Type Flattening Obfuscation

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* [4] 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 [4, 10]: 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 [7] 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.

In the opposite direction, binary code *obfuscation* is a method to protect the low-level code from being decompiled, or from being analyzed in general. Because the code analysis contains of different interdependent phases, the obfuscation [12, 25] can proceed at any of them, e.g. anti-disassembly (binary packer, self-modifying code), binary stripping, control-flow flattening, virtualization (for both data and control obfuscation)... just name a few. Basically, each obfuscation method consists of one or several *semantics-preserving* transformations [8, 12] which hide certain properties of the code.

Context and problem. An optional feature of binary code decompilation is *type reconstruction*, namely to recover high-level types from machine-dependent objects [5, 10]. This is the research objective of some research papers [14, 17, 20, 21], and killing feature of commercial [31, 32] as well as open source [30] binary code analysis tools. Beside decompilation, types and particularly *function signatures* are also essential in numerous applications, e.g. static binary rewriting [13, 16] and raising [26, 28], see for example [18] for a more completed list. Thus the knowledge about types expand the attack surface since more analysis can be applied on the programs.

Contribution. Despite of successes in binary type reconstruction and the need of protecting function signatures, to the best of our knowledge there is no explicit effort in hiding type information. This paper presents a method for type obfuscation, 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 if not impossible 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 tricks in *uCc*, an open source obfuscating C compiler which obfuscates function signatures. The functions in binaries generated by *uCc* can be perfectly analyzed by classical procedures (boundary detection, disassembling, CFG recovery, etc), only their signatures are obfuscated. That way, we can evaluate the effectiveness of type obfuscation tricks on function signatures while excluding unwanted obfuscation effects that may come from (bad) results of other analysis phases. We find that Mixed Boolean Arithmetic (MBA) expressions [11, 23] are a good match for the goal.

In summary, our contributions are as follows:

- We introduce the notion of *type flattening*, 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 build a prototype compiler *uCc* to realize the ideas of obfuscation. *uCc* also implements the permutation polynomials of MBA [11] while other open source state-of-the-art obfuscators (e.g. Tigress [34]) give only basic arithmetic encoding expressions. Other deobfuscation tools (e.g. Syntia [22], QSynth [29]) can profit *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. Brief history of binary type inference

In statically typed languages, the compiler does not need preserve source code level type information in the generated machine code (type erasure), then type recovering requires special techniques. Before presenting the type obfuscation, we give a brief discussion about how current methods on binary type inference work, that gives some intuition about our bypassing technique.

Though a broad survey for research up until 2015 can be referenced in [18], it sustains a storage point of view bias: types are attached always with concrete 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 compile-time constraints, they may or may not have runtime storage imprints. An example is C's *type qualifier* (e.g. `const`, `restrict`), in general any *refinement type* should not leave storage traces, the same thing with generics. More concretely, as we will presented in the section, the low-level polymorphism is a very specific problem that binary type recovery techniques have to deal with. Also, the survey lacks some important papers which are only published until later [20, 21].

We focus only on semantics-based approaches, recent research using machine learning [27] or statistical language model [19] are out of scope of the paper. We omit the phase of variable/function detection, which is an essential step before type recovering, more details on this subject can be referenced in [9]. We avoid also difficulties in disassembling, the binaries are supposed to be perfectly disassemblable.

From now on, unless otherwise stated, the target language is C, this is also the target language of almost all research papers and tools in the domain.

2.1. Initial work

Though earlier ideas have been proposed in another context [3], the research in recovering types from low-level languages may begin with the classic paper of Mycroft [5] for his interest of decompilation. The principal idea is inspired by the work of Damas-Hindley-Milner [1, 2] 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 [10]. 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 (low-level polymorphism). Let's consider an assignment

$$p' = p + n$$

where $\vdash p\colon \mathtt{ptr}(S)$ (p is of type pointer to a struct S) and $\vdash n\colon \mathtt{int}$, Mycroft's rules derive $\vdash p'\colon \mathtt{ptr}(S)$ 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 $\vdash p'\colon \mathtt{ptr}(\mathtt{int})$.

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

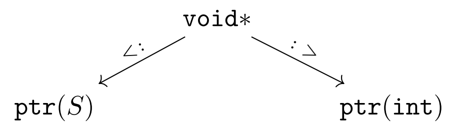

Figure 1. A subtype lattice

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

$$*p' = m$$

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

The lack of an IR with well-defined semantics limits Van Emmerik's work, he had to use ad-hoc type patterns to recognize and propagate type/subtype relations.

2.2. Improvement

Lee et al. in TIE [14] had the same idea of using type lattice for type preciseness, their deduction rules are more detail and support more cases (e.g. calls and dynamic jumps) but basically similar to Van Emmerik's. For example, the previously discussed assignment

$$p' = p + n$$

will generate $\vdash \mathtt{ptr}(T_\beta) <: \tau_{p'}$ where T_β is a type variable and $\tau_{p'}$ means type of p', but T_β is not free (never used outside the assignment) then this constraint is equivalent with $\vdash p'\colon \mathtt{void}*$. The notable improvement is the use of an IR named BIL (BAP Instruction Language), this makes the type analysis simpler and more coherent.

Polymorphism. The approaches discussed until now only consider basic cases of *low-level polymorphism*, e.g. adding a pointer to an integer may result in a pointer of the same type or not, but there are more. For example, `mov` can freely move data between signed and unsigned values, or even a constant can behaves as different types: zero is an integer, but it can be also a `NULL` pointer. Another case is the indistinguishability between a pointer to a struct and a pointer to the first field of this struct. All come from the low-level appearance of *type casting*, more details can be referenced in [6].

Noonan et al. handled these problems in Retypd [20] by first using subtyping in almost all derived constraints. The effect of data moving $x = y$ will be represented by $\vdash \tau_y <: \tau_x$. More importantly, they proposed a *type capability* model: each type variable is attached with several labels representing it capabilities. For example, the pointer dereference and assignment

$$x = *p$$

will result in $\vdash \tau_p.\mathtt{load} <: \tau_x$, 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 (if exists) and p itself.

They used lattices for subtype relations, and type analysis is proceeded on an IR, similar with Lee et al.

2.3. Existing implementations

Only Van Emmerik gives an open-source implementation of type recovery in his Boomerang decompiler, Lee et al and Noonan et al. do not. Published recently, Ghidra [30] is an open-source decompiler which has type recovery, we do not know how it works yet. Other open-source decompilers, Snowman [33] or RetDec [24], do not seem focus much on this kind of analysis. There are also commercial tools whose methods are not published, most notably Hex-Rays [31] and JEB [32].

3. Type obfuscation

In this section, we present proposal for type flattening obfuscation and techniques used for that notion. We focus only on obfuscating scalar types (pointers, integers, but not floats), supporting aggregate types (e.g. struct, nested struct) is still the ongoing work.

3.1. Type flattening

A common point of type recovery techniques is to use some *subtype lattice* which represents also the preciseness of inferred types. In the lattice, the bottom type $\perp$ means that the variable violates some constraints in the type system [14]. Ideally, $\perp$ 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.*

Uncertainty. 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 fig. 2). This is actually what are being performed by some binary raising projects [26, 28] and decompiler [15].

Goal. Our goal in type flattening obfuscation is to make any referred type become $\top$. In our specific context, under AMD SystemV ABI, it is `size64`.

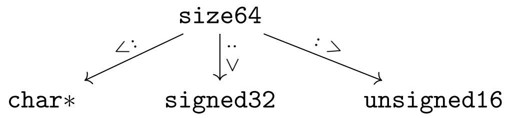

Figure 2. $\top$ as `size64`

3.2. Hiding data-flow

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.

A direct trick for type flattening is to hide the data-flow in the program, that is done by introducing *spurious pointer aliases* [12]. Let's look at the piece of code in listing 1, for illustration purpose we have used a pseudo-C syntax but only add type annotation on the variables whose types are supposed to be already known.

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

Listing 1. Direct data movement

Any type recovery method can find the return type of `foo`. For example, 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 2, we hide the data-flow by adding a spurious alias q for p. TIE would derive $\vdash y$: `size32`, while Retypd is not better $\vdash$ `size32` $<: y$ where $\vdash y <:$ `size32`.

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

Listing 2. Data movement with spurious alias

3.3. Hiding function signatures

We have seen that types of local variables can be hidden by corrupting the data-flow, but hiding types of function parameters require different tricks. For inter-procedural type analysis, the data-flow corruption 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, function parameters are one of type information sources; with the previous trick, 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).

In binaries compiled by a normal compiler, the size of a function parameter can be detected by the size of the register used inside the function. For example, if the first argument is of type `int` (i.e. `signed32`) then the register `edi` is used, not `rdi`. In our compiler *uCc*, the function parameter size is hidden by following tricks.

Signature rewriting. For each function,

Inner trampoline. sdfsd

3.4. Other tricks

3.5. Unused techniques

4. Implementation and evaluation

5. Related work

6. Ease of Use

6.1. Maintaining the Integrity of the Specifications

The IEEEtran class file is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

7. Prepare Your Paper Before Styling

Before you begin to format your paper, first write and save the content as a separate text file. Complete all content and organizational editing before formatting. Please note sections 7.1–7.5 below for more information on proofreading, spelling and grammar.

Keep your text and graphic files separate until after the text has been formatted and styled. Do not number text heads—LaTeX will do that for you.

7.1. Abbreviations and Acronyms

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, ac, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

7.2. Units

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive".
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter", not "webers/m^2". Spell out units when they appear in text: ". . . a few henries", not ". . . a few H".
- Use a zero before decimal points: "0.25", not ".25". Use "cm^3", not "cc".)

7.3. Equations

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)", not "Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

7.4. LaTeX-Specific Advice

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BIBTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BIBTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

7.5. Some Common Mistakes

- The word “data” is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter “o”.
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an “inset”, not an “insert”. The word alternatively is preferred to the word “alternately” (unless you really mean something that alternates).
- Do not use the word “essentially” to mean “approximately” or “effectively”.
- In your paper title, if the words “that uses” can accurately replace the word “using”, capitalize the “u”; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones “affect” and “effect”, “complement” and “compliment”, “discreet” and “discrete”, “principal” and “principle”.
- Do not confuse “imply” and “infer”.
- The prefix “non” is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the “et” in the Latin abbreviation “et al.”.
- The abbreviation “i.e.” means “that is”, and the abbreviation “e.g.” means “for example”.

An excellent style manual for science writers is [**b7**].

7.6. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

7.7. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is “Heading 5”. Use “figure caption” for your Figure captions, and “table head” for your table title. Run-in heads, such as “Abstract”, will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

7.8. Figures and Tables

Positioning Figures and Tables. Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation “Fig. 3”, even at the beginning of a sentence.

TABLE 1. TABLE TYPE STYLES

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity “Magnetization”, or “Magnetization, M”, not just “M”. If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write “Magnetization (A/m)” or “Magnetization {A[m(1)]}”, not just “A/m”. Do not label axes with a ratio of quantities and units. For example, write “Temperature (K)”, not “Temperature/K”.

Acknowledgment

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks …”. Instead, try “R. B. G. thanks…”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

References

[1] R. Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Science* 17 (1978), pp. 348–375.
[2] L. Damas and R. Milner. “Principal Type-Schemes for Functional Programs”. In: *POPL*. 1982.
[3] O. Shivers. “Data-Flow Analysis and Type Recovery in Scheme”. In: *Topics in Advanced Language Implementation*. MIT, 1990.
[4] C. Cifuentes. “Reverse Compilation Techniques”. PhD thesis. 1994.

Figure 3. Example of a figure caption.

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

[6] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. "Coping with Type Casts in C". In: *FSE*. 1999.

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

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

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

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

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

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

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

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

[15] 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, pp. 51–60.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[30] *Ghidra*. URL: https://ghidra-sre.org/.

[31] *Hex-Rays Decompiler*. URL: https://www.hex-rays.com/.

[32] *JEB Decompiler*. URL: https://www.pnfsoftware.com/.

[33] *Snowman decompiler*. URL: http://derevenets.com/.

[34] *Tigress: A Source-to-Source-ish Obfuscation Tool*. URL: https://tigress.wtf.