Automatic Type Reconstruction in Disassembled C Programs

K. Dolgova
Institute for System Programming
Russian Academy of Sciences
25, B. Kommunisticheskaya, Moscow, Russia
katerina@ispras.ru

A. Chernov
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
cher@unicorn.cmc.msu.ru

Abstract

This paper presents an algorithm for automatic type reconstruction from target assembly code compiled by a C compiler. The primitive language types are recovered by an iterative algorithm, which operates over the lattice of primitive types' properties. Layout of composite types is reconstructed by building set of accessible offsets for each composite type. The algorithm is the essential part of a tool for program decompilation being developed by the authors.

1 Introduction

In the last years there is an increasing need for tools to help programmers and security analysts understanding executables. For instance, the military and companies use outsourcing development of some components for their final products in order to reduce the cost of software development. They are interested that such components do not perform malicious code. Also there are a lot of old executables without source code that need to be improved. So decompilation is an actual problem in reverse engineering.

In the scope of this work we consider C as the high-level language and i386 assembly language as the low-level language. The long-term goal of our work is to develop decompilation tool translating executables to C language. Currently we are focused on a simpler problem of translating assembly code generated by a C compiler from a C program back to C.

As C itself has low-level features like union, pointer conversions, etc we conseder a limited subset of the C language called *strict C*. The restrictions are as follows: 1) explicit casts `(Type) Expr` are disallowed, 2) unions are disallowed.

The decompilation problem can be performed in the following steps: 1) detecting function in instruction flow, 2) detecting parameters and return value of functions, 3) reconstruction of high-level language structures (conditions, loops, etc), 4) reconstruction of high-level language types.

The first three aspects of the decompilation problems are well-covered both in theory [7] and in practice [1], while the latter aspect of decompilation, namely type reconstruction, has had little attention.

The remainder of the paper is organized as follows: section 2 discusses related work, section 3 outlines our algorithm, section 4 describes an algorithm for reconstruction of basic types, and section 5 describes an algorithm for reconstruction of composite types. In the conclusion we state the primary results and outline the directions of future work.

2 Related Work

There exist not many works on type reconstruction during decompilation. One of the most closely related to our algorithms is algorithm VSA presented in [3]. However, the goal of VSA is value propagation and certain attributes such as size and memory layout of structure types are recovered as a side effect. The method presented in [2, 4] addresses the problem of recovering variable-like entities when analyzing executables by VSA, a combined numeric-analysis and pointer-analysis algorithm, and aggregate structure identification, an algorithm to identify the structures of aggregates [3]. These algorithms are based on subdividing the memory-regions of the executables into variable-like entities called *a-loc*. The intuition behind this approach is that data access patterns in the program provide clues about how data is laid out in memory. This method is based on interval analysis and does not join separated uses of the same structure type, because the main purpose of the VSA is security vulnerability analyses, not data type reconstruction. Also no attempt is performed to reconstruct built-in types of structure fields and arrays.

Mycroft [9] gives a method of recovering types of variables during program decompilation. He presents a unification-based algorithm for performing type reconstruc-

1095-1350/08 $25.00 © 2008 IEEE
DOI 10.1109/WCRE.2008.20

tion. For instance, when a register is dereferenced with an offset of 4 to perform a 4-byte access, the algorithm infers that the register holds a pointer to an object that has 4-byte fields at offset 4. The type system uses disjunctive constraints when multiples type reconstructions from a single usage pattern are possible. However, Mycrofts algorithm has several weaknesses. It is unable to recover information about type signedness. In case of conflicts unions are created that makes the resulting C program unreadable. Also conflicting terms are not deleted from the working set that may cause rapid growth of the working set making the algorithm inefficient in memory and time. The working set is unlimited and in many cases the algorithm does not converge.

Past work on decompiling assembly code to high-level language is also relates to our goals [6, 7, 5]. However, that works have not done much to address the problem of recovering information about types of variables.

3 Algorithm Outline

The ideas behind our algorithm may be outlined as follows.

1. A variable in a program may hold any value from the value set determined by the type of the variable. But the type of the variable does not change between locations of the source code satisfying our strict C language subset. So the program is actually a constant expression over types, therefore we apply a kind of constant propagation data-flow analysis algorithm to the domain of datatypes.

2. Type of a variable is constrained by instructions used for processing the variable value, registers used to store it and the program environment (for example, standard library functions) affecting it. These constraints may work as flow functions for type propagation.

3. Assume that each memory access may be represented in the form $(base + offset + \sum_{j=0}^{n} C_j x_j)$, where $C_j x_j$ are multiplicative components of address expression. However, multiplication in address expressions corresponds to array element access in high-level language programs. For the strict C subset we may assume, that `a[i]` and `a[0]` have the same type for any array `a`, thus multiplicative components of address expressions are irrelevant for type reconstruction and may be dropped. Note, however, that certain vagueness exists for arrays of element size 1 (for example, character buffers or strings).

4. After dropping multiplicative components of address expressions, the assembly program contains finite number of constant offsets. Those offset are treated as structure field offsets. Thus pairs $(base + offset)$ are handled much like simple scalar variables on stack, in static area or in registers, as described above. The whole structure type may be recovered by collecting all the offsets used with the same base.

5. Assume that C_j are constants. The constants used in address expression give us information about array element sizes. Let m be $\min_{j=0}^{n} C_j$, then we may assume, that m is the size of the array elements. Situation complicates a bit, if expressions like `a[2*i]` are used in the C program, but it can be resolving by calculating $\hat{m} = GCD(m_1, \ldots, m_n)$, where m_i are mins of array access expressions of the array `a`.

6. Arrays of structs can be automatically recovered using the following heuristics. Let m be the size of the array element. If there exists memory accesses into the same array $(base + offset_1)$ and $(base + offset_2)$, and $|offset_1 - offset_2| < m$, then the array element is actually a structure.

4 Basic Type Reconstruction

In this section we limit the set of reconstructed types to the set of basic C types and the generic pointer type (`void*`). The structure of memory blocks reconstructed into structure and array types (or their combination) is reconstructed by a memory markup algorithm described in the next section.

Let $\Omega_0 = \{$ unsigned char, char, unsigned short, $\ldots\}$ be the set of basic scalar types of the C language and the "pseudo"-type *void*. It is used as a special datatype as described below. Let $\Omega_1 = \{\text{void*}\}$ be the set consisting only of the generic pointer type. The full set of reconstructed types is $\Omega = \Omega_0 \cup \Omega_1$. Each type T from the set Ω is identified by three attributes: $core \in \{\text{integer}, \text{pointer}, \text{float}\}$, $size \in \{1, 2, 4, 8\}$, and $sign \in \{\text{signed}, \text{unsigned}\}$, i. e. type T is represented by a triple $T = \langle \tau^{core}, \tau^{size}, \tau^{sign} \rangle$. The *void* type is $void = \langle \emptyset, \emptyset, \emptyset \rangle$.

Certain C types may have indistinguishable attributes on some platforms. For example, both *int* and *long* are $\langle \{\text{integer}\}, \{4\}, \{\text{signed}\} \rangle$ on i386. However, both of them remain in the Ω set to maintain platform-independency. The decompiler may choose any of them still maintaining the correct decompilation property.

We assume, that pointers are stored and operated as 32-bit unsigned integral quantities. If the size of an object is less than four bytes, such an object cannot hold a pointer value. Any pointer type has attributes $\langle \{\text{pointer}\}, \emptyset, \emptyset \rangle$ and its *size* and *sign* attributes are ignored.

The primary goals of the basic type reconstruction algorithm are as follows: 1) determine, whether an object holds a pointer, floating-point, or integral value, 2) determine the size of the value stored in object, and 3) determine, whether an object holds signed or unsigned value.

4.1 Analysis objects

Certain platforms (especially, i386) have a very limited set of the general purpose CPU registers and they are often reused. Thus, the sole register name (e.g. `%eax`) is ambiguous. One proposed solution is to use SSA representation for registers, but this implies use of φ-functions and a number of copy operations. Instead we use register objects corresponding to connected components of the DU-chains (also called *webs* [8]) graph for the CPU registers.

Objects for type analysis are extracted from the following assembly program constructs: 1) CPU register webs. For simplicity we denote them just as register names, e.g. `%eax`. 2) Memory locations at absolute addresses (global variables). 3) Memory locations at fixed offsets from the current stack frame pointer register, e.g. `-2(%ebp)` (local variables). 4) Memory locations at fixed offsets from the stack pointer register `%esp` and pushed on stack by corresponding instructions (subroutine parameters).

Indirect memory accesses are also considered as objects, on the later iterations of type recovery algorithm.

5) Object dereference (corresponds to pointer dereference in C programs). 6) Offsetted object dereference (corresponds to array or field access via pointer in C programs).

Each object obj_i is characterized by a type $\Theta_i \in \Omega$. This is the type used for the object in the source C program, and the Θ_i type is called an *ideal type*. The decompiler may reconstruct another type $T_i \in \Omega$, provided that T_i is semantically equivalent to the Θ_i with respect to the decompiled program. The type T_i is called *reconstructed* type and denoted as $obj_i : T_i$.

The reconstructed types for the object of the assembly programs are computed using an iterative algorithm for each attribute τ^{core}, τ^{size}, τ^{sign} of types.

4.2 Type Dependency Tree

A type dependency tree is a tree, similar to an expression tree. A type dependency tree contains operations from the assembly program or objects in its nodes, only objects in its leaves. Edges correspond to value use dependency. Nodes corresponding to the assembly program instructions are called *instruction nodes*, all other nodes are called *object nodes*. The parent to a instruction node is the *operation result node*. The children of an instruction node are the *operand nodes*.

The tree consists of the following five instruction node types: 1) **Sequence** nodes, 2) **Evaluation** nodes, 3) **Copying** nodes, 4) **Subroutine call** nodes, 5) **Condition** nodes.

For each instruction tree node with the operand objects obj_1, obj_2 and the operation result object obj_{res} in the working set of objects *OBJ* an equation is written as follows: $obj_i : t_{n,1} \bigcirc obj_j : t_{n,2} \Rightarrow obj_k : t_{n,3}$, where obj_i, obj_j, obj_k are the objects, $t_{n,1}$,$t_{n,2}$, t_{n_3} their respective types in the n-th equation, $\bigcirc$ — operation in the instruction node. obj_i and obj_j are the operands, and obj_k is the result. All equations constitute the system of equations. Each type $t_{n,i}$ corresponds to some type T_j and thus has attributes $t_{n,i} = \langle \tau^{core}_{n,j}, \tau^{size}_{n,j}, \tau^{sign}_{n,j} \rangle$. The system of equations and the type dependency tree have one-to-one correspondence.

4.3 Constraints and join function

The type constraints are computed using properties of assembly instructions in the decompiled program.

1. $C_{reg} \models$ is a *register constraint*. It affects the *core* and the *size* attributes of the type variable.
2. $C_{cmd} \models$ is an *instruction constraint*. It affects the *core*, the *size*, and the *sign* attributes of involved type variables.
3. $C_{flag} \models$ is a *flag constraint*. It affects the *sign* attribute of involved type variables.
4. $C_{env} \models$ is an *environment constraint*. It affects the all type variable attributes.

The join function $\sqcup$ for types is defined as follows. A type is described by three attributes *core*, *size*, and *sign*, so let $\sqcup^{core}$, $\sqcup^{size}$, $\sqcup^{sign}$ be the join functions for corresponding components. Then for $T_1 = \langle \tau_1^{core}, \tau_1^{size}, \tau_1^{sign} \rangle$ and $T_2 = \langle \tau_2^{core}, \tau_2^{size}, \tau_2^{sign} \rangle$ $T_1 \sqcup T_2 = \langle \tau_1^{core} \sqcup^{core} \tau_2^{core}, \tau_1^{size} \sqcup^{size} \tau_2^{size}, \tau_1^{sign} \sqcup^{sign} \tau_2^{sign} \rangle$. Define $\sqcup^{core} \equiv \cap$, $\sqcup^{size} \equiv \cap$, $\sqcup^{sign} \equiv \cap$, i. e. the three join functions are all set intersection. The three join functions are monotone.

4.4 Tree Traversal Rules

For the type dependency tree we define two types of traversal: 1) bottom-up traversal, and 2) top-down traversal.

During the bottom-up traversal the information about the object types flows from operands to the results of operations. From the system of equation point of view this is left-to-right propagation. During the top-down traversal the information about the object types flows from the operation result to the corresponding operands. From the point of view of system of equations this is right-to-left propagation.

Currently we do not consider interprocedural analysis, so we distinguish between two types of tree nodes: the nodes, which propagate type information and the ones, which do not propagate. If a node propagates type information, then during the bottom-up traversal new information about types is propagated from the operands to the result, and during

the top-down traversal new information about types is propagated from the result to the operands.

Consider the type propagation rules for **bottom-up** traversal. *Binary operation* is demonstrated with an example of subtraction. Let the equation be $e_1 : t_{e_1} - e_2 : t_{e_2} \Rightarrow e_{res} : t_{e_{res}}$. Define the type propagation rules for the binary subtraction operation for the *core* attribute as follows.

$$e_1 : t_1 - e_2 : t_2 \Rightarrow e_{res} : t_{res} \models$$

$$\frac{\text{pointer} \in \tau^{core}_{e_1}, \text{pointer} \in \tau^{core}_{e_2}}{\text{integer} \in \tau^{core}_{e_{res}}},$$

etc.

Define an *upper intersection* operation $\sqcap$ on two set of integers as follows. If S_1 and S_2 are sets of integers, the upper intersection $S = S_1 \sqcap S_2$ is defined as $S = \{s | s = \max(s_1, s_2) : \forall (s_1, s_2) \in S_1 \times S_2\}$. For example, for sets $\{1,2\}$ and $\{2,4\}$ the upper intersection is $\{2,4\}$. The propagation rule for the *size* attribute is defined as $\tau^{size}_{res} = \tau^{size}_{e_1} \sqcap \tau^{size}_{e_2}$.

The propagation rules for the binary subtraction operation for the *sign* attribute are defined similarly to the *core* attribute as follows.

$$e_1 : t_1 - e_2 : t_2 \Rightarrow e_{res} : t_{res} \models$$

$$\frac{\text{unsigned} \in \tau^{sign}_{e_1}, \text{signed} \in \tau^{sign}_{e_2}}{\text{unsigned} \in \tau^{sign}_{res}},$$

etc.

Consider the type propagation rules for unary operation during bottom-up traversal. If an equation has the form $\bigcirc e_1 : t_1 \Rightarrow e_{res} : t_{e_{res}}$, the rule is $t_{e_{res}} = t_{e_1}$.

If a *copy* operation has the form $e_1 : t_{e_1} \rightarrow e_2 : t_{e_2} \Rightarrow e_{res} : t_{e_{res}}$, the type propagation rule for it is $t_{e_{res}} = t_{e_1} \sqcup t_{e_2}$.

For *dereference* operation the type propagation rule is $\tau^{core}_{res} = \tau^{core}_{e_1} \cup \{\text{pointer}\}$, if e_1 is used for memory dereferencing. The τ^{size}_{res} and τ^{sign}_{res} are unchanged.

Rules for **top-down** traversal are basically similar to those of bottom-up traversal. However, during the top-down traversal *alternate* attribute values may appear due to the type balancing rules of the C language. For example, the result of binary operation is unsigned, if at least one operand is unsigned. When type information is propagated from the result to the operands all the possibilities should be considered.

4.5 Basic Type Reconstruction Algorithm

The working set of objects *OBJ* consisting of *N* objects $OBJ = \{obj_1, \ldots, obj_n\}$ is given as the parameter of the algorithm.

The algorithm is presented below. The algorithm converges, as it manipulates the finite set of objects and each object's properties are finite lattices with monotone join functions.

```
Tree = make_tree(program);
Equation = make_equation(Tree);
init (Equation);
set_constraint(Equation);
flag = true;
while (flag) :
        flag = false;
        for all equation: Equation :
                spread_info_lr(equation);
        for all obj:OBJ :
                flag ||= left_join(Type(obj));
        for all equation: Equation :
                spread_info_rl(equation);
        for all obj:OBJ :
                flag ||= right_join(Type(obj));
        for all obj:OBJ :
                flag ||= full_join(Type(obj));
for all obj:OBJ :
        Tobj = match_type((Type(obj));
```

5 Composite Type Reconstruction

Assume, that each memory access may be represented in the form $(base + offset + \sum_{j=0}^{n} C_j x_j)$. This assumption holds provided the source C program is written in the strict C as stated above. Let m be $\min_{j=0}^{n} C_j$ and M be $\max_{j=0}^{n} C_j$. *base* is initially a register and ultimately an object. *offset* is a constant, as well as C_j. Later this memory access expression will be transformed into an object expression in form $(obj + offset)$.

The composite type reconstruction stages are as follows.

Assign_labels. Let l be labels chosen from an enumerable set. Let assign labels to the following elements of the assembly program: 1) memory accesses: $l_i : (b_i, o_i, m_i, \ldots, M_i)$; 2) return values of subroutines (content of `%eax`). Statements that supply pointer values to the program are a subset of the labeled program elements, because some labeled elements may give non-pointer values.

Build_reg_label_sets. Let $LS_{in}(reg, n)$ be a set of labels, which correspond to values that can be stored in the register *reg* at the point immediately preceding the program line *n*. Similarly, let $LS_{out}(reg, n)$ be a set of labels for the register *reg* at the point immediately following the program line *n*. The *label-set analysis* determines the *LS* sets for each register and each line of the assembly program. It is a forward iterative data-flow analysis with set union as the join function. Since the set of assigned labels is finite, the label-set analysis stops after a finite number of iterations. For simplicity we further omit $_{in}$ qualification of *LS* sets. Since the

label-set analysis is performed only for registers, no aliasing problem appears, as registers cannot be aliased.

Build_label_equiv_sets. Based on the computed label-sets the equivalence relation between labels is established as follows:

$$\frac{\exists l_1, l_2, reg, n : \{l_1, l_2\} \subseteq LS(reg, n)}{l_1 \equiv l_2} \quad (1)$$

$$\frac{l_1 : (b_1, o_1, m_1, \ldots, M_1), l_2 : (b_2, o_1, m_2, \ldots, M_2), b_1 \equiv b_2}{l_1 \equiv l_2} \quad (2)$$

Equality of m_1 and m_2 and M_1 and M_2 is not required in rule 2. The defined relation is reflexive, symmetric and transitive by construction. Thus, all labels are separated into equivalence classes, and an arbitrary label is chosen as the representative for each equivalence class. An equivalence class corresponds to a data type in C. Let t_j be a representative of j-th equivalence class. For each memory access $(b_i, o_i, m_i, \ldots, M_i)$ its base b_i may be replaced by the representative of the b_i equivalence class.

Two labels l_1 and l_2 have *common base*, if $l_1 : (b_1, o_1, \ldots)$, $l_2 : (b_2, o_2, \ldots)$, and $b_1 \equiv b_2$.

Build_aggreg_sets. Now define an *array aggregation* relation between the labels, denoted $AA(l_1, l_2)$, by the following inference:

$$\frac{l_1 : (t_1, o_1, m_1, \ldots, M_1), l_2 : (t_1, o_2, m_1, \ldots, M_2), |o_1 - o_2| \leq m_1}{AA(l_1, l_2)} \quad (3)$$

Array aggregation is also an equivalence relation inducing equivalence classes. Let $\{l_1, l_2, \ldots, l_m\}$ be a set of labels of some equivalence class. Let $o = \min_{j=1}^{m} o_j$. Denote $t' = \langle t_1, o \rangle$ as the label for an array of an inner structure, where $\langle b, o \rangle$ denotes offsetting from the given base, but without dereferencing. The labels $l_1, \ldots l_m$ are rewritten as $l_i : (t', o_i - o, m_i', \ldots, M_i)$, where m_i' is the second least multiplier in the memory access expression, and the least multiplier is removed. The array aggregation operation is repeated while there exist array aggregation equivalence classes with more than one representative. The aggregation does not change the base of labels, so if l_1 and l_2 have the common base, after aggregation they still have the common base.

Reconstruct_structs. Finally, structure types may be reconstructed using the computed information. Let S be the set of all labels. The labels may be divided into equivalence classes $S_1, S_2, \ldots, S_m$ based on the common base relation, i.e. each set contains the labels with the common base. Let O_i be the set of immediate offsets used in labels from S_i. Those offsets are considered to be field offsets in a new structure type t_i corresponding to the label set S_i.

Update_object_sets. For all reconstructed structure fields the corresponding objects are created and added to the set of the working objects for the basic type reconstruction algorithm. The basic type reconstruction algorithm then reconstruct types for the object corresponding to the structure fields.

6 Conclusion

In this paper we presented an algorithm for automatic type reconstruction in assembly programs generated by C compilers. Algorithm performs both precise reconstruction of basic types and structure types provided that certain restrictions on the source C programs are met. The presented algorithm converges as operates over finite lattices of built-in type properties and finite sets of labeled memory accesses.

The proposed algorithm is the essential part of a tool for program decompilation being developed by the authors.

Currently we are working on gaining statistics of application of the algorithm to real-world examples. The directions of further research include on enhancing the static analysis (as presented in this article) with the results obtained from tracing of the program being decompiled.

References

[1] Hex-rays decompiler sdk, http://www.hex-rays.com/.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses x86 executables. *Compiler Construction*, 2985:5–23, February 2004.

[3] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. *Verification, Model Checking, and Abstract Interpretation*, 4349:1–28, November 2007.

[4] G. Balakrishnan and T. Reps. Improved memory-accesses analysis for x86 executables. *Compiler Construction*, 4959:16–35, April 2008.

[5] C. Cifuentes, M. Emmerik, B. Lewis, and N. Ramsey. Experience in the design, implementation and use of a retargetable static binary translation framework. *Technical Report*.

[6] C. Cifuentes and A. Fraboulet. Interprocedural static data flow recovery of high-level language code from assembly. *Technical Report*.

[7] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. *Int. Conf. on Softw. Maint.*

[8] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[9] A. Mycroft. Type-based decompilation. *European Symp. on Programming*, 1576:208 – 223, 1999.