A Lightweight Obfuscation for Function Signature Flattening

Ta Thanh Dinh
tathanhdinh@gmail.com

***Abstract*—Beside data and control flow, high-level types are important in binary code analysis, particularly in decompilation. Some research papers have introduced methods to map machine-dependent objects into types of some C-like type system. For the obfuscation/anti-decompilation purpose, we present a technique which bypasses existing type recovery approaches. We have implemented a prototype obfuscating C compiler to demonstrate the technique, the compiler is given open source.**

***Index Terms*—type recovery, decompilation, obfuscation**

1. Introduction

Binary code *decompilation* [6] is to transform the low-level, machine-dependent code of a program into a high-level form, like code of a high-level language. In almost all academic research papers and commerical products, the target language is C. Similar to compilers, a modern binary code decompiler consists of many phases [6, 18]: disassembly, function boundary detection, immediate representation (IR) lifting, control-flow graph (CFG) recovery, high-level variables detection, type (i.e. variable types and function signatures) recovery, etc. Each phase requires particular but not independent [9] analysis techniques: the results of one can affect another. The analyzed program is transformed gradually into a higher-level, more abstract and more understandable representation.

On the contrary, binary code *obfuscation* protects the machine code from being decompiled, or from being analyzed in general. Because the analysis contains of different interdependent phases, the obfuscation may attack any of them, e.g. anti-disassembly (binary packer, self-modifying code), binary stripping, control-flow flattening, virtualization (for both data and control obfuscation). Basically, each obfuscation method consists of *potent transformations* [7, 12] that hide certain properties of the code.

Context. An optional feature of binary code decompilation is *type reconstruction*, namely to recover high-level types from machine-dependent objects [8, 18]. This is the research objective of some research papers [22, 25, 29, 30], and the killer feature of commercial [42, 43] as well as open source [41] binary code analysis tools. Beside decompilation, types and particularly *function signatures* are also essential in numerous applications: static and dynamic binary rewriting [21, 24], binary raising [36, 38]..., see for example [26] for a more completed list.

The knowledge about types in general and particularly function signatures expands the attack surface since more analysis can be applied on the programs. For example, the dynamic binary instrumentation framework Pin [14] provides a method `RTN_Replace` (Routine Replace) which allows to replace a function in the binary by another function, but it requires that the new signature must be the same as the replaced one, another instrumentation tool Dyninst [21] has a similar method `replaceFunction`. For static analysis, the exploit generation for mimicry attacks given in [13] requires a capability of

> ...tracking function calls together with the corresponding parameters.

just quote from the paper.

Problem. Despite of successes in binary type recovery and the need for protecting function signatures, to the best of our knowledge, there is currently little effort in hiding type information. A part from some low-level techniques such as binary stripping, the most related research that we can find is the work of Drape [10, 15, 17] about abstract data type obfuscation, but he focused on complex data types (array, list, tree) and used the array splitting technique proposed previously by Collberg [7]. Actually, the structure of such a type makes more room for obfuscation, and his approach is orthogonal with ours (see Section 6). We consider them heavyweight since they always insert much code size and runtime overhead.

Contribution. We present a new method for hiding type information, the principal idea is based on the fact that the compiler does not need to preserve all information about high-level types (type erasure), then with specific tricks we can exploit the *semantics gap* between the high-level language and machine code to make some information very hard to be recovered. We do not claim that all type information can be hidden, the attacker can eventually know some but it would be hard to distinguish the concrete underlying types from one to another, thus the proposed notion of *type flattening*.

We implement the method in a prototype C compiler *uCc* [46] which hides function signatures. The functions in binaries generated by *uCc* can be perfectly analyzed by traditional procedures (boundary detection, disassembling, CFG recovery, etc), only their signatures are obfuscated. That way, we can evaluate the effectiveness of type flattening while excluding unwanted obfuscation effects that may come from (bad) results of other analysis phases.

Type flattening obfuscation is lightweight, between the compiled binaries with and without type flattening, the former incurs only a small code size and runtime overhead; more sophisticated obfuscation can be added optionally. Though the paper presents only type flattening for primitive types and function signatures, the method can be easily extended for complex types.

In summary, our contributions are as follows:

- We give a critical analysis (Section 2.2) about how current binary type recovery methods deal with the

low-level polymorphism problem, as it turns out there exist always instances of this problem that these methods cannot capture.

- We introduce the notion of *type flattening* obfuscation (Section 3.1), it aims at protecting a high-level property (types) of the program, in contrast with classical methods which focus on lower properties as data or control flow.
- We present some techniques to obtain the notion and implement an open source prototype compiler *uCc* to realize these techniques. The reverse would be hard thanks to the resilience of MBA obfuscation [27, 31].
- We give also in *uCc* an implementation for the permutation polynomials of MBA, other open source state-of-the-art obfuscators (e.g. Tigress [47]) have only basic arithmetic encoding expressions. The deobfuscation tools (e.g. Syntia [32], QSynth [39]) can profit from *uCc* to test their capabilities of MBA simplification.
- We evaluate the binaries generated by *uCc* against decent decompilers, the results show that no one can detect correctly the underlying types of arguments on function signatures: the original types are indistinguishable from the highest types in the C's integer conversion rank.

2. Binary type recovery

In statically typed languages, the compiler does not need preserve source code level type information in the generated machine code (type erasure), then recovering types from binary code requires special techniques. Before presenting the type flattening obfuscation, we give a brief discussion about current approaches on binary type inference and *the problems they aim to solve*. That gives some intuition about our goal and the techniques proposed to achieve this goal.

From now on, unless otherwise stated, the (high-level) type system is that of C. We assume that the binaries are compiled by some C compiler from source codes, and binary type recovery means assigning C's types for low-level primitives. For illustration purpose, we use a pseudo-C syntax for low-level primitives (e.g. `mov rax, rdi` is expressed as $a = i$) but the concrete meaning is mostly direct from the context.

2.1. Brief history

A broad survey for research on binary type recovery until 2015 is [26], but it sustains a storage point of view bias: types are attached always with physical storage primitives (e.g. registers, memory), there are no essential differences between types and data structures, so are the techniques to recover them. Actually, types are logical compile-time constraints, they may or may not have storage imprints. An example is C's *type qualifier* (e.g. `const`, `volatile`), and in general any *refinement type* should not leave storage traces, the same thing with generics. More concretely, as we will present in the section, the *low-level polymorphism* is a specific problem that binary type recovery techniques have to deal with. Also, the survey lacks some important papers which are only published until later [29, 30].

We focus only on program semantics/type theoretical approaches, the research using machine learning [37] or statistical language model [28] are out of scope of the paper. We omit the phase of variable/function detection, which is a vital step before type recovering, more details on this subject can be referenced in [16]. We avoid also difficulties in machine code disassembling, the binaries are assumed to be perfectly disassemblable.

Initial work. Though earlier ideas have been proposed in another context [4], the research in recovering types from low-level languages may begin with the classic paper of Mycroft [8] for his interest of decompilation. The principal idea is inspired by the work of Damas-Hindley-Milner [1, 2, 3] in the ML language: types of variables and functions are checked/referenced automatically from how they are used in the program's source code. For example, given an expression

$$x + y$$

then at least x or y must have integer type, it is impossible that both of them are pointers since adding two pointers does not type check.

The method of Mycroft has several limits, as pointed out by Van Emmerik [18]. One of them comes from the fact that the low-level languages take care mostly on the value of the computation, then (the result of) an expression can be used in several ways and it behaves as different types (see Section 2.2). Let's consider an assignment

$$p' = p + n$$

where $\vdash p\colon \texttt{ptr}(S)$ (p is of type pointer to a struct S) and $\vdash n\colon \texttt{int}$, Mycroft's rules derive $\vdash p'\colon \texttt{ptr}(S)$ since $p+n$ is considered as the offset calculation to access some element of an array of S. But $p+n$ can be also an offset calculation to access some field of type, e.g. `int`, of the struct S, then $\vdash p'\colon \texttt{ptr}(\texttt{int})$.

To overcome these problems, Van Emmerik has proposed a *data-flow based* (in contrast with Mycroft's *constraint based*) approach where type information of an object will be refined gradually, instead of binding it early to some fixed type. He proposed using *subtype lattices* to express the preciseness of type information: p' will not

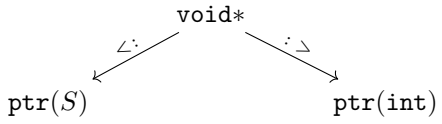

Figure 1. A subtype lattice

be early bound as $\texttt{ptr}(S)$, instead $\vdash p'\colon \texttt{void}*$ (adding integer to pointer does not always result in pointer of the same type) and $\texttt{ptr}(S) <: \texttt{void}*$. The precise type is only assigned later, when enough constraints are derived from other uses, e.g.:

$$*p' = m$$

where $\vdash m\colon \texttt{int}$, it then derives $\vdash p'\colon \texttt{ptr}(\texttt{int})$, finally $\vdash p'\colon \texttt{ptr}(\texttt{int})$ from meet $\texttt{ptr}(\texttt{int}) = \texttt{ptr}(\texttt{int}) \sqcap \texttt{void}*$.

The lack of an IR with well-defined semantics and a type constraint solver limits Van Emmerik's work, he had to use heuristics as type patterns (e.g. adding a non-constant to some pointer implies an array access, not a

struct field access) to recognize and propagate type/subtype relations.

Improvement. The later work always lifts machine code to an IR before type analysis. Lee et al. in TIE [22] had a similar idea of using subtype lattice, their type system are more detailed and support more cases (e.g. struct/function types, constraints for call/jump) but basically similar to Van Emmerik's. For example, the assignment

$$p' = p + n$$

will generate $\vdash \texttt{ptr}(T_\beta) <: \tau_{p'}$ where T_β is a type variable and $\tau_{p'}$ means type of p'. Since T_β is not free (never used outside the assignment), this constraint is more or less equivalent with $\vdash p' : \texttt{void}*$. The type propagation in TIE, thanks to the type constraint solver, does not need type pattern heuristics.

Noonan et al. in Retypd [29] examined several more instances of low-level polymorphism problem (see Section 2.2). As with *type as labeled tree* idiom [5], they proposed a *type capability* model where each type variable may attach with labels representing it capabilities: labels are actually unary type constructors and capabilities are derived types. Retypd uses subtyping in almost all derived

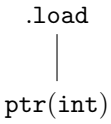

Figure 2. Tree for derived type when reading a pointer of type `int`

type constraints. For example, the pointer dereference and assignment

$$x = *p$$

will result in $\vdash \tau_p.\texttt{load} <: \tau_x$, means p is a readable pointer (`.load` label), and the type of the dereferenced value is a subtype of type of x. The labels on τ_p allows to represent constraints on the inner structure of p and p itself. For example, we may have $\tau_p.\texttt{load} <: \tau_p$, which means p is a pointer to some struct whose the first field is also of type pointer to this struct. That way, Retypd supports recursive types, while TIE cannot.

Untyped decompilation. Some decompilers [34, 45] do not focus much on type analysis, instead they focus on correct disassembling. That allows to *raise* [36, 38] the binary to an IR, then the type system of the IR is used directly to emulate high-level types. A notable example is the retargetable decompiler RetDec [34] which raises binary codes to LLVM [11]. To get the idea, let us consider the small piece of assembly code in Listing 1 (the original source code is given in Listing 2).

```
1  mov eax, [rdi+0x4]
2  ret
```

Listing 1. Pointer increment then dereference

```
int foo(int *x) {
  x += 1;
  return *x;
}
```

Listing 2. Pointer increment then dereference (source)

RetDec raises the assembly code into LLVM (Listing 3), then the decompilation result (Listing 4). As we can see, RetDec does not bother to detect whether p is a pointer to `int`, it uses the System V ABI convention (the first parameter is passed via `rdi`) to map p to `i64`, then casts expressions containing p to the needed types.

```
define i64 @inc_deref(i64 %p) {
dec_label_pc_0:
  %v1_0 = add i64 %p, 4
  %v2_0 = inttoptr i64 %v1_0 to i32*
  %v3_0 = load i32, i32* %v2_0, align 4
  %v4_0 = zext i32 %v3_0 to i64
  ret i64 %v4_0
}
```

Listing 3. Pointer increment then dereference (LLVM)

```
int64_t foo(int64_t p) {
  return (int64_t)*(int32_t *)(p + 4);
}
```

Listing 4. Pointer increment then dereference (decompilation)

Though only giving poor results about types, this approach has advantages: the decompilation is always possible after disassembling, the type system contains just basic types (being mapped from IR types) and casts.

2.2. Polymorphism

One of the problems that all binary type recovery methods aim at is to capture the *low-level polymorphism* in their type systems. This kind of polymorphism means an low-level object may have different high-level types. It either comes from ① type erasure in compilation (high-level objects with distinguished types are compiled into a single low-level object, or ② casts used in the source code which subvert the type checking. There are many instances of low-level polymorphism.

One of the first recognized instances is *pointer arithmetic*; the expression $p + c$ where p is a pointer and c is some constant, may be of type τ_p (same type with the pointer p) or just an unknown pointer type τ_α. This instance is captured by the type system of Van Emmerik (as with TIE and Retypd). Another instance is *symbolic constants*, e.g. after being reset `xor rax,rax`, the register `rax` can be used both as an integer (of value 0) and as a pointer (of value `NULL`); the instance is captured by Retypd only.

```
1  mov eax, edi    ; eax = x
2  cdq
3  idiv esi        ; signed division x / y
4  xor edx, edx
5  mov ecx, eax
6  mov eax, edi    ; eax = x
7  div esi         ; unsigned division x / y
8  add eax, ecx
9  ret
```

Listing 5. Signed and unsigned division

```
function u0:0(τx, τy) -> τ {
block0(x: τx, y: τy):
  z = sdiv x, y
  t = udiv x, y
  s = iadd z, t
```

```
  return s
}
```

Listing 6. Signed and unsigned division (cranelift)

Union. A more complex instance is *incompatible types*, an example is given in Listing 5 (see Listing 7 for the source code). To avoid unnecessary details when using the assembly code for type analysis, it would be better to look at the cranelift IR version given in Listing 6 (since the binary type recovery eventually works on some IR). Type constraints for x and y can be derived in a traditional way, for example with x

$$\frac{\vdash \tau_x <: \texttt{i32}\ (\text{since } \texttt{sdiv}) \qquad \vdash \tau_x <: \texttt{u32}\ (\text{since } \texttt{udiv})}{\vdash \tau_x <: \texttt{i32} \sqcap \texttt{u32}}$$

TIE interprets $\vdash \tau_x <: \texttt{i32} \sqcap \texttt{u32}$ as $x \in [0, 2^{31} - 1]$, which is not a correct constraint; while Retypd uses *union* for the meet $\texttt{i32} \sqcap \texttt{u32}$, which is a reasonable result. Actually, using union to combine incompatible types is similar to the type casting approach of untyped decompilers discussed above.

```
int foo(int x, int y) {
  int z = x / y;
  unsigned t = (unsigned)x / (unsigned)y;
  return z + t;
}
```

Listing 7. Signed and unsigned division (source)

The illustrative assembly code and cranelift IR show also that in low-level, data copy does not always result in *type copy*: two identical values may have different types. For example, the register `eax` at line 1 and line 6 in Listing 5 are two identical copies of the original argument x, but of different types.

3. Type flattening

In this section, we present the notion of type flattening obfuscation and techniques used in *uCc* to obtain that notion. We focus only on scalar types (pointer, integral, but not float nor packed), and function signatures where parameters and return are of scalar types; supporting aggregate types (e.g. struct, nested struct) is still the ongoing work. From now on, unless otherwise stated, we assume that the binaries follow the System V ABI for AMD64 architecture [44].

3.1. Notion

To avoid some ambiguities, we introduce a simple type system with subtyping upon this the type flattening will be expressed. Let $\mathcal{P}$ denote a finite set of primitive type constants, each element of $\mathcal{P}$ is either of form $\texttt{i}N$ (for signed types) or $\texttt{u}N$ (for unsigned types):

$$P = \{\texttt{i}N, \texttt{u}N \mid \texttt{N} = 8, 16, 32, 64\}$$

The function `sizeof` computes the size in bytes of type constants, e.g. $\texttt{sizeof}(\texttt{i32}) = 4$. The unary type constructor `ptr` defines pointer types over primitive types, e.g. $\texttt{ptr}(\texttt{i8})$ is a pointer to some value of type `i8`. The function `sizeof` is extended for pointer types, though $\texttt{sizeof}(\tau) = 8$ for any pointer type τ. The binary type constructor `union` defines a new union type for two types of the same size, the `sizeof` function is extended also for union types:

$$\texttt{sizeof}(\texttt{union}(\tau_x, \tau_y)) = \texttt{sizeof}(\tau_x) = \texttt{sizeof}(\tau_y)$$

We extend `union` to accept multiple type variables of the same size, also `union` has the following properties:

$$\begin{aligned} \texttt{union}(\tau_x, \tau_y) &= \texttt{union}(\tau_y, \tau_x) && \text{symm.} \\ \texttt{union}(\tau_x, \tau_y, \tau_z) &= \texttt{union}(\texttt{union}(\tau_x, \tau_y), \tau_z) \\ &= \texttt{union}(\tau_x, \texttt{union}(\tau_y, \tau_z)) && \text{assoc.} \end{aligned}$$

The constructor `ptr` is extended to cover union types, naturally the size of a pointer to union is 8. The binary type constructor $\rightarrow$ defines a new function type of some return and parameter types, i.e. if some function $f : \tau_x \rightarrow \tau_y$ then f is of type $\rightarrow (\tau_x, \tau_y)$. For typing functions of multiple parameters, we need an implicit product type constructor $()$, e.g. if $f : \tau_x \rightarrow (\tau_y, \tau_z)$ then f is of type $\rightarrow (\tau_x, (\tau_y, \tau_z))$. For simplification, we limit that f is only defined over primitives, pointers of primitives, and unions of primitives.

Subtyping. `union` is symmetric, i.e. $\texttt{union}(\tau_x, \tau_y) = \texttt{union}(\tau_y, \tau_x)$

Suppose that there is a subtype lattice with $\top$ and $\bot$, and an

The type recovery approaches proposed different techniques to deal with the problem of low-level polymorphism, but a common point is to use some *subtype lattice* for the preciseness of inferred types. In the lattice, the bottom type $\bot$ means that the variable violates some constraints in the type system [22]. Ideally, $\bot$ should not occur since in the worst case, the decompiler can simply simulate the "weak" type system of the low-level language, we consider only $\top$.

The top type $\top$ means universal or any, intuitively if a variable is of type $\top$ then we only know the most trivial information about its type. The idea of *type flattening* is similar, removing useful information about type of an object means making the type recovery algorithm infer the object's type as $\top$.

Definition 1. *A high-level object is called type flattened up to a type inference algorithm with subtyping if its type is inferred as $\top$ in the subtype lattice of the algorithm.*

Top types. Unsurprisingly, under some real world conditions, $\top$ type does not mean we do not know anything, we actually know some properties. Recall that in our context, the binaries are disassemblable, function boundaries can be recognized correctly. Thus the binary, as our goal is to make it reusable, must respect the ABI (Application Binary Interface). For example, AMD64 System V ABI specifies that the first parameter of a function is passed via `rdi` register, thus in the worst case of the binary type inference, the type of the first argument is `size64`. The actual type may be `char*`, `signed32`, `unsigned16`, etc. but it is always subtype of `size64` (see Figure 3). This is actually what are being performed by some binary raising projects [36, 38] and decompiler [23].

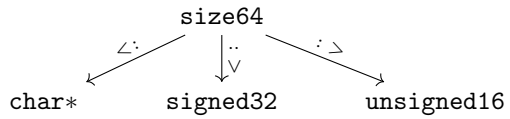

Figure 3. $\top$ as `size64`

Goal. Our goal in type flattening obfuscation is to make any inferred type become $\top$. In our specific context, under AMD SystemV ABI, it is `size64`. For that goal, *uCc* combines two techniques: *data-flow disrupting* and *function signature masquerading*. The former bypasses the intra-procedural data analysis, the later tricks the inter-procedural one.

3.2. Techniques

Though quite different about type systems, constraint generation rules and constraint solvers; later type recovery techniques follow the *data-flow based* approach proposed by Van Emmerik. Actually, most of implementation code in TIE is for the data-flow analysis and CFG building. Retypd uses an external abstract interpreter which does the data-flow analysis, this interpreter contributes greatly to the preciseness of the type inference.

Data-flow disrupting. A direct technique for type flattening is to disrupt the data-flow analysis, that is done by introducing *pointer aliases*. Let's look at the piece of code in Listing 8, for illustration purpose we have used a pseudo-C syntax but only put type annotation on the variables whose types are supposed to be already known.

```
foo(signed32 x) {
  p = &x;
  y = *p;
  return y;
}
```

Listing 8. Direct data movement

Any type recovery method can find the return type of `foo`. Indeed, with TIE: $\vdash p$: `ptr(signed32)` from the first assignment, then the second infers $\vdash y$: `signed32`. Retypd gives a slightly different result $\vdash$ `signed32` $<: y$ where $y <:$ `size32`, which is also more precise.

In Listing 9, we insert an alias q for p to disrupt the data-flow. TIE derives $\vdash y$: `size32`, Retypd is not better $\vdash$ `size32` $<: y$ where $\vdash y <:$ `size32`.

```
foo(signed32 x) {
  p = &x;
  q = an alias of p;
  y = *q;
  return y;
}
```

Listing 9. Data movement with pointer alias

In summary, before an use (which needs to be protected) of a variable, *uCc* ① creates a stack slot to store the variable, ② generates an alias for the stack slot, ③ loads the variable from the alias.

Pointer alias generation. Thanks to results about invertible *permutation polynomials* [19], the generation of pointer aliases become straightforward. Basically, for each case where a pointer needs to be aliased, *uCc* generates randomly an invertible polynomial MBA expression P over $\mathbb{Z}/2^n$ where n is the machine word bit size ($n = 64$), and its inverse $Q = P^{-1}$ (see Figure 5 for an example of P with degree 7 and its inverse Q), the aliases of some value v is $Q(P(v))$. The calculation of $Q(P(v))$ is embedded into the generated binary code.

3.3. Function signature masquerading

While types of local variables can be obfuscated by disrupting the data-flow, obfuscating types of function parameters needs different techniques. For inter-procedural type analysis, the data-flow disrupting still works because the type of passed arguments are already hidden: they are local variables of the calling function. But for intra-procedural type analysis, we can still hide the sign property (signed, unsigned) or whether they are pointers or not, but we cannot hide their size (recall that our goal is to obtain $\top$ for all referred types). That is because of the size of a function parameter can be detected by looking at the register used to access this parameter inside the function. For example, if the first argument is of type `int` (i.e. `signed32`) then the register `edi` is used, not `rdi`. The parameter size is masqueraded by the following techniques:

Signature rewriting. For each function, *uCc* keeps an original copy of the signature but creates another signature where size of each parameter is extended to `size64`. The masqueraded signature is used for any call to the function in the translation unit, so for from an external (but in the translation unit) view, the function behaves as it has the masqueraded signature. The original signature is used inside the function.

Trampoline block. In the function, *uCc* creates a trampoline as the pseudo-entry basic block, this trampoline uses the masqueraded signature to retrieve the parameters but then convert them to the real ones using the original signature. The conversion applies the data-flow disrupting to hide the source of the parameters. The real parameters then passed into the real entry basic block.

Semantics gap and type erasure. The techniques used for function signature masquerading have actually profited from *semantics gap* between the high-level and the machine language. In the high-level language, a function parameter has some high-level type and the program will not type check if this type violates the type system (e.g. passing an integer in a function whose the parameter type is pointer). But the ABI does not have such a constraint, it simply states that, for example, the first parameter is passed via `rdi`, no matter what its type is. In another words, the semantics gap is an instance of *type erasure*.

We can see also that the signature rewriting and inner trampoline profit the low-level polymorphism to wrap a type by its supertype, this is exactly an inverse process of binary type recovery which tries to unveil a type to get the underlying subtype.

3.4. Other obfuscation tricks

uCc implements also some supplemental techniques to augment its resilience from type analysis algorithms and general deobfuscation efforts.

Split and merge. To bypass the type constraints generated from `load/store` operations in TIE, as well as the derived types `.load/.store` in Retypd. *uCc* may split the loading of a value into several loads, each of them retrieves a part of the value, then merges these parts. This is a basic but useful obfuscation trick.

MBA rewriting. *uCc* uses some MBA rewriting rules to obfuscate basic arithmetic operations, `add`, `and`, `or`, `xor`. The rules are taken from [33].

$$
\begin{aligned}
x + y &\rightarrow (x \wedge y) + (x \vee y) \\
x \vee y &\rightarrow (x + y) - (x \wedge y) \\
x \times y &\rightarrow (x \wedge y) \times (x \vee y) + (x \wedge \bar{y}) \times (\bar{x} \wedge y)
\end{aligned}
$$

Figure 4. Some rewriting rules used in *uCc*

Code diversity. *uCc* is a *probabilistic compiler*, the generation of pointer aliases is not the same over generated binary codes: the coefficients and even the degree of invertible polynomials are generated randomly for each run of *uCC*. The number of split parts in a value loading is randomized, as well as the size of each part.

4. Implementation

The open source implementation in Rust of *uCc* is given at [46]. The frontend is implemented from scratch, the code generation part of the backend uses Cranelift [40]. Beside the final result is an obfuscated ELF object file, *uCc* has also several options, for example JIT code generation, showing IR code, and selecting obfuscation level.

Notes. The principal goal of *uCc* is type flattening obfuscation, but there is another requirement that is the capability of evaluating the single effect of type flattening. Concretely, we want that the functions in binaries generated by *uCc* would be perfectly detectable, disassemblable and decompilable (but with flattened types). For these purposes, we try to avoid any other obfuscation tricks that may bring unwanted effects. Many popular obfuscation techniques [20, 35] (control-flow flattening, dead-code insertion, virtualization, code packing, self-modifying code, etc) cannot be applied because they either protect the binaries from disassembling, or prevent the detection of functions in the binary, as well as obfuscating the function control-flow graph.

5. 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*.

6. Related work

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 [26] for a more complete list.

Acknowledgment

Thanks to my wife and my children who are always with me.

References

[1] J. R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60.

[2] R. Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Science* 17 (1978).

[3] L. Damas and R. Milner. “Principal Type-Schemes for Functional Programs”. In: *POPL*. 1982.

[4] O. Shivers. “Data-Flow Analysis and Type Recovery in Scheme”. In: *Topics in Advanced Language Implementation*. MIT, 1990.

[5] D. Kozen, J. Palsberg, and M. I. Schwartzbach. “Efficient Inference of Partial Types”. In: *POPL*. 1993.

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

[7] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. University of Auckland, 1997.

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

[9] B. Schwarz, S. Debray, and G. Andrews. “Disassembly of Executable Code Revisited”. In: *WCRE*. 2002.

[10] S. Drape. “Obfuscation of Abstract Data Types”. PhD thesis. The University of Oxford, 2004.

[11] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *CGO*. 2004.

[12] M. Dalla Preda and R. Giacobazzi. “Semantic-Based Code Obfuscation by Abstract Interpretation”. In: *ICALP*. 2005.

$$P(x) = 562949953421312 - 6992785447134243183 \times x + 9007199254740992 \times x^2 + 1125899906842624 \times x^3 + 8796093022208 \times x^4 + 4611686018427387904 \times x^5 + 35184372088832 \times x^6 - 9223372036854775808 \times x^7$$

$$Q(x) = -6116451243922554880 + 8805192775852862065 \times x - 5341269158061408256 \times x^2 - 4540754324296302592 \times x^3 - 6409053278306304000 \times x^4 - 4611686018427387904 \times x^5 - 1711965992726298624 \times x^6 - 9223372036854775808 \times x^7$$

Figure 5. An invertible polynomial and its inverse generated by *uCc*

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)`

[13] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. "Automating Mimicry Attacks Using Static Binary Analysis". In: *USENIX Security*. 2005.

[14] C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *PLDI*. 2005.

[15] S. Drape. "An Obfuscation for Binary Trees". In: *TENCON*. 2006.

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

[17] S. Drape. "Generalising the Array Split Obfuscation". In: *Information Sciences* 177.1 (2007).

[18] M. Van Emmerik. "Static Single Assignment for Decompilation". PhD thesis. 2007.

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

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

[21] A. R. Bernat and B. P. Miller. "Anywhere, Any-Time Binary Instrumentation". In: *PASTE*. 2011.

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

[23] L. Durfina, J. Kroustek, P. Zemek, and B. Kabele. "Detection and Recovery of Functions and their Arguments in a Retargetable Decompiler". In: *WCRE*. IEEE, 2012.

[24] K. Anand et al. "A Compiler-level Intermediate Representation based Binary Analysis and Rewriting System". In: *EuroSys*. 2013.

[25] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. "Scalable Variable and Data Type Detection in a Binary Rewriter". In: *PLDI*. 2013.

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

[27] N. Eyrolles, L. Goubin, and M. Videau. "Defeating MBA-based Obfuscation". In: *SPRO*. 2016.

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

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

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

[31] F. Biondi, S. Josse, A. Legay, and T. Sirvent. "Effectiveness of Synthesis in Concolic Deobfuscation". In: *Computers and Security* 70 (2017).

[32] T. Blazytko, M. Contag, C. Aschermann, and T. Holz. "Syntia: Synthesizing the Semantics of Obfuscated Code". In: *USENIX Security*. 2017.

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

[34] J. Křoustek, P. Matula, and P. Zemek. "RetDec: An Open-Source Machine-Code Decompiler". In: *Botconf*. 2017.

[35] S. Banescu and A. Pretschner. "A Tutorial on Software Obfuscation". In: *Advances in Computers*. Vol. 108. Elsevier, Jan. 2018, pp. 283–353.

[36] P. Goodman and A. Kumar. "Lifting program binaries with McSema". In: *ISSISP*. 2018.

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

[38] S. B. Yadavalli and A. Smith. "Raising Binaries to LLVM IR with MCTOLL (WIP Paper)". In: *LCTES*. 2019.

[39] R. David, L. Coniglio, and M. Ceccato. "QSynth - A Program Synthesis based approach for Binary Code Deobfuscation". In: *BAR*. 2020.

[40] *Cranelift Code Generator*. URL: https://github.com/bytecodealliance/cranelift.
[41] *Ghidra*. URL: https://ghidra-sre.org/.
[42] *Hex-Rays Decompiler*. URL: https://www.hex-rays.com/.
[43] *JEB Decompiler*. URL: https://www.pnfsoftware.com/.
[44] H. J. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*.
[45] *Snowman decompiler*. URL: http://derevenets.com/.
[46] T. D. Ta. *uCc: an untyped C compiler*. URL: https://github.com/tathanhdinh/uCc.
[47] *Tigress: A Source-to-Source-ish Obfuscation Tool*. URL: https://tigress.wtf.

TABLE 2. TYPE FLATTENING EFFECT

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