TIE: Principled Reverse Engineering of Types in Binary Programs

JongHyup Lee, Thanassis Avgerinos, and David Brumley
Carnegie Mellon University
{jonglee, thanassis, dbrumley}@cmu.edu

Abstract

A recurring problem in security is reverse engineering binary code to recover high-level language data abstractions and types. High-level programming languages have data abstractions such as buffers, structures, and local variables that all help programmers and program analyses reason about programs in a scalable manner. During compilation, these abstractions are removed as code is translated down to operations on registers and one globally addressed memory region. Reverse engineering consists of "undoing" the compilation to recover high-level information so that programmers, security professionals, and analyses can all more easily reason about the binary code.

In this paper we develop novel techniques for reverse engineering data type abstractions from binary programs. At the heart of our approach is a novel type reconstruction system based upon binary code analysis. Our techniques and system can be applied as part of both static or dynamic analysis, thus are extensible to a large number of security settings. Our results on 87 programs show that TIE is both more accurate and more precise at recovering high-level types than existing mechanisms.

1 Introduction

Reverse engineering binary programs to recover high-level program data abstractions is a recurring step in many security applications and settings. For example, fuzzing, COTS software understanding, and binary program analysis all benefit from the ability to recover abstractions such as buffers, structures, unions, pointers, and local variables, as well as their types. Reverse engineering is necessary because these abstractions are removed, and potentially completely obliterated, as code is translated down to operations on registers and one globally addressed memory region.

Reverse engineering data abstractions involves two tasks. The first task is *variable recovery*, which identifies high-level variables from the low-level code. For example, consider reverse engineering the binary code shown in Figure 1(b) (where source operands come first), compiled from the C code in Figure 1(a). In the first step, variable recovery should infer that (at least) two parameters are passed and that the function has one local variable. We recover the information in the typical way by looking at typical access patterns, e.g., there are two parameters because parameters are accessed via `ebp` offsets and there are two unique such offsets (0xc and 0x8).

The *type recovery* task, which gives a high-level type to each variable, is more challenging. Type recovery is challenging because high-level types are typically thrown away by the compiler early on in the compilation process. Within the compiled code itself we have byte-addressable memory and registers. For example, if a variable is put into `eax`, it is easy to conclude that it is of a type compatible with 32-bit register, but difficult to infer high-level types such as signed integers, pointers, unions, and structures.

Current solutions to type recovery take either a dynamic approach, which results in poor program coverage, or use unprincipled heuristics, which often given incorrect results. Current static-based tools typically employ some knowledge about well-known function prototypes to infer parameters, and then use proprietary heuristics that seem to guess the type of remaining variables such as locals. For example, staple security tools such as the IDA-Pro disassembler [2] use proprietary heuristics that are often widely inaccurate, e.g., Figure 1(e) shows that Hex-rays infers both the `unsigned int` and `unsigned int *` as `int`. For example, the Hex-rays default action seems to be to report an identified variable as a signed integer.

The research community has developed more principled algorithms such as the REWARDS system [12], but has limited their focus to a single path executed using dynamic analysis. The focus on dynamic analysis is due to the perceived difficulty of general type inference over programs with control flow [12]. In this line of work types are inferred by propagating information from executed "type sinks", which are calls to functions with known type signatures. For example, if a program calls `strlen` with argument a, we

```
unsigned int foo(char *buf,
         unsigned int *out)
{
  unsigned int c;
  c = 0;
  if (buf) {
    *out = strlen(buf);
  }
  if (*out) {
    c = *out - 1;
  }
  return c;
}
```

(a) Source code

```
push   %ebp
mov    %esp,%ebp
sub    $0x28,%esp
movl   $0x0,-0xc(%ebp)
cmpl   $0x0,0x8(%ebp)
je     8048442 <foo+0x23>
mov    0x8(%ebp),%eax
mov    %eax,(%esp)
call   804831c <strlen@plt>
mov    0xc(%ebp),%edx
mov    %eax,(%edx)
mov    0xc(%ebp),%eax
mov    (%eax),%eax
test   %eax,%eax
je     8048456 <foo+0x37>
mov    0xc(%ebp),%eax
mov    (%eax),%eax
sub    $0x1,%eax
mov    %eax,-0xc(%ebp)
mov    -0xc(%ebp),%eax
leave
ret
```

(b) Disassembled code

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
movl   $0x0,-0x4(%ebp)
cmpl   $0x0,0x8(%ebp)
je     0x0000000008048402
mov    0x8(%ebp),%eax
mov    %eax,(%esp)
call   0x00000000080482d8
mov    %eax,%edx
mov    0xc(%ebp),%eax
mov    %edx,(%eax)
mov    0xc(%ebp),%eax
mov    (%eax),%eax
test   %eax,%eax
je     0x0000000008048416
mov    0xc(%ebp),%eax
mov    (%eax),%eax
sub    $0x1,%eax
mov    %eax,-0x4(%ebp)
mov    -0x4(%ebp),%eax
leave
ret
```

(c) Trace1 (buf = “test”, *out = 1)

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
movl   $0x0,-0x4(%ebp)
cmpl   $0x0,0x8(%ebp)
je     0x0000000008048402
mov    0xc(%ebp),%eax
mov    (%eax),%eax
test   %eax,%eax
je     0x0000000008048416
mov    -0x4(%ebp),%eax
leave
ret
```

(d) Trace2 (buf = Null, *out = 0)

Variable	Hex-Rays	REWARDS(Trace1)	REWARDS(Trace2)	TIE
`buf` (char*)	char *	char *	32-bit data	char *
`out` (unsigned int*)	int	unsigned int *	pointer	unsigned int *
`c` (unsigned int)	int	unsigned int	32-bit data	unsigned int

(e) Inferred types

Figure 1. Example of binary programs and inferred types.

can infer that a has type `const char *` from `strlen`'s type signature. If the program then executes $a = b$, we can infer b has the same type.

Unfortunately, dynamic analysis systems such as REWARDS are fundamentally limited because they cannot handle control flow. As a result, these approaches cannot be generalized to static analysis, e.g., as commonly encountered in practice. Further, these approaches cannot be generalized over multiple dynamic runs since that would require control flow analysis, which by definition is a static analysis. For example, Figure 1(c,d,e) shows the output of REWARDS on two inputs, which results in two different and incompatible sets of results which dynamic systems alone cannot resolve.

In this paper, we propose a principled inference-based approach to data abstraction reverse engineering. The goal of our approach is to reverse engineering as much as we can infer from the binary code, but never more by simply guessing. Our techniques handle control flow, thus can be applied in both static and dynamic analysis settings. We implement our techniques in a system called TIE (Type Inference on Executables).

The core of TIE is a novel type reconstruction approach based upon binary code analysis for typing recovered variables. At a high level, type reconstruction (sometimes called type inference) uses hints provided by how code is used to infer what type it must be. For example, if the signed flag is checked after an arithmetic operation, we can infer both operands are signed integers. Type reconstruction builds a set of formulas based upon these hints. The formulas are solved to infer a specific type for variables that is consistent with the way the code is actually used. Our implementation can perform both intra- and inter-procedural analysis. Figure 1 shows TIE's approach correctly infering the types of the running example.

We evaluate TIE against two state-of-the-art competing approaches: the Hex-rays decompiler [2] and the REWARDS [12] system. We propose two metrics for reverse engineering algorithms: how *conservative* they are at giving a type the correct term, and how *precise* they are in that we want terms to be typed with as specific a type as possible. We show TIE is significantly more conservative and precise than previous approaches on a test suite of 87 programs.

Contributions. Specifically, our contributions are:

- A novel type inference system for reverse engineering high-level types given only the low-level code. The process of type inference is well-defined and rooted in type reconstruction theory. In addition, our type-inference approach is based upon how the binary code is actually used, which leads to a more conservative type (the inferred type is less often completely incorrect) and more precise (the inferred type is specific to the original source code type).
- An end-to-end system approach that takes in binary code and outputs C types. All our techniques handle control flow, thus can be applied in both the static and dynamic setting unlike previously demonstrated work.
- We evaluate our approach on 87 programs from `coreutils`. We evaluate our approach against REWARDS [12] and the Hex-rays decompiler. We show that TIE is more conservative and up to 45% more precise than existing approaches. We note that pre-

vious work has considered a type-inference approach impractical [12]; our results challenge that notion.

2 Background

In this section we review background material in subtyping, typing judgements, and lattice theory used by TIE. A more extensive explanation of subtyping can be found in programming language textbooks such as Pierce [17].

Inference Rules. We specify typing rules as inference rules of the form:

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C}$$

The top of the inference rule bar is the premises P_1, P_2, etc. If all premises on top of the bar are satisfied, then we can conclude the statements below the bar C. If there are no premises, then the rule is axiomatic. Inference rules not only provide a formal compact notation for single-step inference, but also implicitly specify an inference algorithm by recursively applying rules on premises until an axiom is reached.

Typing. Every term t, whether it be a variable, value, or expression, has a type T. The types of terms are specified via inference rules, where the type of the term is the conclusion given that the sub-terms type as specified by the premise. In order to make sure variables are typed consistently, we also include a *context* that maps variables to their types in rules, denoted as Γ by tradition. The type of a term is denoted $\Gamma \vdash t : T$, which can be read as "t has type T under context Γ".

For example, when a variable x is declared as an `int`, Γ would be updated to include a new binding x : `int`. Later on if we want to find the type of x, we simply need to look it up in Γ, denoted as:

$$\frac{x : \text{int} \in \Gamma}{\Gamma \vdash x : \text{int}}$$

The typing of expressions is performed by recursively typing each sub-expression. For example, the type of the expression `x+y` would be inferred as `int` when x : `int` $\in \Gamma$ and y : `int` $\in \Gamma$. In C, we would infer the same expression has type `float` when x, y : `float` $\in \Gamma$ since the plus ("+") function accepts both floats and ints as arguments.

Subtyping. A type T_1 is a *subtype* of T_2, written as $T_1 <: T_2$, iff any term of type T_1 can be safely used in a context where a term of type T_2 is expected. The subtype relation is reflexive (any type is a subtype of itself) and transitive (if $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$). A subtype relation $T_1 <: T_2$ may also be written as $T_2 :> T_1$; the two representations are interchangeable.

Subtyping and typing judgements are bridged by the *subsumption* rule:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

This rule says that with the current typing of variables in Γ, term t has type T when i) we can infer that term t has type S, and ii) type S is a subtype of type T.

Subtyping is extended to records, function arguments, and so on in the obvious way. For example, the rule for subtyping record fields (i.e., C structures) where each of the n fields is labeled l_i is:

$$\frac{\text{for each } i \; S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

This rule says something very simple: if the record field l_i has type S_i, and $S_i <: T_i$, then any time we want we can also conclude that l_i has type T_i. In addition, subtyping is applied to function types, $T_1 \to T_2$, where T_1 is the argument type and T_2 is the result type. The rule for subtyping function types is:

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

When two function types are in a subtype relationship, the result type is covariant (i.e, $S_2 <: T_2$) while the argument type is contravariant (i.e, $T_1 <: S_1$).

Lattices. A *lattice* is a partial order among the values in a domain in which any two elements have an upper and lower bound. If the lattice is bounded, the elements of the lowest and the highest order are $\top$ and $\bot$, respectively. Lattices define two operations: the least upper bound, denoted by the "join" operator $\sqcup$, and the greatest lower bound, denoted by the "meet" operator $\sqcap$.

3 TIE Overview

TIE is an end-to-end system for data abstraction reverse engineering. The overall flow and components in TIE are shown in Figure 2. In this section we describe the overall work-flow of TIE, and then give an example of the steps on our running example.

Lifting to BIL. TIE begins with the binary code we wish to reverse engineer. TIE uses our binary analysis platform, called BAP, to lift the binary code to a binary analysis language called BIL. The BIL code provides low-level typing for all registers and memory cells, e.g., a value loaded into `eax` has type `reg32_t` since `eax` is a 32-bit long register. BAP considers two possible analysis scenarios: a dynamic analysis scenario and a static analysis scenario. In the static analysis scenario, we disassemble the binary and identify functions using existing heuristics, e.g., [11]. In the dynamic analysis scenario, we run the program within a dynamic analysis infrastructure and output the list of instructions as they are executed. In both cases, the output is an assembly program: for static we have the program,

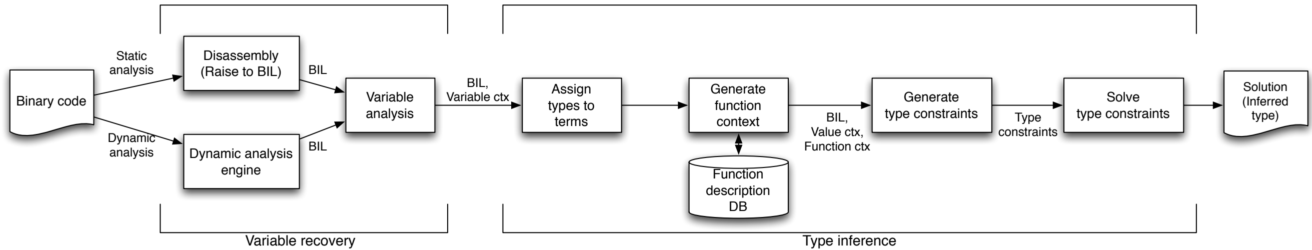


Figure 2. TIE approach for type inference in binary code

and for dynamic analysis we have the single path actually executed, which is then lifted to BIL via a syntax-directed translation. Subsequent analysis is performed on BIL while maintaining a mapping to the original assembly so that we can report final results in terms of the original assembly.

Variable Recovery. The variable recovery phase takes the BIL code produced by the pre-processor as input. The variable recovery step runs our DVSA algorithm to infer high-level variable locations. DVSA infers variables by analyzing access patterns in memory. For example, an access to `ebp+0xc` is a passed parameter since `ebp` is the base parameter and positive offsets are used for parameters. Our algorithm for variable site recovery builds conceptually upon value set analysis (VSA) which determines the possible range of values that may be held in a register [5]. We also return a VSA context which is used during the final inference phase to determine aliasing and reuse stack slots (§ 6.3.5).

Type Reconstruction. The variables recovered by DVSA are passed to our type reconstruction algorithm, along with the BIL code. Type reconstruction consists of three steps:

Step 1: Assign Type Variables. TIE assigns each variable output from DVSA and all program terms a type variable τ_t, representing an unknown type for term t.

Step 2: Constraint Generation. TIE generates constraints on the type variables based upon how the variable is used, e.g., if a variable is used as part of signed division $/_s$, we add the constraint that τ_t must be a signed type. One of our main contributions is that the constraint system can be solved and lead to accurate and conservative results.

Step 3: Constraint Solving. TIE solves the constraints on each type variable τ to find the most precise yet conservative type. Conservative means we do not infer types that cannot be supported by the code, e.g., an unreferenced variable loaded into `eax` but never used will have type `reg32_t` since that is the most informative type possible to infer from the code. Precision is a metric for how close our inferred type is to the original type, e.g., if the variable was originally a C `int`, the most precise type we could infer would be an `int`, a slightly less precise type would be "the variable is a 32-bit number", and the least precise is we could infer nothing at all.

The output of type reconstruction is a type in our type system for each recovered variable. Our type system makes heavy use of sub-typing to model the polymorphism in assembly instructions. For example, the `add` instruction can be used to add two numbers, but also to add a number and a pointer. We use sub-typing to bound what we infer, e.g., for `add` the arguments are either two numbers (either signed or unsigned) or a pointer and a number.

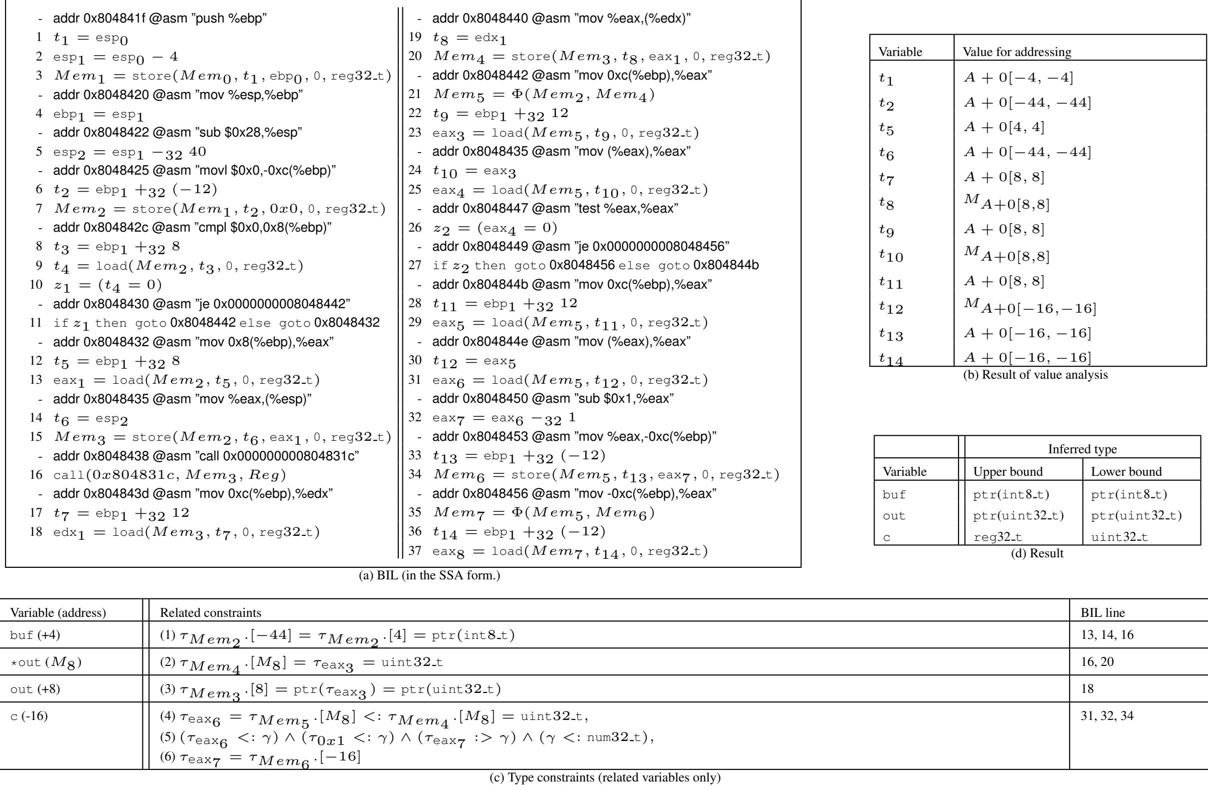
The types we infer are within the TIE type system. In order to output C types we translate TIE types into C. The benefits of this design are that TIE can be retargetted to output types for other similar languages by only retargetting the translator component, and that we are not restricted to C's informal and sometimes wacky type system during type inference itself.

3.1 Example

Figure 3 shows the TIE analysis steps applied to the running example from Figure 1. The function `foo` in Figure 3 has two arguments and one local variables. We perform static analysis in the example. Figure 3 (a) shows the BIL raised from the binary for `foo` (BIL is explained in §4.2). The bold texts consists of annotations indicating the assembly addresses and instructions.

The next step is our DVSA to recover local variables. Figure 3 (b) shows the result of the analysis. The output is a list of identified variables along with the location of each variable in memory (expressed as an `SI` range). Two variables that reference the same memory location (i.e., have the same `SI`) are identical if they always operate on the same SSA memory instance, else we consider them possible places for stack slot reuse of two different variables.

Type inference takes the variables and first assigns a type variable to every BIL program term. We denote by τ_v the type variable for a variable v. Note that the type variable for memory is a record, thus $\tau_{Mem_1}.[a]$ represents the type variable at address a in memory Mem_1. We then analyze



```
 -  addr 0x804841f @asm "push %ebp"
 1  t1 = esp0
 2  esp1 = esp0 - 4
 3  Mem1 = store(Mem0, t1, ebp0, 0, reg32_t)
 -  addr 0x8048420 @asm "mov %esp,%ebp"
 4  ebp1 = esp1
 -  addr 0x8048422 @asm "sub $0x28,%esp"
 5  esp2 = esp1 -32 40
 -  addr 0x8048425 @asm "movl $0x0,-0xc(%ebp)"
 6  t2 = ebp1 +32 (-12)
 7  Mem2 = store(Mem1, t2, 0x0, 0, reg32_t)
 -  addr 0x804842c @asm "cmpl $0x0,0x8(%ebp)"
 8  t3 = ebp1 +32 8
 9  t4 = load(Mem2, t3, 0, reg32_t)
10  z1 = (t4 = 0)
 -  addr 0x8048430 @asm "je 0x0000000008048442"
11  if z1 then goto 0x8048442 else goto 0x8048432
 -  addr 0x8048432 @asm "mov 0x8(%ebp),%eax"
12  t5 = ebp1 +32 8
13  eax1 = load(Mem2, t5, 0, reg32_t)
 -  addr 0x8048435 @asm "mov %eax,(%esp)"
14  t6 = esp2
15  Mem3 = store(Mem2, t6, eax1, 0, reg32_t)
 -  addr 0x8048438 @asm "call 0x000000000804831c"
16  call(0x804831c, Mem3, Reg)
 -  addr 0x804843d @asm "mov 0xc(%ebp),%edx"
17  t7 = ebp1 +32 12
18  edx1 = load(Mem3, t7, 0, reg32_t)
 -  addr 0x8048440 @asm "mov %eax,(%edx)"
19  t8 = edx1
20  Mem4 = store(Mem3, t8, eax1, 0, reg32_t)
 -  addr 0x8048442 @asm "mov 0xc(%ebp),%eax"
21  Mem5 = Φ(Mem2, Mem4)
22  t9 = ebp1 +32 12
23  eax3 = load(Mem5, t9, 0, reg32_t)
 -  addr 0x8048435 @asm "mov (%eax),%eax"
24  t10 = eax3
25  eax4 = load(Mem5, t10, 0, reg32_t)
 -  addr 0x8048447 @asm "test %eax,%eax"
26  z2 = (eax4 = 0)
 -  addr 0x8048449 @asm "je 0x0000000008048456"
27  if z2 then goto 0x8048456 else goto 0x804844b
 -  addr 0x804844b @asm "mov 0xc(%ebp),%eax"
28  t11 = ebp1 +32 12
29  eax5 = load(Mem5, t11, 0, reg32_t)
 -  addr 0x804844e @asm "mov (%eax),%eax"
30  t12 = eax5
31  eax6 = load(Mem5, t12, 0, reg32_t)
 -  addr 0x8048450 @asm "sub $0x1,%eax"
32  eax7 = eax6 -32 1
 -  addr 0x8048453 @asm "mov %eax,-0xc(%ebp)"
33  t13 = ebp1 +32 (-12)
34  Mem6 = store(Mem5, t13, eax7, 0, reg32_t)
 -  addr 0x8048456 @asm "mov -0xc(%ebp),%eax"
35  Mem7 = Φ(Mem5, Mem6)
36  t14 = ebp1 +32 (-12)
37  eax8 = load(Mem7, t14, 0, reg32_t)
```

(a) BIL (in the SSA form.)

Variable	Value for addressing
t_1	$A + 0[-4, -4]$
t_2	$A + 0[-44, -44]$
t_5	$A + 0[4, 4]$
t_6	$A + 0[-44, -44]$
t_7	$A + 0[8, 8]$
t_8	$M_{A+0[8,8]}$
t_9	$A + 0[8, 8]$
t_{10}	$M_{A+0[8,8]}$
t_{11}	$A + 0[8, 8]$
t_{12}	$M_{A+0[-16,-16]}$
t_{13}	$A + 0[-16, -16]$
t_{14}	$A + 0[-16, -16]$

(b) Result of value analysis

	Inferred type	
Variable	Upper bound	Lower bound
buf	ptr(int8_t)	ptr(int8_t)
out	ptr(uint32_t)	ptr(uint32_t)
c	reg32_t	uint32_t

(d) Result

Variable (address)	Related constraints	BIL line
buf (+4)	(1) $\tau_{Mem_2}.[-44] = \tau_{Mem_2}.[4] =$ ptr(int8_t)	13, 14, 16
$\star$out (M_8)	(2) $\tau_{Mem_4}.[M_8] = \tau_{\text{eax}_3} =$ uint32_t	16, 20
out (+8)	(3) $\tau_{Mem_3}.[8] =$ ptr(τ_{eax_3}) = ptr(uint32_t)	18
c (-16)	(4) $\tau_{\text{eax}_6} = \tau_{Mem_5}.[M_8] <: \tau_{Mem_4}.[M_8] =$ uint32_t, (5) $(\tau_{\text{eax}_6} <: \gamma) \wedge (\tau_{0x1} <: \gamma) \wedge (\tau_{\text{eax}_7} :> \gamma) \wedge (\gamma <: \text{num32_t})$, (6) $\tau_{\text{eax}_7} = \tau_{Mem_6}.[-16]$	31, 32, 34

(c) Type constraints (related variables only)

Figure 3. An example of TIE

the function and find there is a call to a well known function strlen. The function description that strlen has an argument of type pointer to char (ptr(int8_t) in TIE) and a return value of type unsigned integer (uint32_t in TIE) is stored in the function context.

TIE then generates constraints for the statements. The constraints are built upon how the variables are used, e.g., if a variable is used in signed division it must be signed. Table Figure 3 (c) shows a simplified excerpt of the constraints related to variables only (Detailed constraints generation rules are explained in §6.2.). According to the constraints, buf is of the same type as the variable at -44, which is used as an argument in the call to strlen. Thus, from the function description of strlen, we infer the variable at 4 is of ptr(int8_t), which is equivalent to char * in C. Likewise, $\star$out is inferred as uint32_t. However, the address $\star$out is unknown because it is passed through the pointer out, thus its address is M_8, which means it is contents from the address 8 (second argument). If a variable is used as a pointer, we also provide the constraint to show the pointer-value relation. Line 18 shows out is the pointer of $\star$out, ptr(uint32_t). To infer c, TIE uses the merged information from multiple paths. In the two branches of the first if of foo, only the true branch gives a hint for inferring c, where the type of $\star$out is revealed as uint32_t. Since c is the result of a subtraction with $\star$out, TIE generates the constraint (5). Through the transitivity of the subtype relation, $\tau_{\text{eax}_6} <: \gamma <: \tau_{\text{eax}_7}$, TIE infers the type of c. Figure 3 (d) shows the inferred types.

4 Lifting Binary Code

4.1 Binary Analysis Platform (BAP)

The input provided to TIE is a binary to either perform static or dynamic analysis. We have developed a single binary analysis platform, called BAP, to raise binary code in either case to a single analysis language. Our design is motivated by the observation that the main difference between the two is that in the dynamic analysis scenario we only analyze the straight-line assembly code produced by a single execution, while in the static analysis case we analyze the assembly code produced by a disassembler. This observation prompted us to design a single back-end for faithful analysis of assembly code fed by static and dynamic-specific front-ends.

In the static analysis scenario, BAP disassembles the binary to produce a stream of assembly instructions. BAP currently implements a linear sweep disassembly. However, other disassembles are possible, e.g., Kruegel *et al.*

program	::=	(*label stmt**)* (i.e., functions)
stmt	::=	*var* := *exp* | `goto` *exp* | `if` *exp* `then goto` *exp* `else goto` *exp* | `return`
		| `halt` *exp* | `assert` *exp* | `label` *label_kind* | `call` *exp* `with` *argument* `ret` *var* | `special` string
exp	::=	*exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | `lab` string | *integer* | `load`(*exp*, *exp*, *exp*, τ_{reg}) | `store`(*exp*, *exp*, *exp*,*exp*,τ_{reg})
		| `cast`(*cast_kind*,τ_{reg},*exp*) | Φ(*var**)
label_kind	::=	*integer* | string
cast_kind	::=	`unsigned` | `signed` | `high` | `low`
var	::=	(string, id_v, τ)
$\Diamond_b$	::=	$+, -, *, /, /_s, \ldots$
$\Diamond_u$	::=	$-$ (unary minus), $\sim$ (bit-wise not)
memory	::=	{ *integer* $\rightarrow$ *integer*, *integer* $\rightarrow$ *integer*, ... } (:τ_{mem})
argument	::=	*(var)*$^+$
τ	::=	τ_{reg} | τ_{mem}
τ_{mem}	::=	`mem_t`(τ_{reg})
τ_{reg}	::=	`reg1_t` | `reg8_t` | `reg16_t` | `reg32_t` | `reg64_t`

Figure 4. The Binary Intermediate Language. Note commas separate operators.

have shown that even obfuscated programs can accurately be disassembled [11]. BAP raises the stream of instructions up to its intermediate language, called BIL. All subsequent analysis is performed on BIL.

In the dynamic analysis scenario, a user first executes the program within an emulation framework to produce an instruction trace. Within BAP there are currently two options. First, we provide a PIN-based implementation that works on user-land programs. Second, BAP interfaces with TEMU [3], a whole-system emulator which can produce an instruction stream for the entire OS and application stack. In either case the stream of executed instructions are translated into BIL, on which all subsequent analysis is performed.

4.2 Binary Intermediate Language (BIL)

BIL serves as the first step to reverse engineering by assigning all variables a low-level type. In addition, by raising to BIL we can focus subsequent analysis on a small, well-specified language instead of dealing with the hundreds of assembly instructions that often have intricate and non-intuitive semantics. Lifting allows us to focus getting the semantics right at only one point instead of throughout all analysis.

The BIL language is shown in Figure 4. Each term in a BIL statement has an initial low-level type for which we use as a basis for high-level type reconstruction. Here we provide an overview; the formal semantics are given elsewhere [1]. The base types τ_{reg} in BIL IL are `reg`i`_t` where $i \in \{1, 8, 16, 32, 64\}$ for i-bit registers (i.e., n-bit vectors). Memories are given type `mem_t`(τ_{reg}), where τ_{reg} determines the type for memory addresses, e.g., $\tau_{\text{reg32_t}}$ has 32-bit memory addresses. Memory is modeled as an array: you give BIL an unsigned integer, it returns a value.

Our type inference engine infers most constraints by analyzing how variables are used in expressions, e.g., the signed division "$a /_S b$" allows us to conclude both a and b are signed numbers. Expressions in BIL are similar to those found in most languages. BIL has binary operations $\Diamond_b$ (note "&" and "|" are bit-wise), unary operations $\Diamond_u$, constants, and casting. Casting is used when indexing registers under different addressing modes. For example, the lower 8 bits of `eax` in x86 are known as `al`. When lifting x86 instructions, we use casting to project out the lower-bits of the corresponding `eax` register variable to an `al` register variable when `al` is accessed. Thus, BIL makes explicit the relationship between register addressing modes, e.g., `al` is the lower 8 bits of `eax`.

Memory operations are representative as either `load` and `store` operations in BIL. While the syntax may seem complicated, the notation simply conveys what, where, and how many bytes we are storing or loading. The semantics of `load`$(e_1, e_2, e_3, \tau_{\text{reg}})$ is to load from the memory specified by e_1 at address e_2. In C, this would loosely be written $e_1[e_2]$ (which as we will see helps us infer that e_2 is a pointer).[1] Finally, τ_{reg} tells us how many bytes to load. In C, if e_1 is of type τ, then $e_1[e_2]$ loads `sizeof(`τ`)` bytes. τ_{reg} similarly tells us how many bytes to load from memory. The semantics of `store`$(e_1, e_2, e_3, e_4, \tau_{\text{reg}})$ are similar where e_1 is the memory, the dereferenced address is e_2,

[1] The parameter e_3 tells us the endianness to use when loading bytes from memory. In BIL, we use 0 for little endian and 1 for big endian (since 0 is "smaller" than 1). We include the endianess so that BAP can be retargetted to different endian architectures, e.g., ARM memory operations can specify the endianness as a parameter.

and the value written in e_3. `store` syntactically returns a new memory name indicating that an update has happened, which is used extensively when we translate BIL to single static assignment (SSA) form (§5.1).

Statements and Programs. A program in BIL is a label followed by a sequence of statements. For example, in the static case the label is the beginning of the function and the statements are the lifted assembly. Every function implicitly takes as arguments the current set of registers and the current state of memory, each of which may be typed. BIL statements are straight-forward: there are statements for assignments, jumps, conditional jumps, and labels. In addition, we include `special` statements to identify invocation points for externally defined procedures and functions, e.g., `int 0x80` is translated to special. The `id` of a `special` indexes the kind of special, e.g., what system call, which is used in subsequent analysis. As we will see, if we know the types of a system call, we can use that information to infer high-level types.

5 Variable Recovery

5.1 Static Single Assignment (SSA)

High-level languages typically allow an arbitrary number of variables, each of which must be assigned to a fixed number of registers (e.g., the 6 general purpose registers on x86). Further, stack slots may be reused for different variables of different types, thus we may need to deconflict when multiple variables are given to a single stack slot. [2]

TIE's first step in variable recovery is to deconflict different uses of the same register and memory cells by transforming the BIL program into single static assignment form (SSA). SSA on registers is performed by giving each new register assignment a unique name. For example, each time `eax` is assigned a value we will invent a new instance name eax_1, eax_2, etc. Note SSA deconflicts multiple writes to the same register; it does not recover the original variable names as that information is lost during compilation.

While SSA on scalar registers is quite common, we also put memory into SSA, e.g., the x86 instruction `mov [eax], 0xaabbccdd` is written as `mem1 = store(mem0, eax, 0xaabbccdd, 0, reg32_t)`. During type reconstruction, having different names allows us to type a stack slot in `mem0` different than `mem1`. This feature improves our accuracy since a single stack slot may be used for many different variable instances, e.g., when designated as a generic register spill during the register allocation compilation phase [4].

If a variable v is updated in two different branches, the variable has a different instance v_1 and v_2 for each branch. When the branches meet, we need to give a unique name to subsequent references regardless of which branch was executed. This is denoted by introducing a ϕ function at each branch confluence point, e.g., $v_3 = \Phi(v_1, v_2)$.

[2] An example of when a single stack slot is used for different variables is when the variables have disjoint live ranges.

5.2 DVSA Algorithm

The variable recovery algorithm DVSA takes in the program in SSA form, and outputs variable locations in memory along with alias information. The candidate variables are later typed, and the alias information is used to determine when a single memory slot is used for two different types.

A variable is represented as a symbolic memory load or store. We determine the variables by building up a conservative estimate of the memory configuration by analyzing SSA memory operations. For example, the BIL statements shown in Figure 3(a) show the SSA form of our running example. We generate a map that shows mem_0 has memory configuration $\{t_1 \rightarrow ebp_0\}$, mem_1 has configuration $\{t_2 \rightarrow 0\} \cup mem_0$, etc. For straight-line code, each new memory state will be derived from exactly one previous memory state. However, with control flow we must consider memory at merge points, e.g., the memory state after an if-then-else is the confluence of all paths. For control flow we consider all the confluence of all possible memories, e.g., if there is an update to t_1 with value v_1 along path 1 and an update along path 2 with value v_2, we say that $t_1 \rightarrow \{v_1, v_2\}$.

We use a variant of Value Set Analysis (VSA) [5] to determine possible alias relationships in subsequent steps. VSA is an assembly code abstraction interpretation analysis that approximates the range of values a variable may take on as a strided interval (`SI`). `SI` has a form of $s[lb, ub]$, where s is a stride and $[lb, ub]$ is the interval, the lower and upper bound of the value. For example, the `SI` $2[0, 10]$ represents the set $\{0,2,4,6,8,10\}$. Strided intervals are more specific than simple ranges. However, they over-approximate the set of possible values, e.g., the `SI` for set $\{1,2,4\}$ is $1[1, 4]$, which is an over-approximation since this `SI` represents the set $\{1,2,3,4\}$. The main difference between our variant and the original VSA [5] is we generalize VSA to create an `SI` based upon linear combinations of scalars. The original VSA would implicitly define the `SI` as being with respect to a single reference register, e.g., `ebp`. We allow for VSA to include any combination of scalar variables in our SSA form. This leads to more accurate results in our experiments. More details about the specifics of our algorithm can be found elsewhere [8].

$$
\begin{array}{lll}
T & ::= & \tau_{data} \mid \tau_{fun} \mid \top \mid \bot \mid T \cap T \mid T \cup T \mid T \rightarrow T \\
\tau_{data} & ::= & \tau_{base} \mid \tau_{mem} \\
\tau_{base} & ::= & \tau_{reg} \mid \tau_{refined} \\
\tau_{reg} & ::= & \texttt{reg1_t} \mid \texttt{reg8_t} \mid \texttt{reg16_t} \mid \texttt{reg32_t} \\
\tau_{refined} & ::= & \texttt{num}n\texttt{_t} \mid \texttt{uint}n\texttt{_t} \mid \texttt{int}n\texttt{_t}\ (n = 8,16,32) \mid \\
 & & \texttt{ptr}(T) \mid \texttt{code_t} \\
\tau_{mem} & ::= & \{\forall \text{ addresses } i \mid l_i : T_i\} \\
\tau_R & ::= & \{var_1 : T_1, \cdots, var_n : T_n\} \\
\tau_{fun} & ::= & \tau_{mem} \rightarrow \tau_R \rightarrow T \\
\textit{constraints} & ::= & T = T \mid T <: T \\
 & & \mid \textit{constraints} \wedge \textit{constraints} \\
 & & \mid \textit{constraints} \vee \textit{constraints}
\end{array}
$$

Figure 5. Types and constraints of TIE

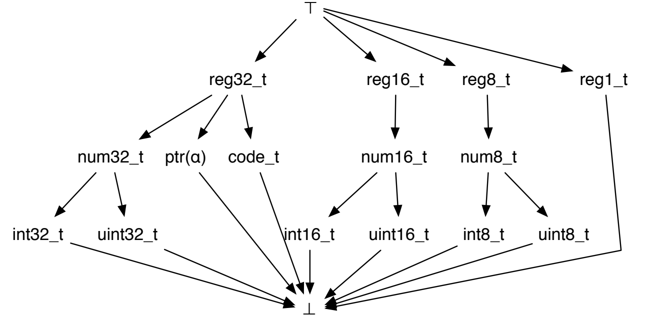


Figure 6. Type lattice showing the hierarchy of types in τ_{base}

6 TIE: Type Inference on Binary Code

6.1 Type System

The input to the type reconstruction phase of TIE is the BIL code annotated with simple register types. Type reconstruction in TIE is the process of collecting and analyzing hints based upon how binary operations treat variables in order to infer high-level types. We use subtype theory to express the amount of uncertainty that may exist on the exact higher-level type of a variable. For example, we may know that a variable is used as an integer, but not know whether it is signed or not. In our system we have a number supertype which is less specific than either a signed or an unsigned integer.

TIE enriches the basic binary-code types from BIL to include:

- $\top$, which corresponds to a variable being "any" type, and $\bot$, which corresponds to a variable being used in a type-inconsistent manner.
- Numbers (`num`n`_t`), signed integers (`int`n`_t`), and unsigned integers (`uint`n`_t`).
- Pointers of type `ptr`(T) where T is some other type in the system.
- Records of a fixed number of variables, which map the variable name var_i to its type T_i. We also distinguish the memory type τ_{mem}, which maps each memory cell address to the type stored at that address.
- General function types $T_1 \rightarrow T_2$. In addition, we denote by τ_{fun} the type for high-level C functions we infer, which takes the current memory state and a list of registers as arguments, and returns something of type T.
- Intersection types $(T \cap T)$ and union types $(T \cup T)$, which we explain below.

The base integer types in TIE form a subtyping lattice, shown in Figure 6. We omit additional edges between base types to keep the diagram simple, e.g., `reg16_t` $<:$ `reg32_t`, though they do exist in our type system. The subtyping relationships are extracted from the lattice as follows: if $S \leftarrow T$ is an edge in the lattice, then $S <: T$ (i.e., think of "$<:$" as an arrow in the lattice). Following multiple edges in the lattice corresponds to the transitive nature of subtyping. Since a type hierarchy is a lattice, the $\sqcap$ and $\sqcup$ operations are also applied in the context of subtyping, e.g., $T_1 \sqcap T_2 = M$ where M is the greatest lower bound of types T_1 and T_2 in the subtyping hierarchy.

Intersection and Union Types. TIE's type system has intersection ($\cap$) and union ($\cup$) types. Mathematically, something is of type $T_1 \cap T_2$ if it can be described by both types, e.g., if T_1 is the C type for signed characters of range -128 to 127 and T_2 is the C type for unsigned characters of range 0 to 255, then $T_1 \cap T_2$ is the type for range 0 to 127. In our type system, we use intersection types during constraint solving where something is of type $T_1 \cap T_2$ if it is the order-theoretic meet (i.e., greatest lower bound) of $T_1 \sqcap T_2$.

The dual of intersection types is union type $T_1 \cup T_2$, corresponding to the order-theoretic join (i.e., lowest upper bound) $T_1 \sqcup T_2$. The values of $T_1 \cup T_2$ include the union of values from both types. Note that union types are not the same as sum types. A sum type adds a tag, while a union type extends the range of values.

Output. The output of type inference is an upper and lower bound on the type for each variable. The user is then free to pick either as the inferred type. Of course if we can completely infer the type we output the same type for the upper and lower bound. However, outputting a range is much more general, and can make explicit the uncertainty due to the nature of the problem. For example, C unions may be compiled down so that a single stack slot holds dif-

C types	Corresponding types in our type system
int	int32_t
unsigned int	uint32_t
short int	int16_t
unsigned short int	uint16_t
char	int8_t
unsigned char	uint8_t
* (pointer)	$\texttt{ptr}(\alpha)$
void *	$\texttt{ptr}(\top)$
void	$\bot$
union	$T \cup T$
struct, [] (array)	$\{l_i : T_i\}$
function	$\tau_{mem} \to R \to T\ (:\tau_{fun})$

Table 1. Mapping between the C types the types in our type system.

ferent typed variables. The only reason we know the type of the variable at any point in time is due to a user annotation, which is completely lost during compilation. Our type system allows us to express such uncertainty by expressing one bound as a type conflict($\bot$) and one as a C union ($\cup$).

Correspondence to C Types. TIE's type system contains features of modern programming languages which are then translated in a post-processing step to types in a specific language. Currently TIE is targeted to output C types by translating internally-inferred TIE types into C types using the translation shown in Table 1. Structure types in C correspond to record types in our language, as expected. The void and void * rules may seem strange for those unfamiliar with typing systems. It may seem that void * is a pointer to void, however this is an unfortunate naming problem in C. void corresponds to no type which is why we equate it with $\bot$ in our system. [3] void * can point to anything at all, which is why we equate it to ptr($\top$).

Evaluation Metrics. Our goal is to infer conservative yet accurate types. Conservativeness means we want to infer no more than the binary code tells us, e.g., never guess a type. In addition, we want to be precise by inferring as much as possible from the code.

More formally, let τ_t be a type variable for a term t. We allow a typing algorithm to output both a lower bound $\mathcal{B}^{\downarrow}(\tau_t)$ and an upper bound $\mathcal{B}^{\uparrow}(\tau_t)$ for a variable. If the typing algorithm wants to indicate a specific type it outputs $\tau = \mathcal{B}^{\downarrow}(\tau_t) = \mathcal{B}^{\uparrow}(\tau_t)$.

An algorithm is conservative if the real type τ for a program is between the upper and lower bound. More formally:

[3] Another option would be to introduce the unit type. We did not see any advantage in doing so.

Definition 6.1. [CONSERVATIVENESS] For a term t, let τ_{src} be the type of t in the source code. Given a well-typed source code program, we say a typing algorithm for t is *conservative* with respect to the real type of t (τ_{src}) if

$$\mathcal{B}^{\downarrow}(\tau_t) <: \tau_{\text{src}} <: \mathcal{B}^{\uparrow}(\tau_t).$$

Being conservative is desirable, but not sufficient. For example, an algorithm could output $\top$ for the upper bound and $\bot$ as a lower bound and always be conservative. Therefore, we also define the notion of how precise the inferred type is by measuring the distance between the upper and lower bound. For example, if an algorithm outputs the same upper and lower bound, and the output type is the same as the source code type, the distance is 0 and the algorithm is completely precise.

The distance is measured with respect to our type lattice. If two types can form a type interval, i.e., one is a subtype of the other, their distance is the number of edges separating them on the lattice. If the two types are incompatible, their distance is set to the maximum value. More formally, the distance between types is defined as:

Definition 6.2. [DISTANCE] For two types, τ_1 and τ_2, the *distance* between them, denoted by $||\tau_1 - \tau_2||$, is the difference between the level of each in the type lattice if $\tau_1 <: \tau_2$ or $\tau_2 <: \tau_1$. Otherwise, it is the maximum distance, which is $||\top - \bot||$.

In our type system shown in Figure 6, the maximum value of $||\top - \bot||$ is 4.

Structural types. Our accuracy metric handles structures as two dimensions in terms of distance: how many fields were inferred vs. the real number of fields, and the distance between the real type for a field and its inferred type.

We formalize the distance between two structure types $A = \{l_i : S_i^{i \in 1..n_A}\}$ and $B = \{l_i : T_i^{i \in 1..n_B}\}$ as $||A - B||$ by measuring the difference between the number of fields and type for each field:

$$||A - B|| = \left|(1 - \frac{1}{n_A}) - (1 - \frac{1}{n_B})\right| + \frac{\text{avg}||S_i - T_i^{i \in 1..\max(n_A, n_B)}||}{||\bot - \top||}, \quad (1)$$

where n_A and n_B denote the number of fields for A and B, respectively. The first term shows the difference in the number of fields and the second term measures the distance of each field element using the subtype relation. Note that, to be compatible with the levels in the type lattice the distances are normalized over the total lattice height.

For example, suppose two record types $U = \{0 : \texttt{int32_t}\}$ and $V = \{0 : \texttt{int32_t}, 4 : \texttt{uint32_t}\}$. The number of fields n_U and n_V are 1 and 2, respectively. Thus, $||U - V|| = |(1 - \frac{1}{1}) - (1 - \frac{1}{2})| + \frac{(0+4)/2}{4} = 1$. In addition, $||\texttt{reg32_t} - V||$ is computed as the difference of the levels of reg32_t and V, which is $||\texttt{reg32_t} - \{0 : \top\}|| + ||\{0 : \top\} - V|| = 1 + 1.5 = 2.5$.

Statement	Generated constrains
$x := e$	$\tau_x = \tau_e$
$\texttt{goto}\ e$	$\tau_e = \texttt{ptr}(\texttt{code_t})$
$\texttt{if}\ e\ \texttt{then goto}\ e_t\ \texttt{else goto}\ e_f$	$\tau_e = \texttt{reg1_t} \wedge \tau_t = \texttt{ptr}(\texttt{code_t}) \wedge \tau_f = \texttt{ptr}(\texttt{code_t})$
$\texttt{call}\ f\ \texttt{with}\ m\ v^*\ \texttt{ret}\ r$	$\tau'_m = \tau_{m\,f} \wedge \bigwedge_v (\tau_v = \tau_{v\,f}.[v]) \wedge \tau_r = \tau_{r\,f}$ (where $F \vdash f : \tau_{m\,f} \rightarrow \tau_{v\,f} \rightarrow \tau_{r\,f}, \tau'_m = \#\texttt{update}(\tau_m)$)

Expression	Generated constraint for term with type variable τ
x (variable)	τ_x
v (integer)	τ_v
$-_n e$ (unary neg)	$\tau_e <: \texttt{int}n\texttt{_t} \wedge \tau :> \texttt{int}n\texttt{_t}$
$e_1 +_{32} e_2$	$(\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num32_t})$ $\vee(\tau_{e_1} <: \texttt{ptr}(T_\alpha) \wedge \tau_{e_2} <: \texttt{num32_t} \wedge \tau :> \texttt{ptr}(T_\beta))$ $\vee(\tau_{e_1} <: \texttt{num32_t} \wedge \tau_{e_2} <: \texttt{ptr}(T_\alpha) \wedge \tau :> \texttt{ptr}(T_\beta))$
$e_1 +_{n \neq 32} e_2$	$\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num}n\texttt{_t}$
$\sim_n e$	$\tau_e <: \texttt{uint}n\texttt{_t} \wedge \tau :> \texttt{uint}n\texttt{_t}$
$e_1 <_{S_n} e_2$	$\tau_{e_1} <: \texttt{int}n\texttt{_t} \wedge \tau_{e_2} <: \texttt{int}n\texttt{_t} \wedge \tau :> \texttt{reg1_t}$
$\texttt{load}(m, i, d, \texttt{reg}n\texttt{_t})$	$\tau_i = \texttt{ptr}(\tau_m.[i]) \wedge \tau = \tau_m.[i] \wedge \tau <: \texttt{reg}n\texttt{_t}$
$\texttt{store}(m, i, v, d, \texttt{reg}n\texttt{_t})$	$\tau_i = \texttt{ptr}(\tau_v) \wedge \tau = \tau_m\{i : \tau_v\} \wedge t.[i] <: \texttt{reg}n\texttt{_t}$
$\Phi(e_1, \cdots, e_n)$	$\tau' <: \tau_{e_1} \cap \cdots \cap \tau_{e_n}$

Figure 7. Example rules for type constraint generation

6.2 Generating Type Constraints

The constraint generation step takes as input a BIL program, and outputs a system of type constraints over BIL terms (i.e., variable, expression, or value). In constraint generation each term is initially given a unique type variable. The constraints represent what we can infer about the terms based upon the operations performed. We generate constraints by syntactically inspecting each term. A summary of representative constraint generation rules is shown in Figure 7.

The constraints are expressed in terms of subtyping theory, which has the effect of bounding the type of each variable. For example, if we generate the constraints that $\texttt{int32_t} <: \tau_x$ and $\tau_x <: \texttt{num32_t}$, we know that type τ_x is either $\texttt{num32_t}$ or $\texttt{int32_t}$.

The typing constraints for assignment statements assert that the type of the assigned variable must be the same as the computed expression. The constraint for `goto` says that the type of the computed address must be code. The constraint for `if-then-else` states that the condition must evaluate to a Boolean, and that both true and false jump targets must be code. We discuss `call` and inter-procedural analysis in *inter-procedural constraint generation.*

When typing statements, we recursively descend into any sub-expressions being computed and add the appropriate type constraints, as shown by the rules for expressions in Figure 7. For variables x, we simply return the type variable τ_x. For a single integer v, we simply return that v's type variable. When v is used in an operation additional constraints will be added, e.g., if v is used as a constant for signed comparison with a 32-bit number we add the constraint that v must also be a signed 32-bit number.

Unary and binary operations are more involved. For example, consider 32-bit addition "$e_1 +_{32} e_2$" (e.g., the `add` instruction in x86), which is a function that takes two arguments τ_{e_1} and τ_{e_2} and returns a value of type τ. Our type constraints consider three cases, as indicated by the three parts of the disjunction:

1. Addition is being used to add two numbers, and thus the result is a number. In this case we add the constraint that the operand types τ_{e_1} and τ_{e_2} must be at least a $\texttt{num32_t}$, and the result τ must be at least $\texttt{num32_t}$. We express this constraint as: $\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num32_t}$
2. Addition is being used to add a pointer of type τ_α to a number of type $\texttt{num32_t}$. The resulting type is τ_β because there is no guarantee that after adding a number to a pointer it points to something of the same type. We express this constraint as: $\tau_{e_1} <: \texttt{ptr}(T_\alpha) \wedge \tau_{e_2} <:$

$\texttt{num32_t} \wedge \tau :> \texttt{ptr}(T_\beta)$

3. Addition is being used to add an integer of type num32_t to a pointer of type τ_α. This case is symmetric to the above.

Unary negation ($-_n$), bitwise complementing ($\sim_n$), arithmetic operations on variables other than 32-bits, and signed operations allow us to be much more specific in the constraints generated. For example, unary negation and signed comparison allow us to infer that the types of the operands are at least of signed type. Note that the subtype relation is covariant for the result type τ and contravariant for the operand types since binary and unary operations are functions with specific types, e.g., $-_n : \texttt{int}n\texttt{_t} \rightarrow \texttt{int}n\texttt{_t}$.

Typing Memory Operations. Recall we model memory as an array of elements indexed by an address. We denote by $[i]$ the DVSA value of i. The typing rules for load and store should propagate types so that if we store a value of type α at address $[i_1]$, and then subsequently load a value from $[i_2]$, then the resulting type is α when $[i_1] = [i_2]$. We first describe how the typing rules express this idea, and then discuss how equality between symbolic values is calculated.

The load rule states that when we are given an index i into memory m, then i) we can treat the type of i like a pointer to the type of values at $m.[i]$, and ii) the returned type τ is the same as the type of values stored at $m.[i]$, and iii) since the load was t bytes, the resulting type must be a subtype of t (i.e., $\tau <: \texttt{reg32_t}$ for a 32-bit load). The store operation is symmetric: if we are storing a value v of type τ_v at $[i]$, then the memory cell at $[i]$ has type τ_v.

We introduce equality constraints between memory dereferences when the DVSA SI's overlap. For example, if we store to address j and load from address i, and the DVSA SI's say that j and i are the same address range, we add a type constraint between the referenced memory cells. This is imprecise and possibly un-conservative, e.g., if j's address range is from $[0, 100]$ and i's address range is $[2, 4]$, then the load from i may completely, partially or not at all overlap. As we show in our evaluation our conservativeness in practice is typically above 90% even with such issues.

Inter-procedural constraint generation. As a pre-processing step, we create a context F of type signatures for known functions such as those in libc. When a call to a function f is issued, we first search the function description for the function f in F. If we find a function description $(\tau_{mf} \rightarrow \tau_{vf} \rightarrow \tau_{rf})$ for the function from F, we match it caller's type variables for arguments and return value. The function $\#\texttt{update}(\tau_{mem})$ uses information about calling conventions to match up offsets correctly. For example, in the standard call of 32-bit x86 architecture, #update updates $\{l_i \mapsto T_i\}$ to $\{l_i + \texttt{esp} - \texttt{ebp} + 4 \mapsto T_i\}$.

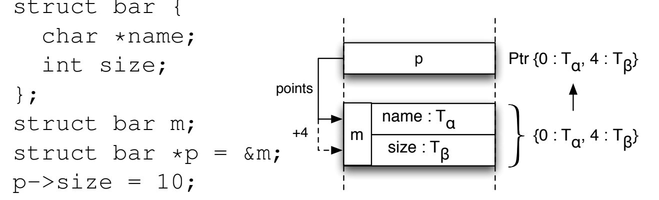


Figure 8. Access to structural data (source code and stack)

TIE also handles calls to functions without known parameter types, e.g., when one local procedure calls another local procedure. During pre-processing TIE adds all function names to F. When a call is made to a function, TIE matches up the callers arguments with the number of parameters inferred for the callee during DVSA. TIE then adds the appropriate equality constraints, e.g., the type of the first passed argument must be the same as the first parameter. This process is dependent on calling conventions and platform. For example, in the standard call of 32-bit x86 architecture, we follow the memory access with the address above ebp, such as $\texttt{ebp} + 4$ or $\texttt{ebp} + 8$, to find the number and variables for the arguments, and the last value of eax will be the return value. Supporting additional calling conventions is straight-forward, e.g., adding fastcall would entail adding constraints that the first two arguments are passed in ecx and edx.

Constraints for structural data. To infer the type of structural data, we restate the rules in Figure 7. Binary programs access the structural data in two ways: 1) access via pointers, where binary programs calculate the address of fields by arithmetic operations on the base pointer, and 2) direct access, where the structural data reside in the stack as multiple variables, and the fields are accessed directly. Unfortunately, directly accessing structure fields is indistinguishable from accessing local variables on the stack. Therefore, we use the access via pointers as a hint that reveals the presence of structural data.

Figure 8 gives the intuition of constraint generation for structural data. A variable m of a two-field structure bar is located on the stack and a pointer p points to m. When the second field size is accessed through p, its address is calculated by applying an add operation with pointer p. Thus, we can tell that p points to structural data, which has at least one field at offset 4.

To handle constraints for such hints, we redefine the rules for $e_1 +_{32} e_2$. If a pointer is used in addition with a constant value, we infer it as a pointer to structural data as follows.

$$\begin{aligned}
&(\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num32_t}) \\
\vee&(\tau_{e_1} <: \texttt{ptr}(\{0 : T_\alpha, [e_2] : T_\beta\}) \wedge \tau_{e_2} <: \texttt{num32_t} \wedge \tau :> \texttt{ptr}(T_\beta)) \\
\vee&(\tau_{e_1} <: \texttt{num32_t} \wedge \tau_{e_2} <: \texttt{ptr}(\{0 : T_\alpha, [e_1] : T_\beta\}) \wedge \tau :> \texttt{ptr}(T_\beta)),
\end{aligned}$$

where $[e_1]$ and $[e_2]$ are the value of e_1 and e_2, respectively. For example, the constraint generated for p is (we show the 2nd disjunctive constraint) $\cdots \vee (\tau_{\mathtt{p}} <: \mathtt{ptr}(\{0 : T_\alpha, 4 : T_\beta\}) \wedge \tau_4 <: \mathtt{num32_t} \wedge \tau :> \mathtt{ptr}(T_\beta)) \vee \cdots$.

6.3 Constraint Solving

The final step in TIE is solving the generated type constraints. In this section we present our constraint solving algorithm, that takes in a list of type constraints $\mathbb{C}$ and returns a map from type variables to the inferred type interval. We use a unification algorithm extended to support subtypes. During the constraint solving process, we keep a *working set* that contains the current state of our algorithm. The working set consists of the following:

- $\mathbb{C}$: the list of type constraints that remain to be processed by our algorithm.
- $\mathbb{S}_{<:}$: a set containing all the inferred subtype relations.
- $\mathbb{S}_{=}$: a map from bound type variables to types.
- $\mathcal{B}^{\uparrow}$: a map from type variables to the upper bound of the inferred type.
- $\mathcal{B}^{\downarrow}$: a map from type variables to the lower bound of the inferred type.

The final solution returns the pair $\langle \mathcal{B}^{\uparrow}, \mathcal{B}^{\downarrow} \rangle$. During each step of constraint solving, we remove and process a single element from the constraint worklist $\mathbb{C}$. Our algorithm terminates when the constraint list is empty ($\mathbb{C} = \emptyset$). The initial constraint list contains the constraints from the constraint generation process (described in § 6.2) while all other contexts ($\mathbb{S}_{<:}$, $\mathbb{S}_{=}$, $\mathcal{B}^{\uparrow}$, $\mathcal{B}^{\downarrow}$) are initialized to empty ($\emptyset$).

In sections § 6.3.3-§ 6.3.4 we describe how we solve each constraint type shown in Figure 5. We solve each constraint in the following order: equality, subtype relation, conjunctive and disjunctive constraints.

6.3.1 Equality constraints ($T_\alpha = T_\beta$)

Equality constraints are solved through unification. Unification is essentially a substitution process, where given $S = T$ we replace all occurrences of S with T in our worklist $\mathbb{C}$. Algorithm 1 shows how the working set is updated for every equality constraint.

Note that before performing substitution, we perform the occurs check [17], in which we check whether a type variable occurs in both hands of an equality, e.g., $\alpha = \mathtt{ptr}(\alpha)$. Since we do not support recursive types, we raise type error and drop the current working set whenever we have an occurs check violation. Also note that the last option in our algorithm is to drop the constraint. We do this to continue typing the remaining variables, instead of failing and returning the type $\perp$ for all variables.

6.3.2 Subtype relation constraints ($T_\alpha <: T_\beta$)

To solve subtype relation constraints, we use a type closure algorithm. Subtype relation constraints describe type inequality relations, thus the solution provided by the closure algorithm maps each type variable to an interval. For each type variable α, we keep track of its upper bound $\mathcal{B}^{\uparrow}(\alpha)$ and lower bound $\mathcal{B}^{\downarrow}(\alpha)$, which are initialized as $\top$ and $\perp$, respectively. Every time we process a subtype relation constraint that involves a type α, the closure algorithm uses the constraint to refine the type interval of α. For example, the constraint $\alpha <: \mathtt{ptr}(\mathtt{int32_t})$ makes the closure algorithm update the upper bound of α to $\mathtt{ptr}(\mathtt{int32_t})$ ($\mathcal{B}^{\uparrow}(\alpha) \leftarrow \mathtt{ptr}(\mathtt{int32_t})$).

The core idea behind the closure algorithm is the propagation of type information via transitive subtype relations. We use $\mathbb{S}_{<:}$, to keep track of the transitive subtype relation. $\mathbb{S}_{<:}$ is always closed under the subtype relation for a given set of subtype relation constraints. For example, if $\mathbb{S}_{<:} = \{\alpha <: \beta\}$ and we append the constraint $\beta <: \gamma$, the new $\mathbb{S}_{<:}$ will become $\{\alpha <: \beta, \beta <: \gamma, \alpha <: \gamma\}$. If another constraint $\mathtt{int32_t} <: \alpha$ is given, the lower bound of α, β, γ will be updated to $\mathtt{int32_t}$. Note that for constraints on constructed types, e.g., $\mathtt{ptr}(\alpha) <: \mathtt{ptr}(\beta)$ we recurse into the $\mathtt{ptr}()$ and infer the implicit constraints $\alpha <: \beta$.

Meet and Join. We populate $\mathcal{B}^{\uparrow}$ and $\mathcal{B}^{\downarrow}$ during the subtype resolution process. We define rules for $\sqcap$ and $\sqcup$ operations in Figure 9 to compute the upper bound $\mathcal{B}^{\uparrow}$ and lower bound $\mathcal{B}^{\downarrow}$ respectively. M-SUBTYPE and M-NOREL show the basic behavior of the $\sqcap$ operation. As shown in M-TYPEVAR, if $\sqcap$ is applied with a type variable α, it produces the intersection type with α. In M-PTR, we apply $\sqcap$ on the parameters of pointers, recursively. For the entries that have no matched label in both record types, we add them to the result of $\sqcap$. As shown in M-RECBASE, if a record type meets a base type τ, then the base type is converted to an equivalent record type $\{0 : \tau\}$, and $\sqcap$ is applied to them again. The rules for $\sqcup$ work in a similar manner.

Algorithm 1: Solving a single equality constraint from the constraint list $\mathbb{C}$.

```
SolveEquality (S = T, ℂ)
    if IsFreeVariable(S) then
        ℂ = ℂ[S\T] ;   𝕊= = 𝕊={S ↦ T} ;
    else if IsFreeVariable(T) then
        ℂ = ℂ[T\S] ;   𝕊= = 𝕊={T ↦ S} ;
    else if S = ptr(S1) and T = ptr(T1) then
        ℂ = ℂ ∪ {S1 = T1} ;
    else if S = {li : Si} and T = {li : Ti} then
        ℂ = ℂ ∪ ⋃i {Si = Ti} ;
    else
        Drop_Constraint
end
```

$$\frac{S <: T \quad S \in \tau_{base} \quad T \in \tau_{base}}{S \sqcap T \vdash S}\ \text{M-SUBTYPE} \qquad \frac{S \not<: T \quad T \not<: S}{S \sqcap T \vdash \bot}\ \text{M-NOREL} \qquad \frac{}{\alpha \sqcap T \vdash \alpha \cap T}\ \text{M-TYPEVAR}$$

$$\frac{T_1 : \mathtt{ptr}(P_1) \quad T_2 : \mathtt{ptr}(P_2) \quad \mathtt{ptr}(P_1 \sqcap P_2) \vdash T_3}{T_1 \sqcap T_2 \vdash T_3}\ \text{M-PTR} \qquad \frac{U : \{l_i : S_i\} \quad V \in \tau_{base} \quad U \sqcap \{0 : V\} \vdash T}{U \sqcap V \vdash T}\ \text{M-RECBASE}$$

$$\frac{S <: T \quad S \in \tau_{base} \quad T \in \tau_{base}}{S \sqcup T \vdash T}\ \text{J-SUBTYPE} \qquad \frac{S \not<: T \quad T \not<: S}{S \sqcup T \vdash \top}\ \text{J-NOREL} \qquad \frac{}{\alpha \sqcup T \vdash \alpha \cup T}\ \text{J-TYPEVAR}$$

$$\frac{T_1 : \mathtt{ptr}(P_1) \quad T_2 : \mathtt{ptr}(P_2) \quad \mathtt{ptr}(P_1 \sqcap P_2) \vdash T_3}{T_1 \sqcup T_2 \vdash T_3}\ \text{J-PTR} \qquad \frac{U : \{l_i : S_i\} \quad V \in \tau_{base} \quad U \sqcup \{0 : V\} \vdash T}{U \sqcup V \vdash T}\ \text{J-RECBASE}$$

Figure 9. Rules for $\sqcap$ and $\sqcup$ operations

Decomposition Rules. Before solving a subtype relation constraint, we try to simplify it by applying decomposition rules. For instance, if a type is a subtype of an intersection type, $S <: T \cap U$, we divide it into two constraints, $S <: T$ and $S <: U$. Likewise, if a type is a supertype of a union type, $S \cup T <: U$, it is decomposed into $S <: U$ and $T <: U$. For extended types, the decomposition rules are applied recursively, i.e., $\mathtt{ptr}(S) <: \mathtt{ptr}(T)$ is changed to $S <: T$. We denote the application of decomposition rules with $\Upsilon()$ and a subtype constraint c is decomposed into c' as $\Upsilon(c) \rightsquigarrow c'$. For constraints on basic type variables, where further decomposition is not possible, the decomposition operator returns an empty set ($\emptyset$).

The overall closure algorithm for each subtype relation constraint $S <: T$ is:

1. Remove $S <: T$ from $\mathbb{C}$ and update $\mathbb{C}$ with $\Upsilon(S <: T)$.
2. $\forall(\alpha <: S)$ in $\mathbb{S}_{<:}$, add $\alpha <: T$ to $\mathbb{S}_{<:}$ and update $\mathcal{B}^{\uparrow}(\alpha) \leftarrow \mathcal{B}^{\uparrow}(\alpha) \sqcap \mathcal{B}^{\uparrow}(T)$ and $\mathbb{C}$ with $\Upsilon(\alpha <: S)$
3. $\forall(T <: \beta)$ in $\mathbb{S}_{<:}$, add $S <: \beta$ to $\mathbb{S}_{<:}$ and update $\mathcal{B}^{\downarrow}(\beta) \leftarrow \mathcal{B}^{\downarrow}(\beta) \sqcup \mathcal{B}^{\downarrow}(S)$ and $\mathbb{C}$ with $\Upsilon(T <: \beta)$

For a given constraint $S <: T$, we search $\mathbb{S}_{<:}$ and find all type variables that are subtypes of S and supertypes of T. We then update the upper bound of all subtype variables of S with the upper bound of T and the lower bound of all supertype variable of T with the lower bound of S. For updating the lower bound and the upper bound, we use the $\sqcup$ and $\sqcap$ operations we defined above.

Similarly to the occurs check, we perform a cycle check in $\mathbb{S}_{<:}$. While computing the closure of $\mathbb{S}_{<:}$ for a newly added subtype relation, if the same type variable occurs in both hands of an inequality, it means a cycle exists in $\mathbb{S}_{<:}$. In this case, we drop the constraint that makes the cycle and remove it from $\mathbb{C}$.

6.3.3 Conjunctive constraints ($C_1 \wedge C_2$)

To solve a conjunction of type constraints, we decompose the conjunction to a list and append the constraints to our worklist $\mathbb{C}$.

6.3.4 Disjunctive constraints ($C_1 \vee C_2$)

Whenever we need to solve a disjunction of type constraints, TIE tries to find a solution for each type constraint separately. Type constraints that force type variables to go to $\bot$ (due to incompatible constraints) are rejected. The remaining solutions are merged using composition rules. When none of the type constraints in the disjunction are satisfiable, TIE raises a type error. If more than one constraints are satisfiable, we either (1) keep the constraints from all satisfiable constraints, or (2) produce a different solution for each satisfiable constraint. In the first case, we have a single general conservative solution that satisfies all compatible constraints, while in the latter, we can have an exponential number of more specific solutions in the number of disjunctions. TIE always applies the former, since our goal is to always infer types conservatively. The *composition rules* described below, show how to merge satisfiable disjunctive constraints.

Composition Rules. Our composition rules take advantage of the intersection and union types to merge multiple subtype relation constraints (regarding a certain type variable) into a single constraint. These rules are applied for different type variables that refer to the same memory location according to DVSA.

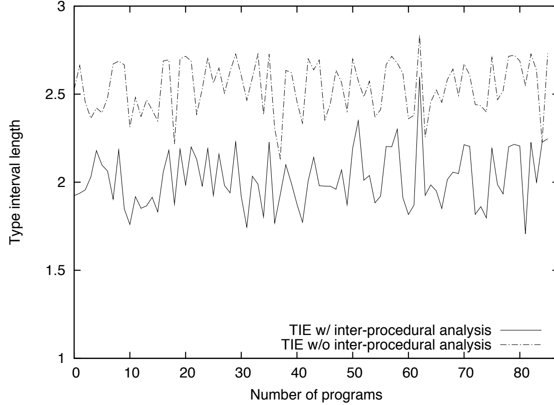
Suppose, we have two constraints $\alpha <: Q \wedge \beta :> T$ and $\alpha <: U \wedge \beta :> V$, both satisfiable with the current working set. The disjunctive constraints are

$$\begin{aligned}(\alpha <: Q \wedge \beta :> T) \vee (\alpha <: U \wedge \beta :> V) = \\ (\alpha <: Q \vee \alpha <: U) \wedge (\beta :> T \wedge \beta :> V)\end{aligned} \tag{2}$$

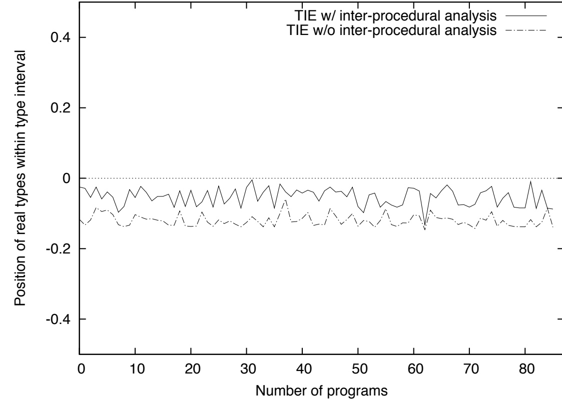
If Eq.2 holds for α and β , the more conservative condition $\alpha <: Q \cup U$ and $\beta :> T \cap V$ also holds. Thus, the above disjunctive constraints can be conservatively merged into conjunctive constraints: $(\alpha <: Q \cup U) \wedge (\beta :> T \cap V)$.

6.3.5 Collection and interpretation of results

After solving all the constraints ($\mathbb{C} = \emptyset$), each working set is a solution. Since our type system has richer types than C, we generalize the solution.



(a) Precision



(b) Conservativeness rate

Figure 10. Summary of precision (left) and conservativeness (right)

Type interval. In the working set, the upper bound and lower bound provide the range of the inferred type for the variable. We call the interval from the lower bound to the upper bound of a variable α as a "type interval," denoted by $[\mathcal{B}^{\downarrow}(\alpha), \mathcal{B}^{\uparrow}(\alpha)]$.

Structural equality in results. The real types in C source codes has struct and arrays for structural types. However, pointers are also used interchangeably with them. Thus, the three types are not distinguishable because their behavior is the same in binary programs. In our type system, we assume they are in structural equivalence as follows:

$$\texttt{ptr}(\alpha) \equiv \{0 \mapsto \alpha\} \equiv \alpha[](\text{array})$$

When we compare two structural equivalent types. We convert the type into a record type $\{0 \mapsto \alpha\}$, which is the most general form.

Collection and generalization. We convert each working set into a final solution. First, if a type variable, T_α, is remained in $\mathcal{B}^{\uparrow}$ and $\mathcal{B}^{\downarrow}$, we replace it as $\mathcal{B}^{\uparrow}(T_\alpha)$ in $\mathcal{B}^{\uparrow}$ and $\mathcal{B}^{\downarrow}(T_\alpha)$ in $\mathcal{B}^{\downarrow}$, respectively. We also generalize record types. If $\mathcal{B}^{\uparrow}$ of record types has fields of $\top$, we remove the fields. If all the fields in $\mathcal{B}^{\uparrow}$ are of the same type, $\{l_i : T^{\forall i}\}$, we convert it to more general structure form, $\{0 : T\}$, according to the structural equivalence. Second, for each term t, we find a matching type variable T_t and a type interval of $[\mathcal{B}^{\downarrow}(\alpha), \mathcal{B}^{\downarrow}(\alpha)]$. If $\mathbb{S}_=$ has a mapping for T_t, we use the equivalent type variable of T_t in $\mathbb{S}_=$ instead of T_t. At last, we collect the result for memory and group them by its address. Since we are based on the SSA form, we may have more than one variable for each memory address and it means the memory location is reused by various data of different type. Thus, we combine members for the same address as a union type of them. Through this collection process, each variable has a final type interval.

7 Implementation

TIE is primarily implemented in 29k of OCaml code, most of which is generic binary analysis code such as a data-flow framework, building CFG's, etc. Our static analysis component uses a linear sweep disassembler written in C build on top of `libopcodes` (more advanced disassembly such as dealing with obfuscated code is left outside the scope of this paper, but such routines can be plugged into our infrastructure). We base our dynamic analysis on PIN [13], and currently there are about 1.4k lines of C++ code to create the instruction trace. The variable recovery and type inference are approximately 3.6 and 2.1K lines of OCaml, respectively.

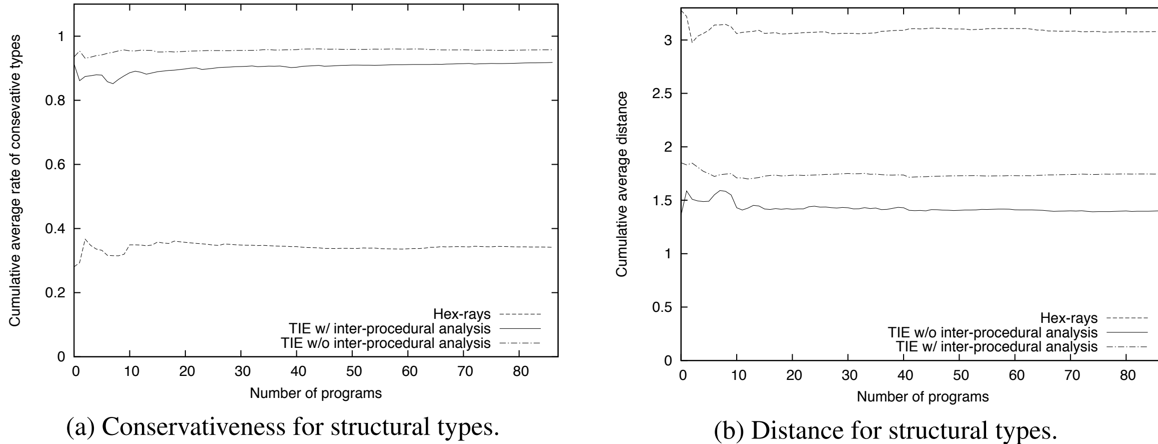
8 Evaluation

8.1 Evaluation Setup

We have evaluated TIE on 87 programs from `coreutils` (v8.4). We compiled the programs with debug support but only use the information for measuring type inference accuracy. The type information was extracted using `libdwarf`. TIE's inter-procedural analysis used function prototypes for `libc`, as extracted from the appropriate header files. All experiments are performed on 32-bit x86 binaries in Linux.

We measure the accuracy of TIE against the REWARDS [12] code given to use by the authors for the dynamic analysis setting, and against Hex-rays decompiler (v1.2) in IDA Pro [2] for static analysis setting.

TIE outputs an upper and lower bound on the inferred type. We use this to measure the conservativeness of TIE. For precision, we translate the bound output by TIE into a single C type by translating the lower bound unless it is $\bot$. If the lower bound is $\bot$, we output the upper bound. We found this heuristic to provide the most accurate results.



(a) Conservativeness for structural types.

(b) Distance for structural types.

Figure 11. Conservativeness and distance for structural types

Caveats. Hex-rays and REWARDS have an inherently less informative type system than ours since they are restricted to only C types. For example, REWARDS guesses that a register holds an `uint32_t` with no additional information for 32-bit registers, while Hex-rays guesses `int32_t` in the same situation. Our `reg32_t` reflects in the same situation that we do not know whether the register is a signed number, unsigned number or pointer.

Thus, we must either convert their types to ours, or vice-versa. We've experimented with both. In order to provide the most conservative setting for REWARDS and Hex-rays, we translated each type output as τ by them as the range $[\bot, \tau]$.

8.2 Static analysis

Figure 10 shows the overall results for TIE. Figure 10(a) shows the type intervals with and without inter-procedural analysis are roughly correlated. However, the inter-procedural analysis has shorter type intervals because it has more chance to refine types. Figure 10(b) represents the position of real types within the type interval normalized as $[-0.5, 0.5]$. As the position is closer to the bottom, the real types are closer to the lower bound in type lattice. As shown in Figure 10(b), for the both cases, the real types are closer to the lower bound, which is the most specific type in the type interval. However, the inter-procedural analysis has the real types closer to the middle of type interval. Thus, Figure 10 tells the inter-procedural analysis provides tighter type interval from the real types at center.

Figure 12 shows the per-program breakdown on conservativeness and precision for TIE and Hex-rays on the test suite. In the intra-procedural case our inferred type is 28% more precise than Hex-rays, and with inter-procedural analysis we are 38% better. While Hex-rays algorithm is proprietary, it appears that in many cases Hex-rays seems to be guessing types, e.g., any local variable moved to `eax` is a signed integer. TIE, on the other hand, is a significantly more principled approach.

We investigated manually cases where TIE inferred an incorrect bound. We found that one of the leading causes was typing errors in the original program. For instance, in the function `decimal_ascii_add` of `getlimits`, a variable of signed inter stores the return value of `strlen`, but the type signature for `strlen` is unsigned.

Structures. Structure are challenging to infer because we must identify the base pointer, that fields are being accessed, the number of fields, and the type for each one. Figure 11 breaks out the conservativeness and precision for only structural types. TIE's accuracy is conservative 90% on structural types, while Hex-rays is less than 45%. TIE's precision of about 1.5 away from the original C type, which is about 200% better than Hex-rays. We conclude that TIE identifies structural types significant more conservatively and precisely than Hex-rays.

8.3 Dynamic analysis

Table 2 shows the result of TIE and REWARDS with dynamic analysis. The coverage column measures how many variables are typed. As expected, a dynamic approach only infers a few variables since only a single path is executed. Unlike TIE, REWARDS guesses a variable has type `uint`n`_t` when no type information is available, which reduced overall precision since REWARDS would mis-classify signed integers, pointers, etc. as unsigned integers. We modified REWARDS to more conservatively use type `reg`n`_t` to get a best case scenario for REWARDS (called REWARDS-c in the table). However, in all cases TIE is more precise (i.e., has a lower distance to the true type) and is more conservative than REWARDS.

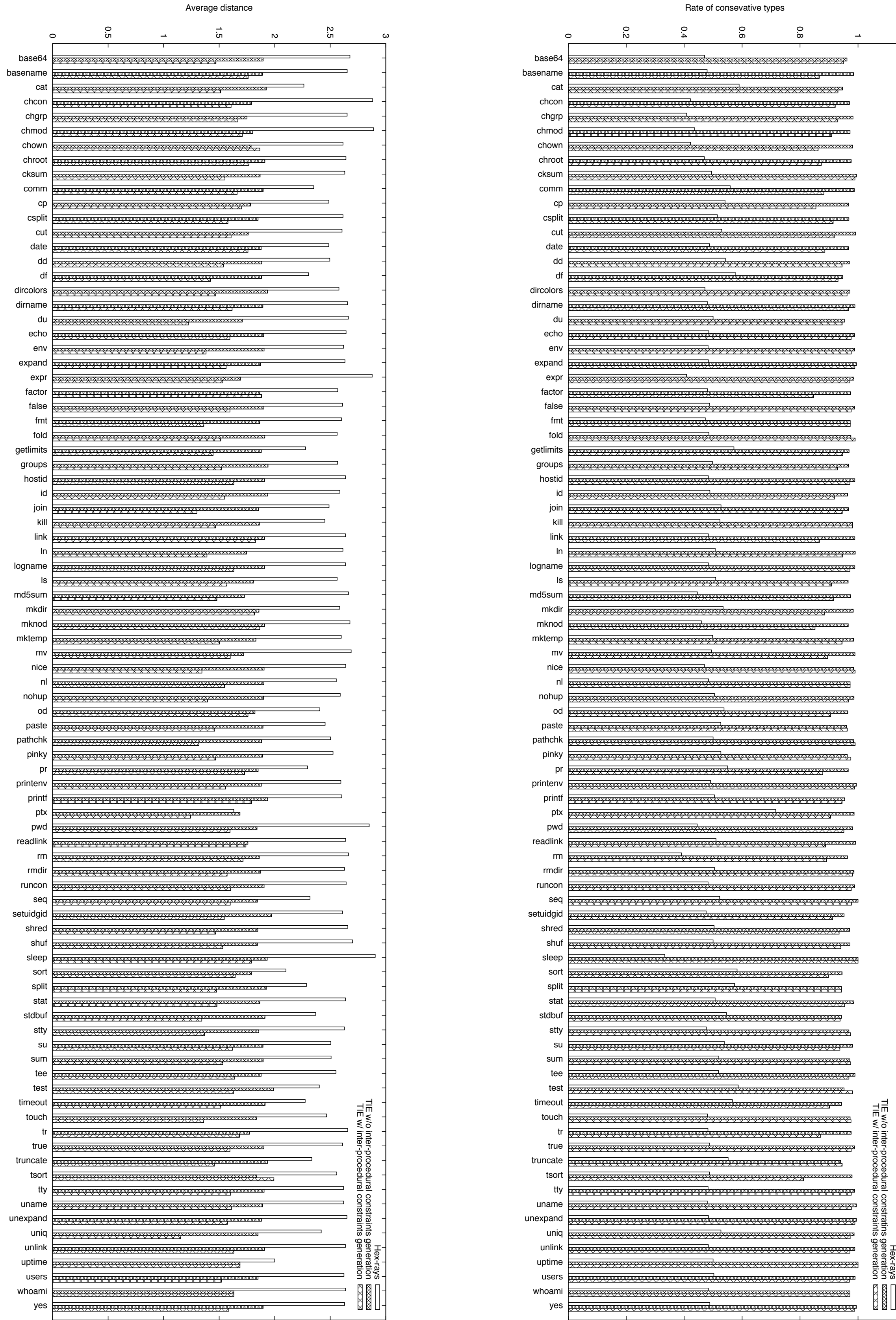


Figure 12. Overall results summary for precision measured by distance of inferred type to real type (left) and conservativeness (right).

		TIE (dynamic)		REWARDS		REWARDS-c		TIE (static w/ 100% coverage)	
Program	Coverage	Conserv.	Distance	Conserv.	Distance	Conserv.	Distance	Conserv.	Distance
chroot	9.23 %	1.0	0.88	0.56	2.3	1.0	1.42	0.87	1.76
df	6.9 %	1.0	1.39	0.46	2.73	1.0	1.62	0.942	1.42
groups	11.11 %	1.0	1.47	0.48	2.22	0.96	1.7	0.93	1.52
hostid	6.89 %	1.0	0.92	0.71	1.82	1.0	1.25	0.97	1.63
users	9.52 %	1.0	1.09	0.73	1.87	0.95	1.64	0.97	1.51
average	8.73 %	1.0	1.15	0.59	2.19	0.98	1.53	0.93	1.57

Table 2. TIE vs. REWARDS with dynamic analysis. TIE is both more precise and more conservative.

A more interesting metric is comparing TIE using static analysis of the entire program against REWARDS using dynamic analysis of a single path. This is interesting because one of the main motivations for REWARDS dynamic approach is the hypothesis that static analysis would not be accurate [12]. We show that in our test cases static analysis is about as precise as REWARDS but can type 100% of all variables. We conclude that a static-based approach can provide results comparable to dynamic analysis, while offering the advantages of working on multiple paths and handling control flow.

9 Related Work

Type reconstruction. Our approach to reverse engineering using type reconstruction is founded upon a long history of work in programming languages. Using type inference to aid decompilation has previously been proposed by others, e.g,. [16, 10, 19]. However, previous work typically tries to infer C types directly, while we use a type range and a type system specifically designed to reflect the uncertainty of reconstruction on binary code. Further, to the best of our knowledge previous work has not been implemented and tested on real programs.

As mentioned, the REWARDS system [12] takes a dynamic approach to reverse engineering data types. We compare TIE to REWARDS [12] because it is among the latest and most modern work in the area. We also compare our system against the Hex-rays decompiler, which is widely considered the industry standard in decompilation including data abstraction reverse engineering.

Semantic Types. REWARDS [12] calls the types printed by their system "semantic types". A semantic type in REWARDS is a manually specified name for a type signature, e.g., a structure may have the semantic type `sockaddr_t` instead of the complete type signature for the structure. Type ascriptions are useful for printing out complex types, e.g., printing out `sockaddr_t` as the type of variable is less cumbersome than printing out the full type signature `struct {short, ushort, ushort char[14], char[8]}`.

REWARDS has code that ascribes their pre-defined types to the arguments of pre-defined common library functions. The ascribed types are propagated like any other type. For example, the type signature for `open` has return type `int`. Whenever REWARDS sees a call to `open`, REWARDS assigns the return variable the ascribed type name `file_d` instead of the actual type `int`. If the return variable is then assigned to another variable the new variable also gets the ascribed type. REWARDS has manually defined about 150 different type names and manually ascribed those types to about 84 standard library calls.

Type ascription is simple to add as part of type inference, e.g., the rules can be found in standard type theory textbooks such as Pierce [17]. While ascribing manually-defined type names to function arguments as in REWARDS would certainly make TIE a better tool, it adds no power to the overall type system thus is left outside the scope of this work.

Variable Recovery. Our approach for recovering variables is based upon Balakrishnan *et al.* [6, 7]. However, their work only recovers variable locations, and does not infer types. While TIE could plug in DIVINE [7] to recover variables, we use our own algorithm based upon data flow lattices. BAP and the BIL language are redesigns of the Vine [3] changed to follow the formal semantics from [1].

Typed Assembly Language. Despite the similarity in name, Typed Assembly Language (TAL) [14, 15, 9] addresses a different challenge that ours. The idea of TAL is to maintain user-provided types for type-safe programs down through code generation in the compiler in order to find program bugs. After code generation the types are thrown away. Our problem is to infer types on unsafe programs (e.g., C types) from binary code. However, type reconstruction for TAL types is an open, related, and interesting problem [18].

10 Conclusion

In this paper we presented an end-to-end system for reverse engineering data abstractions in binary code called

TIE. At the core of our system was a novel type reconstruction algorithm for binary code. Unlike previous approaches in research such as REWARDS [12], which are limited to dynamic analysis of a single execution trace, we handle control flow and thus are amenable to both a static and dynamic analysis setting. We do so while providing a more precise yet conservative type than REWARDS. Our implementation also shows that TIE is about significantly more precise than the leading industry product (the Hex-rays decompiler). We conclude that our type reconstruction techniques and approach is a promising alternative over current methods in research and practice.

11 Acknowledgements

We thank Zhiqiang Lin and Dongyan Xu for the REWARDS code and their helpful comments. We also thank William Lovas for his advise and our anonymous reviewers for their useful comments and suggestions. This work was supported by the National Research Foundation of Korea Grant funded by the Korean Government [NRF-2009-352-D00282]. This material is also based upon work supported by the National Science Foundation under Grant No. 0953751. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Binary analysis platform (BAP). http://bap.ece.cmu.edu.

[2] The IDA Pro disassembler and debuger. http://www.hex-rays.com/idapro/.

[3] BitBlaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[4] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[5] G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin at Madison, Aug. 2007.

[6] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, 2004.

[7] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. In *Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation*, Jan. 2007.

[8] D. Brumley. Carnegie mellon university security group. http://security.ece.cmu.edu.

[9] K. Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, 2003.

[10] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105 – 119, Mar. 2009.

[11] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

[12] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2005.

[14] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Language Systems*, 21(3):527–568, 1999.

[16] A. Mycroft. Type-based decompilation. In *European Symposium on Programming*, Mar. 1999.

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

[18] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[19] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, 1999.