Practical Type Flattening

Ta Thanh Dinh

tathanhdinh@gmail.com

Abstract. High-level type reconstruction is an important step in machine code decompilation. Researchers show possibilities of mapping untyped machine-dependent primitives into types of a C-like type system. We show that the techniques used for type reconstruction can be bypassed, and we claim that the bypass is always possible given *semantics gap* between type systems of high-level and machine language.

1 Introduction

Binary code type recovery is to restore high-level type information of variables, functions from machine codes. It is useful for reverse code engineering (e.g. decompilation) or static binary rewriting/instrumentation, just to name a few. Type recovery is available in several binary analysis tools [17, 18, 19] as a result of machine code decompilation. Academic research includes the classic paper of Mycroft [6] which is probably the first work on the domain, though earlier ideas [5] have been used to recover types of programs in Scheme - a dynamically typed language. Lee et al. [13] extend Mycroft's work by introducing a form of subtyping based on C's integer conversion rank, but lack of recursive type. Polymorphic Type Inference for Machine Code. Caballero and Lin give a survey [14] for research up until 2015. Recent directions are using machine learning [16] or statistical language model [15], which are out of scope of the paper. All approaches tend to output a C-like type system.

Basically, the binary code type recovery starts by detecting high-level objects from machine-dependent primitives: local variables are stored in function stack frame, parameters are passed from registers, see [9] for related techniques. Type constraints are generated gradually through the *low-level use* of objects. The system of constraints is solved then, each object is assigned a certain (polymorphic) type.

This bird view may give off an impression that the binary code type reconstruction is a well established research domain, incorrect results are because of foreign reasons: incorrect binary code disassembling, high-level variable detection, etc. But we claim that there are probably intrinsic problems.

First, there is a common agreement that type information is removed through compilation [12, 13, 14], but there is no explicit explication of why and how some type information is not needed, so removed, in machine codes. The result of reconstructing *the most precise yet conservative* type [13], which may suggest the well-known *principal types* [1, 3], is sound in a limited context only. Actually not all type information can be restored.

Second, though some techniques of data structure reverse engineering [10, 14] can be reused for type recovery, types do not always attach with storage specific primitives (for example `int` is some `32-bit` signed integer stored in a register or stack). The storage-based point of view is almost correct in low-level languages like C, it does not reveal the nature of types. In other languages, types can be zero-sized [21], another examples are *regions* [8] or *units-of-measure* [11] which leave no storage imprint.

Third, current research uses some variants of *Algorithm W* [2, 4] to resolve type constraints in a C-like language. But the polymorphism in C is rather *ad-hoc* [7]: some functions are trivially typed in C by casting, but not typeable in Hindley-Milner type system. This problem is remarked already in [6], but unfortunately omitted in later extension [13]. In real world, it is not rare to see type inconsistency in decompilation results of security tools.

Contribution We first discuss the type information loss in compilation and some cases where assigning a high-level *type* for a low-level primitive is not possible. Though this part is not novel, we bring it to the context of binary code analysis to show limits of type reconstruction. Next, we present a proof-of-concept C compiler which tries to hide types from reconstruction techniques described in current researches.

2 Type information loss

Informally speaking, a type system is some set of constraints and rules to infer constraints on a program. The type system decides which are allowed or forbidden on the semantics of the language, for example adding two pointers is not allowed in C because this operation is semantically nonsense. Type helps avoid bugs as summarized in Milner's famous slogan "well-type program cannot go wrong" [2].

In statically typed languages, the compiler checks the type constraints then generates machine codes (we consider here only languages which generates machine codes), but then most of constraints are lost because they are not needed for running the program. There are at least two sources of information loss: *type erasure* and *data indistinguishability*, happening at different phases of compilation.

2.1 Type erasure

Type erasure means the "intensional" type of an object is hidden/inaccessible or completely removed in some context. It appears under several concepts: subtype substitution, generic specialization (or monomorphization), or boxing, just name a few. In some cases, e.g. subtype substitution, the (sub)type of the wrapped object does not completely removed, it is just not accessible in contexts where only behaviors of the super type are allowed. Such a case will not be discussed in the paper since we look at the low-level machine code.

Once the program is type checked, the compiler will select a consistent machine-dependent representation for each high-level type, this representation may only have little relation with the original type but it does not violate the fact that the program is checked, then safe.

Example 1. `C++`'s typed enum.

```
enum E { one = 1, two };
void foo(enum E e) {
  switch (e) {
  case E::one:
  case E::two:
    printf("ok\n");
    break;

  default: assert(false);
  }
}

enum E bar() {...return some enum E }

int main() {
  enum E e = bar(); foo(e);
  return 0:
}
```

The program is safe when type-checked: `assert(0)` will never be reached when the program runs. In machine code, we may observe that `foo` is a function accepting a `32-bit` signed integer, i.e. information about `enum E` type is erased. However there is no need to add runtime checking for the case where `foo` is passed an argument of value not in `enum E` (e.g. `3`), this case is eliminated by the compiler's type checker.

2.2 Data indistinguishability

When generating binary code for a specific hardware/operating system, the compiler will use a consistent representation for low-level primitives (e.g. registers used in function parameters or return value, alignment for fields of aggregate types), so that the output binary can be used by other programs on the same system. It it possible that two types are distinguished at high-level but they have the same data representation at low-level.

Example 2. Small struct passing.

```
struct S { int a; int b; };
int foo(struct S s) {
  return s.a + s.b;
}
```

```
int bar(long long l) {
  return (int)l + (int)(l >> 32);
}
```

Under System V AMD64 ABI [20], the aggregate type `S` has class `INTEGER`: the argument `s` is passed just in register `rdi`. At the ABI level, `foo` and `bar` is interchangeable, but they are not at the type level.

2.3 Untypeable

There is no type casting in Hindley-Milner's type system (abbr. HM), so when using a variant of *Algorithm-W* [2] for binary code type inference, the output (as a C program without type casting) may not exist: there are programs which are trivially typed in C, but not in HM.

Example 3. Untypeable function.

```
int foo(int i, int *f) {
  return ((int (*)(int (*)(int, int*), int))f)(foo, i);
}
```

Disassembled code:

```
0x0     55                              push rbp
0x1     48 89 e5                        mov rbp, rsp
0x4     48 83 ec 10                     sub rsp, 0x10
0x8     89 7d fc                        mov [rbp-0x4], edi
0xb     48 89 75 f0                     mov [rbp-0x10], rsi
0xf     48 8b 45 f0                     mov rax, [rbp-0x10]
0x13    8b 75 fc                        mov esi, [rbp-0x4]
0x16    48 bf 00 00 00 00 00 00 00 00   mov rdi, 0x0
0x20    ff d0                           call rax
0x22    48 83 c4 10                     add rsp, 0x10
0x26    5d                              pop rbp
0x27    c3                              ret
```

From disassembled code, it is direct to detect that the second argument of `foo` is of functional type (since `call rax` at address `0x20`), but assigning an functional type for `f` is not possible.

The inevitability of type information loss means that the binary code type reconstruction cannot always reaches the notion of *the most precise yet conservative type* [13]. Since compilation is a “many-to-one mapping” [6], real world decompilers simply look for one of possible maps, sometimes they accept even inconsistencies in reconstructed types.

3 Untyped C

References

[1] J. R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60.

[2] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Science* 17 (1978), pp. 348–375.

[3] L. Damas and R. Milner. "Principal Type-Schemes for Functional Programs". In: *POPL*. 1982.

[4] L. Cardelli. "Basic Polymorphic Typechecking". en. In: *Science of Computer Programming* 8.2 (Apr. 1987), pp. 147–172.

[5] O. Shivers. "Data-Flow Analysis and Type Recovery in Scheme". In: *Topics in Advanced Language Implementation*. MIT, 1990.

[6] A. Mycroft. "Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)". In: *ESOP*. 1999.

[7] C. Strachey. "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13.1-2 (2000), pp. 11–49.

[8] D. Grossman et al. "Region-Based Memory Management in Cyclone". In: *PLDI*. 2002.

[9] G. Balakrishnan. "WYSINWYX: What You See Is Not What You eXecute". PhD thesis. University of Wisconsin–Madison, 2007.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song. "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis". In: *CCS*. 2007.

[11] A. Kennedy. "Types for Units-of-Measure: Theory and Practice". In: *Central European Functional Programming School* (2010), pp. 268–305.

[12] Z. Lin, X. Zhang, and D. Xu. "Automatic Reverse Engineering of Data Structures from Binary Execution". In: *NDSS*. 2010.

[13] J. Lee, T. Avgerinos, and D. Brumley. "TIE: Principled Reverse Engineering of Types in Binary Programs". In: *NDSS*. 2011.

[14] J. Caballero and Z. Lin. "Type Inference on Executables". In: *ACM Computing Surveys* 48.4 (2016).

[15] O. Katz, R. El-Yaniv, and E. Yahav. "Estimating Types in Binaries using Predictive Modeling". In: *POPL* (2016).

[16] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck. "TypeMiner: Recovering Types in Binary Programs Using Machine Learning". In: *DIMVA*. 2019.

[17] *Ghidra*. URL: `https://ghidra-sre.org/`.

[18] *Hex-Rays Decompiler*. URL: `https://www.hex-rays.com/`.

[19] *JEB Decompiler*. URL: `https://www.pnfsoftware.com/`.

[20] H. J. Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*.

[21] *Phantom Type*. URL: `https://doc.rust-lang.org/std/marker/struct.PhantomData.html` (visited on 01/02/2010).