

Practical Type Blurring

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 inference 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 reconstruction is available in several binary analysis tools [13, 14, 15] as a result of machine code decompilation. Academic research includes [5] which is one of the pilot papers on this domain, an improvement supporting a limited form of subtyping is presented in [10], a survey of research up until 2016 is given in [11]. One of recent directions is to use machine learning [12], which is out of scope of the paper. All approaches tend to output a C-like type system.

Basically, the type inference starts by detecting high-level variables from machine-dependent primitives: local variables are stored in function stack frame, parameters are passed from registers, see [7] for example. Type constraints are generated then through the use of these variables. Finally, the constraints are solved, each variable is annotated with a type given by the solver.

Though most papers agree that type information is removed in compilation [9, 10, 11], there is no explicit explication of *why and how* some type information is not needed (so removed) in machine codes. A popular point of view that types should attach with storage specific primitives (e.g. `int` is some 32-bit signed integer stored in a register or stack), though mostly correct, does not reveal the nature of types. This is one of the reasons why some researchers unify data structure reverse engineering with type reconstruction [8, 11]. In some languages other than C, types can be zero-sized [17], another example is *regions* [6] which have no storage imprint.

The binary type reconstruction is influenced by *Algorithm W* used originally to type check/inference for programs of Hindley-Milner's type system (abbr. HM) [1, 2, 4]. While the supposed target is a C-like type system, some authors claim to replicate the result of *principal types* [1, 3] by recovering *the most precise yet conservative type* [10]. Such a result is sound in a limited context

only, C’s type system is not HM: there are functions that are trivially typed in C by casting, but not typeable in HM’s. Actually, it is not rare to observe 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 C compiler which tries to hide types from reconstruction techniques described in current researches.

2 Type information loss

Type system of a language can be considered as a set or rules about constraints on programs of this language. On statically typed languages, the constraints can be mechanically checked and proved without running the program, it helps avoid bugs as Robin Milner’s famous slogan “well-type program cannot go wrong” [2].

There exists at least two sources information loss: *type erasure* and *data indistinguishability* [18], happening at different phases of compilation.

2.1 Type erasure

Once the program is type checked, the type information is normally not needed for evaluation (i.e. running program): the compiler can 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 is 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 x86-64 ABI [16], 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	<code>push rbp</code>
0x1	48 89 e5	<code>mov rbp, rsp</code>
0x4	48 83 ec 10	<code>sub rsp, 0x10</code>

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* [10]. Since compilation is a “many-to-one mapping” [5], real world decompilers simply look for one of possible maps, sometimes they accept even inconsistencies in reconstructed types.

3 Untyped C

3.1 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation etc. does not need an indent, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 1 gives a summary of all heading levels.

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	Lecture Notes	14 point, bold
1st-level heading	1 Introduction	12 point, bold
2nd-level heading	2.1 Printing Area	10 point, bold
3rd-level heading	Run-in Heading in Bold. Text follows	10 point, bold
4th-level heading	<i>Lowest Level Heading.</i> Text follows	10 point, italic

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{1}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 1).

Fig. 1. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [ref_article1], an LNCS chapter [ref_lncs1], a book [ref_book1], proceedings without editors [ref_proc1], and a homepage [ref_url1]. Multiple citations are grouped [ref_article1, ref_lncs1, ref_book1], [ref_article1, ref_book1, ref_proc1, ref_url1].

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”. EN. In: *POPL*. 1982.
- [4] L. Cardelli. “Basic Polymorphic Typechecking”. en. In: *Science of Computer Programming* 8.2 (Apr. 1987), pp. 147–172.
- [5] A. Mycroft. “Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)”. In: *ESOP*. 1999.
- [6] D. Grossman et al. “Region-Based Memory Management in Cyclone”. In: *PLDI*. 2002.
- [7] G. Balakrishnan. “WYSINWYX: What You See Is Not What You eXecute”. PhD thesis. University of Wisconsin–Madison, 2007.
- [8] J. Caballero, H. Yin, Z. Liang, and D. Song. “Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis”. In: *CCS*. 2007.
- [9] Z. Lin, X. Zhang, and D. Xu. “Automatic Reverse Engineering of Data Structures from Binary Execution”. In: *NDSS*. 2010.
- [10] J. Lee, T. Avgerinos, and D. Brumley. “TIE: Principled Reverse Engineering of Types in Binary Programs”. In: *NDSS*. 2011.
- [11] J. Caballero and Z. Lin. “Type Inference on Executables”. In: *ACM Computing Surveys* 48.4 (2016).
- [12] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck. “TypeMiner: Recovering Types in Binary Programs Using Machine Learning”. In: *DIMVA*. 2019.
- [13] Ghidra. URL: <https://ghidra-sre.org/>.
- [14] Hex-Rays Decompiler. URL: <https://www.hex-rays.com/>.
- [15] JEB Decompiler. URL: <https://www.pnfsoftware.com/>.
- [16] H. J. Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*.
- [17] Phantom Type. URL: <https://doc.rust-lang.org/std/marker/struct.PhantomData.html> (visited on 01/02/2010).
- [18] Struct reconstruction in decompilers. URL: <https://reverseengineering.stackexchange.com/questions/22634/struct-reconstruction-in-decompilers>.