Type Flattening Obfuscation

Ta Thanh Dinh

tathanhdinh@gmail.com

Abstract. Beside data and control flow, high-level types are also important in program analysis, indeed type recovery is an essential step in binary code decompilation. For the purpose of anti-decompilation, the paper presents a novel obfuscation technique which makes types become harder to be reconstructed.

1 Introduction

An important step in binary code *decompilation* [1] is to detect high-level objects (e.g. functions, variables) from low-level machine dependent objects [3] (e.g. registers, raw memory accesses, machine instructions) before annotate them with types of the "assumed" original language. In a statically typed language like C, the compiler doesn't preserve the high-level type information in the generated machine code, then the *type reconstruction* (or type recovery) from machine code requires special techniques, a survey up until 2015 can be referenced in [8], more recent research includes [9, 10, 11, 13].

In the opposite direction, binary code *obfuscation* is to make the decompilation (or in general code analysis) harder. Since the data and control flow are principal elements for the program analysis, current obfuscation methods [5] focus mostly on them. But to the best of our knowledge, there is still no explicit effort in obfuscating high-level types. There would be several reasons for this lack of effort.

First, type obfuscation is unsurprisingly a side effect of data or control flow obfuscation. Indeed, type reconstruction algorithms need data and control flow to build type constraints, if any of them is hidden then the algorithms cannot work correctly. Or if the function boundary is not found (because of anti-disassembly tricks, for example), then high-level objects cannot be recognized. *Second*, high-level types seem too coarse to be worthy of being protected, in many cases just knowing certain values of the input which make the program exploitable is enough. But knowledge about types expands attack surfaces because more analysis can be proceeded, beside decompilation see the survey [8] for a more complete list.

Type flattening obfuscation

Under our interest, the studies in binary code type reconstruction can be roughly divided into two directions: one is based on semantics of programs, the other is not. We focus on the former, and also on methods based on static analysis. The research in this vein starts with Mycroft [2], and improved in [6, 7, 10, 11]. One of the notable improvement is to propose some notion of *subtyping* to formally estimate the closeness of inferred types over the original types. For example, the subtyping in [6] is based on a lattice which simulates the *conversion rank* of C's integer types, e.g. `int16_t` is subtype of `int64_t`. Basically, if the inferred type is a supertype of the original one then the result, though may be still sound, is less accurate. We should note that sometimes, the supertype is the best result possible.

In this work, we limit on primitive types and their derived pointer types. Our purpose is to neutralize the subtyping relation: a type can only be inferred as its supertype. Concretely, `char`, `short`, `int`, etc. and pointers will be recognized as either `uint64_t` or `int64_t`. We call this technique *type flattening obfuscation*. Also we focus only on obfuscating types of function signatures, local variables are weaker primitives since the compiler can remove them, e.g. through optimization.

2 Compiler for type flattening

We present *uCc*, an open source C compiler that explicitly obfuscates high-level types. The compiler is implemented in Rust, the source code and a brief user guide are given at [15]; different from other code obfuscation implementations, it uses cranelift [14] as backend.

uCc attacks the core of type reconstruction algorithms, these are the data flow used to build type constraints and the *semantic gap* between types in the high-level language and their representations in the low-level machine code. Beside custom tricks, *uCc* uses extensively transformations based on Mixed Boolean Arithmetic expressions [4, 12]. To make some code diversity, the compiler is *probabilistic*: given a source code, *uCc* generates each time a different (but computationally equivalent) output machine code.

Table 1. Functions and original types

Description	Signature
identity	`int id(int)`
division	`int div(short, char)`
modular	`char mod(short, char)`
increase $p \leftarrow p + 1$ then dereference	`int inc_deref(int *p)`
strlen	`int slen(char* s)`
sdbm hash	`int sdbm(char *s)`
djb2 hash	`int djb2(char *s)`
sum of array a	`long sum(short *a, int n)`
n-th fibonaci number (recursive impl.)	`int fibo(int n)`
num. of steps until $n \rightsquigarrow 1$ (Collatz's conj.)	`int collatz(int n)`

Evaluation

We test outputs of *uCc* against some decompilers, table 1 describes briefly tested functions and their signatures in source codes, table 2 shows results of decompilers on binaries generated by *uCc*. Basically, any argument in the function signature is simply assigned by either `int64` or `uint64`.

Table 2. Type flattening effect

Original type	Hex-Rays	JEB	Ghidra	Snowman
`int id(int)`	`int64 id(int64)`	`ulong id(ulong)`	`ulong id(void)`	`int64 id(int64)`
`short div(short, char)`	`int64 div(uint64, uint64)`	`div_t div(int, int)`	`div_t div(int, int)`	`int64_t div(int64, int64)`
`char mod(short, char)`	`int64 mod(int64, int64)`	`ulong mod(ulong, ulong)`	`ulong mod(void)`	`int64 mod(int64, int64)`
`int inc_deref(int*)`	`int64 inc_deref(int64)`	`ulong inc_deref(ulong)`	`ulong inc_deref(undefined8)`	`int64 inc_deref(uint64)`
`int slen(char*)`	`int64 slen(int64)`	`ulong slen(ulong)`	`ulong slen(undefined8)`	`int64 slen(uint64)`
`int sdbm(char*)`	`int64 sdbm(uint64)`	`ulong sdbm(ulong)`	`ulong sdbm(undefined8)`	`int64 sdbm(uint64)`
`int djb2(char*)`	`int64 djb2(uint64)`	`ulong djb2(ulong)`	`ulong djb2(undefined8)`	`int64 djb2(uint64)`
`long sum(short*, int)`	`int64 sum(int64, uint64)`	`ulong sum(ulong, ulong)`	`undefined8 sum(undefined8)`	`int64 sum(uint64, int64)`
`int fibo(int)`	`int64 fibo(uint64)`	`ulong fibo(ulong)`	`ulong fibo(undefined8)`	`int64 fibo(int64)`
`int collatz(int)`	`int64 collatz(int64)`	`ulong collatz(ulong)`	`ulong collatz(undefined8)`	`int64 collatz(int64)`

It may worth to note that the *type flattening* is given by *uCc*, not by the source code complexity of tested functions. Indeed if they are compiled by, *clang* or *gcc* for example, then their signatures can be reconstructed precisely by any tested decompiler. Last but not least, *uCc* doesn't use obfuscation techniques whose side effects may affect the type reconstruction: functions in tested binaries are perfectly detectable, disassemblable and decompilable; only their inferred signatures are in question.

References

[1] C. Cifuentes. "Reverse Compilation Techniques". PhD thesis. 1994.

[2] A. Mycroft. "Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)". In: *ESOP*. 1999.

[3] G. Balakrishnan and T. Reps. "DIVINE: DIscovering Variables IN Executables". In: *VMCAI*. 2007.

[4] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms". In: *WISA*. 2007.

[5] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* 1st. Addison-Wesley Professional, 2009.

[6] J. Lee, T. Avgerinos, and D. Brumley. "TIE: Principled Reverse Engineering of Types in Binary Programs". In: *NDSS*. 2011.

[7] K. ElWazeer et al. "Scalable Variable and Data Type Detection in a Binary Rewriter". In: *PLDI*. 2013.

[8] J. Caballero and Z. Lin. “Type Inference on Executables”. In: *ACM Computing Surveys* 48.4 (2016).

[9] O. Katz, R. El-Yaniv, and E. Yahav. “Estimating Types in Binaries using Predictive Modeling”. In: *POPL* (2016).

[10] M. Noonan, A. Loginov, and D. Cok. “Polymorphic Type Inference for Machine Code”. In: *PLDI*. 2016.

[11] E. Robbins, A. King, and T. Schrijvers. “From MinX to MinC: Semantics-Driven Decompilation of Recursive Datatypes”. In: *POPL*. 2016.

[12] N. Eyrolles. “Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools”. PhD Thesis. Université Paris-Saclay, 2017.

[13] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck. “TypeMiner: Recovering Types in Binary Programs Using Machine Learning”. In: *DIMVA*. 2019.

[14] *Cranelift Code Generator*. URL: https://github.com/bytecodealliance/cranelift.

[15] T. D. Ta. *uCc: an untyped C compiler*. URL: https://github.com/tathanhdinh/uCc.