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 [2] (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 does not preserve the high-level type information in the generated machine code, then the *type recovery* (or type reconstruction) from machine code requires special techniques, a survey up until 2015 can be referenced in [5].

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 [4] 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 both 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-disassembing 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 [5] for a more complete list.

2 Type flattening compiler

We present *uCc*, an open source C compiler that explicitly obfuscates high-level types, its objective is to make reconstructing types from machine codes harder. It is implemented in Rust, the source code and a brief user guide are given at (cite).

Our effort is to attack 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 obfuscation transformations based on Mixed Boolean Arithmetic expressions [3, 6]. 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.

We test outputs of *uCc* against some decompilers, table 1 describes briefly tested functions and their types in source codes, table 2 shows results of using 4 decompilers to decompile binaries generated by *uCc*. The test suite is available at the repository.

Table 1. Functions and original types

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

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)`
`long sum(short*, int)`	`int64 sum(int64, uint64)`	`ulong sum(ulong, ulong)`	`undefined8 sum(undefined8)`	sn
`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 noting that *uCc* makes no effort in making disassembling or function boundary detection hard. Indeed, functions in tested binaries are perfectly detectable, disassemblable and decompilable.

References

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

[2] G. Balakrishnan and T. Reps. “DIVINE: DIscovering Variables IN Executables”. In: *VMCAI*. 2007.

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

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

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

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