Profile-Based Type Reconstruction for Decompilation

K. Troshina
Institute for System Programming RAS
25 Alexander Solzhenitsyn st., Moscow, 109004, Russia
katerina@ispras.ru

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

A. Fokin
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
apfokin@gmail.com

Abstract

Decompilation is reconstruction of a program in a high-level language from a program in a low-level language. In most cases static decompilation is unable to completely reconstruct high-level data types due to loss of typing information during compilation. We present several profile-based techniques that help to recover high-level types. The techniques include pointer/integer determination by value profiling and composite type identification by heap profiling.

1 Introduction

A typical software development uses many third-party software components, which are often provided without source code. Also it is a common situation when some legacy application runs for years and no source code is available and there are no means of obtaining it. In both cases it is sometimes necessary to fix bugs or adapt the components without source code to changing requirements.

Such problems are addressed by reverse engineering. A typical reverse engineering track starts from executable files and goes through an assembly program, high-level program (for example, in C programming language) and ends on specification level. The task of obtaining a high-level program from an assembly program is solved by decompilation.

We assume that the source program, which is not available for reverse engineering, is written in C and compiled by some (unknown to us) C compiler. Existing decompilers to the C language often fail to reconstruct data types correctly and produce C programs with explicit type casts and unions, even if the source program does not have such. A good decompiler should try to reconstruct as many as possible of data types.

Often it is not possible to reconstruct all data types precisely using only static decompilation techniques. At least pointer and integer variables can not be unambiguously reconstructed if there was no dereference. The article is devoted to discussion of several heuristics which may provide hints for the static analysis in cases of ambiguity.

One of the closely related to our static type reconstruction algorithms is algorithm VSA presented in [1]. 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. Mycroft [5] gives a method of recovering types of variables during static program decompilation. He presents a unification-based algorithm for performing type reconstruction.

As static analysis is not sufficiently powerful to reconstruct all types, dynamic (profiled-based) information may be useful to resolve ambiguities.

Dynamic inference of abstract types is presented in [3]. The goal of inference of abstract types is to obtain finer-grained type information than use of single program language types.

Run-time type checking for compiled C programs is described in tool *Hobbes* [2]. This tool is developed for identifying defects in C program caused by improper use of unsafe language features.

Unfortunately, program decompilation based on profile-based information is not yet well researched.

The remainder of the paper is organized as follows. Brief description of static type reconstruction algorithm is given in section 2, section 3 is devoted to profile-based type hinting. Section 4 presents experimental results.

2 Static Type Reconstruction

Type reconstruction is an essential part of decompilation, algorithms used in the other stages of decompilation are out of scope of this work.

Objects for our type analysis algorithm are connected components of the DU-chains (also called *webs* [4]). It allows handling reuse of the limited number of the hardware registers. We prefer DU-chains over SSA representation to avoid excessive copying in join sites. DU-chains are created for CPU registers, memory locations at fixed offsets by the stack frame and stack pointer and memory locations at fixed addresses and object dereferences corresponding to pointer deferences in C programs, and offseted object dereferences corresponding to array or field accesses in C programs.

Our static type reconstruction algorithm consists of two phases: 1)basic type reconstruction and 2)composite type reconstruction.

The primary goals of the basic type reconstruction algorithm are determine if an object holds a pointer, floating-point or integral, determine the size of the integral and floating-point type, stored in an and for integral type determine if an object holds a signed or unsigned.

The set of the types being reconstructed is the set of the basic C types and the generic pointer type `void *`. Each type is identified by three attributes: *core*, *size*, and *sign*. A correspondence between the C types and the triples of attributes is established.

A system of equations is written for an assembly program as follows. The instruction *add* r_1, r_2, r_3 is mapped to an equation $obj_3 : T_3 = obj_1 : T_1 + obj_2 : T_2$, where objects obj_3, obj_1, obj_2 were formed for registers r_1, r_2, r_3 and types T_1, T_2, and T_3 are being reconstructed. The types of the objects are computed by an iterative algorithm for each attribute. The initial values for type attributes are taken from constraints. There exist four types of constraints: instruction, flag, register, and environment. For computing object types we use a subset lattice over types attributes. The type attribute values are joined by the set intersection operation, so the join operation is monotone. If the attribute set for some type become empty, a conflicting object usage is detected.

We use a *canonic form* of any address expression for composite type reconstruction as follows: $(base + offset + \sum_{j=1}^{n} C_j x_j)$,

where *base* is the base address, which is an untraceable value, *offset* is a constant (numeric) offset, x_j are untraceable values, C_j are constants. Without loss of generality, assume that $0 < C_1 < C_2 < \ldots < C_n$. *A multiplicative part* of the address expression is $\sum_{j=1}^{n} C_j x_j$. In code generated from strictly conforming C programs *base* always has some pointer type.

Backward propagation algorithm is executed to build address expression terms for memory access instructions. Then all address expression terms are divided into sets of terms with equivalent bases. Each of such sets corresponds to one composite type.

To build disjoint sets of type-equivalent memory base addresses a forward label propagation algorithm is used. If the number of disjoint sets is greater than the number of composite types in the original C program, the static composite type reconstruction algorithm is unable to reveal equivalence of bases. This issue is addressed by use of profile hints and described later.

After this, the offsets from equivalent bases are collected. A skeleton of a C structure is constructed from the set of offsets, where types of structure fields still need to be reconstructed. The new objects are constructed for the structure fields and added to the working set of the basic type reconstruction algorithm.

Arrays are reconstructed in a similar way. If all offsets from base set have the same multiplicative component, an array is identified. A new object is created for the first array element.

For the arrays the GCD of multiplicative constants in address expressions is assumed to be the size of the array elements. In some cases it is possible to reconstruct the array size.

The overall structure of basic and composite type reconstruction algorithm is shown in Fig. 1.

The initial object set is obtained from the assembly program. Than it is passed to the module that implements basic type reconstruction algorithm. As the result of its executing we get reconstructed basic C types, pointers (indirect objects), and not reconstructed types. The set of indirect objects (pointers) is passed to the composite type reconstruction module, which builds skeletons of composite types and makes new objects for structure fields and array elements. This set is merged with set of objects with not reconstructed types and is passed back to the basic type reconstruction module. Objects that correspond to structure fields or array elements may be also indirect. Algorithm stops when the set of unrecognized objects and pointers set become empty. Algorithm converges as pointers have finite levels of indirection and basic type reconstruction algorithm operates with finite lattice (type attribute set) with monotone join function (set intersection).

Static type reconstruction algorithm does not use profile-based information. This storage is used in modified type reconstruction algorithm, as static type reconstruction is not always capable of reconstructing a datatype for a variable unambiguously.

To resolve such problems we use profile-based type decompilation hints as described below.

3 Profile-Based Type Hinting

Memory Copying and Zeroing Identification. The C standard permits byte access to objects of arbitrary types. For example, even in a strictly conforming C program it is valid to copy the contents of one structure into another structure of the same type byte by byte, and a naive implementation of `memcpy` function does so. A typical C compiler usually inlines a highly optimized for the given processor version of memory copying functions directly at call sites. From the point of view of decompilation it means that almost any function may contain memory accesses violating strict typing rules and producing conflicts during type reconstruction. Fortunately, a rare C program does low-level byte access to memory by passing the certain C functions like `memcpy`, `memmove`, `memset`, etc.

On the other hand, the compiler generates code, which copies one variable to another by moving around machine words instead of fields. This also produces multiple conflicts during basic type reconstruction.

A decompiler should identify inlined calls to memory manipulation functions and replace them with calls of the corresponding C standard function. It should greatly improve precision of type recovery algorithms. One possible approach would be a signature search with a signature database with all variants of the memory manipulation functions for different compilers and different optimization settings.

On the other hand, it is possible to monitor run-time behavior of a process and identify locations in the process code that behave like memory copying operations.

For example, the `memset` function is used to fill up the specified memory region with a specified byte value (typically 0). By monitoring memory stores it is possible to identify series of sequential writes of some fixed byte value into a contiguous memory region. The addresses of machine instructions producing this pattern are saved and then transformed into code locations in the disassembled listing. The assembly instructions at the specified locations are ignored by the type reconstruction algorithm.

The `memcpy` function also has a distinctive run-time pattern as sequential read/write operations from one contiguous memory region to another contiguous memory region (either in the ascending order of memory addresses or in the descending order). The corresponding code addresses are also mapped to the code locations in the disassembled listing and ignored by the type reconstruction algorithm.

Type Reconstruction by Value Profiling. At runtime it is possible to determine whether the register value is pointer with a reasonable level of confidence. Similarly we may determine whether a variable holds signed or unsigned integer quantities.

This can be supported by the following empirical observations:

1. Pointers hold either values, which are valid addresses in the virtual address space of the process, or zero.
2. Signed integer variables take positive and negative values grouped around zero. -1 is more expected value for a 32-bit signed quantity, than, say, $-2 \cdot 10^9$.
3. Unsigned integer variables rarely take values corresponding to -2 or -3, and, say, $3 \cdot 10^9$ is more expected values for a 32-bit unsigned quantity, than value corresponding to -2. The unsigned value corresponding to -1 is considered as a valid unsigned value as because 1) it corresponds to the maximum unsigned value, 2) it is often used as an error indicator.

On a 32-bit architecture we need to profile 32-bit quantities, as pointers have 32 bits. Quantities of other sizes are of less interest. The first issue is selection of objects for value profiling. We might want to directly profile memory locations (memory addresses), but this is not practical due to the following reasons: 1)different variables may be stored at the same address, 2)stack memory as well as heap memory (to lesser extent) is heavily reused and 3)a local variable or a structure field may have different addresses at different instants.

However, in order to be used in computation a value has to be loaded onto a register by a memory load instruction. We assume that no distinct operations in the source C program are mapped into one memory load instruction, so the memory load instruction manipulates with values of one type, provided that the corresponding operation in the source program manipulates with values of one type. Thus, instead of profiling memory addresses, memory load instructions are profiled. A value profile *VP* of a program is a mapping from the set *Addr* of virtual memory addresses in the program code segment to the set of sets of 32-bit values *Val*. Addresses that do not belong to the program code segment are excluded from the value profile.

The virtual memory addresses need to be mapped to disassembled instructions. Such a correspondence *DM* between the set *Addr* of virtual memory addresses and the set of locations *Loc* in the disassembled program is built during disassembling. So there are mappings $VP : Addr \to 2^{Val}$ and $DM : Addr \to Loc$. A mapping *IVP* from disassembled instructions to the set of sets of 32-bit values is constructed using the given mappings *VP* and *DM*. They are $IVP : Loc \to 2^{Val}$ and $IVP(x) = VP(DM^{-1}(x))$.

As a result, some memory load instructions in the disassembled program are annotated with a set of values. Non 32-bit loads and 32-bit loads, which were not covered by the profiling runs, remain unannotated.

A memory load instruction loads the value of one object obj_1 to a register web object obj_2 (see Section 2). The value profile of the object obj_1 is the union of the load instruction profiles of all memory load instructions using the object obj_1. Let $VP(obj_1)$ be the value profile for the object obj_1. On the other hand, a triple $T_i = \langle \tau_1^{core}, \tau_1^{size}, \tau_1^{sign} \rangle$ is built for each object obj_1 as the result of static type reconstruction. The reconstructed type T_i information may be refined by the profile information using the following heuristics.

Let *ValidAddr* be the set of valid virtual addresses for the profiled process. A pointer confidence *PC* value for the object obj_i is defined as

$$PC(obj_i) = \frac{|VP(obj_i) \cap ValidAddr|}{|VP(obj_i) - \{\texttt{null}\}|}$$

where `null` is the null pointer value. *PC* value of 0 means that the object is definitely not a pointer value, and value of 1 means that the object is almost surely a pointer.

To define a confidence level function for unsigned values we use the following heuristics. If the value profile contains -2, it gives confidence level of 1/2 of being signed, values -3 and -4 contribute 1/4 to the confidence level of being signed, etc. So, an unsignedness confidence value for the object obj_i is defined as

$$UC(obj_i) = 1 - \sum_{x=-2^{31}+1}^{-2} \frac{\sigma_{obj_i}(x)}{2^{2 \cdot \lfloor \log_2 |x+1| \rfloor + 1}}$$

where $\sigma_{obj_i}(x) = 1$, if $x \in VP(obj_i)$, and $\sigma_{obj_i}(x) = 0$ otherwise.

If the unsignedness confidence value is close to 1, there is no evidence that the value is signed.

The pointer confidence level function is used in cases, when $\{\text{integer}, \text{pointer}\} \subseteq \tau_i^{core}$, to help resolving integer/pointer ambiguity. The unsigned confidence level function is used in cases, when $\tau_i^{core} = \{\text{integer}\}$ and $\{\text{signed}, \text{unsigned}\} \subseteq \tau_i^{sign}$ to help resolving signed/unsigned ambiguity.

Type Identification by Heap Profiling. As discussed above structures are reconstructed from the set of offsets used in the memory access operations. Structure base addresses are grouped into equivalence classes using a disjoint-set data structure after elimination of copying and zeroing memory accesses. However, several disjoint sets may still correspond to a single high-level data type due to imprecise tracking of pointer copying.

Structure objects are typically allocated on heap using the `malloc` family of standard library calls. Such library calls are intercepted and tracked at run-time. Typically it is used to track programming errors like double freeing, but can also be used for enhancing type reconstruction as described.

An *access footprint* of a memory block is a set of addresses of machine instructions from which the memory block is accessed either for read or for write. Instructions corresponding to memory copying or zeroing are not included in the access footprint. Several memory blocks having the same access footprint are actually of the same static type. Consider two memory access operations belonging to some access footprint. Let $\langle b_1, o_1 \rangle$ and $\langle b_2, o_2 \rangle$ are bases and offsets for the first and the second memory accesses. It is possible that b_1 and b_2 are not in the same disjoint set after the static analysis as discussed above. Then, if $o_1 > 0$ and $o_1 = o_2$, then the disjoint sets of b_1 and b_2 are merged. The $o > 0$ requirement cuts off false positives that might happen if the address of the field with offset 0 is taken and used on its own. By merging two disjoint sets we may obtain a richer set of structure offsets and recover both the structure skeleton and the types of the fields with a better precision.

For the sake of simplicity let's assume that the heap blocks are never reused, thus, a base address of some heap block unambiguously corresponds to some call of a `malloc`-family function. For the real-world memory management this restriction may be remedied by introducing generations of the virtual addresses.

The memory blocks with the same access footprint are merged into one equivalence class. Let *AF* be the set of the base virtual addresses of the blocks. As discussed above each memory load operation is value profiled. If the value profile *VP* of the instruction is a subset of $AF \cup \texttt{null}$ we may safely assume that a pointer value of respective type is loaded. This hint is passed back to the static analysis.

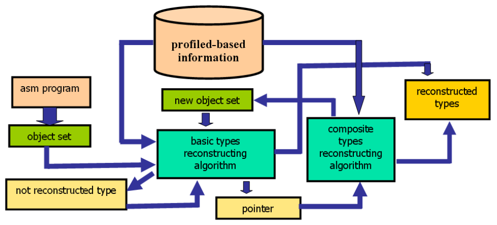
profiled-based information
asm program
object set
new object set
basic types reconstructing algorithm
composite types reconstructing algorithm
reconstructed types
not reconstructed type
pointer

Figure 1. Type reconstruction algorithm

So, static type reconstruction algorithm presented in section 2 is modified for using profile-based information as shown in the Fig. 1. Heap and value profiles are collected during one or several profiling runs and are stored in the database. They are used by the basic type reconstruction algorithm as additional type constraints affecting the *core* and the *sign* attributes of involved type variables. These constraints may indicate that type *is not* a pointer or *is not* unsigned, and are optional. Composite type reconstruction algorithm use profile-based information for building type skeletons and for more precise sets of equivalent bases.

4 Experimental results

This section discusses results of application of the described algorithm to a number of sample C programs.

We use the Valgrind [6] framework to implement profile collection for assisting the static type reconstruction algorithm.

All source code is taken from FreeBSD-4.0 distribution and compiled with gcc-4.3.2 compiler at the default optimization level on a Linux system. The `execute` function is compiled at -O2 optimization level, so all variables except one were put on the registers. Only one variables was stored in the stack frame. The summary of hand-checked code is shown in the Table 1. The 'CLOC' column shows the number of LOC in the C source, and the 'ALOC' column shows the number of LOC in the corresponding assembly listing.

Example	CLOC	ALOC	Description
`35_wc.s`	107	245	`cnt()` function from wc utility (file 'wc.c')
`36_cat.s`	27	104	`raw_cat()` function from cat utility (file 'cat.c')
`37_execute.s`	486	1167	`execute()` function from bc utility (file 'execute.c')
`38_day.s`	173	346	`isnow()` function from calendar utility (file 'day.c')

Table 1. Hand-checked sample code

The type reconstruction statistics for all examples is shown in Table 2.

	BT	Exact BT	Corr BT	Fail BT	PTRs	Exact PTRs	Corr PTRs	Fail PTRs
35	7	1	6	0	2	1	1	0
36	7	4	3	0	1	0	0	0
37	1	0	1	0	0	0	0	0
38	9	8	1	0	4	2	1	1*

Table 2. Hand-checked reconstructed type statistics

The 'BT' column in Table 2 shows the total number of stack variables of basic types. Variables taken on the registers are now counted. The 'Exact BT' column shows the number of variables, which types were reconstructed exactly, and the 'Corr BT' column shows the number of variables, which types were reconstructed correctly. The 'Fail BT' column shows the number of incorrectly reconstructed basic types. The 'PTRs' column shows the total number of stack variables of pointers to basic types. The 'Exact PTR', 'Corr PTR', 'Fail PTR' columns show the numbers of exactly, correctly and incorrectly reconstructed types respectively.

Our heap profiling and value profiling tools are implemented by using the Valgrind framework.

The value profiling metrics were collected for `gzip` program invoked on `configure` file from the Valgrind distribution. The value profile of the whole gzip application is shown in Fig. 2. As gzip operates with compression windows of size less than 2^{16}, values greater than 100 000 are relatively rarely used.

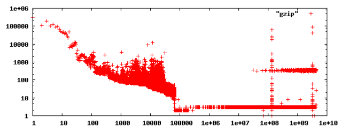

Figure 2. Value profile of gzip

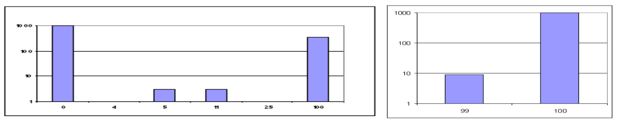

Figure 3. Distribution of the *PC* and the *UC* metrics in gzip

We computed the *PC* metrics for all memory loads and stores, and the resulting table is shown in Fig. 3. Most loads and stores are unambiguously identified as being integer operations (*PC* value of zero) or pointer operations (*PC* value of 100). Only a few integer variables occasionally take values in the pointer domain.

The results of calculation of the *UC* metrics are shown in Fig. 3. Unfortunately, this metrics works not as good, as expected. For almost all loads and stores it shows as 100. This does not mean that all variables in gzip are unsigned (however, it is so), but rather that profiling is unable to discover evident use of negative values.

We have done the benchmark programs when profiling is enabled and disabled. The program being profiled runs up to 100 times slower and requires up to 100–200 times more memory space. However, we work on improving these figures.

References

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

[2] M. Burroes, S. Fruend, and J. Wiener. Run-time type checking for binary programs. *Proceedings of CC*, pages 90–105, April 2003.

[3] P. Cuo, J. Perkins, S. McCamant, and M. Ernst. Dynamic inference of abstruct types. *Proceedings of ISSTA*, pages 749–754, July 2006.

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

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

[6] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proc. of PLDI*, June 2007.