Type Analysis of Low-level Code

Robin Eklind
KTH Royal Institute of Technology
Stockholm, Sweden

Keywords type analysis, type recovery, type inference, type constraints, type lattice, variable recovery, decompilation, reverse engineering, low-level code, assembly, LLVM IR, SSA, formal verification, binary analysis

1 Introduction

Even a brief reflection on the magnitude at which we rely on the *correct implementation* of software to provide the most vital infrastructure of society is likely to leave one astounded. With increasingly interconnected and interdependent systems – often in polyglot environments – faults may have far reaching real-world consequences. Implementation correctness may be validated through proactive formal verification and static analysis of source code to prevent entire categories of security vulnerabilities and limit the occurrence of bugs. However, frameworks for validation of source code are often tied to a specific source language or small set of languages (e.g. ADA and SPARK) and therefore have limited effectiveness in polyglot projects. Furthermore, inconsistent interpretation of source code between static analysis tools and compilers may lead to run-time errors even when statically proven never to occur [1, 6]. Finally, as demonstrated in Ken Thompson's 1984 Turing award lecture *Reflections on Trusting Trust*, the output of a compiler may not be trusted even if its input is formally verified, as the compiler environment may have been tampered[1] with:

> *"No amount of source-level verification or scrutiny will protect you from using untrusted code."* [8]

For these reasons, conducting formal verification and semantic analysis on the output rather than the input of a compiler has importance. To facilitate formal verification efforts and enable rich static analysis of binary executables, type analysis of low-level code (e.g. assembly) is essential as type information is lost during the process of compilation and has to be recovered.

Problem definition Type inference on low-level code is the problem of inferring the typing information of source programs from corresponding binary executables. Source programs contain ground truth typing information and binary executables are used as input to type analysis frameworks.

[1]Tempered environments is a real-world issue – not only theoretically – which affect large corporations such as Ericsson and Saab of Sweden on a daily basis.

Aim The aim of this meta-study is to critically assess research related to type analysis of low-level code, provide an overview of fundamental techniques used for type recovery in binary executables, and try to envision the role of and enhancements to type analysis in future applications of binary analysis.

Scope The scope of this study is limited to type analysis and variable recovery, as they are intimately related. While type analysis is heavily dependent on other research topics related to binary analysis – such as binary lifting[2] (i.e. translating machine code to platform-independent intermediate representations), control flow analysis (e.g. required to track floating-point stack access [5]), data flow analysis and pointer analysis – these have intentionally been excluded from the scope.

2 Background

This section provides a brief background on important terminology and concepts related to type analysis and binary analysis.

To help explain different aspects of variable recovery and type analysis, two running examples are used throughout this paper. The first example (see figure 3 of appendix A) details a function `f` which allocates local variables on the function stack frame. The second example (see figure 4 of appendix B) details a function `g` which operates on an array of structures, and a function `h` which operates on a structure pointer. Subsequent in-text references to `f`, `g` and `h` correspond to these three functions, and the `f:12` notation represents a reference to the function `f` at line number 12 in the source listing.

2.1 Static Single Assignment

Static Single Assignment (SSA) form is a property of an intermediate representation (IR) in which each variable is defined before use and assigned exactly once. SSA-form is commonly used in the IR of compilers as it facilitates data flow analysis and simplifies some optimization passes (e.g. dead code elimination).

Within type analysis, SSA-form help distinguish distinct variables stored in the same register by tracking their live ranges. For instance, in the running example (see listing 3b of appendix A) the `eax` register is defined at lines 15, 19, 23 and 25, and refers to different variables of different types. When

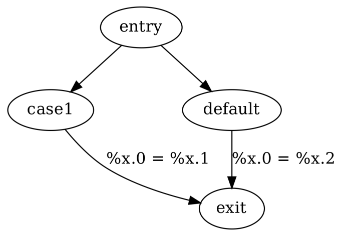
[2]Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode: https://github.com/trailofbits/mcsema

converted to SSA-form each of these assignments would be to a unique *version* of the eax variable; e.g. `%eax.0`, `%eax.1`, `%eax.2`, etc.

Multiple assignments to the same source variable are represented in SSA-form using Φ (*Phi*) instructions (also known as Φ functions). A Φ instruction essentially merges (joins) data flow, such that the resulting value of the Φ instruction depend on the branch taken in the control flow graph (as further illustrated in figure 1c).

```c
int f(int a) {
    int x;
    switch (a) {
    case 5:
        x = a * 2;
    default:
        x = a * 3;
    }
    return x;
}
```

(a) C source code.



(b) Φ-node semantics: value of `%x.0` in `%exit` basic block depend on branch taken.

```llvm
define i32 @f(i32 %a) {
entry:
  switch i32 %a, label %default [
    i32 5, label %case1
  ]
case1:
  %x.1 = mul i32 %a, 2
  br label %exit
default:
  %x.2 = mul i32 %a, 3
  br label %exit
exit:
  %x.0 = phi i32 [ %x.2, %default ], [ %x.1, %case1 ]
  ret i32 %x.0
}
```

(c) LLVM IR in SSA-form: each variable is assigned exactly once.

Figure 1. C source code converted to LLVM IR in SSA-form and associated CFG illustrating Φ-node semantics..

2.2 Type Lattice

A type lattice may be thought of as a set of subtyping relationships, represented as a directed graph from the *top* type ⊤ to the *bottom* type ⊥; where every type is a subtype of ⊤, and no type is a subtype of ⊥.

- ⊤: any type
- ⊥: inconsistent type

In type primitive type lattice of TIE (see figure 2), for instance, both signed and unsigned 32-bit integers (`int32` and `uint32`, respectively) are subtypes of 32-bit integers (`num32`) [5].

In the context of type recovery, a type lattice may specify the set of possible types for the variable through upper

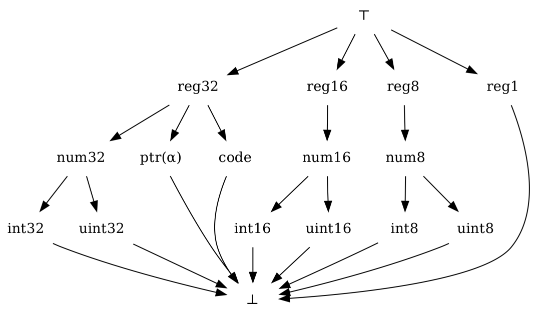


Figure 2. Primitive type lattice of TIE.

and lower bounds; thus imposing type constraints on the variable.

3 Variable Recovery

Variable and typing information is lost during the process of compilation, as local and global variables, function arguments and function parameters are lowered from source code to machine code and mapped onto type-less registers and memory locations. Debug information of binary executables may record this mapping. When debug information is limited or absent however, the source variables have to be recovered from low-level code using variable recovery methods. Since type analysis is based on inference between type typing relations of variables, variable recovery is required for type recovery.

3.1 Value Set Analysis

To recovery variables stored in memory regions (e.g. global, heap, stack), the Value Set Analysis (VSA) algorithm is key; as it determines the possible ranges of values that registers may hold at every point of execution (i.e. registers holding addresses to memory locations). The value ranges (upper and lower bounds, and stride) give a conservative estimate of possible memory locations for variables [1]. While few of the reviewed papers have mentioned it, symbolic execution should be able to improve on this conservative estimate.

4 Type Recovery

The key concepts used in type recovery are *type inference*, *type propagation* and *unification* [7]; and their shared representation is *type constraints* (e.g. upper and lower bounds in a primitive type lattice) associated with the variables determined through variable recovery (see section 3). The underlying theory is based on algorithm W [3].

- Type inference infers type constraints of variables (based on the values they may contain, their use in instructions and as function arguments).
- Type propagation propagates the type constraints between variables which interact (based on data flow analysis).

- Unification unifies the type constraints of the entire program, and may be seen as calculating the set of solutions (which may be empty) to the type constraint equations.

4.1 Type Inference

Value-based type inference The type of a variable may be inferred from the values stored in its register (e.g. 42) or memory location. For instance, the type of the global variable `unk_8000040` (see line 32 in figure 3b) may be inferred from the sequence of bytes stored at its memory location; given that the characters of the byte sequence (`'foo', 0`) are within ASCII range (0-127) and ends with a zero, the source type may be a NULL-terminated string.

One issue with value-based type inference is that it is difficult to distinguish pointer values from integer values or arbitrary sequences of bytes (e.g. `0x414243` could either refer to an address, the integer value `4276803` or the byte sequence `ABC`). Therefore, value-based type inference may lead to misclassification of types [2].

One benefit of value-based type inference is that it only requires examining the contents of registers and memory. Given that no extensive code analysis is required, this approach fits dynamic analysis well.

Instruction type sources The type of a variable may be inferred from the operational semantics (e.g. assignment, comparison, arithmetic) of instructions (also known as *type revealing instruction*). An assignment instruction may reveal the size of a memory location, a comparison instruction the signedness of an integer variable, and an arithmetic instruction the relationship between variables. Further more, the use of FPU registers may reveal the source type of a floating-point variable.

Compared to value-based type inference, instruction type source may extract more information. For instance, given value-based inference a variable assigned the value `321` may be classified as a signed or unsigned integer of bit size at least 16 (since the value `321` does not fit in 8 bits). More concretely, the `mov` instruction at `f:14` (see figure 3b) constrains the local variable at `ebp-4` to either *num16* or *num32* in the primitive type lattice on a 32-bit system (see figure 2). However, there is more information contained within the type revealing `mov` instruction at `f:14`, namely the size of the memory location (`DWORD`) used in the assignment operation. With this added knowledge, the only valid type constraint is `num32`.

One issue with relying on instruction type sources to determine the size of variables in memory, is that optimizing compilers may produce executables containing distinct instructions accessing the same memory location using different data size specifiers. This is common when using `memcpy` and `memmove` on structure types in binaries produced by optimizing compilers; as further illustrated in figure 4b where `g` and `h` refer to the structure type `T` using different size specifiers (i.e. `DWORD` for offset 4 at `h:49` and `BYTE` for offset 4 at `g:21`).

Function type sources Type information derived for arguments in calls to functions with known functions signatures (e.g. those of the standard library) [2].

Type sink A type sink is a type which can be resolved directly and is known to be correct. For instance, the type of an argument in a function call to a standard library function.

Format-string based type inference The types of arguments in function calls to the `printf` family of functions in the standard library may be inferred from the corresponding format string verbs (e.g. `%d` for integer and `%f` for floating-point numbers).

For instance, based on the format string ``` ``%s, %d'' ``` used in the call to `printf` at `f:20` (see figure 3b) the types of the corresponding variables `ebp-8` and `ebp-4` may be inferred from the string verbs `%s` for NULL-terminated string and `%d` for integer number, respectively.

Here, it is important to note that the `eax` register containing the address to the format string at `f:19` is used at two other locations in `f`, each referring to distinct variables. At `f:15`, `eax` refers to the string literal ``` ``foo'' ``` stored in the global memory region, and at `f:25`, `eax` refers to the integer return value. To separate these variables from one another, SSA-form is used to track live ranges and create distinct *versions* of the `eax` register.

5 Evaluation Metrics

To enable objective comparison of different type recovery methods, a set of shared evaluation metrics is required. For this purpose, TIE proposed two evaluation metrics: *distance* and *conservativeness* [5]. Other research projects have proposed metrics to compare class hierarchies.

For benchmark comparison of different type recovery methods, the open source tools of Coreutils[3] are often used. Executables may be compiled to include debug information as a ground truth for source variables and types. The debug information is only used for validation, and no type recovery method assessed in this essay depend on the information to be available.

5.1 Distance

The distance metric measures the distance in height between the recovered type and the source type in the primitive type lattice (see figure 2). The calculation of distance is meaningful only for subtypes (e.g. `int32` is a subtype of `reg32` at distance 2), otherwise the maximum lattice height is used.

[3]Coreutils - GNU core utilities: http://www.gnu.org/software/coreutils/coreutils.html

5.2 Conservativeness

Reports whether the source type exists within the type range (lower and upper bound) of the recovered type.

The distance measurement as defined by TIE did not distinguish between multi-level pointers (e.g. `int*` and `int**` are equivalent, i.e. on the same level in the type lattice). Thus SecondWrite proposed a refinement to measure the ratio between the recovered pointer level and the source pointer level [4].

6 Conclusion

This section concludes the paper, and offers reflections on future applications and future research related to type analysis of low-level code. For the remainder of this section, I will switch to a first person narrative and share my subjective views.

6.1 Future Applications

Personally, I predict that binary analysis and formal verification of binary executables will be of increasing importance in the future as software continues to be more intimately integrated in the most essential infrastructures of our society. Furthermore, based on Ken Thompson's *Reflections on Trusting Trust* and given the occurrence of tampered binary releases of open source repositories[4] these validation efforts must take place not only at source code level, but also at binary executable level.

6.2 Future Research

To handle issues with inconsistent type constraints inferred from instruction type sources (e.g. memory access instructions with different size specifiers), some type recovery methods do not infer size type constraints from `load` and `store` instructions. This limits the information inferred from executables but resolves unification problems.

As future research, it would be interesting to evaluate whether the type inference, type propagation and unification methods of type recovery may be adapted to support *fuzzy* type constraints; and what impact this would have to resolve the aforementioned issues with unification of inconsistent type constraints.

Fuzzy type constraints With terminology adapted from fuzzy logic, a fuzzy type constraint is a type constraint with a certainty of accuracy (0 through 1), as compared to the presence or absence of a type constraint (0 or 1) used in current type recovery methods.

For instance, type constraints inferred from function type sources may be assigned a high certainty of accuracy, while type constraints inferred from instruction type sources (e.g. memory access with size specifiers) may be assigned a low certainty of accuracy. The full detains of how unification of fuzzy type constraints would work is left for future research.

Structure pointers with offsets There is also research to be done related to the correct inference and propagation of pointer type information for pointers into structures at positive and negative offsets. It would be interesting to define an internal type representation of *pointing into a structure type at an offset*, and evaluate whether this representation may resolve the issues of the current type recover approach employed by IDA, as outlined in appendix C.

[4]Giteabot account was compromised: https://github.com/go-gitea/gitea/issues/4167

References

[1] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 23.

[2] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 65.

[3] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 207–212.

[4] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Notices* 48, 6 (2013), 51–60.

[5] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[6] Yannick Moy. 2015. How Our Compiler Learnt From Our Analyzers. (22 May 2015). https://blog.adacore.com/how-our-compiler-learnt-from-our-analyzers

[7] Alan Mycroft. 1999. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming.* Springer, 208–223.

[8] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (1984), 761–763.

Appendices

A Local variable example

```c
#include <stdio.h>

int f(int y) {
   int x = 321;
   char *s = "foo";
   printf("%s, %d", s, x);
   return x+y;
}
```

(a) Example C source code.

```nasm
; --- [ f ] -------------------------------------------------------

; int f(int y)
;
;    ebp-8: s
;    ebp-4: x
;    ebp+0: old ebp
;    ebp+4: return address
;    ebp+8: y
f:
        push    ebp                   ; store old stack frame
        mov     ebp, esp              ; create new stack frame
        sub     esp, 8                ; allocate local variables
        mov     DWORD [ebp-4], 321    ; int x = 321
        lea     eax, [unk_8000040]    ;    "foo"
        mov     DWORD [ebp-8], eax    ; char *s = "foo"
        push    DWORD [ebp-4]         ;    arg3: x
        push    DWORD [ebp-8]         ;    arg2: s
        lea     eax, [unk_8000044]    ;    "%s, %d"
        push    eax                   ;    arg1: format
        call    printf                ; printf("%s, %d", s, x)
        add     esp, 12               ; caller cleanup
        mov     eax, [ebp-4]          ;    x
        mov     ebx, [ebp+8]          ;    y
        add     eax, ebx              ; return x + y
        mov     esp, ebp              ; deallocate local variables
        pop     ebp                   ; restore old stack frame
        ret     4                     ; callee cleanup

[section .data]

unk_8000040: db 'foo', 0
unk_8000044: db '%s, %d', 0
```

(b) Corresponding assembly code in NASM syntax.

Figure 3. Local variable example used to illustrate different aspects of variable recovery and type analysis.

B Structure type example

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

typedef struct {
   int32_t x;
   int8_t y;
   char name[6];
} T;

void g(T ts[], int n) {
   for (int i = 0; i < n; i++) {
      printf(
         "%s (%d, %d)\n",
         ts[i].name,
         ts[i].x,
         ts[i].y
      );
   }
}

void h(T *ts) {
   memset(ts, 0, sizeof(T));
}
```

(a) Struct example C source code.

```nasm
; --- [ g ] -------------------------------------------------------

; void g(T ts[], int n)
;
;     esp+0:  stored ebx
;     esp+4:  stored esi
;     esp+8:  return address
;     esp+12: ts
;     esp+16: n
g:
        push    esi                 ; store esi
        push    ebx                 ; store ebx
        mov     eax, [esp+16]       ; eax = n
        test    eax, eax
        jle     .loc_ret            ; if (n <= 0) { return }
        mov     edx, [esp+12]       ; edx = ts
        lea     eax, [eax+eax*2]    ; eax = n*3
        lea     ebx, [edx+5]        ; edx = ts.name
        lea     esi, [ebx+eax*4]    ; end = ts.name + n*sizeof(T)
  .loc_loop:
        movsx   eax, BYTE [ebx-1]   ; eax = ts->y
        push    eax                 ;    arg4: eax
        push    DWORD [ebx-5]       ;    arg3: ts->x
        push    ebx                 ;    arg2: ts->name
        add     ebx, 12             ; ts++
        push    format              ;    arg1: "%s (%d, %d)\n"
        call    printf              ; printf(
                                    ;    "%s (%d, %d)\n",
                                    ;    ts->name,
                                    ;    ts->x,
                                    ;    ts->y
                                    ; )
        add     esp, 16             ; caller cleanup
        cmp     esi, ebx
        jnz     .loc_loop           ; if (ts >= end) { break }
  .loc_ret:
        pop     ebx                 ; restore ebx
        pop     esi                 ; restore esi
        ret

; --- [ h ] -------------------------------------------------------

; void h(T *ts)
;
;     esp+4: ts
h:
        mov     eax, [esp+4]
        mov     DWORD [eax], 0
        mov     DWORD [eax+4], 0
        mov     DWORD [eax+8], 0 ; memset(ts, 0, sizeof(T))
        ret
```

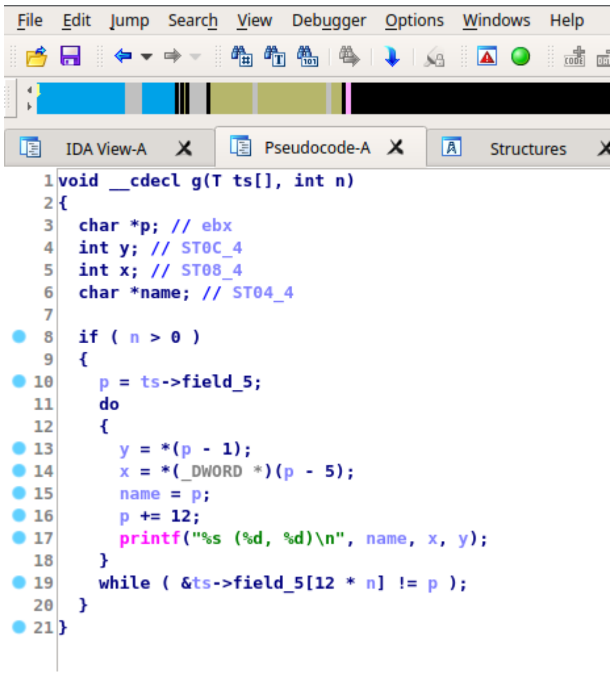
(b) Corresponding assembly code in NASM syntax.

Figure 4. Struct example used to illustrate type recover of structure types.

C Structure pointers with offsets

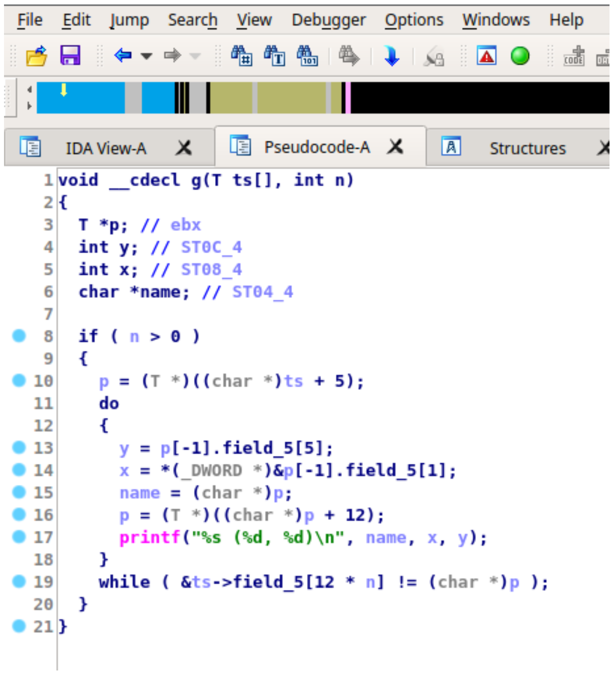
When decompiling the assembly of the structure type example (see figure 4b) using the state-of-the-art decompiler IDA Hex-Rays (version 7.1.180227), an issue in the type recovery of pointers to structure types has been identified; where IDA lacks a type representation of *pointing into a structure type at an offset.*

The type of p in figure 5 should be *pointer to T at offset 5*, but C lacks a representation for such a type, and IDA seem to lose this information in its internal representation; thus resulting in suboptimal type recovery, as the structure field accesses p->y, p->x and p->name are not recovered at line 13, 14 and 15, respectively.



```
 1 void __cdecl g(T ts[], int n)
 2 {
 3   char *p; // ebx
 4   int y; // ST0C_4
 5   int x; // ST08_4
 6   char *name; // ST04_4
 7
 8   if ( n > 0 )
 9   {
10     p = ts->field_5;
11     do
12     {
13       y = *(p - 1);
14       x = *(_DWORD *)(p - 5);
15       name = p;
16       p += 12;
17       printf("%s (%d, %d)\n", name, x, y);
18     }
19     while ( &ts->field_5[12 * n] != p );
20   }
21 }
```

(a) Pointer to char; correct representation but loses information about T.



```
 1 void __cdecl g(T ts[], int n)
 2 {
 3   T *p; // ebx
 4   int y; // ST0C_4
 5   int x; // ST08_4
 6   char *name; // ST04_4
 7
 8   if ( n > 0 )
 9   {
10     p = (T *)((char *)ts + 5);
11     do
12     {
13       y = p[-1].field_5[5];
14       x = *(_DWORD *)&p[-1].field_5[1];
15       name = (char *)p;
16       p = (T *)((char *)p + 12);
17       printf("%s (%d, %d)\n", name, x, y);
18     }
19     while ( &ts->field_5[12 * n] != (char *)p );
20   }
21 }
```

(b) Pointer to T; incorrect representation.

Figure 5. IDA fails to recover (or has no valid internal representation of) the pointer type of p, which should be *pointer to T at offset 5.*