website [27], which are new to the training dataset above. Table XV gives the results, which show that the classifier trained with type feature can detect two more malware samples than the one without. Moreover, the classifier trained with only type feature can detect 287. Although the number is less than the one (295) of the samples detected by the classifier trained without type feature, it can enhance the confidence of the detection.

V. DISCUSSION

In this section, we discuss some limitations of our approach.

A. Program Analysis and Machine Learning

Different from most existing work, our approach employs machine learning, rather than program analysis. So we first discuss limitations of these two approaches in binary type inference.

Table XVI shows the comparisons between program analysis and machine learning in binary type inference. Generally, machine learning approach is easier to start than program analysis approach, since program analysis approach requires much prior knowledge, such as control-flow and typing rules (*i.e.*, the behaviour patterns for types). In particular, there may be too many patterns for a type or too many possible types for some patterns to figure out manually. Another limitation for program analysis approach is the scalability. As mentioned in Section I, "DIVINE [7] spends 2 hours while analyzing programs of the order of $55,000$ assembly instructions" [4]. While for machine learning approach, once a classifier is trained, it can learn the types efficiently.

TABLE XVI
COMPARISON BETWEEN PROGRAM ANALYSIS AND MACHINE LEARNING

	Program Analysis	Machine Learning
Easy to start	hard	easy
Prior Knowledge	needed	not or less needed
Scalability	uncertain	relatively good
Rich samples	not need	require
Explicable	easy to reasoning	hard to reasoning
Accuracy	over-approximate or under-approximate	uncertain

However, one limitation of machine learning approach is that it requires rich samples to train or mine a classifier. More samples always can get a better classifier. Moreover, different samples (*i.e.*, dataset) affects the performance of the classifier. As demonstrated in our experiments in Section IV-D, training BITY on coreutils can get a better classifier with respect to diffutils and findutils than training BITY on our own dataset. Another problem of machine learning approach is explainable. It is hard to reason why the result type is learnt in machine learning approach.

Concerning accuracy, program analysis approach always infers types in over-approximate or under-approximate mode. Thus it may be too conservative or incorrect (see examples in Section II). While machine learning approach uses types as labels of the classifier, and does not take any approximated mode. Generally, if samples are rich enough, the classifier can learn types quite accurately.

In fact, we employ both program analysis and machine learning in our approach: we use program analysis techniques to extract semantic information for variables, which can characterise the behaviour pattern of types and improve the accuracy, and use machine learning technique to train a classifier based on these semantic information, which enables us to learn types efficiently. This enables us to make full use of the advantages of machine learning and program analysis and to strive a balance between accuracy and scalability.

In addition, taking explainable into account, similar to Yang et al.'s work [28], we can learn the typing rules in logic form rather then types, by combing program analysis and machine learning, which is left as a future work.

B. Struct and Struct Pointer

One drawback of BITY is that it cannot recover structs that are in global memory regions and the stack, which is an open challenge in binary type inference [29]. As explained in Section III-D1, a variable of type struct in global memory regions or the stack is always complied into several ones corresponding to its components, acting like it were the definition of these component variables rather then the single struct variable. Therefore, in this paper, we focus on structs in the heap area, that is, struct pointers.

It is worth noting that, although our variable recovery and points-to analysis work well in practice, there are still some limitations on struct pointers, such as the non-accessed fields and nested structs. Moreover, as shown in our experiments in Section IV-C, there are still 62.9% variables of type **struct* that cannot be recovered correctly.

Let us consider the example shown in Figure 14, where struct Point is defined with a nested struct Nest. But after compiling, the nested struct information is lost in the assembly codes. From the assembly codes, BITY can only recover a whole struct with all the three components.

In fact, we can enhance our analysis to recover the tree layout of memory more precisely with a structure analysis, such as Value-Set Analysis (VSA) [14] and DIVINE [7] (a combined analysis consisted of VSA and Aggregate Structure Identification (ASI) [30]). Once the tree layout of memory is constructed, the leaf nodes are identified as target variables. Then we can carry on to extract their features to learn their types with the classifier. At last, composite types like *struct* and *array* are constructed according to the tree layout. This is left as a future work.

```
struct Point {
 int x;
 struct Nest {
  int y;   int z;
 } n;
};
void func() {
 Point* p =  (Point*)
       malloc(sizeof(Point));
 p->x = 2;
 p->n.y = 4;
 p->n.z = 6;
}
```

```
func proc near
 ......
 ......
 mov eax, [ebp-8]
 mov dword ptr [eax], 2
 mov eax, [ebp-8]
 mov dword ptr [eax+4], 4
 mov eax, [ebp-8]
 mov dword ptr [eax+8], 6
 ......
 ......
func endp
```

Fig. 14. Snippet Code for nested struct programs

```
int main() {
 ......
 int a, b;
 double c = (double) a;
 b = a;
 ......
 char d, e;
 int f = (int) d;
 e = d;
 .......
}
```

```
 ......
 cvtsi2sd xmm0, dword ptr a$[ebp]
 movsd qword ptr c$[ebp], xmm0
 mov eax, dword ptr a$[ebp]
 mov dword ptr b$[ebp], eax
 ......
 movsx eax, byte ptr d$[ebp]
 mov dword ptr f$[ebp], eax
 mov  dl, byte ptr d$[ebp]
 mov  byte ptr  e$[ebp], dl
 ......
```

Fig. 15. Snippet Code for type cast programs

C. Type Cast

By now, we do not handle type casts in BITY. Our approach is motivated by "duck types", that is, the type of a variables is determined by its features and properties rather than being explicitly defined. If a variable needs a type cast and succeeds, we believe that it can be operated as well as if it were defined by the casted type.

Nevertheless, let us discuss how to revise our approach for type casts. For that, let us consider the programs shown in Figure 15, which contains two assignments with type casts, namely, the converting from *int* to *double* and the converting from *char* to *int*, and two assignments without type casts, where the assembly codes are obtained via MSVC, and the member block v\$[ebp] in the assembly codes corresponds to the variable v in C codes. As discussed in Section III, we will treat the variables through assignments as a single special variable and merge the behaviours together to learn a type for it. Thus type cast information are lost. For example, the variable a and c will be treated as the same variable. Let us see the assembly codes. We found that the converting from integer to double (the converting from char to int, resp.) is encoded with the opcode *cvtsi2sd* (*movsx* resp.). Moreover, compared to the assignments without type casts, both of which are represented as "mov, mov", the assignments with type casts are compiled into the instruction sequence "cvtsi2sd, movsd" and "movsx, mov", respectively.

This indicates that there are some differences for assignements with type casts. We have also tried different compilers such as MSVC, gcc, g++, clang and clang++ and found that the instructions used for type casts are similar. So for type casts, a solution is to revise our approach: not to merge the variables and their related instructions for such kinds of assignments "cvtXX2XX, movXX" or "movXX, movXX", where X denotes a character.

D. Function Boundaries

In our variable recovery, we assume that function boundaries are recovered correctly, since we recover variables function by function. Let us assume that there are two functions in a program, both of which have a parameter with different types. Without loss of generality, the parameters of both functions are represented as [ebp+8]. If the function boundary cannot be recovered, then we have to treat the whole program as a single function as mentioned in Section III. In that case, all the [ebp+8] would be treated an identity variable, . So there is at least one parameter lost. Moreover, the learnt type is not correct, since all the related instructions, belonging to different types, are collected. Similarly to the case where function boundaries are recovered incorrectly. In a word, the wrong function boundaries can affect our variable recovery and instruction extraction, and thus type learning.

In this paper, we focus on type learning and resort to IDA Pro to solve the function boundaries.

E. Architecture, Compiler and Optimization

One challenge of binary analysis is that binaries are architecture and compiler dependent, plus the compiler optimizations specific to the architecture. Generally, binaries with the similar configurations share similar features, such as instructions. As discussed in Section V-A, BITY is sample dependent, and thus is configuration dependent in some sense. Our dataset are collected from binaries which are complied by gcc and MSVC with none optimization on x86_64 platforms. So we think it would perform better on the binaries with the similar configurations than others. For example, another reason that why BITY-Core performs better than BITY-Own on diffutils and findutils is that both the training dataset and the test dataset used by BITY-Core are collected via gcc, while the training dataset used by BITY-Own is via MSVC and gcc but the test dataset is via gcc.

F. Obscured or Encrypted Binaries

Nowadays, there are many binaries that are obscured or encrypted, and anti-obfuscation and binary-decryption can be regard as independent research topic [31]. This paper focuses on type learning, and we assume that binaries are not obscured or encrypted in this paper.

In our tool BITY, we use IDA Pro as our front end. One can try to handle these obscured or encrypted binaries with IDA Pro, if they can be disassembled, then BITY can be apply on them. Otherwise, one can use any anti-obfuscation or binary-decryption tools to disassemble these binaries, and then pass the assembly codes to BITY.

VI. Related Work

There have been a large body of work on binary type inference. In this section we discuss a number of recent related work. Interested readers can refer to [29] for a more comprehensive survey.

TIE [6] is a static tool to infer primitive types for variables in binaries, which are limited to integer and pointer

types. Moreover, rather than the specific types, its output is the upper bounds or the lower bounds, which may not be accurate enough for binary engineers. Binary type inference in PointerScope [13], a tool to detect the pointer misuses, focuses on the pointer types. Aiming for scalability, SecondWrite [4] combines a best-effort VSA variant for points-to analysis with a unification-based type inference engine, but the accuracy depends on high-quality points-to data. Robbins *et al.* [24] reduce the binary type inference problem into a rational-tree constraint problem, which is then solved through an SMT solver. Yan and McCamant's work [32] proposes a graph-based algorithm to check whether the variables typed by *int* are declared with unsigned or signed. Retypd [2] is a novel static tool for type inference on machine code, which supports subtyping, recursive types, and polymorphism. Hex-Rays [1] is a popular commercial tool for binary code analysis, whose exact algorithm is proprietary. All these tools above resort to static program analysis techniques, which are too heavy-weight for practical use or too conservative to recover types with high accuracy.

Howard [33] and REWARDS [3] adopt a dynamic approach to detect data structures, by generating type constraints from execution traces. ARTISTE [34], another tool to detect data structures dynamically, takes a combination of value invariants, cycle invariants, and points-to relationships to generate hybrid signatures that minimize false positives. MemPick [35], [36] focuses on the high-level data structures such as singly- or doubly-linked lists, graphs, and many types of trees like B-trees and AVL. DSIbin [37] uses a combination of DSI and the type excavator Howard for the inspection of C/C++ binaries to identify dynamic data structures. However, as approaches based on dynamic analysis, these tools cannot achieve full coverage of variables defined in a program.

Some tools concern recovering object-oriented features from C++ binaries [5], [38]–[40]. Most of them adopt program analysis techniques, except for Katz *et al.*'s work [40], which uses object tracelets to capture potential runtime behaviours of objects and ranks the possible types based on object tracelets. Similar to this work, we use the related instruction set to capture potential behaviours of variables, without considering the order, yielding a simpler solution.

Moreover, Raychev *et al.* [41] propose a new approach to predict from "big code" program properties, including types. Their approach leverages program structures to create dependencies and constraints, which are used for probabilistic reasoning. As lots of program structures can be easily discovered at high-level source code, this approach works well. However, less program structures can be recovered for stripped binaries. Recently, Zheng *et al.* [10] present the system EKLAVYA which trains a recurrent neural network to recover function type signatures from disassembled binary code. While our solution recovers types for not only parameters of functions but also local and global variables. Moreover, our solution considers the multi-level pointer such that our types are more expressive than theirs.

VII. Conclusion

Binary type inference is valuable for binary analysis. In this work, we have proposed a new approach to learning the most possible type for a recovered variable. Different from existing work, our approach is based on classifiers, without resorting to program analysis techniques such as constraint solving techniques. To demonstrate the viability of our approach, we have implemented our approach in a prototype tool BITY and carried out some interesting experiments. Our experiments have shown that BITY returns more precise results than the commercial tool Hey-Rays, the open source tool Snowman and a recent tool EKLAVYA using machine learning, and can help detect malware.

As for future work, we will take type quantifiers (*e.g.*, signed) into account. We can enhance our analysis with VSA or DIVINE to recover more structures for composite types. We can implement our approach in some open source tools or as a plus-in. We can also try to learn some typing rules in logic form rather than type to improve interpretability.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308 and 61772347, Science and Technology Foundation of Shenzhen City under Grant No. JCYJ20170302153712968, Project 2016050 supported by SZU R/D Fund and Natural Science Foundation of SZU (Grant No. 827-000200).

References

[1] *The IDA Pro and Hex-Rays*, http://www.hex-rays.com/idapro/.
[2] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *ACM Sigplan Conference on Programming Language Design and Implementation*, 2016, pp. 27–41.
[3] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Network and Distributed System Security Symposium*, 2010.
[4] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *ACM Sigplan Conference on Programming Language Design and Implementation*, 2013, pp. 51–60.
[5] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "Smartdec: Approaching c++ decompilation," in *Reverse Engineering*, 2011, pp. 347–356.
[6] J. H. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Network and Distributed System Security Symposium*, 2011.
[7] G. Balakrishnan and T. Reps, "Divine: discovering variables in executables," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2007, pp. 1–28.
[8] O. Kramer, *Scikit-Learn*. Springer International Publishing, 2016.
[9] *Snowman*, https://derevenets.com/.
[10] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *the 26th USENIX Security Symposium*, 2017.
[11] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *International Conference on Formal Engineering Methods*, 2017, pp. 430–446.
[12] *GNU Diffutils*, https://www.gnu.org/software/diffutils/.
[13] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis," *Proceedings of the Western Pharmacology Society*, vol. 47, no. 47, pp. 46–49, 2013.
[14] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 binary executables," *University of Wisconsin-Madison Department of Computer Sciences*, 2012.

[15] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: learning to recognize functions in binary code," in *Usenix Security Symposium*, 2014.

[16] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Usenix Conference on Security Symposium*, 2015, pp. 611–626.

[17] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 2017, pp. 201–212.

[18] B. C. Pierce, *Types and Programming Languages*. MIT Press,, 2002.

[19] IntelCorporation, *Intel 64 and IA-32 Architectures Software Developer Manuals*, December, 2016.

[20] J. Crnic, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[21] D. Brumley and J. Newsome, "Alias analysis for assembly," School of Computer Science, Carnegie Mellon University, Tech. Rep., 2006, cMU-CS-06-180.

[22] *178 algorithm C language source code.*, http://www.codeforge.com/article/220463.

[23] X. Shiliang, *Commonly Used Algorithm Assembly (C Language Description)*. Tsinghua University Press, 2004, in Chinese.

[24] E. Robbins, J. M. Howe, and A. King, "Theory propagation and rational-trees," in *Symposium on Principles and Practice of Declarative Programming*, 2013, pp. 193–204.

[25] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *IEEE European Symposium on Security and Privacy*, 2017, pp. 177–189.

[26] Z. Xu, C. Wen, S. Qin, and Z. Ming, "An effective malware detection based on behaviour and data features," in *2nd International Conference on Smart Computing and Communication*, 2017, pp. 55–68.

[27] *DAS MALWERK*, http://dasmalwerk.eu/.

[28] F. Yang, Z. Yang, and W. W. Cohen, *Differentiable Learning of Logical Rules for Knowledge Base Reasoning*, 2017.

[29] J. Caballero and Z. Lin, "Type inference on executables," *Acm Computing Surveys*, vol. 48, no. 4, p. 65, 2016.

[30] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, 1999, pp. 119–132.

[31] M. Mateas and N. Montfort, "A box, darkly: Obfuscation, weird languages, and code aesthetics," in *the 6th Digital Arts and Culture Conference*, 2005, pp. 1–22.

[32] Q. Yan and S. McCamant, "Conservative signed/unsigned type inference for binaries using minimum cut," *Technical Report. University of Minnesota*, 2014.

[33] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Network and Distributed System Security Symposium*, 2011.

[34] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Artiste: Automatic generation of hybrid data structure signatures from binary code executions," *Technical Report TRIMDEA-SW-2012-001. IMDEA Software Institute*, 2012.

[35] I. Haller, A. Slowinska, and H. Bos, "Mempick: High-level data structure detection in c/c++ binaries," in *Reverse Engineering*, 2013, pp. 32–41.

[36] ——, "Scalable data structure detection and classification for c/c++ binaries," *Empirical Software Engineering*, vol. 21, no. 3, pp. 778–810, 2016.

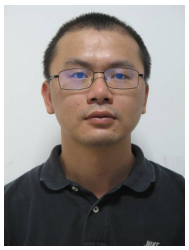
[37] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lttgen, and H. Bos, "Dsibin: Identifying dynamic data structures in c/c++ binaries," in *IEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 331–341.

[38] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering c++ objects from binaries using inter-procedural data-flow analysis," in *ACM Sigplan on Program Protection and Reverse Engineering Workshop*, 2014, p. 1.

[39] K. Yoo and R. Barua, "Recovery of object oriented features from c++ binaries," in *Asia-Pacific Software Engineering Conference*, 2014, pp. 231–238.

[40] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," in *ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, 2016, pp. 313–326.

[41] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *The ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, 2015, pp. 111–124.



Zhiwu Xu Zhiwu Xu received his Ph.D. in Computer Science from University Paris Diderot - Paris 7 and University of Chinese Academy of Sciences under the joint cultivation in 2013. Zhiwu Xu is currently an assistant professor at Shenzhen University. His research is in the area of program analysis and verification, type systems, software security, and machine learning.

Cheng Wen Cheng Wen received his Master's degree in Software Engineering from Shenzhen University in 2018. Cheng Wen is currently a Ph.D candidate in Computer Science in Shenzhen University. His research is in the area of program analysis and verification, binary code type inference and machine learning.

Shengchao Qin Shengchao Qin got his PhD in Applied Mathematics from Peking University and also worked as a Postdoctoral Research Fellow in National University of Singapore under the Singapore-MIT Alliance program, before moving his job to UK. His research interests lie mainly in formal methods, software engineering and programming languages, in particular, formal specification and modelling, program analysis and verification, theories of programming, program logic such as separation logic. To this date he has published over 100 papers in international journals and peer-refereed international conferences.