RADBOUD UNIVERSITY

MASTER'S THESIS
COMPUTING SCIENCE

# Seeing through obfuscation: interactive detection and removal of opaque predicates

*Author:*
Thomas RINSMA
thomasrinsma@gmail.com

*Supervisor:*
Dr Erik POLL
e.poll@cs.ru.nl

*Second reader:*
Prof. Marko VAN EEKELEN
m.vaneekelen@cs.ru.nl

*External supervisor:*
Eloi SANFELIX GONZALEZ
sanfelixgonzalez@riscure.com

*A thesis submitted in fulfilment of the requirements for the degree of Master of Science*

*in the*

Digital Security Group
Institute for Computing and Information Sciences

August 23, 2017

# Abstract

A program can be obfuscated to impede analysis of its inner workings or data while retaining its original functionality. The effectiveness of software obfuscation depends highly on the technique used, on the program that is being obfuscated and on the type of secret that is being hidden. Yet, under the right circumstances, an obfuscated program can require a considerably larger amount of time and effort to analyse than its non-obfuscated counterpart.

Different methods of obfuscation provide protection against different kinds of attackers. The focus of this thesis is on control-flow obfuscation and specifically on one method of control-flow obfuscation: the insertion of opaque predicates.

While control-flow obfuscation techniques were originally mostly intended to protect against manual analysis, upcoming automated software testing and analysis techniques such as symbolic execution have created a new challenge for obfuscators. Several automated deobfuscators make use of such analysis techniques [22] [6]. However, some of these automated deobfuscation techniques are either not available in (easy-to-use) tools or are limited to a very specific type of obfuscation and lack versatility.

This thesis gives an overview of the current state-of-the-art in opaque predicates and control-flow obfuscation in general, showing that for some of these obfuscations, little to no automated deobfuscation tools are available, at least not outside the realm of academic proofs-of-concept. It describes an attempt at partly filling this gap by the creation of an IDA Pro plugin called Drop, which implements an advanced opaque-predicate detection algorithm (Ming et al. [22]) in a versatile manner.

We show that Drop is able to detect and remove *invariant* and *contextual* opaque predicates in both artificial and non-artificial environments. Additionally, we show that the ability for an analyst to provide interactive input during the deobfuscation process can be rather useful in various scenarios, and the ability to provide a specific type of constraint on program arguments can even help detect and reverse an otherwise very strong type of control-flow obfuscation: *input-invariant* opaque predicates.

# Acknowledgements

I would like to thank my supervisor, Erik Poll, for his valuable feedback and his time spent reading many draft versions of this document over the past six months.

I also want to thank the people at Riscure. Specifically my external supervisor, Eloi Sanfelix Gonzalez, for his feedback and our insightful conversations; and Ileana Buhan, for helping me to find a topic and allowing me to stay at Riscure for this thesis project in the first place. The atmosphere and the people at Riscure have been great in general but I want to thank my friends and fellow interns specifically. Nils, Karolina, Parul, Roeland, Petros, Danilo and Romain: thank you for the support and the much needed distractions during these sometimes stressful months.

Finally, I'm very thankful to my parents for motivating me and encouraging me to pursue my computer (security) hobby in the form of this degree.

# Contents

# Chapter 1

# Introduction

Software *obfuscation* allows one to protect parts of an application such as algorithms or secret constants from analysis through reverse-engineering efforts. While a perfect protection from such analyses is probably practically – if not theoretically – impossible (see Section 2.2), obfuscation can still considerably increase the effort required by an analyst to obtain the sought-after secrets.

As is laid out in Section 2, there are many possible applications for software obfuscation including nefarious ones such as malware camouflaging. While this alone warrants the research on *deobfuscation* techniques, even legitimate users of obfuscation tools can benefit from such research because it gives us a better idea of the strengths and weaknesses of both obfuscation and deobfuscation techniques.

With the exception of several academic tools, there exist very few tools for automatic deobfuscation, especially when compared with the number of obfuscation tools out there (Section 2.3.6). This makes intuitive sense because it is harder to remove "noise" than it is to add it. Yet, the fundamental properties of obfuscation – specifically the functionality property – combined with practical limitations severely limit the possible "noisiness" of any applied obfuscation. For example, string constants in an application's source code can be obfuscated by encrypting them, but in order for the application to maintain its functionality they need to be decrypted again before they are printed to the screen. Similarly, one can insert bogus control-flow (e.g. by inserting dead code, see Section 2.3.1) into an application to obfuscate a secret algorithm, but careful analysis should show that such code is either never executed or does not contribute to the algorithm's result. This indicates that there is the possibility of automatic detection or removal of these obfuscations, and there are indeed techniques that do exactly this (e.g. [22] and [11]). Regarding control-flow obfuscation specifically (Section 2.3), it is sometimes possible to use modern code exploration techniques such as *symbolic execution* to detect or even revert these obfuscating transformations.

A problem arises when obfuscators start to exploit weaknesses of such automated deobfuscation techniques in the design of their obfuscating transformations (as Wang et al. [30] and Banescu et al. [4] do by causing path explosion in symbolic execution engines, see Section 2.3.1). Blindly applying an automated deobfuscation technique on programs obfuscated using these transformations will not be effective. In this thesis, we hypothesise that – especially in these cases, but also for less advanced obfuscations – there is a use for a *hybrid* deobfuscation workflow, i.e. not entirely by hand or fully automated, but somewhere in between.

## 1.1 Motivation

Existing techniques for automatic deobfuscation of control-flow obfuscations make little to no use of human assistance. Because human analysts are good at detecting patterns, they could possibly provide valuable input to a deobfuscation algorithm, *interactively* steering it in the right direction by providing additional context and thereby speeding up the reverse-engineering process. This could be especially helpful in cases where a fully automated deobfuscator would be very slow or would fail entirely, for example by spending large amounts of time analysing irrelevant code.

A notion that is almost intrinsically linked with interactivity is *visualisation*. The utility of any automatic deobfuscator highly depends on the type and quality of its output. It might act like a black box, transforming some obfuscated program $\mathcal{O}(P)$ into a (possibly partly deobfuscated) program $P'$, or it could (additionally) produce some other type of output representing the result of the analysis performed, such as a simplified control-flow graph. An *interactive* deobfuscator would need to produce a good visualisation of the obfuscated program and of the results of its analysis in order for the analyst to provide it with valuable input. There might be different types of visualization or hybrid methods (e.g. colouring basic-blocks or annotating the control-flow graph with symbolic constraints) that improve comprehensibility of the deobfuscated program.

We hypothesise that a deobfuscation tool with these two properties – i.e. interactivity and proper visualisation – can be helpful for an analyst looking to reverse-engineer certain types of control-flow obfuscations, particularly certain advanced variants of opaque predicates.

## 1.2 Thesis overview

This rest of this thesis is organised as follows:

- Chapters 2 and 3 provide an overview of background and related work on the topics of software obfuscation and deobfuscation respectively, both specifically focussing on opaque predicates and other control-flow obfuscations. Section 3.5 summarises some perceived problems with the state-of-the-art of control-flow deobfuscation.

- Chapter 4 introduces Drop, the tool produced for this thesis with the intention of providing a possible solution to some of the problems stated in the previous chapters.

- Chapter 5 evaluates the effectiveness of Drop in several experiments that aim to test different aspects of the tool.

- Chapter 6 draws conclusions from literature research and from the results obtained from the experiments in Chapter 5. Additionally, it lays out weak points of Drop and more generally details some possible topics for future work.

# Chapter 2

# Obfuscation

Software *obfuscation* is the act of obscuring algorithms or data contained in a program in such a way that it becomes hard (or at least *harder*) to extract or determine this content by someone with full access to the program and the system it runs on.

In a sense, software obfuscation is a method of *security through obscurity.* In cryptography, this concept is widely rejected: the security of a (crypto)system must not depend on the secrecy of its mechanism (i.e. *Kerckhoffs' principle*). Yet, when it comes to hiding the mechanism of a piece of software from an attacker who must still be able to execute that software on his own system (a so called *man-at-the-end* attacker), obfuscation – or at least a weak form of it – seems to be the only option available from a purely software perspective. Section 2.2 goes into detail on why "perfectly strong" obfuscation in the theoretical sense is not possible and why obfuscation is therefore usually (including in this thesis) discussed in a heuristic and practical sense.

What this means in practice is that given enough time and resources any obfuscated program can be reverse engineered. While this might seem discouraging, software obfuscation is useful in practice because it can nevertheless be quite strong and there are use cases where permanent protection is not necessarily required.

In the video game industry for example, where a large portion of sales often occur in a short period after the launch of a product, obfuscation is sometimes used to protect anti-piracy and anti-cheating code. In such a scenario, the focus is on protecting the code for (at least) that initial period of time, not permanently. Similar reasoning holds – more generally – for all proprietary software that receives frequent updates: if every version of a program is obfuscated in a different manner, the obfuscation only needs to "hold up" until the next version of the program is released. Of course, some reverse engineered knowledge about one version of the program might be applicable to the next version, so the efficacy of such a model highly depends on what the software author is trying to protect.

Software that handles copyright-protected media with certain restrictive measures (i.e. DRM[1]) is often obfuscated in order to protect decryption keys or algorithms. Specifically, *white-box cryptography* is often used for such applications, which by itself can be considered a form of obfuscation.

On a different note, malware is also often obfuscated. This is done in order to protect it from being detected by anti-virus software and to make manual analysis as difficult as possible. Additionally, if it exploits any yet-unknown vulnerabilities (i.e. *0-days*) then it is beneficial for the author to obfuscate code related to exploitation in order to hide details of the vulnerabilities and thereby delay their discovery by security researchers. On the other side of this arms race, software vendors will often obfuscate their released security *patches* in order to thwart analysis of the vulnerabilities that were patched.

[1]Digital Rights Management

## 2.1 Related concepts

Two closely related concepts to software obfuscation are software *watermarking* and software *tamper-proofing* [7]. There is a lot of overlap between techniques used for implementing these concepts and those used for (de)obfuscation. Generally, all transformations used for these goals – i.e. the obfuscation of (part of) a program, the application of a watermark to a program and the application of tamper-protection code to a program – share the notion that it should be *hard* to undo them. For this reason, watermarks and tamper-proofing code are often obfuscated or embedded within obfuscated code or data.

While these concepts are outside the scope of this thesis, their similarity to software obfuscation means that any results obtained here are likely to be relevant and applicable to these concepts as well.

## 2.2 Formal obfuscation theory

Theoretical research on obfuscation has been progressing ever since the 2001 paper of Barak et al. [5]. In this paper, Barak et al. provide a strong formal definition of an obfuscator (see below). Importantly, they also prove that obfuscation according to that (*virtual black-box*) definition is impossible in general, because there exists a family of programs that cannot be obfuscated.

Recently, however, there has been a surge of new research after Garg et al. [13] showed that a weaker form of obfuscation – so-called *indistinguishability obfuscation* – could be possible. Informally speaking, this roughly means that given an obfuscated program and several candidates for the corresponding original program, it is impossible to determine which of the candidates is the original program.

While these new results are promising and could potentially be useful for various kinds of cryptographic applications (such as fully homomorphic encryption [1]), they are far from being suitable for the purpose of practical software obfuscation. For this reason, work on practical software obfuscation approaches has largely been heuristic and empirical in nature.

Still, it is important to define the properties of an ideal obfuscator. Barak et al. [5] state the properties of an obfuscated program $\mathcal{O}(P)$ obtained by applying an obfuscator $\mathcal{O}$ to a program $P$ as follows (paraphrased informally):

**functionality** $\mathcal{O}(P)$ computes the same function as $P$.

**efficiency** $\mathcal{O}(P)$ is at most polynomially slower and larger than $P$.

**virtual black box** any information that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to $P$.

To state the last property in other words: there is no information to be gained from the obfuscated program that can not be gained from the input/output behaviour of the original program (nor from the input/output behaviour of the obfuscated program because they are functionally equivalent).

Several years before the publication of the formal definition above, Collberg, Thomborson, and Low [9] provided a more practical definition of an *obfuscating transformation* $\mathcal{T}$: Let $P$ and $P'$ be the source and target program respectively, then $P \xrightarrow{\mathcal{T}} P'$ is an obfuscating transformation from $P$ to $P'$ if $P$ and $P'$ have the same *observable behaviour*. The authors specifically note that this definition allows $P'$ to have extra side-effects as long as these are not experienced by the user. Additionally, in the case where $P$ encounters an error condition, $P'$ may or may not terminate [9]. This relaxation of the functionality property – compared to the definition of Barak et al. – allows for more flexibility in the design of obfuscating transformations.

In the same paper, Collberg et al. propose two variable properties in their attempt to assess the quality of an obfuscating transformation:

**potency** *To what degree is a human reader confused?* How much higher is the *complexity* of $P'$ compared to $P$, as measured by one of several measures of program complexity?

**resilience** *How well are automatic deobfuscation attacks resisted?* How much time is required to construct an automatic deobfuscator that is able to effectively reduce the potency of $\mathcal{T}$, and what is the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of $\mathcal{T}$?

While both of these properties are inherently imprecise, they clearly specify the distinction between the goals of obfuscating software against *human* attackers versus *machine* attackers.

## 2.3 Control-flow obfuscation

A *control-flow obfuscation* is any obfuscating transformation that obfuscates the control-flow of a program, as opposed to its data. This can be done by modifying the actual control-flow of the program statically (e.g. control-flow flattening, Section 2.3.2, virtualization, Section 2.3.3, or return-oriented programming, Section 2.3.5) or on-the-fly (self-modification, Section 2.3.4), by obscuring the actual control-flow paths within a larger collection of *bogus* control-flow (i.e. opaque predicates, Section 2.3.1) or by sequentially applying two or more of these transformations.

The main focus of this section is on opaque predicates because they are the main target of the deobfuscation tool produced for this thesis. Nevertheless, several other control-flow obfuscation techniques are summarised in Sections 2.3.2 to 2.3.5 to provide some broader context.

### 2.3.1 Opaque predicates

An opaque predicate is a predicate that is traditionally either always *true* or always *false*, but in such a way that this is hard to detect by human analysts and automated analysis. Opaque predicates typically allow one to insert pieces of bogus (i.e. dead) code and control-flow into a program which are never executed and obscure the actual functionality of the program. Interestingly, they can also be used in various ways for watermarking by inserting unique information in pieces of dead code [7] or as constants within predicates [3]. Furthermore, they can be used for non-control-flow obfuscation as well, e.g. as a basis for data obfuscation by constructing secret constants with arithmetic expressions containing opaque predicates.

The concept of the *opaque predicate* has been around for at least 20 years and has seen various improvements over the years to combat increasingly more advanced deobfuscation techniques. Collberg formalised opaque constructs based on simple number-theoretic properties and on array and pointer aliasing in 1997 [9] and various more advanced variants of them in *Surreptitious Software* in 2009 [8]. These constructs would traditionally be referred to as just *opaque predicates*, but in order to make the distinction with more advanced opaque predicate constructions more clear, we will refer to them as *invariant* opaque predicates (following the terminology of Ming et al. [22]):

**Definition 2.3.1** *An **invariant opaque predicate** is a single predicate whose value is constant, independent of the values of its contained variables, program state or user input. It is an invariant in and of itself.*

Described below are two common concepts on which such invariant opaque predicates (but also more advanced types of opaque predicates, as described further along in this section) can be based: number theory and aliasing.

### Number-theoretic predicates

Number-theoretic constructs used in opaque predicates often follow the form:

$$\forall x \in \mathbb{Z} : n | f(x)$$

i.e. $f(x) \equiv 0 \pmod{n}$, where $f(x)$ is usually a simple polynomial over $x$. Examples of such predicates are $x^2 + x \equiv 0 \pmod{2}$ and $x^3 - x \equiv 0 \pmod{3}$, both of which are opaquely true for all integer values of $x$. Similar constructs can however also be created with multiple variables. An example of this is the following opaquely false predicate on all integers $x$ and $y$: $7y^2 - 1 = x^2$.

### Aliasing-based predicates

Opaque predicates based on so-called *array aliasing* and *pointer aliasing* are both largely based on the principle of constructing a data-structure in a seemingly random manner but following certain invariants. Predicates can then be constructed from properties derived from these invariants. Additionally, to increase potency (and possibly resilience), the data-structure can be updated freely during the execution of the program as long as the invariants are not violated. Natural language examples of such invariants are: *"every third element of the array is 1 (mod 5)"* in the case of an array-like data-structure, or *"pointers $p_1$ and $p_2$ never point to the same node"* in the case of a linked-list or graph-like data-structure [8].

### Contextual opaque predicates

In their overview on intellectual property protection, Drape et al. [12] highlight the lack of stealthiness as a disadvantage of constructs such as those proposed by Collberg, arguing instead for (what are now often called) *contextual* opaque predicates. The truth value of a contextual opaque predicate is derived from its context within the program, specifically on one or more invariant properties. This invariant context is created by one or more pre-condition predicates which are already present in the program (or are implied). The contextual opaque predicate is then constructed such that the conjunction of these pre-conditions implies the truth value of the contextual opaque predicate.

**Definition 2.3.2** *A **contextual opaque predicate** is an opaque predicate whose value **is not** constant by itself, but **is** constant given a certain invariant (i.e. some other property which always holds at the location of the predicate). It is therefore not self-contained, but apart from that it has the same effect as an invariant opaque predicate.*

To give an example, the predicate $3x > 14$ is not opaquely true by itself, but together with the context of $x > 4$ it forms a contextual opaque predicate[2]. For maximum

[2]Ignoring the possibility of an integer-overflow.

stealthiness this context is chosen to be created as implicitly as possible, e.g. as a side-effect of a pre-existing calculation.

### Input-invariant opaque predicates

A related concept to the above is a construct based on what Banescu et al. [4] call an *input invariant.* When applied to opaque predicates, as the authors of Tigress did with their `input`-type predicates (see Section 2.3.6), the result is effectively a contextual opaque predicate whose context is based on properties of the program's input.

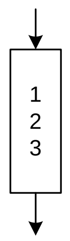
**Definition 2.3.3** *An **input-invariant opaque predicate** is an opaque predicate whose value **is not** constant by itself, but **is** constant given that a certain predicate over properties over the arguments of the program (the "input invariant") holds.*

This input invariant is specified by the author of the program at obfuscation-time and is usually chosen to be based on some weak restrictions that already trivially hold during normal runs of the program (e.g. a parameter specifying a count of something will never be negative). However, when an input is provided such that the invariant does *not* hold, the predicate might evaluate to a different value, causing arbitrary behaviour. This technique violates the functionality property defined by Barak et al. [5] because the obfuscated program does not exhibit the same functionality as the original program when such arbitrary behaviour occurs. However, because arbitrary behaviour only occurs in what are effectively error conditions anyway, this *violation* of the functionality property is in fact rather similar to its *relaxation* by Collberg and Thomborson [7] (as described in section 2.2).

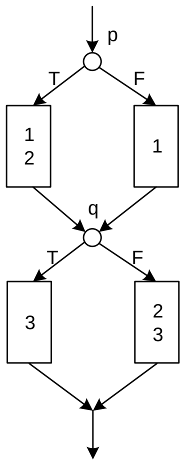
At the cost of this, Banescu et al. [4] argue that programs obfuscated with input-invariant opaque predicates (and other input invariant constructs) are harder to deobfuscate automatically using symbolic execution–based techniques. This is in fact their main argument for the use of input-invariant opaque predicates. The idea being that the set of constraints encountered by the symbolic execution–engine will not logically imply the opaqueness of input-invariant opaque predicates, as it does for other types of opaque predicates (see Section 3.3.1). After all, these predicates are not technically opaque in the traditional sense, because there are executions of the program resulting in a different value for the predicate (namely those where the invariant does not hold). Determining the validity of one such execution (i.e. *Is this the intended execution path?*) is a very hard and entirely different problem, and is outside the scope of a symbolic execution–engine.

However, the invariants provided to the obfuscator must be predicates over the program's arguments that hold for every valid execution. They can therefore not be much[3] weaker than the program's (presumably documented) constraints over these arguments, as provided to the user. This means that it might be relatively easy for an analyst to make educated guesses as to what invariants were provided to the obfuscator. Given such guessed invariants and given that they are strong enough, a deobfuscator *can* use logic again to determine the opaqueness of a predicate. In this thesis (Section 3.5 specifically), it is hypothesised that a deobfuscator with a certain amount of interactivity can in fact be used to attack such constructs in practice.

[3]Depending on to what degree the functionality property of the obfuscator is allowed to be violated.



(A) Before insertion



(B) After insertion

FIGURE 2.1: Inserting the correlated predicates $p$ and $q$ that together form a dynamic opaque predicate (figure from [31]).

### Dynamic opaque predicates

Palsberg et al. [23] classify both invariant and contextual opaque predicates as static opaque predicates and introduce the concept of the *dynamic opaque predicate*[4]:

**Definition 2.3.4** *A* ***dynamic opaque predicate*** *is part of a tuple of correlated predicates which always either evaluate to the* same *or the* opposite *truth value, which might be different at different runs of the program. These predicates are inserted in such a way that all possible control-flow paths along the predicates have the same functionality.*

Figure 2.1 shows such a case where two correlated predicates $p$ and $q$ are created and inserted such that their truth value is always equal, yet can arbitrarily differ at different runs of the program without altering the functionality of the program. In this case, this is visualised by the instruction sequence $\{1, 2, 3\}$ which is equivalent to both $\{1, 2\}$ followed by $\{3\}$ *and* to $\{1\}$ followed by $\{2, 3\}$.

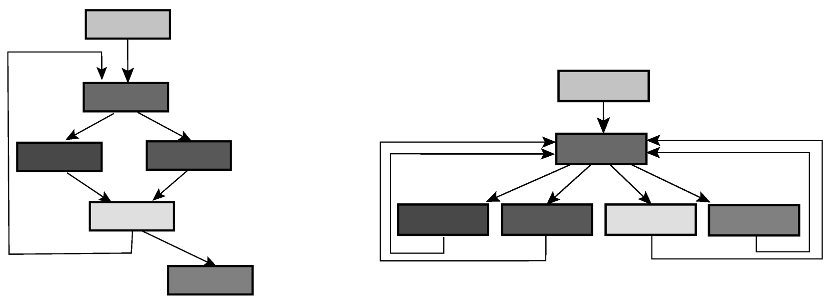
These dynamic opaque predicates are arguably more difficult to construct than static opaque predicates because both branches of each condition need to contain valid code and the functionality of all possible code-paths along the branches needs to be the same. An obfuscator is restricted in its ability to insert such predicates by the type and number of instructions in a basic-block, i.e. the longer a basic-block, the more ways in which it can be split up.

A generalized method of automatically inserting such dynamic opaque predicates in structures such as branches and loops in addition to straight-line code was developed by Xu, Ming, and Wu in 2016 [31]. They argue that these *generalized dynamic opaque predicates* are harder to deobfuscate than regular dynamic opaque predicates because the generalization of the insertion algorithm allows iterative insertion of predicates (i.e. running the obfuscator on its own output iteratively), which obfuscates which predicates are correlated.

### Other anti–symbolic execution opaque constructs

Meanwhile, more research has also been conducted on improving the robustness of non-dynamic opaque predicates against deobfuscation techniques. Wang et al. [30]

[4]Sometimes instead referred to as a *dynamically* opaque predicate.



(A) Before control-flow flattening (B) After control-flow flattening

FIGURE 2.2: Control-flow graph of an example function before and after applying control-flow flattening (figures from Tigress documentation).

proposed the use of sequences such as the Collatz sequence to cause path explosion in symbolic execution engines. The Collatz (or 3n+1) sequence is created by iteratively applying the following function to its own output, starting with a positive integer $n$:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n+1 & \text{if } n \equiv 1 \pmod{2} \end{cases}$$

The resulting sequence of this iterative application always seems to converge to a fixed value (namely 1) after a finite number of steps. Collatz conjectured that this property holds for all positive initial values, but this has not been proven. This (unproven) property holds for a generalized set of similar *Collatz functions*.

The idea of Wang et al. is to construct a loop in which such a formula is iteratively applied until the fixed-point is reached, starting with a value of $n$ that is dependent on user-input. As soon as the fixed-point is reached, the obfuscated piece of code is executed and the loop exits. Because of the apparent dependency on a symbolic value (user input), a naive symbolic execution engine will have to consider both possible branches at each loop iteration, quickly generating rather complex (non-linear) constraints which are difficult to solve.

### 2.3.2 Control-flow flattening

Another well-established method of code obfuscation is *control-flow flattening*, as formalised by Chenxi Wang in 2001 [29] (and therefore sometimes nicknamed *chenxification*). The core idea of this technique is the transformation of all control-flow transfers into indirect transfers. This is achieved by effectively turning the function into a switch statement within an unconditional loop. Each basic block is assigned a branch of the switch statement, allowing control-flow transfers to occur by assigning to the *control variable* the value of the basic block that is to be executed next. The resulting control-flow graph shows a single *dispatcher* block as the sole successor and predecessor of every other basic block in the function[5], thereby completely hiding the actual control-flow of the function. Figure 2.2 shows the effect of this technique on the control-flow graph of a small function.

[5]Depending on both the compiler and the disassembler used, the dispatching can be shown to occur in several steps, creating a (still obscure) "ladder" of dispatcher blocks.

### 2.3.3 Virtualization obfuscation

Virtualization-based techniques can result in a control-flow graph that looks similar to the result of control-flow flattening. Yet, these techniques work according to an entirely different principle. During obfuscation, a selected portion of code is translated (i.e. compiled) into a custom (often RISC-like) instruction-set. This translated code together with an interpreter of its instruction-set is what then makes up the obfuscated program. To impede reverse-engineering, a virtualization obfuscator will often randomize the instruction-set encoding for each program it obfuscates. Additionally, further obfuscation can be performed on parameters or on the ordering of parameters of specific instructions [24].

Because of the indirect nature of the control-flow in the resulting obfuscated programs, they can be difficult to analyse. An attacker is forced to reverse engineer both the instruction-set encoding and the translated program itself.

### 2.3.4 Runtime code generation and modification

A more efficient and versatile alternative to virtualization is runtime code generation (i.e. just-in-time (JIT) compilation). Obfuscators employing this technique generate obfuscated code which itself generates native code during runtime, which is subsequently executed. Somewhat similarly to virtualization obfuscation, the original program is transformed by the obfuscator into a list of instructions in an intermediate language. These instructions are in this case converted to *jitting* code, which itself can be obfuscated and diversified in a variety of manners (e.g. Tigress's jitting obfuscation allows this with options such as `--JitObfuscateArguments` and `--JitImplicitFlowHandle`).

A disadvantage of runtime code generation is that the obfuscator needs to have knowledge of the target architecture. This disadvantage is not necessarily shared by any of the other previously mentioned types of obfuscating transformations.

Instead of generating all of the native code at once before executing it, the generation and execution can be intertwined in various ways. A piece of generated code itself can for example contain a key which is needed to generate the next piece of code. This general concept is known as *self-modification* [16] [2].

### 2.3.5 Return-oriented programming

Return-oriented programming (ROP) is an exploitation technique that – under the right conditions – allows arbitrary code execution in environments with $W \oplus X$ protection (i.e. environments where writeable memory cannot be executed). It works by chaining together pieces of pre-existing code to obtain and execute a malicious program, circumventing the need to write such a program to an executable memory segment.

While it was originally designed for exploitation, return-oriented programming can also be used as an obfuscation technique, i.e. by hiding a (sub-)program within a larger innocuous program. Lu, Xiong, and Gao [20] put this in practice with their *RopSteg* obfuscator which allows for program *steganography* by using return-oriented programming.

### 2.3.6 Tools

It is difficult to get a good picture of the features and functionality of all the obfuscators that are being used because many of them are proprietary or in-house products. Still, many tools are publicly available to use for different use-cases.

One such use-case is the protection of Android applications. Android applications are usually Java-based, and Java code can often be reconstructed rather accurately from compiled byte-code when no obfuscation is applied. Tools like ProGuard[6] and Shield4J[7] protect Java-based applications from reverse-engineering by (among other things) obfuscating the names of classes and class members. Other tools such as Allatori[8], DashO[9] and DexGuard[10] are able to also perform control-flow obfuscation. Similar tools exist for .NET-based applications for Windows, which are similarly easy to decompile when no obfuscation is applied.

More interesting however for this thesis is the obfuscation of native code. Two publicly available academic tools which provide control-flow obfuscation functionality for native code are Obfuscator-LLVM (OLLVM)[11][15] and Tigress[12][4]. Whereas OLLVM only supports control-flow flattening and the insertion of one type of opaque predicates and is not very configurable, Tigress supports almost every control-flow obfuscation technique that has been theorized while being configurable with over a hundred command-line options.

[6] `https://www.guardsquare.com/en/proguard`
[7] `http://shield4j.com/`
[8] `http://www.allatori.com/`
[9] `https://www.preemptive.com/solutions/android-obfuscation`
[10] `https://www.guardsquare.com/en/dexguard`
[11] `https://github.com/obfuscator-llvm/obfuscator/`
[12] `http://tigress.cs.arizona.edu/`

# Chapter 3

# Deobfuscation

Software *deobfuscation* can be defined as the inverse of software *obfuscation*. Nevertheless, obfuscation transformations are often not completely invertible, i.e. for most obfuscators $\mathcal{O}$, there is no possible deobfuscator $\mathcal{O}^{-1}$ such that $\mathcal{O}^{-1}(\mathcal{O}(P)) = P$ for all programs $P$. The mere transformation of source code to a binary executable file – as performed by a compiler – is an example of an irreversible obfuscation: information such as comments and variable names is irreversibly lost during compilation. For this reason, *deobfuscation* of an obfuscated program is often taken to mean the act of extracting any obfuscated "secrets" from the program to the degree needed for the use case of the analyst. In the compiler example where variable names are lost, one can still extract a proprietary algorithm from the compiled program by analysing its disassembly output.

Section 3.1 describes a typical reverse-engineering workflow and the incorporation of deobfuscation tools. Section 3.2 introduces the different types of analyses techniques that are used by automatic deobfuscation techniques. Section 3.3 lists some previous work on automatic deobfuscation that targets specific transformations, while Section 3.4 details more general and transformation-oblivious attacks.

## 3.1 Workflow and tools

The process of deobfuscation often consists at least partially of manual analysis. However, more and more free and proprietary deobfuscation tools are being created, most of which are designed for a specific architecture, platform, or category of obfuscations. This section will give a short and non-exhaustive overview of the available tools for a few different use-cases.

One such use-case is the reverse-engineering of (usually Windows and x86) malware. Malware applications often make use of data obfuscation transformations such as packing and string obfuscation. Various tools are available for reverse-engineering such transformations such as PackerAttacker[1] and FLOSS[2]. When control-flow is obfuscated as well, one can use VirtualDeobfuscator[3] for virtualization-based obfuscations and de4dot[4] for .NET executables obfuscated with various kinds of data and control-flow transformations.

Another category of software for which many deobfuscation tools are available is Android software, and to a lesser degree, Java applications in general. Many tools are able to transform obfuscated Java or Dalvik byte-code back into Java code. Examples

[1] `https://github.com/BromiumLabs/PackerAttacker`
[2] `https://github.com/fireeye/flare-floss`
[3] `https://github.com/jnraber/VirtualDeobfuscator`
[4] `https://github.com/0xd4d/de4dot`

of this are JMD[5], DeGuard[6] and simplify[7]. Additionally, there is JEB[8], which will decompile both Dalvik and native Android code with support coming for several deobfuscation transformations including control-flow unflattening and the removal of opaque predicates in the upcoming version 2.3.

Outside of the Android domain, there exist several more general tools for control-flow deobfuscation. JAKSTAB[9] implements several static analysis techniques to allow for accurate control-flow recovery and automatic reversal of virtualization-based obfuscations on (x86) Windows and Linux platforms [18] [17]. The *logic oriented opaque predicate checker* (LOOP)[10] is a research tool based on Intel's PIN and the Binary Analysis Platform (BAP) that allows for the automatic detection of several types of opaque predicates on x86 platforms [22]. Section 3.3.1 contains a description of the techniques implemented in this tool.

There are also several IDA PRO plugins related to deobfuscation: VMAttack[11], which enables an analyst to attack (stack-based) virtualization obfuscations; and Optimice[12], which is able to detect some simple opaque predicates and remove dead code. Sadly, it seems that Optimice has been abandoned while it is still in a relatively undeveloped state.

## 3.2 Analysis techniques for automated deobfuscation

Software analysis techniques can roughly be divided into two categories: *static* and *dynamic*. Both of these are described below in terms of how they apply to (control-flow) deobfuscation.

### 3.2.1 Static analysis

The field of static program analysis encompasses all techniques that don't necessitate execution of the program under analysis. For the purpose of deobfuscation, various techniques can come in handy. A prerequisite for most analyses is the generation (i.e. recovery) of the control-flow graph of the program or function. This is a hard problem but various heuristics make it possible to recover a good approximation in many cases [27]. The theoretical recovery of a complete and sound control-flow graph in fact inherently requires some control-flow deobfuscation because this graph should not contain unreachable basic-blocks such as those caused by opaque predicates.

In practice, techniques such as *program slicing*, *data-flow analysis* and *abstract interpretation* can be applied on top of a CFG in order to more precisely determine relations between instructions, basic-blocks and functions. Some of these techniques are used by both compilers and deobfuscators to perform optimisations such as dead-code removal and constant propagation.

#### Symbolic execution

A specific static analysis technique that has been used in techniques for automated deobfuscation recently is symbolic execution.

[5]https://github.com/contra/JMD
[6]http://apk-deguard.com/
[7]https://github.com/CalebFenton/simplify
[8]https://www.pnfsoftware.com/jeb2/
[9]http://www.jakstab.org/
[10]https://github.com/s3team/loop
[11]https://github.com/anatolikalysch/VMAttack
[12]https://code.google.com/archive/p/optimice/

The principle behind symbolic execution is as follows: instead of executing a program on inputs of actual values, its execution is simulated by passing *symbolic* values instead. Program variables whose value would depend on these input values now contain symbolic expressions. Additionally, a special symbolic expression called the *path condition* is maintained during execution. The path condition is a predicate which is at all points during the execution a conjunction of conditions over symbolic variables, that must be satisfied in order to reach the current point in the program. A conditional branch instruction therefore splits the symbolic execution up into two "branches", each with a different path condition (namely, the previous path condition conjoined with either the branch condition or the negation of the branch condition). Because a program can have an infinitely large number of such branches (e.g. because of an infinite loop), the symbolic execution *tree* can get arbitrarily large if path-trimming heuristics are not applied.

The concept of symbolic execution was originally formulated by King [19] in 1976 as an alternative to traditional program testing techniques for assuring that a given program meets its requirements. Looking back, King's paper has been very influential and many innovations and different applications of symbolic execution have since arisen as theorem provers and constraint solvers have gotten more efficient.

In the context of control-flow deobfuscation, symbolic execution is interesting for its ability to semantically represent (part of) a program. Ideally, the symbolic formulas constructed by symbolically executing a program represent *what* is being calculated, which is theoretically independent from *how* it is calculated, i.e. how the control-flow is laid out. This property is useful for deobfuscation because control-flow obfuscations often complicate the manner in which a computation is performed without changing the result of the computation (functionality property).

Section 3.3.1 and 3.4 mention some specific cases where symbolic execution has been used for the purpose of deobfuscation.

### 3.2.2 Dynamic analysis

Dynamic analysis techniques are based on the execution of the program under test. Software *testing* is a form of dynamic analysis.

Apart from the program's output, various aspects of the program's behaviour can be analysed as well. A specific form of program behavioural monitoring is *instruction-level tracing*: recording every instruction that gets executed when the program is executed on a certain input. The result of this – a *trace* – can be analysed afterwards with static-analysis techniques.

A somewhat related concept is *concolic* (*conc*rete + symb*olic*) execution. This a hybrid between concrete execution (i.e. testing) and symbolic execution. The key innovation of this concept being that the program under test is initially ran with concrete input values in order to obtain a trace from which the symbolic path condition is extracted. An SMT[13] solver is then used to obtain new concrete input values which cause the execution to take a different branch, by inverting conjuncts of the path condition. This process is repeated in order to quickly explore more possible paths.

## 3.3 Deobfuscation attacks on specific transformations

There are two types of approaches that can be taken when building a deobfuscator. On the one hand, one can take a general and transformation-oblivious approach

[13]Satisfiability Modulo Theories

by using advanced analysis techniques such as symbolic execution to, for example, simplify a program or recover key parts of an algorithm. On the other hand, one can target a specific obfuscating transformation – or even the implementation of a transformation by a specific obfuscator. In this section I will examine deobfuscation techniques and tools of the latter kind, specifically regarding opaque predicates and control-flow flattening.

### 3.3.1 Opaque predicates

In 2006, Madou [21] showed that a dynamic approach can be capable of detecting opaque predicates in an obfuscated program. His technique roughly consists of analysing which branches appear to be unconditional during the runtime of a program, before fuzzing those branches with random inputs to make a more informed guess as to their opacity. While the results obtained from such approaches are neither sound nor complete, they can be efficient and possibly provide a starting point for further analysis.

Dalla Preda et al. [10] have shown that some number-theoretic opaque predicates (i.e. those of the form $\forall x \in \mathbb{Z} : n|f(x)$) can be attacked statically by an abstract interpretation–based approach. Alternatively, symbolic or concolic execution–based techniques can be used to find and eliminate more advanced (e.g. contextual) opaque predicates, as shown by Ming et al. [22] and David et al. [11].

### 3.3.2 Control-flow flattening

Control-flow flattening transformations can sometimes be reversed entirely statically, as shown by Udupa et al. [28]. The authors note however that their approach does not work well when faced with interprocedural interactions or pointer aliasing.

## 3.4 General attacks

In addition to these "attacks" on specific obfuscation transformations, there has been an effort to create a general approach to control-flow deobfuscation by Yadegari et al. [32]. They propose a system that combines concolic execution and taint analysis with a set of transformation-oblivious simplification rules. After experimentation they conclude that this system is effective at deobfuscating programs that have been obfuscated using virtualization and return-oriented programming.

Salwan [25] also successfully took a transformation-oblivious approach by using symbolic execution to attack Tigress's challenge programs, which are obfuscated using multiple combinations of different transformations.

## 3.5 Problem statement

As is outlined in this chapter, multiple methods of attacking control-flow obfuscations have been theorized over the years and several of them have been implemented in publicly accessible tools (e.g. LOOP and Jakstab, see Section 3.1). However, apart from not being very versatile and easy to use, these tools remain mostly ineffective against certain recent innovations in control-flow obfuscations. Specifically, they are unable to detect input-invariant opaque predicates (see Section 2.3.1). Such input-dependent techniques can cause path-explosion in naive symbolic execution–based deobfuscators because these deobfuscators don't possess any knowledge about invariant properties of the program input.

A human analyst, however, might be able to spot such invariants and provide them as additional context to a deobfuscator, allowing it to reason about constructs like input-invariant opaque predicates. In general, for a deobfuscator to be as versatile as possible, it should allow for human input at many points during the deobfuscation process. The analyst could for example select candidate predicates to analyse, specify information about the program's global state, or specify functions to be hooked to simplify analysis. Such an interactive deobfuscator would be more robust against different variations and future enhancements of obfuscation techniques than deobfuscators that currently exist.

Chapter 4 introduces the deobfuscation tool produced for this thesis with the intent to test the hypothesis that such interactivity is helpful in various cases and might even allow it to detect input-invariant opaque predicates, a previously undetectable type of obfuscation.

# Chapter 4

# Introducing Drop

In Section 3.5, a problem with existing deobfuscation tools is identified. Namely, their lack of user-friendliness, versatility and their inability to effectively reverse certain advanced context- and input-dependent control-flow obfuscations. The concept of a more versatile and interactive deobfuscation tool was proposed which would be able to attack various obfuscated functions automatically and certain others by leveraging the skills of a human analyst.

This proposed solution was put into practice in a tool called Drop (short for Drop Removes Opaque Predicates). Drop is a plugin for the widely popular Interactive Disassembler (IDA Pro, also referred to as just IDA). There are several reasons why IDA Pro was chosen as a 'host' platform for Drop: it has a rich plugin API which allows plugins in different languages including Python and C++ to tap into most of its functionality and interface; it has built-in and community-developed support for loading binaries built for almost every architecture and platform; and lastly, it is already widely used for reverse-engineering purposes and is often the main tool in a reverse-engineering workflow.

Section 4.1 specifies the ideal non-functional and functional requirements of Drop. Section 4.2 details the choices that were made regarding the libraries and frameworks used by Drop. Section 4.3 describes how the required functionality was implemented.

## 4.1 Requirements

While Drop is primarily a research tool intended for experimentation with interactive deobfuscation techniques, there are some over-arching requirements that a polished version of such a tool should ideally satisfy:

- It should be *cross-platform*, i.e. it should work on all platforms on which IDA runs: Windows, Mac OS and Linux.
- It should be as *architecture-agnostic* as possible, allowing the analysis of binaries of as many architectures as supported by IDA and the framework(s) used.
- It should be *versatile* in the sense that it can be applied to programs obfuscated with any obfuscator of a certain class.
- It should be *well-integrated* in the user-interface of IDA, using its pre-existing static analysis and visualisation capabilities to the maximum extent.

Adhering to these ideals as much as possible allows us to determine to a degree whether or not such a universal tool is even feasible. These requirements are currently not all satisfied to the fullest degree by Drop, but they served as guidelines during the design and implementation phases. Additionally, the possibility of making Drop publicly available at some point in the future was not ruled out. Specifically, the possibility of a submission to the yearly IDA PRO plugin contest[1] was kept in mind.

### 4.1.1 Specific functionality

The functionality of DROP is focussed on attacking a narrow subset of the set of all types of control-flow obfuscations. Specifically, it is focussed on the detection of several kinds of opaque predicates, either automatically or with the help of interactivity.

From recent literature (Section 3.3.1) it appears that methods using symbolic execution can be effective at detecting opaque predicates in a general manner, i.e. without needing a preprogrammed list of common opaque predicate constructions. For this reason, DROP's opaque predicate discovery functionality was based on the algorithm proposed by Ming et al. [22]. Because this algorithm makes use of symbolic execution, an appropriate symbolic execution engine needed to be chosen. Section 4.2 outlines the options available and what choice was made.

Section 4.3 outlines the details of the implemented opaque predicate detection algorithm and it shows how a symbolic execution–based approach is ideal for the purpose of interactively adding constraints or symbolic values to a function or branch.

In addition to the ability to detect opaque predicates with or without interactive input, several requirements were set for DROP's functionality. Firstly, it would need to to have some method of visualising the detected opaque predicates and bogus code within the IDA PRO interface. Secondly, if possible within the constraints of the API provided by IDA PRO, DROP would need to have the ability to actively remove any detected bogus control-flow and dead code resulting from opaque predicates from the control-flow graph view or even from the assembly listing itself.

All of these requirements are satisfied to some degree in the current version of DROP. Several tests of effectiveness and practicality have been performed, of which the results are shown in Chapter 5.

## 4.2 Architectural choices

One of the goals during the creation of DROP was to expand upon work such as that by Ming et al. [22] and David, Bardin, and Marion [11], both of which use symbolic execution to detect opaque predicates. While it is possible to implement a symbolic execution engine from scratch, this is not necessary. There are several freely available libraries that provide functionality ranging from not much more than just constraint-solving to full-system symbolic or concolic execution. In our case, the main deciding factors were (1) what language it is written in (or which bindings are available) and (2) which architectures are supported for the programs under analysis. Specifically, because DROP is an IDA PRO plugin, a symbolic execution framework is needed that can be controlled through C++ or Python and it should preferably support as many architectures as possible to take advantage of IDA's wide range of supported architectures.

Six symbolic execution frameworks were considered: TRITON[2], BAP[3], KLEE[4], S2E[5], ANGR[6], and MIASM[7]. See Table 4.1 for their respective properties. Out of

[1] `https://www.hex-rays.com/contests/index.shtml`
[2] `https://triton.quarkslab.com/`
[3] `https://github.com/BinaryAnalysisPlatform/bap/`
[4] `https://klee.github.io/`
[5] `http://s2e.epfl.ch/`
[6] `http://angr.io/`
[7] `https://github.com/cea-sec/miasm`

| Framework | Language (bindings) | Architectures | Supported OS | Binary analysis |
|---|---|---|---|---|
| KLEE | C/C++ | LLVM IR | L | No |
| S2E | C/C++ | x86(-64), ARM | L | Yes |
| TRITON | C++, Python | x86(-64) | L,W,M | Yes |
| BAP | OCaml, C, Python, Rust | x86(-64), ARM | L,M | Yes |
| MIASM | Python | x86(-64), ARM(64), MIPS, SH4, MSP430 | L,W,M | Yes |
| ANGR | Python | x86(-64), ARM(64), MIPS(64), PPC(64), AVR | L,W,M | Yes |

TABLE 4.1: Symbolic execution engines that were considered. The third column shows supported architectures of the file under analysis, whereas the fourth column shows the supported host operating systems: L = Linux, W = Windows, M = Mac OS.

these options, for the reasons described above, ANGR was chosen: it is a Python framework, it supports a wide range of architectures and on top of that it has many static-analysis techniques built-in. [26] MIASM would probably be another good choice, given that it is also a Python framework and that it also supports a wide range of architectures. Compared to ANGR however, it is a lot more lightweight and it lacks some of the functionality for high-level control over the symbolic execution engine that ANGR has. KLEE by itself does not support binary analysis without access to an application's source code. S2E alleviates this by building upon KLEE and supporting the lifting of binaries to the intermediate representation used by KLEE (LLVM IR), but only for a select few architectures. TRITON and BAP both appear like well-rounded frameworks, but they too support only a select few architectures.

## 4.3 Implementation

The functionality of DROP is contained within four Python files:

**`drop.py`** The main plugin file which is initially loaded by IDA PRO. It contains the main plugin class which keeps track of plugin- and project-wide state.

**`helpers.py`** This file contains various helper functions for interfacing with IDA PRO and the user interface.

**`gui.py`** This file contains the `PluginForm` class representing the layout of the plugin's panel together with event-handlers for the various actions that can be performed. See Section 4.3.1 for a detailed overview of the graphical user interface.

**`workers.py`** This file contains the worker classes that perform the analyses corresponding with buttons in the panel. Specifically, the following workers are implemented: `ConcreteRunner`, `OpaquePredicateFinder`, `UnreachabilityPropagator` and `UnreachableCodePatcher`. Sections 4.3.2 to 4.3.5 explain these workers and their usage in detail. The workers are initialised with all required information from the IDA database before being executed in a separate thread. During their execution, the user interface is available and the user is able to interact with IDA as normal. When a worker completes, it sends its results back to the main thread through a Qt *signal*, allowing the results to be shown in the plugin panel and the IDA graph view and disassembly listing.

### 4.3.1 User Interface and Workflow

When a file is loaded by IDA, Drop automatically attempts to load it as a new angr project. When activated by pressing the shortcut key (`Alt+F6`) or by selecting it from the *Plugins* menu item, Drop manifests itself as a panel to the side of the main disassembly view in IDA Pro. The panel is an instance of the IDA `PluginForm` object, in which arbitrary Qt widgets can be drawn. Figure 4.1 shows this panel with its components annotated with circled numbers, corresponding to those mentioned below.

The panel is arranged in a three-step layout. Each step can be performed with varying degrees of interactivity. The steps are:

1. Finding candidate code-blocks[8] that could contain opaque predicates.
2. Detecting which code-blocks from the candidate set contain (which type of) opaque predicates.
3. Propagation of unreachability information and dead code removal.

During the first step, the user can either manually mark blocks as candidates ① or let Drop execute the current function concretely to generate a trace ② of visited code-blocks, which will then be marked as candidates.

In the second step, the user is able to provide the symbolic execution engine with three types of additional context ③④⑤ (see Section 4.3.2) before starting opaque predicate detection on all marked code-blocks ⑥ or only on the selected code-block ⑦.

If any opaque predicates were found in the second step, the user can instruct Drop to propagate this unreachability information ⑨ and possibly manually mark code-blocks as unreachable ⑧. Finally, an optional button press ⑩ will cause Drop to attempt to patch no-op instructions over the code-blocks that are marked as unreachable.

Beneath all of these step-by-step controls, the panel contains a field ⑪ where any detected opaque predicates are listed, showing the address of the code-block containing the opaque predicate, its type, and the detected tautology.

The process of using Drop is also demonstrated in three screen-casts that can be found online[9].

[8] A code-block is a node in an IDA function control-flow graph

[9] `https://th0mas.nl/downloads/thesis/drop_demo_videos.zip`

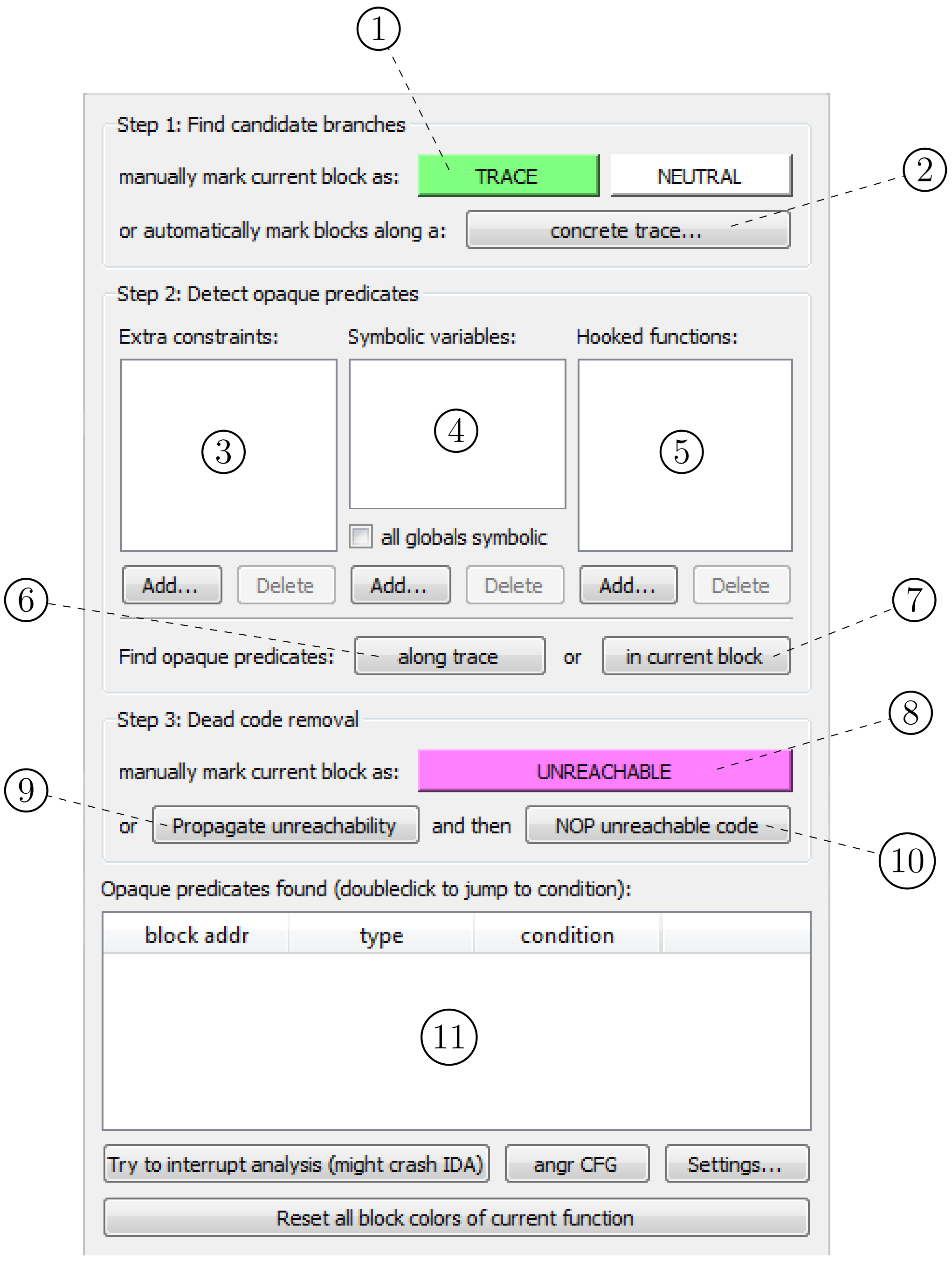

1
2
Step 1: Find candidate branches
manually mark current block as:
TRACE
NEUTRAL
or automatically mark blocks along a:
concrete trace...
Step 2: Detect opaque predicates
Extra constraints:
Symbolic variables:
Hooked functions:
3
4
5
all globals symbolic
Add...
Delete
Add...
Delete
Add...
Delete
6
7
Find opaque predicates:
along trace
or
in current block
Step 3: Dead code removal
8
manually mark current block as:
UNREACHABLE
9
or
Propagate unreachability
and then
NOP unreachable code
10
Opaque predicates found (doubleclick to jump to condition):
block addr
type
condition
11
Try to interrupt analysis (might crash IDA)
angr CFG
Settings...
Reset all block colors of current function

FIGURE 4.1: The main DROP panel with annotated buttons and listings.

### 4.3.2 Interactive input

In addition to the interactive manner in which unreachability and trace information is being stored (through marked code-blocks in IDA PRO's assembly listing and graph view), DROP allows the user to tweak the context of the opaque predicate discovery algorithm in several ways:

- Additional constraints can be specified that will be added to the constraint-solver during the opaqueness checks. These constraints are (in-)equalities over two operands which can be: the value at a specified memory address; a local variable or argument (i.e. a stack offset); a specified register; or a constant.

  In addition to these regular combinations of operands, a special type of constraint can be added: an (in-)equality over the integer-converted value of a specific element of a string-array.[10] This rather specific constraint-type is specifically intended to be used for input-invariant opaque predicates that use the `atoi` function on a specific element of the list of command-line arguments. An example of such a constraint would be: `atoi(str_arr[2]) >= 42`.

- Global variables by default contain their initial value during the symbolic execution. In some cases this is not desired, e.g. when it is modified by a different function during regular execution. It is therefore possible to specify individual global variables to be initialised as an unconstrained symbolic variable instead. Additionally, a checkbox is available which allows *all* global variables to be treated as unconstrained symbolic variables.

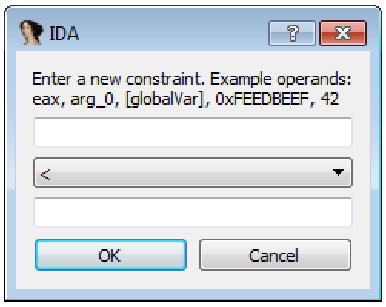
- If the analysed function contains (many) calls to a different function which is either unrelated to any opaque predicates, or of which the analyst knows the output, she can attach a *hook*, i.e. a piece of code which will be executed instead of the original function whenever the original function is called. In cases where such a function is large or would be called often, a hook can significantly simplify and speed up analysis. Ideally, the analyst would be able to specify her own snippet of Python code, but the current version of DROP only allows a choice from several preset hooks. These presets are:
  - Do nothing: don't set a return value.
  - Return a constant word: the analyst can specify the word to return when adding the hook.
  - Return an unconstrained symbolic variable.
  - Return a printable character, i.e. a symbolic 8-bit variable with some constraints that force it to be printable.

### 4.3.3 Concrete execution

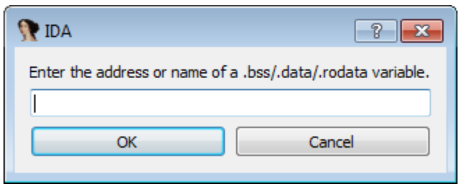
Because the opaque predicate detection process works on a per-block basis, it is useful to limit this process to code-blocks along an execution trace through the function. Dedicated tools exist to obtain such traces from actual executions of a program on a specific architecture (e.g. with Intel's PIN for x86), but even ANGR provides us with similar functionality without this cost of losing architecture-independence.

DROP's `ConcreteRunner` worker generates an execution trace through a single function by letting ANGR 'explore' with all symbolic functionality turned off, starting

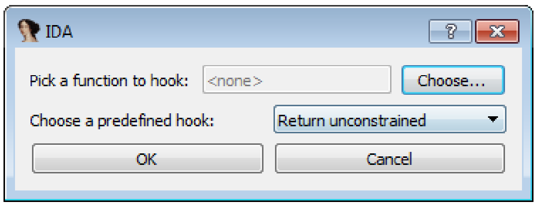
[10]This special constraint-type is referred to as *atoi-on-argv* in this thesis.


IDA
?
Enter a new constraint. Example operands:
eax, arg_0, [globalVar], 0xFEEDBEEF, 42
<
OK
Cancel

(A) Adding a new (in-)equality constraint.



(B) Adding a new symbolic global variable.



(C) Adding a function hook.

FIGURE 4.2: Screenshots of the dialogs for interactive input.

at an initialised function entry state with user provided parameters. When using ANGR in this manner it is effectively an inaccurate machine-simulator because kernel or library functions might not be (fully) implemented, memory might not be simulated properly and most of all: execution is started at the current function without any context of the rest of the program's execution. Nevertheless, the resulting trace does not need to be accurate as it is only meant to be a starting point for opaque predicate detection.

### 4.3.4 Opaque predicate detection

The opaque predicate detection technique that was implemented in the `OpaquePredicateFinder` class of DROP works by performing the following steps for every branching code-block ('branch') in the set of marked blocks (e.g. the concrete trace):

1. Calculate the *dominator set* of the branch, defined as the set of nodes (i.e. basic blocks) in the control-flow graph which could have been visited in a path from the entry-point to the current branch.

2. Perform symbolic execution from the function entry-point to the current branch, avoiding all basic blocks which are not in the dominator set, and taking into account the user-provided constraints, symbolic variables and function hooks. Pick the first path that is found. Note that this shortcut trades soundness for speed.

3. Extract the list of constraints (i.e. predicates) corresponding to the possible branches along the path, invert them if the *false* branch was satisfied in the path. Call these predicates $\psi_1 ... \psi_n$. If any additional regular constraints are provided for this function by the analyst, append them to this list. If any *atoi-on-argv* constraints are provided by the analyst *and* if a call to `atoi` was made on the corresponding provided string-array element: append the provided constraint on the output of that specific `atoi` call to the list of predicates.

Finally, given the list of predicates, use the constraint solver's satisfiability check to determine if the final predicate in the path ($\psi_n$) is an opaque predicate:

- $\psi_n$ is an **invariant** opaque predicate (see Definition 2.3.1) *iff* it is a tautology over its containing variables, which is true *iff* its inverse, $\neg\psi_n$ is not satisfiable. For example, the predicate $x \leq 5 \vee x > 5$ is an invariant opaque predicate because its inverse, $x > 5 \wedge x \leq 5$, is clearly not satisfiable.
- $\psi_n$ is a **contextual** opaque predicate (see Definition 2.3.2) *iff* its truth value is always implied by the conjunction of the previous predicates (i.e. the path condition before reaching the current branch). Therefore we get:

  $$\begin{aligned} & \psi_n \text{ is a contextual opaque predicate} \\ \Leftrightarrow\ & (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{n-1}) \rightarrow \psi_n \\ \Leftrightarrow\ & (\neg\psi_1 \vee \neg\psi_2 \vee \ldots \vee \neg\psi_{n-1}) \vee \psi_n \end{aligned}$$

  Again, this holds for all values of the contained variables *iff* the formula is a tautology, which is true *iff* its inverse is not satisfiable:

  $$\begin{aligned} \Leftrightarrow\ & \neg((\neg\psi_1 \vee \neg\psi_2 \vee \ldots \vee \neg\psi_{n-1}) \vee \psi_n) && \text{not satisfiable} \\ \Leftrightarrow\ & (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{n-1}) \wedge \neg\psi_n && \text{not satisfiable} \end{aligned}$$

  Therefore, $\psi_n$ is a contextual opaque predicate *iff* $(\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{n-1}) \wedge \neg\psi_n$ is not satisfiable. For example, if the path condition contains the conjuncts $\psi_1 : x \equiv 0 \pmod 8$, $\psi_2 : x > 16$, and $\psi_3 : x < 30$, then the predicate $x = 24$ is contextually opaque, because the conjunction of the previous conditions with its inverse (i.e. $x \equiv 0 \pmod 8 \wedge x > 16 \wedge x < 30 \wedge x \neq 24$) is not satisfiable.

This core idea of this method of detecting opaque predicates was first described by Ming et al. [22]. Note that any input-invariant opaque predicates (see Definition 2.3.3) will be detected as contextual opaque predicates, if and only if the correct context (or a stronger form) is provided by the analyst as a special *atoi-on-argv* constraint. For example, if a program was obfuscated with the input-invariant $\text{atoi}(\text{argv}[1]) > 0$, it will still be detected if the analyst realises that this value is always larger than 100 and specifies the constraint $\text{atoi}(\text{argv}[1]) > 100$, because $x > 100$ implies $x > 0$.

### 4.3.5 Visualisation and elimination

After an opaque predicate has been detected, it is added to the opaque predicate table for the current function and it is visualised in the IDA PRO disassembly listing and control-flow graph view by marking the basic block that contains the predicate with a special colour. Additionally, basic blocks that can only be reached by the unsatisfiable branch of a conditional jump guarded by an opaque predicate are marked with a different colour signifying that they consist of dead code.

Following this, the user can click a button to start the `UnreachabilityPropagator` worker which will iteratively mark basic blocks as unreachable if all their predecessors are marked as unreachable (ignoring loop-back edges). After this process, it is immediately more clear to an analyst which branches and basic-blocks belong to *bogus* control-flow (based on the provided context).

Lastly, after a section of the function is marked as unreachable, the user can press a button to run the `UnreachableCodePatcher`, which will try to replace unreachable code by no-op instructions and remove any bogus conditional jumps. Replacing code by no-op instructions is the most practical method of removing code without having to restructure, modify and recompile the disassembled instructions.

After this process, the control-flow graph is simplified and more closely approximates a *sound* control-flow graph of the current function (meaning that after this process, the control-flow-graph contains less nodes and edges that can never be visited). Additionally, if the Hex-Rays Decompiler is installed and the loaded program is of a supported architecture[11], the user can run the decompiler on the patched code, which might result in a decompilation output that is a lot more clean (i.e. containing less bogus code) than before the removal of bogus control-flow.

[11]being ARM, ARM64, x86, x86-64 and PowerPC

# Chapter 5

# Evaluation

The effectiveness of the deobfuscation capabilities of DROP is somewhat difficult to measure properly because of the number of parameters involved. These parameters include: the program or function that was obfuscated, the type of obfuscation used, the compiler (settings) used and the target architecture of the program. Additionally, as established in Section 2.2 and in Chapter 3, the quality of a deobfuscation result can be subjective and can depend on what type of information an analyst is after.

However, when attacking opaque predicates specifically, the primary goal of a deobfuscator is always to detect their presence, which is the very thing these predicates are designed against (hence why they are *opaque*). Therefore, if we have control over the insertion of opaque predicates (or if we can determine how many were inserted), we can use the number of opaque predicates that are properly detected by the deobfuscator as an objective measure of effectiveness. This metric is used for the experiments in sections 5.1 and 5.2, which respectively try to determine how well different variations of opaque predicates are detected by DROP and how well DROP is able to deal with opaque predicates in 'real' code. The experiment in Section 5.1 also tests the unreachability propagation and dead-code removal capabilities of DROP, showing the effect of deobfuscation on a function's control-flow graph and the Hex-Rays decompilation output.

Section 5.3 describes a program that was written and obfuscated with an input-invariant opaque predicate. It shows that DROP is able to detect this predicate if and only if an appropriate invariant is provided as a special additional constraint by the analyst.

Section 5.4 describes an experiment that was performed to test whether DROP is able to perform opaque predicate detection and removal on programs compiled for architectures other than x86.

## 5.1 Effectiveness on a small constructed program

In order to experiment with various types of opaque predicates in a very simple environment, a small C program – `fib_fac.c` – was created. It contains two functions – `fac` and `fib` – which respectively calculate the factorial of $n$, and the $n^{\text{th}}$ Fibonacci number. Figure 5.1 shows the code for these functions.

Eight obfuscated versions of this program were obtained, each obfuscated with a different type of opaque predicate, either manually (for three of them) or automatically (for five of them). Each obfuscated version was compiled to a 32-bit x86 ELF-file for Linux, with compiler optimisations turned off (to prevent the compiler from interfering with any obfuscations). Each executable was then loaded into IDA PRO, where the opaque predicate detection and removal functionalities of DROP were applied to it. Opaque predicate detection was able to be performed in reasonable time on all branching code-blocks because of the small size of both functions. Therefore, the

```
// Calculates the n-th Fibonacci number
int fib(int n) {
  int i, c, d;
  int a = 1;
  int b = 1;

  for (i = 0; i < n; ++i) {
    c = a;
    a = a + b;
    b = c;
  }

  return a;
}
```

```
// Calculates n!
int fac(int n) {
  int res = 1;
  int i;

  if (n < 0) {
    return 0;
  }

  for (; n > 1; --n) {
    res *= n;
  }

  return res;
}
```

FIGURE 5.1: The important functions of `fib_fac.c`

| | Obfuscation type | Extra input required | All predicates detected | Removal successful |
|---|---|---|---|---|
| Manual | Invariant, number-theoretic (1 variable) | no | yes | yes |
| | Invariant, number-theoretic (2 variables) | no | yes | yes |
| | Contextual, number-theoretic | no | yes | yes |
| Tigress | Invariant, linked-lists (`list`) | **yes** | yes | yes |
| | Invariant, arrays (`array`) (CLANG) | no | yes | yes |
| | Invariant, arrays (`array`) (GCC) | no | **no** | n/a |
| | Invariant, number-theoretic (`env`) | no | yes | yes |
| OLLVM | Invariant, number-theoretic (`-bcf`) | no | yes | yes |

TABLE 5.1: Results of applying the IDA plugin to obfuscated versions of `fib_fac.c`, obtained through the manual and automatic insertion of different types of opaque predicates.

generation of a trace through concrete execution was not needed and not part of this experiment.

Table 5.1 shows the results of this experiment. For each obfuscation type, the following results are noted in the table:

- *Extra input needed*: Was the analyst required to provide extra constraints, symbolic variables or function hooks to improve opaque predicate detection? The process of choosing appropriate concrete values in order for the algorithm to analyse all potential predicates can also be interactive but is not included in this metric, as mentioned above.
- *All predicates detected*: Were all the opaque predicates that were present in the executable detected properly by Drop?
- *Removal successful*: Did the unreachability propagation and dead-code removal functionality work properly, i.e. did it remove all dead code (is it *complete*) and did it not remove any possibly reachable code (is it *sound*)?

As can be seen in the table, opaque predicate detection and removal was successful for all samples except for one: Tigress's array-invariant–based opaque predicates, compiled with GCC. For this executable, the analysis got stuck on a large and complex constraint resulting from a C expression similar to the following one:

```
array[3 * (x % 10)]
```

Interestingly, when the same obfuscated program is compiled with CLANG, no such problem occurs. Investigation revealed that in this case, GCC applied an optimisation to the sub-expression `(x % 10)` that results in a series of copy, multiplication and shift instructions being generated instead of – in this case, for x86 – an `IDIV` instruction which is generated by CLANG. In fact, any integer division or modulo operation with a constant number can be replaced by a multiplication by the multiplicative inverse of that number [14]. This transformation is applied by some compilers because integer multiplication is often faster than integer division.

When the result of this optimisation is combined with the multiplication and array-indexing operations already present in the expression, the resulting set of lifted instructions apparently results in a rather difficult-to-solve constraint. This is the only case out of the eight sample files where the total analysis time was not trivially small (i.e. in the order of several seconds). In fact, analysis here was stopped after no solution was found after 30 minutes.

We see that there is only one case in the table where extra user-input was required, namely for Tigress's linked-list–based opaque predicates. This executable is deobfuscated properly when the user specifies the invariant property as an additional constraint. In this case specifically, a linked-list pointer called `_obf_2_main__opaque_list1_1` was created by Tigress, which is initialised in `main` and updated in multiple functions by pointing it to the value of its `next` pointer. The structure is initialised in such a way that the `next` pointer of these links is never `NULL`. Because this initialisation occurs in a different function, Drop has no knowledge of this invariant. The analyst can specify it by marking the pointer as being symbolic (by adding `_obf_2_main__opaque_list1_1` to the list of symbolic variables) and adding a non-zero constraint (by entering `[_obf_2_main__opaque_list1_1] != 0` as a new constraint).

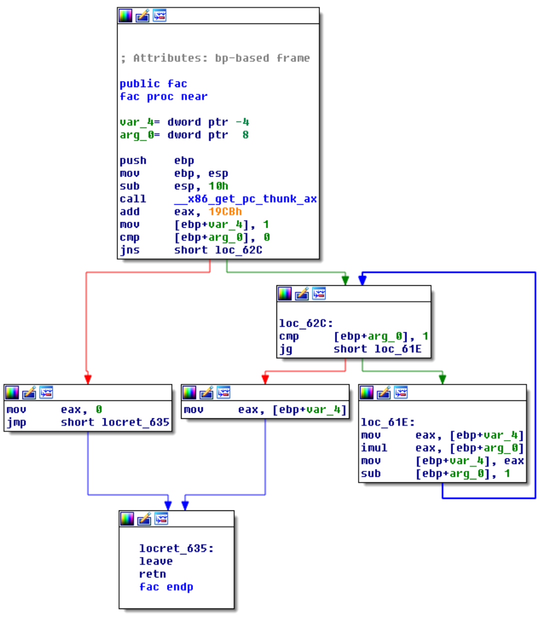
### 5.1.1 Control-flow graphs

Figure 5.2 shows the effect of opaque predicate detection and removal on the control-flow graph of the `fac` function. The obfuscated version shown corresponds to the first row of Table 5.1. In Figure 5.2c it is clearly visible that the three yellow basic-blocks are directly unreachable as the result of invariant opaque predicates. Figure 5.2b shows the control-flow graph of the same function after this unreachable code was removed. The resulting graph is a lot more similar to the control-flow graph of the unobfuscated original (Figure 5.2a) than to the obfuscated version of the program. However, even though the function's control-flow is mostly restored, one can see that many dead instructions within reachable basic-blocks remain. In Figure 5.2b this is most clearly visible in the largest and centre-most code-block in which all but the final two instructions are dead. In fact, apart from the final jump, the only non-dead instruction in this block is the final move instruction (`mov eax, [ebp+var_4]`), which is the only instruction in the original version of this code-block (the centre code-block in Figure 5.2a). The extraneous instructions here were part of the set-up code for the opaque predicate, but because the code-block in which they reside does not consist entirely of dead code, they would be difficult to remove. Removing them would require some form of data dependence analysis.

### 5.1.2 Decompilation output

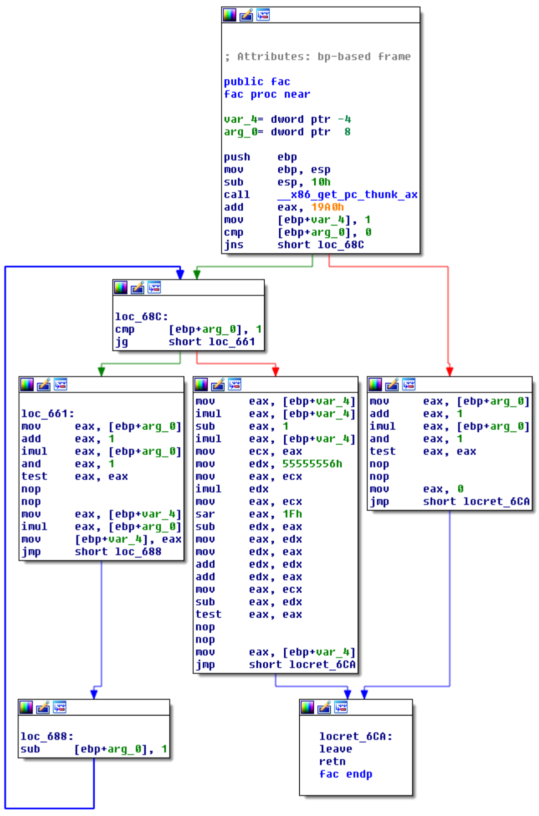
An advantage of Drop's integration within IDA Pro is that if one has an appropriate license and if the obfuscated program is of a supported architecture, one can use the Hex-Rays Decompiler to further deobfuscate the resulting control-flow graph. The decompiler is a deobfuscator in and of itself because firstly, a (pseudo-)C representation is inherently more structured, and therefore more easily readable and understandable than a disassembly listing; and secondly, a deobfuscator performs many transformations to undo compiler tricks and optimisations, especially regarding arithmetic expressions. The Hex-Rays decompiler is also able to detect and ignore dead instructions in certain situations, including those like the one mentioned in the previous sub-section.

An example of the power of the decompiler when used together with Drop is shown in figures 5.3 and 5.4 which respectively show the decompilation output of the OLLVM-obfuscated version (corresponding to the last row of Table 5.1) of the `fac` function before and after the combined process of opaque predicate detection and dead code removal. Apart from some redundant variable assignments, the result of the final decompilation is almost equivalent to the original C source-code. Importantly, it is clear that the removal of bogus control-flow has severely reduced the number of loop-constructs and `goto` statements present in the decompilation output. This difference is especially drastic for programs obfuscated by OLLVM because the dead code OLLVM inserts after each opaque predicate has a control-flow edge that loops back to the basic-block containing the predicate, creating a loop-like construct which makes the decompilation output extra verbose.



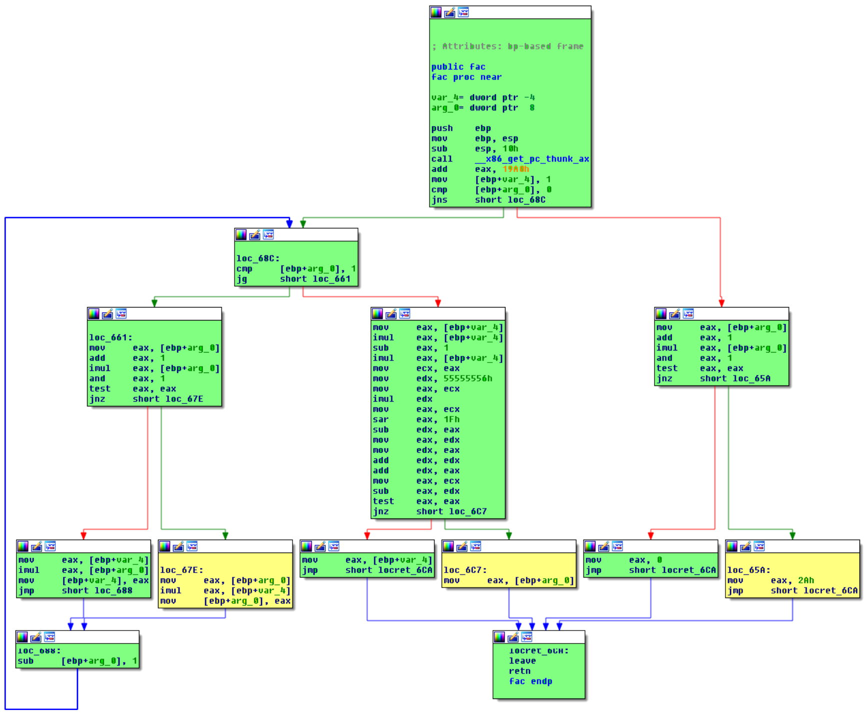
```
; Attributes: bp-based frame

public fac
fac proc near

var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
call    __x86_get_pc_thunk_ax
add     eax, 19CBh
mov     [ebp+var_4], 1
cmp     [ebp+arg_0], 0
jns     short loc_62C

loc_62C:
cmp     [ebp+arg_0], 1
jg      short loc_61E

mov     eax, 0
jmp     short locret_635

mov     eax, [ebp+var_4]

loc_61E:
mov     eax, [ebp+var_4]
imul    eax, [ebp+arg_0]
mov     [ebp+var_4], eax
sub     [ebp+arg_0], 1

locret_635:
leave
retn
fac endp
```

(A) Original CFG



```
; Attributes: bp-based frame

public fac
fac proc near

var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
call    __x86_get_pc_thunk_ax
add     eax, 19A0h
mov     [ebp+var_4], 1
cmp     [ebp+arg_0], 0
jns     short loc_68C

loc_68C:
cmp     [ebp+arg_0], 1
jg      short loc_661

loc_661:
mov     eax, [ebp+arg_0]
add     eax, 1
imul    eax, [ebp+arg_0]
and     eax, 1
test    eax, eax
nop
nop
mov     eax, [ebp+var_4]
imul    eax, [ebp+arg_0]
mov     [ebp+var_4], eax
jmp     short loc_688

mov     eax, [ebp+var_4]
imul    eax, [ebp+var_4]
sub     eax, 1
imul    eax, [ebp+var_4]
mov     ecx, eax
mov     edx, 55555556h
mov     eax, ecx
imul    edx
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
mov     edx, eax
add     edx, edx
add     edx, eax
mov     eax, ecx
sub     eax, edx
test    eax, eax
nop
nop
mov     eax, [ebp+var_4]
jmp     short locret_6CA

mov     eax, [ebp+arg_0]
add     eax, 1
imul    eax, [ebp+arg_0]
and     eax, 1
test    eax, eax
nop
nop
mov     eax, 0
jmp     short locret_6CA

loc_688:
sub     [ebp+arg_0], 1

locret_6CA:
leave
retn
fac endp
```

(B) Deobfuscated CFG



```
; Attributes: bp-based frame

public fac
fac proc near

var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
call    __x86_get_pc_thunk_ax
add     eax, 19A0h
mov     [ebp+var_4], 1
cmp     [ebp+arg_0], 0
jns     short loc_68C

loc_68C:
cmp     [ebp+arg_0], 1
jg      short loc_661

loc_661:
mov     eax, [ebp+arg_0]
add     eax, 1
imul    eax, [ebp+arg_0]
and     eax, 1
test    eax, eax
jnz     short loc_67E

mov     eax, [ebp+var_4]
imul    eax, [ebp+var_4]
sub     eax, 1
imul    eax, [ebp+var_4]
mov     ecx, eax
mov     edx, 55555556h
mov     eax, ecx
imul    edx
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
mov     edx, eax
add     edx, edx
add     edx, eax
mov     eax, ecx
sub     eax, edx
test    eax, eax
jnz     short loc_6C7

mov     eax, [ebp+arg_0]
add     eax, 1
imul    eax, [ebp+arg_0]
and     eax, 1
test    eax, eax
jnz     short loc_65A

mov     eax, [ebp+var_4]
imul    eax, [ebp+arg_0]
mov     [ebp+var_4], eax
jmp     short loc_688

loc_67E:
mov     eax, [ebp+arg_0]
imul    eax, [ebp+var_4]
mov     [ebp+arg_0], eax

mov     eax, [ebp+var_4]
jmp     short locret_6CA

loc_6C7:
mov     eax, [ebp+arg_0]

mov     eax, 0
jmp     short locret_6CA

loc_65A:
mov     eax, 2Ah
jmp     short locret_6CA

loc_688:
sub     [ebp+arg_0], 1

locret_6CA:
leave
retn
fac endp
```

(C) Marked obfuscated CFG. Legend: yellow = unreachable because of an invariant opaque predicate, green = visited in a concrete execution.

Figure 5.2: Control-flow graphs of the original and obfuscated (with manually added number-theoretic opaque predicates, corresponding to the first row of Table 5.1) version of the function `fac` in `fib_fac.c`.

```c
int __cdecl fac(int a1)
{
  bool v1; // bl@2
  int v3;  // [esp+0h] [ebp-28h]@2
  int v4;  // [esp+4h] [ebp-24h]@10
  int *v5; // [esp+8h] [ebp-20h]@2
  int *v6; // [esp+Ch] [ebp-1Ch]@2
  bool v7; // [esp+13h] [ebp-15h]@2
  int *v8; // [esp+14h] [ebp-14h]@2
  int v9;  // [esp+18h] [ebp-10h]@1

  v9 = a1;
  if ( y2 >= 10 && (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) != 0 )
    goto LABEL_12;
  while ( 1 )
  {
    *(&v3 - 4) = v9;
    *(&v3 - 4) = 1;
    v1 = *(&v3 - 4) < 0;
    v8 = &v3;
    v7 = v1;
    v6 = &v3 - 4;
    v5 = &v3 - 4;
    if ( y2 < 10 || (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) == 0 )
      break;
LABEL_12:
    *(&v3 - 4) = v9;
    *(&v3 - 4) = 1;
  }
  if ( v7 )
  {
    if ( y2 >= 10 && (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) != 0 )
      goto LABEL_13;
    while ( 1 )
    {
      *v8 = 0;
      if ( y2 < 10 || (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) == 0 )
        break;
LABEL_13:
      *v8 = 0;
    }
  }
  else
  {
    while ( *v6 > 1 )
      *v5 *= (*v6)--;
    *v8 = *v5;
  }
  do
    v4 = *v8;
  while ( y2 >= 10 && (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) != 0 );
  return v4;
}
```

FIGURE 5.3: Decompilation of the OLLVM-obfuscated version of `fac` in `fib_fac.c`, before deobfuscation

```
int __cdecl fac(int a1)
{
  bool v1; // bl@1
  int v3;  // [esp+0h] [ebp-28h]@1
  int *v4; // [esp+8h] [ebp-20h]@1
  int *v5; // [esp+Ch] [ebp-1Ch]@1
  bool v6; // [esp+13h] [ebp-15h]@1
  int *v7; // [esp+14h] [ebp-14h]@1
  int v8;  // [esp+18h] [ebp-10h]@1

  v8 = a1;
  *(&v3 - 4) = a1;
  *(&v3 - 4) = 1;
  v1 = *(&v3 - 4) < 0;
  v7 = &v3 - 4;
  v6 = v1;
  v5 = &v3 - 4;
  v4 = &v3 - 4;
  if ( v1 )
    return 0;
  while ( *v5 > 1 )
    *v4 *= (*v5)--;
  *v7 = *v4;
  return *v7;
}
```

FIGURE 5.4: Decompilation of the OLLVM-obfuscated version of `fac` in `fib_fac.c`, after automatically detecting opaque predicates and removing the resulting dead code

| | before obfuscation | | after Tigress | | after OLLVM | |
|---|---|---|---|---|---|---|
| Function name | #CB | #instr | #CB | #instr | #CB | #instr |
| `calculate_columns` | 22 | 183 | 40 | 648 | 58 | 356 |
| `init_column_info` | 18 | 142 | 36 | 719 | 53 | 337 |
| `signal_setup` | 23 | 118 | 39 | 594 | 59 | 366 |
| `make_link_name` | 10 | 67 | 22 | 475 | 36 | 262 |
| `long_time_expected_width` | 9 | 59 | 31 | 781 | 20 | 119 |
| `cmp_ctime` | 6 | 59 | 16 | 490 | 9 | 86 |
| `queue_directory` | 7 | 47 | 17 | 378 | - | - |
| `xstrcoll_df_extension` | 8 | 40 | 20 | 478 | 14 | 74 |
| `set_exit_status` | 5 | 18 | 18 | 1339 | 11 | 49 |

TABLE 5.2: Functions from GNU `ls`, before and after obfuscation with Tigress and OLLVM (see sections 5.2.1 and 5.2.2). The original version and the Tigress output were compiled with GCC for x86, with optimisations turned off. OLLVM was also instructed to compile for x86 with optimisations turned off. Legend: #CB: number of code-blocks in function; #instr: number of instructions in function.

## 5.2 Effectiveness against 'real' code

While the Fibonacci and factorial functions of `fib_fac.c` were useful as a clean environment in which different variants of invariant and contextual opaque predicates could be tested, they are not very representative of a typical use-case for obfuscation. It is important to also test how well DROP is able to handle larger functions with various kinds of control-flow constructs, local and global variables, local function calls, library function calls, etcetera.

A program containing many such constructs is `ls` from the GNU `coreutils`[1]. The `coreutils` have been used for the evaluation of deobfuscators and symbolic execution engines in the past[11][6]. For this experiment, the `ls` program was arbitrarily picked from the `coreutils`. Tigress and OLLVM are used in sections 5.2.1 and 5.2.2 respectively to obfuscate nine arbitrarily picked medium-sized functions from GNU `ls`. These functions are listed in Table 5.2.

### 5.2.1 Obfuscated by Tigress

In Section 5.1, it is shown that Tigress's number-theoretic environment-based obfuscation technique (i.e. the `env` option for `--InitOpaqueStructs`) was detected by DROP when applied to the constructed Fibonacci and factorial functions. In this experiment, the same obfuscation was applied to the nine chosen functions from GNU `ls`. Tigress was instructed to insert five opaque predicates per function (using `--InitOpaqueCount`). However, this seems to be treated more as a guideline by Tigress: in some cases more than five opaque predicates are inserted, and some opaque predicates are inserted in the dead-code portion of another opaque predicate, making them unreachable. This is accounted for in the results of this experiments, as explained below. For all Tigress-based experiments, Tigress version 2.1 is used and provided with a random seed value of 42 for consistency and reproducibility.

Table 5.2 shows the size of the functions before and after this obfuscation in terms of the number of code-blocks detected by IDA and the number of instructions per

[1]https://www.gnu.org/software/coreutils/coreutils.html

| | | | trivial trace | | all branches | |
|---|---|---|---|---|---|---|
| Function name | #present | #reachable | #found | #fp | #found | #fp |
| `calculate_columns` | 5 | 5 | 0* | 0 | 0* | 0 |
| `init_column_info` | 6 | 5 | 0 | 0 | **5** | 0 |
| `signal_setup` | 6 | 5 | 1 | 0 | 5 | 2† |
| `make_link_name` | 5 | 4 | 2 | 0 | **4** | 0 |
| `long_time_expected_width` | 5 | 4 | 4 | 1‡ | **5** | 1‡ |
| `cmp_ctime` | 5 | 5 | 0§ | 0 | **5** | 0 |
| `queue_directory` | 5 | 5 | 1 | 0 | **5** | 0 |
| `xstrcoll_df_extension` | 5 | 5 | 3¶ | 0 | **5¶** | 0 |
| `set_exit_status` | 9 | 4 | **4** | 0 | **4** | 0 |

Table 5.3: Number of opaque predicates present, reachable and detected in Tigress-obfuscated functions from GNU `ls`. See Section 5.2.1 for an explanation of the annotations. Legend: #present: number of opaque predicates (OPs) present in the function; #reachable: number of OPs not hidden by other OPs; #found: total number of OPs detected of which #fp are false positives, along a *trivial concrete trace* or *all branching code-blocks* in the function through manual selection. Bold numbers indicate that all reachable OPs were detected.

function. It can be seen that in most cases, the number of code-blocks after obfuscation with Tigress is roughly twice the original number. The number of instructions after obfuscation however is much more random, and anywhere between 3.5 and 13 times the original number. This is because the size of the opaque predicates inserted by Tigress is random but even the smaller ones take up a large amount of instructions.

The obfuscated version of `ls` was loaded into IDA Pro and each of the nine obfuscated functions was 'attacked' with Drop. The following steps were performed:

1. A concrete trace was generated with the value 0 provided for all function parameters for a fair comparison between functions. In some cases this causes the concrete execution to skip most of the function body (where 0 is an invalid or trivial input) but it shows what can be detected without the need for the analyst to have an understanding of the functions in question.

2. Relevant context was provided for this program and the type of opaque predicates that are inserted: the Tigress-generated global variable `_obf_1alwaysZero` was constrained to zero, while all other global variables were set to be symbolic. In a few cases, function hooks were added to aid symbolic exploration by returning an unconstrained value.

3. Opaque predicate detection was performed on the highlighted trace.

4. Opaque predicate detection was performed on all predicates in the function by marking predicate-containing code-blocks as 'trace', and re-running the opaque predicate detection.

Table 5.3 shows how many opaque predicates are contained in each function and how many of those are reachable, together with the results of step 3 and 4 above, where the total number of detected opaque predicates is shown together with the number of false positives, i.e. detected opaque predicates which are either not opaque, or were not inserted by Tigress. Numbers highlighted in a bold font indicate the cases where all opaque predicates were discovered. The following annotations are referred to in Table 5.3:

- * For an unknown reason, none of the opaque predicates in this function are detected. Further investigation is needed to determine why this case fails.
- † These two false positives are caused by the algorithm's greedy behaviour, i.e. only looking at one path from the start of the function to the destination branch. Additionally, because of this, two actual opaque predicates are not detected by the algorithm because they are only reachable through the predicates falsely marked as opaque.
- ‡ This false positive is an interesting case: a stack canary check inserted by GCC, which is marked as an invariant opaque predicate. Under normal circumstances with no illegal manipulations of the stack, this is in fact an accurate conclusion. However, since it was not inserted by Tigress, it is regarded in this experiment as a false positive, even though the algorithm arguably did not make a mistake here.
- § For this function, concrete execution fails entirely. The cause of this is most likely related to the fact that one of the arguments of this function is a function pointer, which is called in the function body. Nevertheless, all opaque predicates are detected after hooking the `timespec_cmp` function (to always return an unconstrained value) before running opaque predicate detection on all branching code-blocks.
- ¶ Here, the function `is_directory` needed to be hooked (to always return an unconstrained value) in order for all opaque predicates to be detected.

The results in Table 5.3 show that for seven out of the nine functions tested, all opaque predicates were discovered. However, for only one of these functions (`set_exit_status`), all opaque predicates were found (and therefore visited) along the generated trace. For a couple of functions, none of the opaque predicates were visited along the trace. While we should not expect a single trace to visit all opaque predicates in a given function, we can at least improve the number of opaque predicates found by picking better values to pass as function arguments for concrete execution. Take the function `init_column_info` for example. It has a single integer parameter which is treated as a boolean value. When a zero is passed, the concrete trace completely skips most of the function body, while a value of one causes it to pass by several of the opaque predicates present in the function. We can conclude that the ability to generate a concrete trace might not always be very helpful, but it can at least provide the analyst with a starting point, especially when provided with cleverly chosen function arguments.

| | | trivial trace | | all branches | |
|---|---|---|---|---|---|
| Function name | #present | #found | #fp | #found | #fp |
| `calculate_columns` | 7 | **7** | 0 | **7** | 0 |
| `init_column_info` | 12 | 2 | 0 | **13** | 1* |
| `signal_setup` | 14 | 2 | 0 | 6† | 0 |
| `make_link_name` | 10 | 4‡ | 0 | **10**‡ | 0 |
| `long_time_expected_width` | 4 | **4**§ | 0 | **4**§ | 0 |
| `cmp_ctime` | 2 | 0 | 0 | **2** | 0 |
| `queue_directory` | 0 | **0** | 0 | **0** | 0 |
| `xstrcoll_df_extension` | 2 | **2**¶ | 0 | **2**¶ | 0 |
| `set_exit_status` | 2 | 0 | 0 | **2** | 0 |

Table 5.4: Number of opaque predicates present and detected in OLLVM-obfuscated functions from GNU `ls`. See Section 5.2.2 for an explanation of the annotations. Legend: #present: number of opaque predicates (OPs) present in the function; #found: total number of OPs detected of which #fp are false positives, along a *trivial concrete trace* or *all branching code-blocks* in the function through manual selection. Bold numbers indicate that all OPs were detected.

### 5.2.2 Obfuscated by OLLVM

The experiment from Section 5.2.1 was repeated, but OLLVM was used instead of Tigress for the insertion of opaque predicates. A difference between Tigress and OLLVM is that – at least for the functions in this experiment – OLLVM never inserts opaque predicates in locations that are unreachable because of another opaque predicate. Therefore, in this experiment, all inserted opaque predicates should be reachable through symbolic execution. Table 5.4 shows the result of this experiment. The following annotations are referred to in Table 5.4:

* In this case, the false positive result is caused by the greedy behaviour of the opaque predicate detection algorithm: in order to minimize analysis time, the first path that is found from the start of the function to the branch in question is treated as the only path. In cases such as this where an essential containing variable is modified along a second path, the predicate can be marked as opaque when it is not.

† Here, a loop-limiting optimisation in the detection algorithm causes it to miss eight opaque predicates. These branches can only be reached when a loop of many iterations is properly executed. Without this time-saving optimisation turned on, all opaque predicates would be found.

‡ § ¶ In these cases, the respective functions `dir_len`, `align_nstrftime` and `is_directory` needed to be hooked (to always return an unconstrained value) in order for all opaque predicates to be detected.

The main result of this experiment is that for eight out of nine functions, all opaque predicates were detected after appropriate context (i.e. correctly hooked functions and a proper selection of *trace* blocks) was provided to the algorithm.

## 5.3 Input-invariant opaque predicates

DROP's capability of detecting input-invariant opaque predicates was put to the test on a Tigress-obfuscated implementation of the Caesar cipher, `caesar.c`. Appendix A lists the contents of this program.

The program's three command-line arguments specify the following respectively: its mode of operation – i.e. encryption or decryption; the number of places to shift (i.e. the key); and the message to encrypt or decrypt. The second argument should always be a positive integer. If it is not, the program exits with an error code. In other words, the condition atoi(argv[2]) >= 1 should always hold for valid invocations of the program. Therefore, that condition was provided as an input invariant to Tigress (by providing the option `--Inputs='+2:int:1?2147483647'`, i.e. the second argument from the left is an integer between 1 and `INT_MAX` inclusive). Tigress was instructed to insert one input-invariant opaque predicate per function.

The resulting obfuscated version of `caesar.c` (see Appendix B) has two interesting changes, compared to the original:

- A global variable called `_obf_2_main__argv` is added, which is assigned the value of `argv` in the `main` function. This allows input-invariant opaque predicates to be added in any function.
- In the `caesar` function, an input-invariant opaque predicate is inserted inside the while-loop. Specifically, line 13 of the original program, i.e.:

```
13  str[i] = ((str[i] - 'a' + num_shift) % 26) + 'a';
```

  is replaced by the following code in the obfuscated program:

```
105 atoi_result7 = atoi(*(_obf_2_main__argv + 2));
106 if ((atoi_result7 - 1 < 0) + (atoi_result7 - 2147483647 >
       0)) {
107   *(str + i) = (char)2;
108 } else {
109   *(str + i) = (char )((((int )*(str + i) - 97) +
         num_shift) % 26 + 97);
110 }
```

Apart from some extra casts and different notations being used, we see that `atoi` is indeed called on the second command-line argument (line 105), and that this value is used in the inserted predicate, which makes sure it is within the valid range (i.e. larger than 0). If it is not, the bogus code on line 107 is executed.

### 5.3.1 Results

The obfuscated program was compiled with GCC for x86, with optimisations turned off. After loading the resulting binary executable into IDA PRO and specifying no additional context to DROP (apart from checking the box to treat all global variables as unconstrained symbolic variables), it did not detect the input-invariant opaque predicate. However, when the input invariant was specified as a constraint (i.e.: atoi(argv[2]) >= 1), the opaque predicate *was* detected. Specifying a stronger invariant (such as atoi(argv[2]) >= 5) will also cause the opaque predicate to be detected.

## 5.4 Architecture-independence

In principle, Drop is architecture-independent in the sense that it supports all architectures supported by both IDA Pro and angr. In practice, all experiments above have been performed on x86 binaries, which was also the main focus during the development of Drop. Each processor architecture or instruction-set has its quirks and peculiarities[2], especially regarding how it is treated by IDA Pro's loader and angr's loader and instruction lifter. In order for us to be able to claim true architecture-independence, proper testing is required. Because this is not the main focus of this thesis, it is left as future work.

Nevertheless, a small experiment was performed where the first obfuscated test from Table 5.1 has been (cross-)compiled using GCC for x86, x86_64, ARM64 and MIPS64. All opaque predicates in the `fib` and `fac` functions were detected in all of these executable files. Patching unreachable code worked well for all architectures, although for ARM and MIPS, the resulting CFG as rendered by IDA after dead-code removal has some extraneous blocks filled with NOP instructions, making it slightly harder to understand compared to the other architectures. For the architectures for which the Hex-Rays decompiler is available (x86-* and ARM*), the decompilation output represented the original program almost precisely (similar to the result in figures 5.3 and 5.4).

[2]Such as ARM's different modes and conditional instructions

# Chapter 6

# Conclusions and reflection

## 6.1 Conclusions

Despite the large volume of literature on opaque predicates – from both a protection and from an attack standpoint, i.e. obfuscation (Chapter 2) and deobfuscation (Chapter 3) – there are not many tools available that allow for automated deobfuscation of control-flow obfuscations. The tools that *are* available [22][6][18] are lacking in accessibility, versatility or both. A tool such as LOOP[22] for example is difficult to set up and has limited applicability because of its lack of manual controllability and its dependency on a specific hardware architecture.

Experiments in Chapter 5 have shown that a more versatile approach to opaque predicate detection is possible and can be effective. By making use of the powerful visualisation and analysis capabilities present in IDA Pro combined with the flexibility of the angr platform, Drop – the tool produced for this thesis – is able to (semi-)automatically detect, visualise and remove the bogus control-flow resulting from various types of invariant and contextual opaque predicates. We saw that dead-code removal by patching code in IDA Pro's assembly listing is especially useful when combined with the power of the Hex-Rays decompiler. The combination of applying Drop's dead-code removal and Hex-Rays decompilation transformation sometimes results in almost completely deobfuscated C-like source-code.

Additionally, we can conclude that the fact that an analyst can interactively specify additional context (i.e. constraints, symbolic variables and hooks) to the opaque predicate detection process allows Drop to be used in a much more versatile manner compared to a fully automated approach. Despite this, experiments also show that there are still many cases where the effectiveness or efficiency of Drop is far from optimal, even when appropriate additional context is provided by the analyst (see Section 5.1).

From experiments in Section 5.3 we draw the conclusion that it is in fact possible to detect and revert obfuscations that forego the functionality property (i.e. obfuscations that require the obfuscated program to produce identical output to the original program only for *valid* inputs) such as input-invariant opaque predicates which are only opaque given the fact that a certain invariant over the program arguments holds. Specifically, if the analyst is able to determine this invariant or a stronger condition, she can specify it to Drop as an additional constraint, and Drop's adaptation of the algorithm of Ming et al. [22] will able to detect the input-invariant opaque predicates as being contextual opaque predicates.

## 6.2 Future work

Despite the positive results from the experiments in Section 5, there are aspects of DROP that could be improved:

- The interface for providing additional constraints is currently limited to (in-)equalities on specific types of operands with the special exception of (in-)equality constraints on the `atoi`-converted value of a string-array element and a constant value. While this is sufficient for the scenarios discussed in this thesis, a more generalised interface allowing other types of user-provided constraints could be useful for other scenarios.
- Similarly, the function hooking functionality is currently limited to a few predefined functions. A more flexible approach would be to allow the analyst to provide her own python function for a given hook.
- Dead-code patching is currently rather rudimentary. It could be improved by performing instruction-level analysis, e.g. using data-dependence information.
- Proper detection and deobfuscation of the *div-by-mul* optimisation (as mentioned in section 5.1) could be implemented.
- DROP could possibly be expanded to detect (generalised) dynamic opaque predicates, or even different types of control-flow obfuscation using similar symbolic-execution–based techniques.

Additionally, there would be many benefits to having better integration of the plugin within IDA. Specifically, the following improvements could be made in this area:

- Making use of IDA's debugging/tracing infrastructure and interface. IDA allows plugins to register a debugger, providing API support for stepping through code, accessing memory, and other operations. Integrating DROP with this API and registering the current 'concrete execution' functionality as a debugger would not only allow other debugger-plugins to be used for trace generation, but also allow for things like exploration and modification of memory space using the pre-existing IDA user interface.
- Currently, ANGR's own control-flow graph generation algorithms are used (`CFGAccurate` and `CFGFast` as a fall-back), and code-block/basic-block addresses are carefully translated between IDA's representation and ANGR's representation when needed, which inevitably causes problems in various edge-cases. A much better approach would be to natively extract IDA's control-flow graph and use it within ANGR.
- The IDA PRO API allows plugins to register context-menus and keyboard short-cuts in various locations within the interface. DROP could greatly benefit from this.
- Right now, settings and internal state of DROP cannot be saved or loaded, and analysis needs to be re-run when a project is closed and re-opened in IDA PRO. An obvious solution to this problem would be to save and load this internal state together with IDA project files.

Apart from the above-mentioned points of improvement, these aspects inherent to the design of either Drop or its dependencies are non-optimal:

- In order for an analyst to use Drop to deobfuscate a function, that function needs to be fully and properly recognised by both IDA Pro and angr. Specifically, functions which are obfuscated to include garbage data behind opaque predicates might not be properly recognised as such and might cause the reconstructed control-flow graph of IDA, angr or both to be incomplete. This, in turn, might cause opaque predicate detection to fail. In order to solve this problem, an entirely different approach would have to be taken to opaque predicate discovery, instead of the current function-based approach. Luckily, such garbage data–inserting obfuscations are not stealthy at all and therefore unlikely to be used in situations where stealth is required.

- During opaque predicate detection, all location-information (e.g. the addresses of predicates) is stored and processed on the code-block level. The same holds for unreachability information. However, for more accurate results it might be better to do so on the instruction level. Because control-flow graph reconstruction can be inaccurate, this might prevent certain edge-cases where Drop's analyses fail. This would however require more resources than the current approach, so it is unknown whether doing this would be worthwhile.

# Appendix A

# Test program: caesar.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void caesar(char *str, int num_shift)
{
  int len = strlen(str);
  int i;

  for (i = 0; i < len; ++i) {
    // Shift if alphabetic
    if (str[i] >= 'a' && str[i] <= 'z') {
      str[i] = ((str[i] - 'a' + num_shift) % 26) + 'a';
    }
  }
}

int main(int argc, char *argv[])
{
  if (argc < 4) {
    return 1;
  }

  // whether to encrypt or decrypt
  int operation = atoi(argv[1]);

  // the number to shift with
  int num_shift = atoi(argv[2]);

  // the string to encrypt/decrypt
  char *str = argv[3];

  // Make sure num_shift is positive
  if (num_shift <= 0) {
    return 1;
  }

  if (operation == 0) {
    // encrypt
    caesar(str, num_shift);
  } else {
    // decrypt
    caesar(str, 26 - num_shift % 26);
  }

  printf("Result: %s\n", str);

  return 0;
}
```

# Appendix B

# Obfuscated version of `caesar.c`

```c
/* Generated by CIL v. 1.7.0 */
/* print_CIL_Input is false */

struct _IO_FILE;
struct timeval;
int _obf_1entropy  =   0;
extern void signal(int sig , void *func ) ;
extern float strtof(char const   *str , char const   *endptr ) ;
typedef unsigned long size_t;
int _obf_2_main__argc  =   0;
typedef struct _IO_FILE FILE;
extern  __attribute__((__nothrow__)) int (
    __attribute__((__nonnull__(1), __leaf__)) atoi)(char const
      *__nptr )  __attribute__((__pure__)) ;
extern int fclose(void *stream ) ;
extern double strtod(char const   *str , char const   *endptr )
    ;
extern void *fopen(char const   *filename , char const   *mode
    ) ;
extern void abort() ;
extern void exit(int status ) ;
extern int raise(int sig ) ;
extern int fprintf(struct _IO_FILE *stream , char const
    *format  , ...) ;
extern int strcmp(char const   *a , char const   *b ) ;
char **_obf_2_main__argv  =   (char **)0;
extern unsigned long strtoul(char const   *str , char const
    *endptr , int base ) ;
extern int rand() ;
void obfmegaInit(void) ;
extern int strncmp(char const   *s1 , char const   *s2 ,
    unsigned long maxlen ) ;
int _obf_1alwaysZero  =   0;
extern int gettimeofday(struct timeval *tv , void *tz  , ...) ;
void caesar(char *str , int num_shift ) ;
extern int printf(char const   * __restrict  __format  , ...) ;
int main(int argc , char **argv ) ;
extern  __attribute__((__nothrow__)) size_t (
    __attribute__((__nonnull__(1), __leaf__)) strlen)(char
    const   *__s )  __attribute__((__pure__)) ;
extern long strtol(char const   *str , char const   *endptr ,
    int base ) ;
extern unsigned long strnlen(char const   *s , unsigned long
    maxlen ) ;
```

```
extern void *memcpy(void *s1 , void const   *s2 , unsigned long
    size ) ;
struct timeval {
   long tv_sec ;
   long tv_usec ;
};
extern void *malloc(unsigned long size ) ;
extern int scanf(char const   *format  , ...) ;
int main(int argc , char **argv )
{
  int operation ;
  int tmp ;
  int num_shift ;
  int tmp___0 ;
  char *str ;
  int _obf_2_main__BARRIER_0 ;
  int atoi_result9 ;
  int atoi_result10 ;

  {
  obfmegaInit();
  _obf_2_main__argc = argc;
  _obf_2_main__argv = argv;
  _obf_2_main__BARRIER_0 = 1;

  if (argc < 4) {
    return (1);
  }
  tmp = atoi((char const   *)*(argv + 1));
  operation = tmp;
  tmp___0 = atoi((char const   *)*(argv + 2));
  num_shift = tmp___0;
  str = *(argv + 3);
  if (num_shift <= 0) {
    return (1);
  }
  {
  atoi_result9 = atoi(*(_obf_2_main__argv + 2));
  if (((atoi_result9 - 1 < 0) + (atoi_result9 - 2147483647 >
      0)) + 1) {
    if (operation == 0) {
      caesar(str, num_shift);
    } else {
      caesar(str, 26 - num_shift % 26);
    }
  } else {
    {
    while ((operation == 0) >= atoi_result9) {
      caesar(str - -1, num_shift - num_shift);
    }
    }
  }
  }
  printf((char const   */* __restrict  */)"Result: %s\n", str);
  return (0);
}
}
```

```
void caesar(char *str , int num_shift )
{
  int len ;
  size_t tmp ;
  int i ;
  int atoi_result6 ;
  int atoi_result7 ;

  {
  tmp = strlen((char const   *)str);
  len = (int )tmp;
  i = 0;
  while (i < len) {
    if ((int )*(str + i) >= 97) {
      if ((int )*(str + i) <= 122) {
        {
        atoi_result7 = atoi(*(_obf_2_main__argv + 2));
        if ((atoi_result7 - 1 < 0) + (atoi_result7 - 2147483647
            > 0)) {
          *(str + i) = (char)2;
        } else {
          *(str + i) = (char )((((int )*(str + i) - 97) +
              num_shift) % 26 + 97);
        }
        }
      }
    }
    i ++;
  }
  return;
}
}
void obfmegaInit(void)
{


  {

}
}
```

# Bibliography

[1] Joël Alwen, Manuel Barbosa, Pooya Farshim, Rosario Gennaro, S. Dov Gordon, Stefano Tessaro, and David A. Wilson. "On the Relationship between Functional Encryption, Obfuscation, and Fully Homomorphic Encryption". In: *IMA International Conference on Cryptography and Coding*. Springer, 2013, pp. 65–84.

[2] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. "A Model for Self-Modifying Code". In: *International Workshop on Information Hiding*. Springer, 2006, pp. 232–248.

[3] Genevieve Arboit. "A Method for Watermarking Java Programs via Opaque Predicates". In: *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*. 2002, pp. 102–110.

[4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. "Code Obfuscation Against Symbolic Execution Attacks". In: *Proceedings of the 32Nd Annual Conference on Computer Security Applications*. ACSAC '16. ACM, 2016, pp. 189–200.

[5] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (Im)Possibility of Obfuscating Programs". In: *Advances in Cryptology — CRYPTO 2001*. Vol. 2139. Springer Berlin Heidelberg, 2001, pp. 1–18.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: *International Conference on Computer Aided Verification*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 463–469.

[7] Christian S. Collberg and Clark Thomborson. "Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection". In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 735–746.

[8] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1 edition. Addison-Wesley Professional, 2009-08-03.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations*. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[10] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. "Opaque Predicates Detection by Abstract Interpretation". In: *International Conference on Algebraic Methodology and Software Technology*. Vol. 4019. Lecture Notes in Computer Science. Springer, 2006, pp. 81–95.

[11] Robin David, Sébastien Bardin, and Jean-Yves Marion. "Targeting Infeasibility Questions on Obfuscated Codes". In: *arXiv preprint arXiv:1612.05675* (2016).

[12] Stephen Drape et al. "Intellectual Property Protection Using Obfuscation". In: *Proceedings of SAS 2009* 4779 (2009), pp. 133–144.

[13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. “Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits”. In: *SIAM Journal on Computing* 45.3 (2016), pp. 882–929.

[14] Torbjörn Granlund and Peter L. Montgomery. “Division by Invariant Integers Using Multiplication”. In: *ACM SIGPLAN Notices*. Vol. 29. ACM, 1994, pp. 61–72.

[15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. “Obfuscator-LLVM – Software Protection for the Masses”. In: IEEE, 2015-05, pp. 3–9.

[16] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. “Exploiting Self-Modification Mechanism for Program Protection”. In: *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*. IEEE, 2003, pp. 170–179.

[17] Johannes Kinder. “Towards Static Analysis of Virtualization-Obfuscated Binaries”. In: *Reverse Engineering (WCRE), 2012 19th Working Conference On*. IEEE, 2012, pp. 61–70.

[18] Johannes Kinder, Florian Zuleger, and Helmut Veith. “An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 214–228.

[19] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[20] Kangjie Lu, Siyang Xiong, and Debin Gao. “RopSteg: Program Steganography with Return Oriented Programming”. In: ACM Press, 2014, pp. 265–272.

[21] Matias Madou. “Application Security through Program Obfuscation”. Dissertation. Ghent University. Faculty of Engineering, 2006.

[22] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. “LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. ACM Press, 2015, pp. 757–768.

[23] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. “Experience with Software Watermarking”. In: *Computer Security Applications, 2000. ACSAC’00. 16th Annual Conference*. IEEE, 2000, pp. 308–316.

[24] Rolf Rolles. “Unpacking Virtualization Obfuscators”. In: *3rd USENIX Workshop on Offensive Technologies.(WOOT)*. 2009.

[25] Jonathan Salwan. *Playing with the Tigress Binary Protection*. URL: https://github.com/JonathanSalwan/Tigress_protection (visited on 2017-02-01).

[26] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: IEEE, 2016-05, pp. 138–157.

[27] Henrik Theiling. “Extracting Safe and Precise Control Flow from Binaries”. In: *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference On*. IEEE, 2000, pp. 23–30.

[28] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. “Deobfuscation: Reverse Engineering Obfuscated Code”. In: *Reverse Engineering, 12th Working Conference On.* IEEE, 2005.

[29] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. “Protection of Software-Based Survivability Mechanisms”. In: *International Conference on Dependable Systems and Networks, 2001. DSN 2001.* IEEE, 2001, pp. 193–202.

[30] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. “Linear Obfuscation to Combat Symbolic Execution”. In: *European Symposium on Research in Computer Security.* Vol. 6879. Lecture Notes in Computer Science. Springer, 2011, pp. 210–226.

[31] Dongpeng Xu, Jiang Ming, and Dinghao Wu. “Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method”. In: *International Conference on Information Security.* Vol. 9866. Lecture Notes in Computer Science. Springer, 2016, pp. 323–342.

[32] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. “A Generic Approach to Automatic Deobfuscation of Executable Code”. In: 2015 IEEE Symposium on Security and Privacy. IEEE, 2015-05, pp. 674–691.