NNT : 2017SACLV031

Thèse de doctorat de l'Université Paris-Saclay préparée à Université de Versailles Saint-Quentin-en-Yvelines

Ecole doctorale n°580
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat: Informatique

Mme Ninon EYROLLES

Obfuscation par Expressions Mixtes Arithmético-Booléennes : Reconstruction, Analyse et Outils de Simplification

Thèse présentée et soutenue à Versailles, le 30 Juin 2017.

Composition du Jury :

Mme.	Sandrine BLAZY	Professeur des universités Université de Rennes 1	Présidente du Jury
Mme.	Caroline FONTAINE	Chargée de recherche, CNRS	Rapporteure
M.	Pascal JUNOD	Professeur, Haute École spécialisée de Suisse occidentale	Rapporteur
M.	Louis GOUBIN	Professeur des universités, UVSQ	Directeur de thèse
Mme.	Marion VIDEAU	Responsable scientifique, Quarkslab	Co-encadrante
M.	Emmanuel FLEURY	Maître de conférences Université de Bordeaux	Examinateur
M.	Johannes KINDER	Senior Lecturer Royal Holloway University of London	Examinateur
M.	Renaud SIRDEY	Directeur de recherche Commissariat à l'Énergie Atomique	Examinateur
M.	Thomas SIRVENT	Ingénieur Direction Générale de l'Armement	Invité

So, the constellations. . . I don't believe there's a whale out there, but I believe the stars exist and that people put the whale up there. Like we're good at drawing lines through the space between stars. Like we're pattern finders, and we'll find patterns, and we really put our hearts and minds into it even if we don't mean to. So I believe in a universe that doesn't care and people who do.

Angus, *Night in the Woods*

Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse Louis Goubin, pour avoir accepté de diriger cette thèse et pour sa disponibilité durant ces trois années et demie.

Je remercie chaleureusement Marion Videau, pour sa bienveillance, la qualité de son encadrement scientifique et son soutien permanent.

Frédéric Raynal m'a donné l'opportunité de cette thèse, ainsi que d'excellentes conditions pour mener à bien mes travaux, et je l'en remercie.

Je remercie aussi Emmanuel Fleury pour m'avoir incitée à remettre en question mes intuitions et mes raisonnements.

J'adresse mes remerciements à Caroline Fontaine et Pascal Junod pour avoir accepté d'être rapporteurs de mon manuscrit, pour leur relecture attentive et toutes leurs remarques.

Je tiens à remercier tous les membres du jury de thèse pour leur présence et pour leurs nombreuses questions : Sandrine Blazy, Johanness Kinder, Renaud Sirdey, et plus particulièrement Thomas Sirvent pour ses nombreux commentaires sur le manuscrit. Je remercie à travers lui la DGA, pour avoir co-financé le projet QBOBF, qui a concerné les travaux sur l'obfuscation à Quarkslab entre 2014 et 2016.

Un très grand merci à Cyril pour m'avoir aidée de nombreuses fois dans la compréhension du fonctionnement de l'UVSQ, et à toute l'équipe crypto du LMV pour leur accueil.

Je suis vraiment reconnaissante à Lucas et Philippe pour avoir lu les premiers jets de mon manuscrit, ainsi qu'à Jérôme et Kévin pour avoir lu jusqu'au bout malgré les formules.

Je remercie aussi l'ensemble des mes collègues de Quarkslab, pour avoir répondu à mes questions métaphysiques (*mais qu'est-ce que le reverse ?*), et pour m'avoir supportée ces derniers mois.

Merci également à Camille de m'avoir poussée à creuser le sujet des MBA, et pour toutes les questions et discussions qui en ont découlé.

Je remercie aussi les « vieux de la vieille », ceux qui étaient déjà passés par là et qui m'ont rassurée quand la fin paraissait parfois trop loin et trop difficile à atteindre : Aymeric, Julien, Marc, Anne, et tous ceux qui m'ont prodigué leur soutien indirectement.

Enfin, à mes parents qui m'ont toujours soutenue, à mon grand-père mon plus grand fan, et à mon gg : *Marcie du pu peurfond de mon thieur* (ça veut rien dire, mais je trouve que ça boucle bien).

Contents

Introduction

Software protection aims at defending programs against unwanted analysis. Its goal is to protect both the author of the program—i.e. the intelligence and contents of the software—but also the users, by assuring for example the confidentiality of their data. With the increasing creation and distribution of applications, and especially because of the burst in usage of mobile platforms in the past few years, the number and scale of the analyses performed on these applications is expanding.

Depending on the type of software and the identity of the analyst, the examination of applications can be considered as malicious or benevolent; and the qualification of one's intentions is not always trivial. For example, one could easily agree that the analysis by a company of a concurrent solution qualifies as industrial espionage and is malicious, while the examination of malware in order to design a defense mechanism is surely of common interest. However, the analysis of a program in order to assess its security is more difficult to sort into one category or the other, as it depends on the way the obtained information is used or disclosed. Therefore in this thesis we tend to use the term *analysis* instead of *attack*, and cast any good or bad intention aside.

There are many reasons why one would want to protect one's software, whether it is to impede security analysis, preserve intellectual property, or prevent non-authorized access to media content... For decades, the notion of *diversity* has been identified as a key factor to increase the security of systems [Coh93, FSA97]. Indeed, the analysis of a program is often based on some kind of pattern recognition, meaning that when a piece of code is associated with its semantics, it can be identified in other locations in the program, or even in other programs. Increasing the diversity of a program, i.e. representing semantically equivalent pieces of software with different sequences of instructions, makes the recognition more difficult and slows down the process of program analysis.

Furthermore, producing diverse instances of the same software reduces the replicability of the analysis. Ideally, an analysis successful on one instance of a system will need to be completely conceived again for another instance of the same system. Diversity techniques can be, for example, adding or deleting non-relevant

code (e.g. calls, jumps), reordering instructions, replacing equivalent sequences of instructions...

Recent examples of program diversification for security demonstrate the continuous interest in this field of research: for example, the idea of replacing equivalent sequences of instructions combined with the process of superoptimization has led to the concept of *superdiversification* [JJN+08]. Adding bogus instructions and shuffling code are also still considered useful techniques in the field of diversification, which can be used for example to conceal the purpose of patches, or to increase the resistance to ROP attacks [CKWG16]. Malicious applications (called *malware*) also use *polymorphism* and *metamorphism* in order to easily generate diverse instances and render their analysis less reproducible [WS06].

Program diversity has also induced the notion of software *obfuscation* [CTL97], namely the transformation of code in order to make its comprehension more difficult. While diversity aims at reducing the reproducibility of analysis on different instances, program obfuscation intends to make the analysis of one particular instance as complex as possible. Obfuscation thus conceals the semantics of a piece of software in two main ways: by mutating the code and data (e.g. changing their representation or value), and by adding irrelevant (and sometimes misleading) information (e.g. junk code). The obfuscation technique we chose to study uses both those principles in order to obfuscate common mathematical expressions. Operators are transformed with rewritings using Mixed Boolean-Arithmetic (MBA) expressions, and useless operators and constants are added during the obfuscation process. Throughout our research, we took an interest in both MBA obfuscation and MBA deobfuscation, as it is fundamental practice in computer security: knowledge about one way always helps to learn more about the other.

Contributions and Thesis Organization

Our work was mainly focused on Mixed Boolean-Arithmetic (MBA) expressions, which are used as a tool for obfuscating both constants and expressions. We use the traditional definition of a mathematical expression, meaning a finite combination of symbols (constants, variables, operators...) that is well-formed.

Several issues were tackled during this work:

- Structuring the state of the art around MBA obfuscation: indeed, the domain is quite young and the directly relevant literature is scarce. It can also borrow from more mature topics, like cryptography or rewriting. In the process, we reconstructed an MBA obfuscation technique from public obfuscated examples.

- Elaborating a definition of *simplification* of an obfuscated expression: we analyzed the factors that could explain the difficulty of designing a deobfuscation solution, and defined our own simplicity metrics for MBA expressions.

- Producing deobfuscation—or simplification—algorithms: this achievement filled the need to provide publicly available tools to analyze and automate, at least partly, the task to reverse the obfuscation technique. There was until now very little public work on the deobfuscation of MBA expressions.

- Assessing the *resilience* of the MBA obfuscation technique, both for expression and constant obfuscation: we define the resilience, in our case, as the difficulty of deobfuscating the obfuscated expressions in general.

- Providing new ideas to improve this resilience: we used both general deobfuscation techniques and our simplification algorithms to propose several ideas in order to improve the overall resilience of the MBA obfuscation technique.

We published our two solutions for MBA deobfuscation: the one based on bit-blasting was presented at Grehack 2016 [GEV16], while the one using word-level symbolic simplification was presented during SPRO 2016 [EGV16], along with other results on permutation polynomials [BERR16] which are not *per se* part of an MBA obfuscation but are nevertheless closely related.

The rest of this thesis is organized in the following way: the first chapter describes the general context inherent to our work, which is the definition and uses of obfuscation and its counterpart, reverse engineering.

The second chapter exposes the state of the art regarding MBA obfuscation, expression simplification, and other related fields such as bit-vector logic, SMT solvers, term rewriting, as well as some deobfuscation techniques that could apply in our context.

The third chapter presents our analysis of the notion of MBA complexity, detailing the issue of characterizing the simplicity (or complexity) of an MBA expression, and proposes our own metrics in this scope.

The fourth chapter depicts the reconstruction of the MBA obfuscation technique, and exhibits our two propositions for MBA simplification, one using bit-blasting and the other one being based on rewriting.

In the final chapter, we assess the resilience of the MBA obfuscation technique. We first study how known black-box attacks can deobfuscate MBA-obfuscated expressions, and then show how our own algorithms offer solutions to the problem of MBA deobfuscation. We conclude that the obfuscation technique does not provide great resilience in its current state, and suggest several improvements that would increase said resilience.

Chapter 1

Context

This chapter aims at giving some context about the work presented in this thesis: what is software obfuscation? What does it protect from? How can it be used? How to evaluate the quality of obfuscation techniques?

1.1 Software Obfuscation

In this section, we define what software obfuscation is, and give a few examples of situations where it is used.

1.1.1 Definition

Obfuscation is a process which consists in transforming a program in order to make its analysis difficult and costly, while preserving its *observable* behavior. Three properties often appear when dealing with obfuscation [BGI+01, CTL97]:

- *Functionality*: the obfuscated program must have the same input/output behavior (or *computational equivalence*) as the original program.
- *Efficiency*: the increase in size and execution time of the obfuscated program compared to the original program must be "acceptable" (depending on the usage context and the hardware constraints).
- *Resilience*: the obfuscated program must be harder to analyze (in terms of skills, time, tools...) than the original program. There are several metrics to assess the "obfuscating" aspect of a transformation, and we detail some examples in Section 1.5.

The field of software obfuscation is included in the wider scope of *software protection*, which can also include software encryption, packing, anti-debugging or

anti-tampering measures. Obfuscation often refers to transformations modifying the code in order to confuse it, while anti-debug and anti-tampering are transformations and/or processes designed to counter specific analysis; but they are quite often combined to offer more protection to the program. Some techniques are also hard to categorize: protections using code virtualization are sometimes called obfuscation, sometimes just software protection. Thus the limit between obfuscation and software protection is sometimes vague.

1.1.2 Uses

Obfuscation exists in a context where the software is distributed, meaning that the attacker/analyst has an instance of the program and completely controls the environment where it is executed—in the field of white-box design (see Section 1.4.3), it is called the *white-box attack context*. The process of analyzing a product based on its finished and distributed form is called *reverse engineering*. Its purpose is to understand one or several features of a program in order to describe a format, an algorithm, a protocol, and/or find an error in development (vulnerability research). Here, to *describe* means to give some high level semantics, e.g. a pseudo-code description, the name of a standard algorithm with inputs and parameters, a source code description (decompilation)... Reverse engineering of programs is commonly considered the inverse practice of obfuscation, and classical techniques are detailed in Section 1.3.

The analyst goals can be numerous: non-authorized use of the software (e.g. in the case of a licensed software), extraction of secrets or proprietary information (keys, protocols), redistribution (e.g. tampered version), or analysis for security purposes or interoperability (e.g. alternative client). Obfuscation traditionally offers protection against reverse engineering actions (sometimes called *Man-At-The-End (MATE) attacks*), mainly in those domains:

- Malware: obfuscating transformations provide confusion and polymorphism, thus allowing malicious software to avoid automatic signature detection by antivirus engines. Moreover, obfuscation also slows down the work of a reverse engineer analyzing the malware for further detection or investigation.

- Protection of intellectual property: obfuscation is a good tool to protect an algorithm or a protocol included in a commercial software. Known examples include Skype [PF06] or DropBox [KW13].

- Rights management: whether it is to protect the access to software with a license check, or to a digital content with a Digital Right Management (DRM) scheme, those technologies often embed critical information (e.g.

cryptographic keys and protocols) and protect this information with obfuscation. DRM schemes embedded in programs (e.g. VOD services or video games protection) are currently one of the main sources of obfuscation in modern settings [MG14].

- Protection of personal or sensitive data: in the case of mobile applications for example, a lot of sensitive data is held in the device (e.g. bank account details). Obfuscation can help secure this data, along with the protocols handling it.

1.2 Theoretical Obfuscation and Practical Obfuscation

In this section, we give a quick overview of the differences between theoretical obfuscation, or *cryptographic obfuscation*, and practical obfuscation as used in common commercial solutions.

1.2.1 Cryptographic Obfuscation

Cryptographic obfuscation aims at giving a formal context to obfuscation, especially around the idea of quantifying the difficulty of analyzing the obfuscated program $\mathcal{O}(P)$ of a given program P.

The formal study of software obfuscation was initiated by Barak et al. in [BGI+01], where they introduced the notion of *Virtual Black Box* (VBB) as the best possible property of an obfuscated program. The VBB property guarantees that anything that can be learned from $\mathcal{O}(P)$, can also be learned from the input/output behavior of P. In the same article, it is also proved that no general obfuscator achieving this property for any input program P can be built. Therefore, Barak et al. suggested in the same article a weaker definition of *indistinguishability obfuscation*: if two programs P_1 and P_2 of same size compute the same function, their obfuscation $\mathcal{O}(P_1)$ and $\mathcal{O}(P_2)$ should be indistinguishable from one another.

While this definition does not appear to give an explicit guarantee that the obfuscated program actually hides information, the work of Goldwasser et al. [GR07] proves otherwise. They provide a new definition of *best-possible obfuscation*: any information extracted from $\mathcal{O}(P)$ can be exposed by every other functionally equivalent program of similar size. They also prove that for efficient obfuscators, the definitions of indistinguishability and best-possible obfuscation are equivalent.

In 2013, the first candidate for general purpose indistinguishable obfuscation was proposed by Garg et al. [GGH+13], based on three components: branching

programs as the computational model, fully homomorphic encryption and multilinear maps. The first implementation of such indistinguishable obfuscation was presented by [DYJAJ14], and shows that its application for practical obfuscation is still not conceivable. Indeed, one example of function they obfuscate—using an Amazon EC2 machine with 32 cores—is a 16-bit point function containing 16 OR gates: the obfuscation process takes around 7 hours, and produces an obfuscated program of 31 GB with an execution time of about 3 hours. Those overheads are huge compared to the practical uses of obfuscation, for example DRM technologies or mobile applications.

Therefore, practical obfuscation often does not rely on such cryptographic concepts for now, but more on successive applications of different program transformations. The security of practical obfuscation can derive from several concept: loss of information, theoretical complexity, or composition of different weaker layers of obfuscation. We discuss this in more detail in Section 1.5.

1.2.2 Practical Obfuscation

As stated in the previous section, a practical obfuscator is based on the composition of different program transformations. Each of those transformation brings its own level of complexity, but the interleaving of the different obfuscation techniques greatly contributes to the global resilience of the obfuscated program. Some of the most used commercial obfuscators are strong.protect[1], Cloakware [LGJ08] (now included in Irdeto's[2] protection solution) and Arxan[3].

An obfuscator can apply its technique on different representations of a program, mainly the source code, the Intermediate Representation (IR) or the assembly language. The main problematic of an obfuscator designer is to find the correct trade-off between the protection brought by the obfuscation and the decline in performances (in terms of memory, time...).

Source Code

An obfuscator operating on the source code is called a *source-to-source* obfuscator, and enables the use of obfuscation techniques exploiting specificities of the input programming language. It is also easier to integrate into an existing compilation chain, as the obfuscation step is taking place before the compilation. Source-to-source obfuscation is also used on interpreted languages where the source code can be retrieved easily with decompilation: we quoted earlier the case of DropBox as an obfuscated software and it is written in Python. The Java language is also a

[1] https://strong.codes/
[2] http://irdeto.com/index.html
[3] https://www.arxan.com/

very productive field for source-to-source obfuscators, e.g. DashO[4] or ProGuard[5]. The main drawback to this approach is that it restricts oneself to one programming language.

Intermediate Representation

Intermediate Representation (IR) is used by compilers, virtual machines or reverse engineering frameworks to represent code. It is designed to be independent of any source or target language. Obfuscators working on the IR level are thus more general than source-to-source obfuscators, and present the ability to work on programs from various source languages and target assembly language. However, the integration is more difficult as it requires the obfuscator to be added to the existing compilation toolchain. Commercial obfuscators using intermediate representation, in this case the LLVM compiler IR [LA04], are for example strong.protect (based on the open-source project Obfuscator-LLVM [JRWM15]), or Epona (a commercial obfuscator developed by Quarkslab).

Assembly Language

The assembly level presents a major loss of information compared to the IR and source levels, thus it is very difficult to implement a general obfuscator working only on assembly. However, the technique of protection by virtualization can be applied directly on binary programs. The protected code then runs on a virtual CPU different from standard CPUs. One example of a commercial obfuscator implementing virtualization is VMProtect[6].

Apart from virtualization, transformations operating on assembly language commonly belong to the larger field of software protection, and more specifically regarding anti-tampering (e.g. integrity checks) or packing (e.g. encryption of the program).

1.3 Program Analysis

In this section, we present a few classical concepts of program analysis (or reverse engineering). A program is composed of both code and data, from which result the notions of *control flow* and *data flow*. Their common representations are detailed in Section 1.3.1. From these representations, two types of reversing approaches are possible, namely *static* analysis and *dynamic* analysis, described in Section 1.3.2.

[4]https://www.preemptive.com/products/dasho/overview
[5]http://proguard.sourceforge.net/
[6]http://vmpsoft.com/

1.3.1 Analysis Levels

From a disassembled program (e.g. with the IDA disassembler[7]), different pieces of information can be inferred, which traditionally belong to the control flow and/or the data flow.

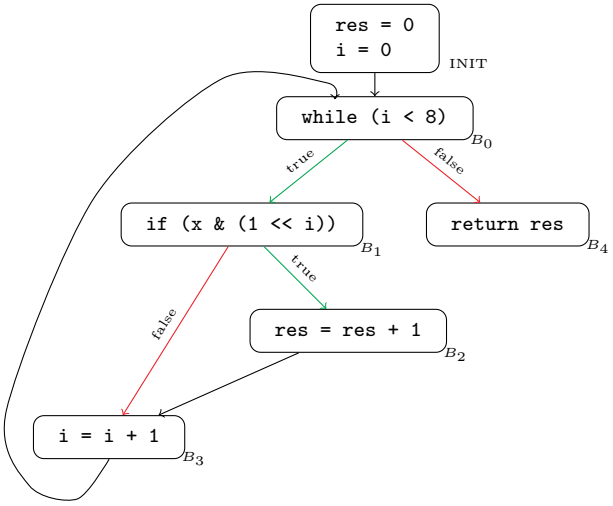
Control Flow

The control flow of the program designates all possible execution paths of the program, and how those paths are chosen. It is commonly represented with two graphs: the *Control Flow Graph* (CFG) and the *Call Graph* (CG).

The Control Flow Graph is specific to a function, and represents all possible executions of this function. Each node of the graph is a *basic block*: a continuous sequence of instructions, without any control transfer instruction (`jump`, `ret`, `call`...) or without being a target of a control transfer instruction. An edge between two basic blocks means that there is a possible execution path linking those blocks. We illustrate a simple example of a function computing the Hamming weight of a value x on 8 bits, and its Control Flow Graph in Figure 1.1—the nodes are labeled (INIT, $B_0, \ldots, B_4$) for further explanations in Section 1.4.1. The CFG of the same function in its compiled form as displayed in IDA is provided in Figure 1.2.

```
uint8_t hamming_weight(uint8_t x)
{
  uint8_t res = 0;
  uint8_t i = 0;
  while(i < 8) {
    if (x & (1 << i))
      res = res + 1;
    i = i + 1;
  }
  return res;
}
```

(a) Function in C.



(b) CFG in pseudo-code.

Figure 1.1: The `hamming_weight` function and its CFG.

[7] https://www.hex-rays.com/products/ida/

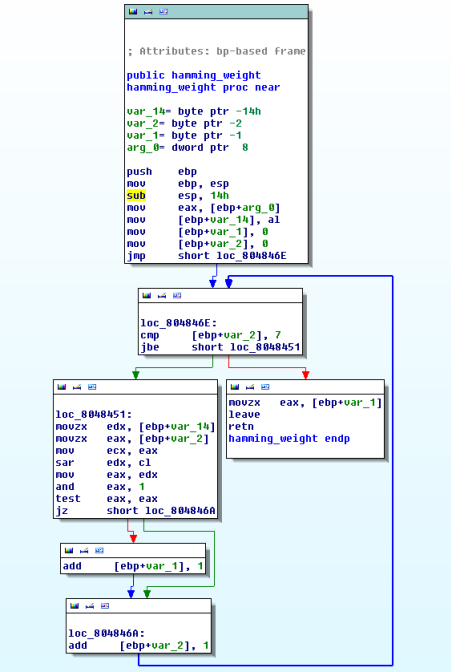


Figure 1.2: The CFG of `hamming_weight`'s assembly displayed in IDA.

The other graph displaying information about the execution flow of the program is the *Call Graph* (CG), where every node represents a function, and an edge from a node f_1 to a node f_2 means that function f_1 contains at least one call to the function f_2.

Data Flow

Compared to the control flow, the data flow does not possess a unique graph representation: different data flow analysis yield different representations. A classical example of a data flow analysis is the *reaching definitions*, which consists in determining where each variable may have been defined. Such information about data is extracted from the CFG of the functions, firstly by stating for each basic block the data-flow values entering and leaving the block. Many other data flow analysis can be performed, for example live-variable analysis or available expressions [ALSU06].

The literature about data flow analysis comes mostly from the fields of program optimization and program testing [KSK09, ALSU06]. In the reverse engineering context, an analyst might consider different notions about the data (not necessarily considering the *flow* of this data):

- the set of values computed at different points,

- the constant data of the program,
- the interactions between code and data (writes, reads).

For example, one can recognize a cryptographic algorithm by finding standard cryptographic constants, or deduce the computations performed in a basic block with its input and output values. In [BHMT16], the authors use a visual representation of the reads and writes of the data to deduce information about the implementation of a white-box and conceive a side-channel attack from the data addresses read or written in memory.

While this type of analysis should be referred as more general *data analysis*, it is very commonly called data flow analysis. In general, any information deduced from the data of a program is referred as belonging to the data flow.

1.3.2 Analysis Methods

Static Analysis

The static analysis of a program is done without executing it, by examining the code of the program itself, very probably in disassembled form. It is sometimes possible to *decompile* the binary program to get a source code representation of it, but the decompilation is a difficult problem and it is not always possible to achieve it, especially on obfuscated programs.

Static analysis can use general information, such as the type of the program or its constant data (strings, numbers...). The first step is often to reconstruct the CFGs of the functions and the CG of the program—this is done automatically with a tool, for example a disassembling framework. Static data flow analysis can also be performed from a CFG.

To simulate the execution of the program, it is possible to use *symbolic execution* [Kin76], which assumes symbolic values for inputs, and expresses the constraints determining the execution paths with regards to those symbols. Examples of reverse engineering tools providing symbolic execution on binary programs are Triton [SS15] and Miasm [Des12].

Static analysis is safer for the analysis of potentially malicious binaries, as no execution is needed. It is sometimes the only possibility of analysis, for example when the target architecture of the executable is not available, or when no dynamic instrumentation tool exists. Because this type of analysis may bring the reverser to consider execution paths that are not actually taken, it is considered an overapproximation of the possible executions of the program. In the case where the analysis would fail to recognize some execution paths of the program (i.e. non-exhaustive analysis), the failure of the overapproximation would be caused by the analysis being unsound.

Dynamic Analysis

The dynamic analysis of a program is done on particular inputs, and infers information on the execution paths taken and data modifications. As only a subset of the possible paths of execution is analyzed, the dynamic analysis is considered an underapproximation of the possible executions of the program.

A few examples of dynamic analysis include:

- Debugging: interactively watch the program during its execution, step-by-step or by using breakpoints.
- Tracing: log every executed instruction, and analyze it off-line (after execution).
- Data tainting: mark one or several interesting variables and watch their impact on other variables of the program during execution.

Most of the dynamic analysis technique can also be used statically with symbolic execution.

1.4 Some Classical Obfuscation Techniques

In this section, we present a few classical obfuscation techniques. To learn more details about those techniques and find more examples, please refer to work such as [CN09, DGBJ14, CTL97].

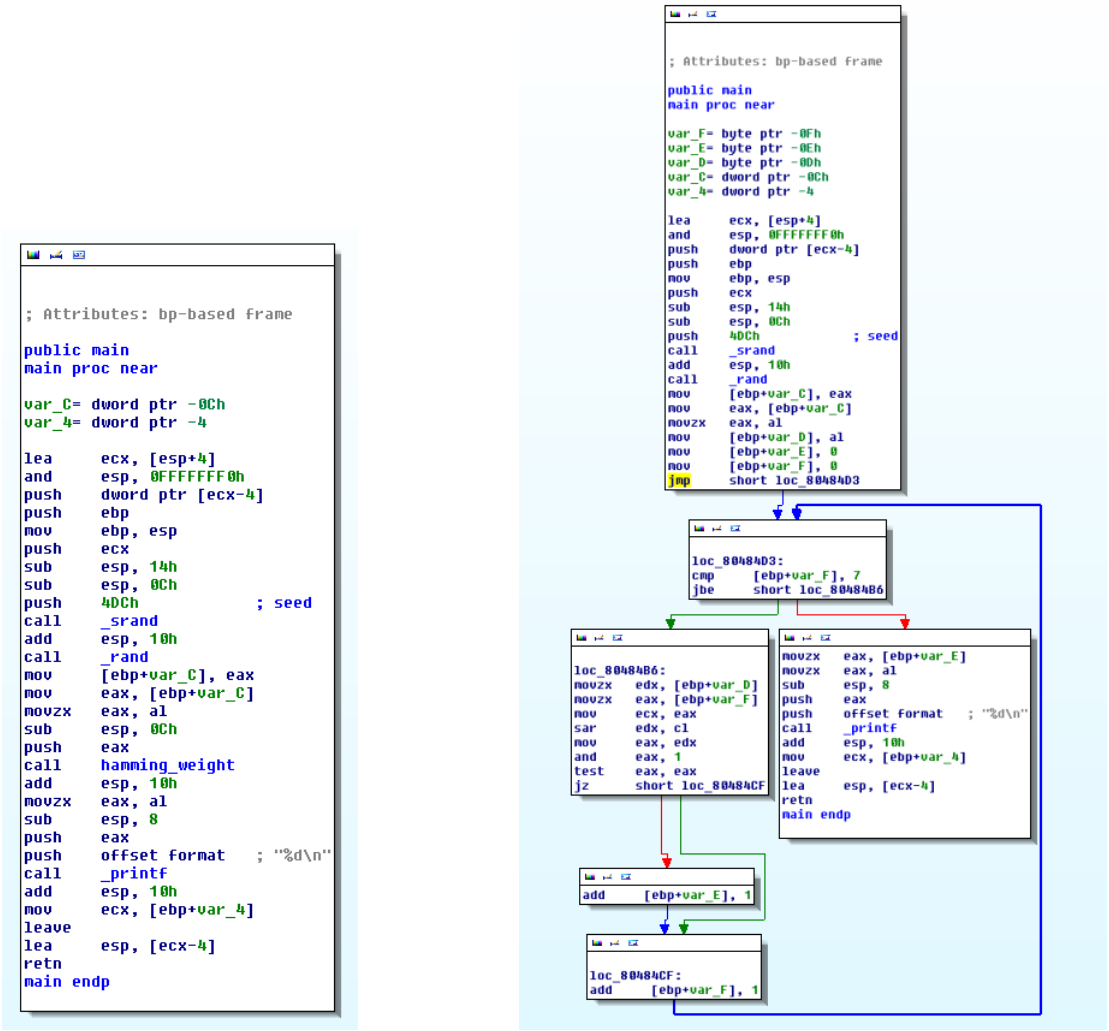
1.4.1 Control Flow Obfuscation

Confusing the control flow of a program means that the different execution paths must not appear clearly to the analyst (at least statically). Two simple examples of obfuscation techniques that obscure the control flow are *inlining* and *outlining* of functions. We start by detailing those simple examples, then present the *opaque predicate* technique, and finally the process of *control flow flattening*. These are all classical examples, but a lot more can be imagined to confuse the control flow.

Inlining and Outlining

The technique of *inlining*—actually coming from the field of program optimization—replaces a call to a function g by the body of the procedure itself. As an obfuscation technique, this has the advantage to modify both the CFG of the calling function, and the CG of the program (since the replaced call is no longer viewed as such). An example of the inlining process is illustrated in Figure 1.3, with the `main` function calling `hamming_weight`. The Figure 1.3a shows the CFG of `main` without

the inlining (the call to `hamming_weight` is visible), whereas in Figure 1.3b, one can recognize the CFG of `hamming_weight` (cf. Figure 1.2) included in the CFG of `main`. In the CG of the second program, the edge from `main` to `hamming_weight` would have disappeared.



(a) The `main` function without inlining.

(b) The `main` function with `hamming_weight` inlined.

Figure 1.3: Illustration of the inlining of `hamming_weight` in `main`.

Inverting the process of inlining creates the *outlining* obfuscation: this technique replaces a piece of code in a function f with a call to a new function g containing this particular piece of code. It changes the CFG of the original function f and adds a new node g in the CG, as well as an edge from f to g.

Opaque Predicate

An opaque predicate is a boolean expression which value is known during obfuscation, but hard to compute for the analyst of the obfuscated program. Opaque predicates can be based on mathematical theorems (e.g. Expression (1.1)), or on information difficult to obtain while analyzing (e.g. aliasing, see [CTL98]).

$$\begin{aligned} &\forall x, y \in \mathbb{N}, \text{ if } x = y^2 \\ &\text{then } P = \Big((x \equiv 0 \mod 4) \vee (x \equiv 1 \mod 4)\Big) \text{ is always true} \end{aligned} \tag{1.1}$$

The traditional construction using opaque predicates such as P in Expression (1.1) is to create fake branches in the CFG of a function: the value of P being always true, the branch corresponding to its evaluation to false will never be on any execution path, and *junk code* (useless code) can be inserted. This obfuscation process is illustrated in Figure 1.4 (the dashed edge represents an execution path that will never happen for any input of the program).

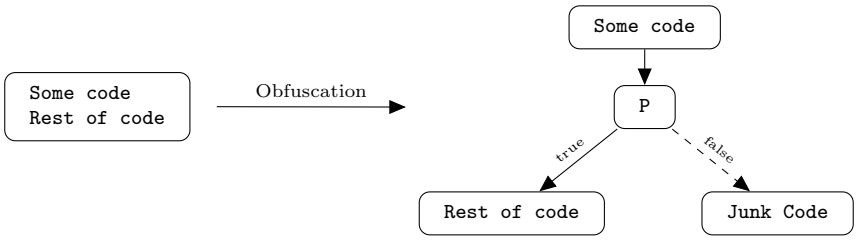


Figure 1.4: Classical opaque predicate construct with P always true.

Another similar way to use opaque predicates is to choose a predicate P evaluating randomly to true or false, and execute the same code on the two conditional branches. This obviously needs to be combined with various obfuscation techniques that will make the two blocks difficult to identify as the same code. This process is illustrated in Figure 1.5.

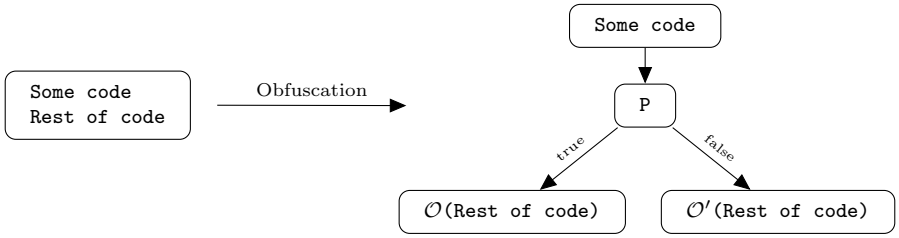


Figure 1.5: Classical opaque predicate construct with P randomly true or false.

Obfuscation by opaque predicates can be attacked for example by using symbolic execution to gather opaque predicates and an SMT solver to solve them.

Control Flow Flattening

The idea of control flow flattening is to completely change the structure of a function's CFG, by encoding the information of the control flow in the data flow. One way of achieving this purpose is to number the basic blocks, and create a *dispatcher* that manages the execution with a variable. This variable determines which block should be executed after the current block. At the end of each basic block, the variable is updated and the execution goes back to the dispatcher. This process operated on the `hamming_weight` function is illustrated in Figure 1.6.

One can see that the value of the variable `next` is controlling the execution flow. The strength and weakness of this technique thus reside in the concealing of this variable. Deobfuscation can be performed statically if the values are easy to read (e.g. in Figure 1.6), or dynamically by following the order of execution of the different basic blocks.

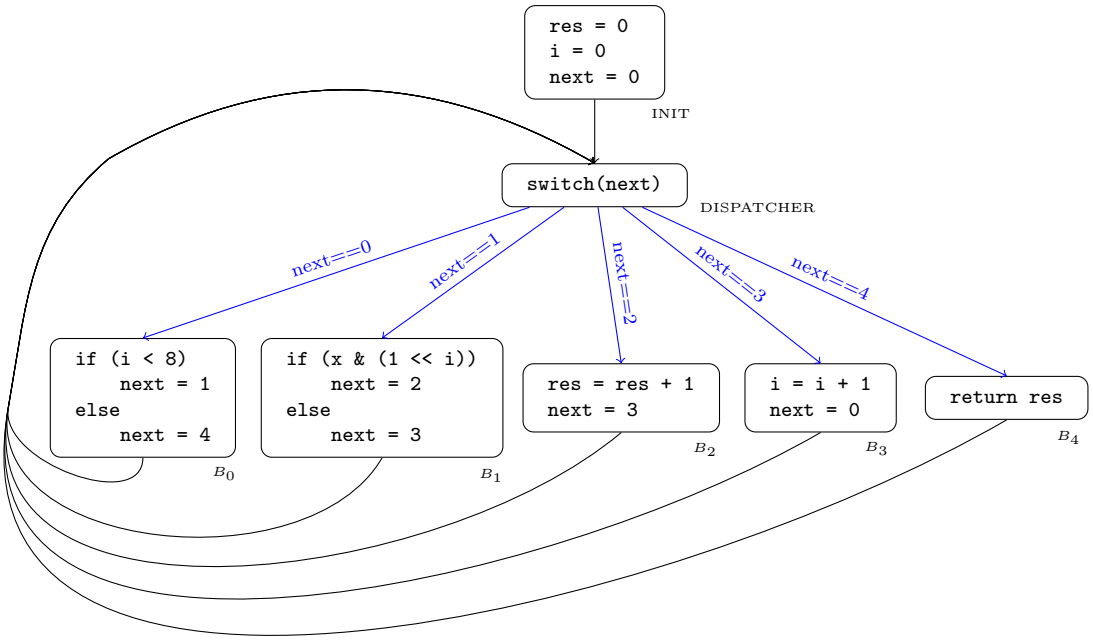


Figure 1.6: Control flow flattening on `hamming_weight`.

1.4.2 Data Flow Obfuscation

While obscuring the data flow, one aims at concealing the values taken during execution, as well as the information that can be inferred from the data organization and interactions. While *splitting variables* scatters the meaning of a variable into several ones, changing their *representation* or using *encodings* focuses on hiding the values and/or operations appearing during execution.

Splitting Variables

It is possible to split a variable into two or more variables, multiplying as much the analysis work to understand the interest of this variable. For example, one can choose a positive integer a and describe an integer variable x of the program as $x_1 \times a + x_2$. Then basic operations such as $x + 1$ or $x \times 2$ would be described in a more complex way, as illustrated with Figure 1.7. In this figure, $\mathcal{O}$ represents the obfuscation transformation.

$$\mathcal{O}(x) \rightarrow x_1 \times a + x_2$$

$$\mathcal{O}(x+1) \rightarrow \begin{cases} x_1 = x_1 + \left\lfloor \dfrac{x_2}{a-1} \right\rfloor \\ x_2 = x_2 + 1 \mod a \end{cases}$$

$$\mathcal{O}(x \times 2) \rightarrow \begin{cases} x_1 = 2x_1 + \left\lfloor \dfrac{2x_2}{a} \right\rfloor \\ x_2 = 2x_2 \mod a \end{cases}$$

Figure 1.7: An example of integer variable splitting.

Changing Variables Representation

It is possible to change the representation of variables, in order to hide both the actual values taken by the variables, as well as the operations applied on them.

Integers are traditionally represented in binary form in the machine level, using powers of two. To obfuscate that representation, the patent [FCMCI12] describes a technique representing integers with another basis (e.g. in powers of three). This allows to conceal both the value taken by the variables, and the computation of a XOR between two variables. The process is described in Figure 1.8: the function f changes to the new representation using basis 3, and g back to the binary representation. An addition in the alternative representation is equivalent to a XOR in the original one, since the carries are not taken into account.

$$\forall x, y, n \in \mathbb{N} \qquad x = \sum_{i=0}^{n} x_i 2^i \text{ and } y = \sum_{i=0}^{n} y_i 2^i$$

$$\text{if } f(x) = \sum_{i=0}^{n} x_i 3^i \quad \text{and} \quad g(x) = \sum_{i=0}^{n} (x'_i \mod 2) 2^i \quad \text{with } x = \sum_{i=0}^{n} x'_i 3^i$$

$$\text{then} \quad x \oplus y = g(f(x) + f(y))$$

Figure 1.8: An example of representation change.

Encodings

Using encodings is one of the most common data flow obfuscation techniques. Encodings aim at preventing the value of a variable to appear in clear in memory at any point of the program execution. This technique is borrowed to the field of cryptographic white-box (see Section 1.4.3), where encoding functions are merged within lookup tables to conceal the values between the different steps of a cipher.

In practice, the encodings in obfuscation are very often affine functions, because they are easily invertible and do not produce a big overhead in terms of performance. Traditionally, the variable must be decoded before any computation and re-encoded after; it is possible to use encodings homomorphic to an operator (so that some computations might be done on the encoded values), but this greatly decreases the diversity of the possible encodings.

1.4.3 White-Box

One area where cryptography and practical obfuscation converge is the subject of white-box cryptography [CEJVO03, Mui13], which aims at protecting cryptographic algorithms embedded in a program in the *white-box attack context*—meaning the environment is under the control of the attacker. Compared to obfuscation that can be applied to theoretically any program and conceal either data or algorithms, only cryptographic algorithms are white-boxed, and the process may use particularities of the algorithm to help conceal the key used. White-box cryptography is recognized as a useful concept to obfuscate cryptographic algorithms and is often considered indispensable in an obfuscator (e.g. the solutions proposed by CryptoExperts[8] or Arxan[9]).

[8] `https://www.cryptoexperts.com/technologies/white-box/`

[9] `https://www.arxan.com/technology/white-box-cryptography/`

1.5 Quality of an Obfuscation Technique

As explained in Section 1.1.2, reverse engineering aims at extracting high-level semantics from parts of a program. The goals may be to identify a standard algorithm from its parameters (e.g. standard cryptographic constants), to understand a custom algorithm in order to invert it or re-use it with different parameters.... In order to achieve these goals, the reverser infers information about the structure and the behavior of the program mainly through its control flow and data flow, as detailed in Section 1.3.

To assess the quality of an obfuscation technique, one would try to determine how the transformation prevents (or at least slows down) this extraction of information. The difficulty of this question relies on the fact that the attacks on obfuscation are composed of both human and automatic analysis: reverse engineers bring their own experience and intuition (which are hard to quantify), as well as their own analysis tools.

Designing metrics to help characterize a "good" obfuscation can take several approaches: a first one is to use classical software complexity metrics (presented in Section 1.5.1) and derive definitions for the quality of obfuscation from those properties (see Section 1.5.2). Another approach detailed in Section 1.5.3, presented by Dalla Preda et al. [DPG05], uses abstract interpretation to give an attack model to obfuscation. We discuss and compare these approaches in Section 1.5.4.

1.5.1 Complexity Metrics of a Program

We give here a few examples of typical software complexity metrics [CTL97]. For a more detailed list of metrics, readers fluent in French can look at the PhD thesis of Marie-Angela Cornelie [Cor16]. Those metrics can be sorted in roughly three types:

- Number of instructions: can be counted as the number of operators or operands (distinct or not), and can also be used to compute more metrics such as the program vocabulary, the volume, etc [Hal77].
- Control flow: cyclomatic complexity (number of linearly independent paths), nesting level, knots...
- Data flow: fan-in/fan-out of instructions, data-flow complexity (number of inter-basic block variable references), data structure complexity...

One could just use these metrics and say that a good obfuscation should increase some chosen ones (depending on the obfuscation technique). They are for example used in the definition of a *potent* obfuscation, detailed in next section.

1.5.2 Metrics for Obfuscation

In [CTL97], Collberg et al. define three metrics that are often used as a basis to characterize a good obfuscation transformation:

- *potency*: uses one of the software complexity metrics (see Section 1.5.1); if the obfuscation transformation increases this complexity, then it is a *potent* transformation.
- *resilience*: determines the resistance of the transformation to the programmer effort (human analysis) and the deobfuscator effort (automatic analysis). Collberg et al. define the programmer effort as local, global, inter-procedural or inter-process, while the deobfuscator effort is either in polynomial time or in exponential time.
- *cost*: the extra execution time and space of the obfuscated program compared to the original. The cost can be free ($\mathcal{O}(1)$ more resources), cheap ($\mathcal{O}(n)$), costly ($\mathcal{O}(n^p)$) or dear (exponentially more resources).

Then the authors define the *quality* of an obfuscation transformation as the combination of these three metrics. Another common metric is also *stealth*, meaning the difficulty to detect the obfuscation. In his book [CN09], Collberg defines both steganographic stealth—the analyst cannot determine if the transformation has been applied or not—and local stealth—the analyst cannot tell where the transformation has been applied.

Other definitions of potency and resilience were also proposed: Karnick et al. [KMM+06] define the potency from three metrics (nesting complexity, control flow complexity and variable complexity) and the resilience using Java decompilers (which is a definition hard to generalize to binary programs).

By just using the software complexity metrics, those definitions do not take into account the reverse engineering process. That is why Dalla Preda et al. proposed an attack model and a definition of potency relying on the properties that a reverser might want to infer from the analysis of the program.

1.5.3 Attack Model from Abstract Interpretation

In their work, Dalla Preda et al. [DPG05] present a model for attacks on obfuscation based on abstract interpretation, which is a way of approximating the concrete semantics of a program. They encode the properties that can interest an analyst as elements φ of an abstract domain (modeling the static and dynamic analyzers). The abstraction model can thus adapt to the context, being fine when the analyst is interested in details of the program. In this setting, if a transformation

is potent to a property φ, it means that a reverser cannot deduce this property from the obfuscated program—this identifies the class of attacks against which the obfuscation is potent. As pointed by the authors, this does not guarantee that the obfuscation cannot be easily undone, i.e. its resilience. However, while they do not provide a general model for resilience estimation, they provide a case study with the assessment of the resilience of opaque predicate insertion.

1.5.4 Discussion

Using software complexity metrics to determine the potency of an obfuscation transformation presents several drawbacks. First, it only characterizes one aspect of the obfuscation and there is no trivial way to combine several aspects. Besides, as these metrics describe the program in a static way, it is quite easy to design an obfuscation transformation that would artificially increase the metrics by inserting dead code (code that is never executed).

The resilience, as defined by Collberg et al. [CTL97], is based on the *programmer effort* (which is the analyst effort) and the *deobfuscator effort* (which is the automatic tool effort). The programmer effort is defined regarding the locality of the obfuscating transformation, and for example, a local obfuscation would only bring weak or trivial resilience—depending on the deobfuscator effort. In the case of expression obfuscation, the original expression is often obfuscated locally (a branch-free expression is typically a basic block in the CFG), but can still be considered to have a good resilience if it is hard to automatically deobfuscate. In the same way, the deobfuscator effort is only defined as its execution time and space (polynomial or exponential), but this does not take into account the difficulty of designing such a deobfuscator.

The abstract model of Dalla Preda (see previous section) and its definition of potency are closer to real settings as they take the reverser interest and methods into account. But the abstraction of the static and dynamic analyzers raises one observation: how are we supposed to know the methods and tools used by the analyst? While there is a lot of public work on the subject, reversers often possess their own private toolkit, and it is not really possible to assess the resistance against unknown analysis.

Another factor to the complexity of quantifying the quality of an obfuscation technique is the composition of obfuscation layers: several weak transformations might produce a resilient obfuscated program in the end—this was studied in [JSV09] with metrics such as instruction count, cyclomatic number and knot count. This kind of metrics can nevertheless help the designer of an obfuscation technique to guide its elaboration. Recent work [MP15] focus more on the *unintelligibility* of the code, by using for example the Kolmogorov complexity. The

authors assume that an obfuscated code exhibiting more irregularities requires a longer description in order to be characterized, and thus makes it harder to comprehend. This does not necessarily indicate if the obfuscation is hard to automatically deobfuscate or not, but it can indeed give some information about the *diversity* of the transformation (how many diverse outputs it can produce, for example). Experimental evaluation of code obfuscation techniques [Mar14] is also a recent field of research.

Chapter 2

Existing Work

In this chapter, we introduce the existing work about the subjects later developed in this thesis: MBA obfuscation, expression simplification, bit-vector logic and other deobfuscation techniques close to our research.

2.1 MBA Expressions

The present section details the definition and uses of *Mixed Boolean-Arithmetic (MBA) expressions*. These are expressions that mix classical arithmetic operators (addition, multiplication...) and boolean operators (exclusive-or, and, or...). In full generality, expressions mixing arithmetic and bitwise operators are already in use in broad contexts without being given a name. Any expression mixing arithmetic operators and bitwise ones, for example applying a boolean mask on an integer before an addition, fulfills the minimal requirements to be called an MBA. Moreover, any bitwise or arithmetic operator available in a processor might be used to construct an MBA expression.

However, if for a general characterization we do not exclude any existing arithmetic or bitwise operator, in this thesis we will consider *polynomial* MBA expressions as defined by Zhou et al. in [ZMGJ07], since all the MBA expressions we have encountered up to now in the context of obfuscation are of this form.

2.1.1 Polynomial MBA Expressions

The definition of MBA expressions as a tool for obfuscating programs was given by [ZM06, ZMGJ07]. We reproduce in this section the important notions around MBA expressions (or as we call them sometimes, *mixed* expressions).

Definition 1 (Polynomial MBA [ZMGJ07])**.** *An expression E of the form*

$$E = \sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_0, \ldots, x_{t-1}) \right) \tag{2.1}$$

where the arithmetic sum and product are modulo 2^n, a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables $x_0, \ldots, x_{t-1}$ in $\{0,1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I, J_i \subset \mathbb{Z}$ are finite index sets, is a polynomial Mixed Boolean-Arithmetic (MBA) expression*. A* linear *MBA expression is a polynomial MBA expression of the form*

$$\sum_{i \in I} a_i e_i(x_0, \ldots, x_{t-1}),$$

For example, the expression E written as

$$E = (x \oplus y) + 2 \times (x \wedge y) \tag{2.2}$$

is a linear MBA expression, which *simplifies* to $E = x + y$. An example of a non-linear polynomial MBA expression could be

$$85 * (x \vee y \wedge z)^3 + (xy \wedge x) + (xz)^2.$$

Since the MBA-obfuscated expressions we have studied so far rely on composing layers of MBA rewritings, the following statement exposed in [ZMGJ07] is essential to us: by definition, the composition of polynomial MBA expressions is still a polynomial MBA expression (as the variables $x_0, \ldots, x_{t-1}$ can be polynomial MBA themselves). This guarantees that we are only working with polynomial MBA expressions.

For conciseness, the term "MBA expression" in the rest of this thesis stands for a polynomial MBA expression. Moreover, this study is limited to the most frequent operators: arithmetic $\{+, -, \times\}$ and boolean $\{\wedge, \vee, \oplus, \neg\}$—we use both the terms *bitwise* and *boolean* for this type of operators. The list of available operators can vary between use cases. For example, in [ZMGJ07], besides the *usual* operators $\{+, -, \times, \wedge, \vee, \oplus, \neg\}$, signed and unsigned inequalities alongside signed and unsigned shifts are also considered. Other operators as shuffle or convolution are not taken into account, even if they are relevant in other contexts [Vui03].

We deliberately chose not to detail the subject of MBA inequalities, since it partially changes the issue to handle. Indeed, an MBA inequality is an assertion, and its value is either true or false. While this value can also be interpreted as a number, the situation is slightly different from an expression that can take a great range of values. Depending on the context, an attacker might just want to check if the inequality is satisfiable or not, instead of recovering a simpler form

of the expression. This problem is related to the constant obfuscation technique proposed in [ZMGJ07], since the only value taken by the obfuscated expression is the constant to be hidden. We detail this technique in the next section, and discuss its security in Section 5.3. An assessment of the security of this constant obfuscation technique was also done by Biondi et al. in [BJLS15], which we discuss further in Section 2.2.4.

2.1.2 MBA-based Obfuscation

Obfuscation of Expressions

The technique to obfuscate one or several operators using MBA expressions was first presented in [ZM06, ZMGJ07] and in various patents [JGZ08, KZ05, GLZ12] with intersecting lists of authors. The process relies on two components:

- MBA rewriting: a chosen operator is rewritten with an equivalent MBA expression, as can be seen in Expression (2.2). A list of rewriting examples is given in the articles and patents from Zhou et al., and in [ZMGJ07], a method to generate new MBA equalities is detailed (see next section). Other examples of MBA equalities can be found in works regarding *bit hacks*, for example [War02].
- Insertions of identities: let us call e any part of the expression being obfuscated, then we can write e as $f(f^{-1}(e))$ with f any invertible function on $\mathbb{Z}/2^n\mathbb{Z}$. In the work of Zhou et al. f is an affine function. As this is very close to the process of encoding (see Section 1.4.2), we often refer to this step as the encoding step.

Those two principles can be observed in the process of getting Expression (2.3), an example of an obfuscated MBA expression on two variables $x, y \in \{0,1\}^8$.

$$\begin{aligned} e_1 &= (x \oplus y) + 2 \times (x \wedge y) \\ e_2 &= e_1 \times 39 + 23 \\ E &= (e_2 \times 151 + 111) \end{aligned} \tag{2.3}$$

The MBA rewriting used for the obfuscation process is given in Expression (2.4), and the encoding affine functions in Expression (2.5).

$$x + y \to (x \oplus y) + 2 \times (x \wedge y) \tag{2.4}$$

$$\begin{aligned} f &: x \mapsto 39x + 23 \\ f^{-1} &: x \mapsto 151x + 111 \end{aligned} \tag{2.5}$$

Then, the following expression stands for $x + y$ on 8 bits:

$$(((x \oplus y) + 2 \times (x \wedge y)) \times 39 + 23) \times 151 + 111.$$

To increase the strength of the obfuscation, one can apply the two principles (rewritings and encodings) iteratively as much as wanted. This technique has proved to be quite popular in obfuscation, in real life settings ([MG14, BS14, JGZ08]).

The MBA-obfuscated expression found by Mougey et al. while analyzing an obfuscated program is reproduced here in the form of a Python function, in Figure 2.1. This version of the expression has been obtained by using the framework Miasm to symbolically execute the MBA-obfuscated assembly, and then displaying Miasm IR with a more high-level representation: the Python language. We detail this process in Section 4.1.

$$\begin{aligned}
a &= 229x + 247 \\
b &= 237a + 214 + ((38a + 85) \wedge 254) \\
c &= (b + ((-2b + 255) \wedge 254)) \times 3 + 77 \\
d &= ((86c + 36) \wedge 70) \times 75 + 231c + 118 \\
e &= ((58d + 175) \wedge 244) + 99d + 46 \\
f &= (e \wedge 148) \\
g &= (f - (e \wedge 255) + f) \times 103 + 13 \\
R &= (237 \times (45g + (174g \vee 34) \times 229 + 194 - 247) \wedge 255)
\end{aligned}$$

Figure 2.1: MBA-obfuscated expression of $(x \oplus 92)$ [MG14].

Opaque Constant

An obfuscation method based on MBA expressions to hide constants is also presented in [ZMGJ07]. It uses *permutation polynomials*, which are invertible polynomials over $\mathbb{Z}/2^n\mathbb{Z}$. They were characterized by Rivest in [Riv99], but no inversion algorithm was given then. Zhou et al. provide a subset of invertible polynomials of degree m noted $P_m(\mathbb{Z}/2^n\mathbb{Z})$, as well as a formula to find the inverse of such polynomials. The constant obfuscation (or *opaque constant*) is constructed in the following way. Let:

- $P \in P_m(\mathbb{Z}/2^n\mathbb{Z})$ and Q its inverse: $P(Q(X)) = X \quad \forall X \in \mathbb{Z}/2^n\mathbb{Z}$,
- $K \in \mathbb{Z}/2^n\mathbb{Z}$ the constant to hide,

- E an MBA expression of variables $(x_1, \ldots, x_t) \in (\mathbb{Z}/2^n\mathbb{Z})^t$ non-trivially equal to zero (e.g. $E = x + y - (x \oplus y) - 2 \times (x \wedge y)$),

Then K can be replaced by $P(E + Q(K)) = P(Q(K)) = K$, whatever the values taken by $(x_1, \ldots, x_t)$. We thus have a function computing K for all its input variables, variables that can be chosen randomly in the rest of the program—this increases the dependency of the opaque constant to the rest of the code. We show a weakness of this technique in Section 5.3.

2.1.3 Generating New MBA Equalities

In their paper [ZMGJ07], Zhou et al. also provide a method to generate new linear MBA equalities, that can later be used to create rewrite rules. This method is derived from their first theorem, that we rephrase here for better clarity. Note that the theorem we reproduce in this thesis differs from the original of [ZMGJ07], because we keep only one direction of the equivalence—this is the only direction we were able to prove, despite the other one being described as "plain" by Zhou et al.

Theorem 1 ([ZMGJ07])**.** *With n the number of bits, s the number of bitwise expressions and t the number of variables, all positive integers, let:*

- $(X_1, \ldots, X_k, \ldots, X_t) \in \{\{0,1\}^n\}^t$ *be vectors of variables on n bits,*
- $e_0, \ldots, e_j, \ldots, e_{s-1}$ *be bitwise expressions,*
- $e = \sum_{j=0}^{s-1} a_j e_j$ *be a linear MBA expression, with a_j integers,*
- $e_j(X_1, \ldots, X_t) = \begin{pmatrix} f_j(X_{1,0}, \ldots, X_{t,0}) \\ \vdots \\ f_j(X_{1,n-1}, \ldots, X_{t,n-1}) \end{pmatrix}$ *with $X_{k,i}$ the i-th bit of X_k and*

$$f_j : \{0,1\}^t \to \{0,1\} \qquad 0 \leq j \leq s-1$$
$$u \mapsto f_j(u)$$

- $F = \begin{pmatrix} f_0(0) & \ldots & f_{s-1}(0) \\ \vdots & & \vdots \\ f_0(2^t-1) & \ldots & f_{s-1}(2^t-1) \end{pmatrix}$ *the $2^t \times s$ matrix of all possible values of f_j for any i-th bit.*

If $F \cdot V = 0$ has a non-trivial solution, with $V = (a_0, \ldots, a_{s-1})^\intercal$, then $e = 0$.

Proof: Let $F \cdot V = 0$, with $V = (a_0, \ldots, a_{s-1})^\intercal$. If we explicit $F \cdot V$, we get:

$$F \cdot V = \begin{pmatrix} f_0(0) & \ldots & f_{s-1}(0) \\ \vdots & & \vdots \\ f_0(2^t-1) & \ldots & f_{s-1}(2^t-1) \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{s-1} \end{pmatrix} = \begin{pmatrix} \sum_{j=0}^{s-1} a_j \cdot f_j(0) \\ \vdots \\ \sum_{j=0}^{s-1} a_j \cdot f_j(2^t-1) \end{pmatrix},$$

meaning that $F \cdot V = 0 \Leftrightarrow \sum_{j=0}^{s-1} a_j \cdot f_j(l) = 0$ for every $l \in 0, 2^t - 1$. This is equivalent to having $\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i} \ldots, X_{t,i}) = 0$ for every i, whatever the values of the $X_{k,i}$.

On the other hand, we can write e as:

$$\begin{aligned} \sum_{j=0}^{s-1} a_j \cdot e_j(X_1, \ldots, X_t) &= \sum_{j=0}^{s-1} a_j \cdot \begin{pmatrix} f_j(X_{1,0}, \ldots, X_{t,0}) \\ \vdots \\ f_j(X_{1,n-1}, \ldots, X_{t,n-1}) \end{pmatrix} \\ &= \sum_{j=0}^{s-1} a_j \cdot \sum_{i=0}^{n-1} f_j(X_{1,i}, \ldots, X_{t,i}) \cdot 2^i \\ &= \sum_{j=0}^{s-1} \Big(\sum_{i=0}^{n-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i}) \cdot 2^i \Big) \\ &= \sum_{i=0}^{n-1} 2^i \Big(\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i}) \Big). \end{aligned}$$

If $F \cdot V = 0$, then $\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i} \ldots, X_{t,i}) = 0$ for every i, thus

$$\sum_{i=0}^{n-1} 2^i \Big(\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i}) \Big) = 0 \quad \text{for all } i, \ 0 \leq i \leq n-1,$$

meaning that $e = 0$.□

One could note that the bitwise expressions e_j cannot contain constants other than 0 and -1. While expressions such as $(x \oplus 0)$ or $(x \wedge (-1))$ can be represented with one truth table that is valid for every bit i (the truth table of x in these examples), this proof does not hold for bitwise expressions such as $(x \oplus 92)$.

From Theorem 1, a method to create new linear MBA equalities can be deduced. From any $\{0,1\}$-matrix of size $2^t \times s$ with linearly dependent column vectors, one can generate a linear MBA expression of t variables equal to zero.

For example, the matrix:

$$F = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

with column-vectors truth-tables for

$$\begin{aligned} f_0(x,y) &= x \\ f_1(x,y) &= y \\ f_2(x,y) &= (x \oplus y) \\ f_3(x,y) &= (x \vee (\neg y)) \\ f_4(x,y) &= (-1) \end{aligned}$$

has the vector $V = (1,-1,-1,-2,2)^\intercal$ as a solution to $F \cdot V = 0$. This yields the following linear MBA equation:

$$x - y - (x \oplus y) - 2(x \vee (\neg y)) - 2 = 0.$$

This equation can be derived in many equalities, for example:

$$\begin{aligned} x - y &= (x \oplus y) + 2(x \vee (\neg y)) + 2 \\ (x \oplus y) &= x - y - 2(x \vee (\neg y)) - 2, \end{aligned}$$

each equality then producing two rewrite rules, as explained in Section 2.4.

2.1.4 MBA in Cryptography vs in Obfuscation

Before being given the name of MBA in the context of obfuscation, such a mixing of bitwise and arithmetic operators was already used in the context of cryptography to design symmetric primitives, with the stated goal of getting efficient, non-linear and complex interactions between operations. Building blocks such as ARX designs (e.g. [KN10]) and some generalizations can be found in major algorithms like hash functions (e.g. the SHA family), stream ciphers (e.g. Salsa family) or block ciphers (e.g. XTEA).

The notion of T-functions [KS03]—i.e. functions where the i-th output bit only depends on the first i input bits—appears both in the context of cryptography and obfuscation, as integer arithmetic operators, such as $(+, -, \times)$ are triangular T-functions and provide efficient non-linear invertible functions.

However, regarding MBA expressions, there is a key difference between what is looked for in cryptography and in obfuscation. In cryptography the MBA expression is the direct result of the algorithm description, and the resulting cryptosystem has to verify a set of properties (e.g. non-linearity, high algebraic degree) from a black box point of view. The complex form of writing is directly related to some kind of hopefully intrinsic computational complexity for the resulting function: one wants the inverse computation to be difficult to deduce without knowing the key. In obfuscation, an MBA is the result of rewriting iterations from a simpler expression which can have very simple black box characteristics. There is no direct relation between the complex form of the expression and any intrinsic computational complexity of the resulting function: on the contrary, when obfuscating simple functions, one knows that the complex form of writing is related to a simpler computational function. Nevertheless, getting the result of the computation for the obfuscated expression requires indeed to get through all the operators in the considered expression which implies somehow a computational complexity.

Therefore, cryptography can provide us with an example study of what means incompatibility between operators and how it can prevent an easy study of MBA expressions in a unified domain. Indeed, we work on n-bit words considered at the same time as elements of different mathematical structures. For example, standard arithmetic operations are considered in $(\mathbb{Z}/2^n\mathbb{Z}, +, \times)$ while bitwise operations belong to $(\{0,1\}^n, \wedge, \vee, \neg)$ or $(\{0,1\}^n, \wedge, \oplus)$. More details on this incompatibility of operators are given in Section 3.1.

2.2 Expression Simplification

In this section, we detail the issues arising when addressing the subject of expression simplification (in general, in the obfuscation context and in our context of MBA obfuscated expressions), then we provide the existing work regarding arithmetic, boolean and mixed simplification.

2.2.1 The Question of Simplification

Considering the general literature on expression simplification, we can retain two types of simplification:

1. computing a unique representation for equivalent objects (*canonical representation*),

2. finding an equivalent, but simpler form ("simpler" being context-dependent).

The first type of simplification is the most studied in literature since it can prove equivalence of expressions and check for equality to zero. Nevertheless, it has already been noted [Car04, BL82] that a canonical form may not always be considered as the simplest form, depending on the definition of simplicity (whether it is cheaper to store, easier to read, more efficient to compute, related to some high level semantics...). This implies that if both problems are related (as a solution for one could solve the other), they may be different because of the context.

As obfuscation is designed to counter both human and automatic analysis, the definition of what is a *simple* program and/or expression is thus double: for a human, the *readability* would probably be of concern, while for a machine, the questions of performances (in terms of memory or computing time) would probably be of importance. Because obfuscation aims mainly at preventing the *analyst* to get information, we focus our work on the readability—or the understandability—of the program.

In the case of MBA obfuscation, *understanding* can have different meanings depending on the context of the attack, for example:

- identifying distinctive constants or operations of a standard algorithm,
- associating a high level semantics to different parts of the formula,
- extracting the formula or part of it and using it in another context, with different parameters,
- inverting the function containing the formula.

With those diverse goals for the analyst, and as readability is not a trivial notion to define, we do not search for a general and perfect definition of what a simple expression is. On the contrary, our focus is on the simplicity of MBA expressions specifically, and a general metric for expressions simplicity (e.g. the Minimum Description Length from [Car04]) might not describe the inherent difficulty of MBA obfuscated expressions. Nevertheless, we may assess that simplifying an MBA-obfuscated expression is somehow close (if not equivalent) to finding the original expression of the non-obfuscated program in our case. Indeed, since the MBA obfuscation we consider is mostly conducted through rewriting, we aim at returning to a former state. We do not consider exceptional situations: for example, it is very unlikely that by simplifying, we produce a program simpler than the original. The idea of finding the original expression (or at least get close enough to it) is also present in the problematic of decompilation, with which we share this interest.

Finally, it is interesting to note the fact that the definition of a simple expression could also depend on the simplification algorithm chosen. For example, in Chapter 4, we detail two simplification approaches: one using a bit-blasting approach—on the bit level—and the other a rewriting approach—on the word level. When bit-blasting expressions, two main factors bring difficulty to the simplification:

1. when the number of bits increases, as it increases the number of bit expressions to manipulate;

2. when the number of arithmetic operations (mainly addition and multiplication) rises, as it increases the dependencies between the bit expressions.

Thus expressions presenting a low number of bits and arithmetic operators might be considered simpler than other expressions when using bit-blasting. Whereas with the rewriting approach, neither of these aspects influences the difficulty of simplification.

With all these factors in mind, we present in the following sections the existing work about arithmetic, boolean and mixed expressions simplification.

2.2.2 Arithmetic Simplification

We can easily illustrate the two types of simplification of the previous section with examples from the computer algebra field concerning pure arithmetic expressions (namely polynomials), where there exist efforts in both canonical form and other simplifications related to the context. The canonical form of polynomials is the expanded form. Depending on polynomials, this form is not necessarily the most readable: for example, the expanded form on the right of Expression (2.6) can be easily considered as simpler than the original form on the left, whereas the factorized form on the left of Expression (2.7) is more readable than its expanded form.

$$(x-3)^2 - x^2 + 7x - 7 = x + 2 \tag{2.6}$$

$$(1+x)^{100} = 1 + 100x + \cdots + 100x^{99} + x^{100} \tag{2.7}$$

Those examples show the difficulty of defining in full generality the notion of simplicity, even when a canonical form is available. In computer algebra software such as Maple [Map], several strategies are often offered to the users, who have to choose the one adapted to their objective. Nevertheless, there are standard simplification steps suitable to all strategies that can be constantly applied (e.g. $x \times 0 = 0$).

2.2.3 Boolean Simplification

The same issues arise in the field of minimization of boolean functions and simplification of logic circuits [Weg87]. While there exist several normal forms for boolean functions (CNF, DNF, ANF), those are not always relevant in the case of circuit simplification. Indeed, the goals of circuit simplification can be various: reducing the number of gates, the depth of the circuit, the fan-out of the gates, etc. We provide with Figure 2.2 an example of such a circuit simplification.

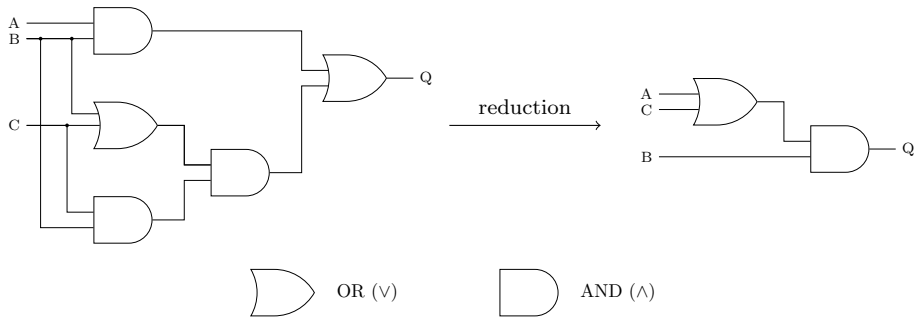


Figure 2.2: An example of circuit reduction from [Kup96].

The formula corresponding to the circuit before reduction is $Q = (A \wedge B) \vee (B \wedge C \wedge (B \vee C))$. The reduced circuit coincides with a Conjunctive Normal Form (CNF), namely $(B \wedge (A \vee C))$. As we can see, another normal form such as Disjunctive Normal Form (DNF), meaning $((A \wedge B) \vee (B \wedge C))$, would not have reduced the number of gates as much as the CNF. In some examples, CNF or DNF would probably not give the simplest circuit, depending on the considered characteristics. In fact, such variations and equivalences in circuit composition are indeed used to provide obfuscation at the circuit level [McD12].

2.2.4 Mixed Simplification

To our knowledge, there exists only one article on simplifying MBA obfuscation [BJLS15], focusing on the constant obfuscation detailed in Section 2.1.2. The authors use three techniques to recover the hidden constant: an SMT-based approach, an algebraic simplification technique and a drill-and-join synthesis method. While their focus is complementary to ours, their solutions do not scale well with our problem of simplifying a non-constant expression. For example, using an SMT solver in our case would require to know the simplified expression, and querying the solver would only validate the equivalence of both obfuscated and simplified expressions. Their algebraic simplification is strongly related to the form of the

obfuscated constant, which is different from obfuscated expressions—as the MBA-obfuscation for expressions does not use permutation polynomials. Furthermore, the program synthesis approach seems too expensive for the general obfuscation case and at least requires further investigation as stated by the authors. In Section 5.3, we also detail another weakness of this constant obfuscation technique.

Regarding tools that could be used for MBA expressions simplification, existing software often do not support both bitwise and arithmetic operators—e.g. Maple, SageMath [S+15]. Those implementing *bit-vector logic* [KS08] (see Section 2.3) allow at least the creation and manipulation of MBA expressions, but those tools are SMT solvers and focus primarily on satisfiability. We develop the topic of bit-vector logic in the next section, and provide a more thorough analysis of MBA simplification with existing tools in Section 3.2.

2.3 Bit-Vector Logic

When working with computer systems, one needs a framework adapted to the machine logic in order to describe elements and operations on those elements. At the machine level, objects are represented using *bit-vectors* (e.g. numbers with the binary numeral system), and any operator available on a processor could be considered to build an expression.

The bit-vector logic [KS08] is adapted to this machine representation: every object is represented by a bit-vector (defined by its size), and traditionally considered operators are: arithmetic $(+, -, \times, /)$, boolean $(\oplus, \wedge, \vee, \neg)$, shifts $(\ll, \gg)$, concatenation $(\circ)$ and extraction ([:]). Our study is limited to arithmetic (without division) and boolean operators, but further work could extend to shifts, concatenation and extraction.

Unfortunately, literature about bit-vector logic is widely focused on proving SAT for a formula [GBD05, WHdM13, BDL98], which is a different problem than ours—proving if an assertion is satisfiable and eventually finding a model does not reach our notion of computing a simpler expression.

Tools commonly implementing the bit-vector logic are SMT solvers, such as Z3 [dMB08] and Boolector [NPB15]. While they are constraint solvers and not simplifiers, they sometimes provide simplifying routines, for example the `simplify` function of Z3, which is rather aiming at the same goal we have but is very limited in the case of MBA—for example, it cannot simplify Expression (2.2) into $x+y$. We provide a more detailed study of the strength and weaknesses of Z3 in Section 3.2.2.

2.4 Term Rewriting

We already mentioned the notion of *rewriting* previously in this thesis, for example to describe MBA obfuscation in Section 2.1.2. Essentially, rewriting systems are composed of a set of objects, and relations on how to transform those objects. In the case of *term* rewriting, those objects are terms (or expressions). In this section, we recall a few important definitions about term rewriting and discuss them regarding our context. We will try to focus on the most basic notions of rewriting here, and recommend [BN99, DJ90, Klo92] to the curious reader.

Let Σ be a set of *function symbols* where each $f \in \Sigma$ comes with a positive integer $n \geq 0$ representing its arity. Function symbols of arity 0 are called *constants*. Let further $\mathcal{X}$ be a set of *variables* such that $\Sigma \cap \mathcal{X} = \emptyset$. Then we can define:

Definition 2 (Set of Σ-terms over $\mathcal{X}$ [BN99])**.** *The set $\mathcal{T}(\Sigma, \mathcal{X})$ of all Σ-terms over X is defined as*

- $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$ *(i.e. every variable is a term),*
- *for all f of arity n and all $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, X)$, we have $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$ (i.e. the application of function symbols to terms yields terms).*

In our case, function symbols will most likely be binary function symbols (e.g. $+, \times, \oplus, \wedge \ldots$) or unary (e.g. $\neg, -$), written in infix form $(x + y) + z$ instead of $+(+(x, y), z)$. For more readability, we use the notation $\mathcal{T}$ to refer to $\mathcal{T}(\Sigma, X)$.

For a term $s \in \mathcal{T}$, we note with $s|_p$ the *subterm* of s at position p, while the term s with its subterm $s|_p$ replaced by a term t is noted $s[t]_p$.

A *substitution* is a mapping from variables to terms $\sigma : \mathcal{X} \to \mathcal{T}$ such that $\sigma(c) = c$ for every constant c, and $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1, \ldots, t_n))$ for every term $f(t_1, \ldots, t_n)$.

The idea around rewriting is to infer rules by orienting equations. For example, if

$$x + y = (x \oplus y) + 2 \times (x \wedge y) \tag{2.8}$$

for all $x, y \in \mathbb{Z}/2^n\mathbb{Z}$, then it is possible to rewrite $x + y$ as $(x \oplus y) + 2 \times (x \wedge y)$ in any given expression. A *rewrite rule* is a pair $(l, r) \in \mathcal{T}^2$ of equivalent terms (i.e. that compute the same result) written $l \to r$. The terms left-hand side (LHS) and right-hand side (RHS) are often used to refer to l and r. For example, Equation 2.8 produces two rewrite rules:

$$x + y \to (x \oplus y) + 2 \times (x \wedge y) \tag{2.9}$$

$$(x \oplus y) + 2 \times (x \wedge y) \to x + y. \tag{2.10}$$

In our case, the relation (2.9) would traditionally be used for MBA obfuscation, while the second could be used for simplification (see Section 4.3). Both l and r may contain variables which refer to arbitrary terms.

A *Term Rewrite System* (TRS) is composed of a set Σ of function symbols and a set $\mathcal{R}$ of rewrite rules over $\mathcal{T}$. Very often, Σ is left implicit and a TRS is identified with its rule set $\mathcal{R}$. Then the action of using a rewrite rule is defined as *term rewriting*.

Definition 3 (Term Rewriting [DJ90])**.** *For a given TRS noted $\mathcal{R}$, a term $s \in \mathcal{T}$* rewrites *to a term $t \in \mathcal{T}$, if*

- $s|_p = \sigma(l)$ *(i.e. $\sigma(l)$ is a subterm of s)*
- *and $t = s[\sigma(r)]_p$ (i.e. t is obtained from s by replacing an occurrence of $\sigma(l)$ by $\sigma(r)$),*

for some rule $l \to r$ in $\mathcal{R}$, position p in s, and substitution σ.

Examples If we have $s = b + (a \wedge b) + (a \wedge b)$, the rewrite rule $x + x \to 2x$ and the substitution $\sigma = \{x \mapsto (a \wedge b)\}$, then:

- $l = x + x$ and $r = 2x$,
- $\sigma(l) = (a \wedge b) + (a \wedge b) = s|_p$ for some position p,
- $\sigma(r) = 2 \times (a \wedge b)$, and thus $s[\sigma(r)]_p = b + 2 \times (a \wedge b)$.

We can therefore rewrite s as $t = b + 2 \times (a \wedge b)$.

For another example, let

- $s = a + (a \oplus (3b + 1)) + 2 \times (a \wedge (3b + 1))$,
- $\mathcal{R} = \{(x \oplus y) + 2 \times (x \wedge y) \to x + y\}$,
- $\sigma = \{x \mapsto a, y \mapsto (3b + 1)\}$.

Then one can rewrite s as $t = a + a + b$. If the previous rule $x + x \to 2x$ was to be added to $\mathcal{R}$, then with the substitution $\sigma = \{x \mapsto a\}$, t could be rewritten as $u = 2a + b$. We show in Section 3.3.1 that finding the right substitution can be a challenge in the obfuscation context, as it does not always exist.

A TRS can have several properties, mainly *termination*, *confluence* and *convergence*.

Definition 4. *A set of rewrite rules is:*

1. ***terminating*** *if after finitely many rules applications, we always reach an expression to which no more rules apply;*

2. ***confluent*** *if when there are different rules to apply from a term x, leading to two different terms y_1 and y_2, we always find a common term z that can be reached from both y_1 and y_2 by successive applications of rewrite rules;*

3. ***convergent*** *if it is both confluent and terminating.*

Figure 2.3 presents a classical illustration of the confluence of a TRS. The relation $\xrightarrow{*}$ represents a certain number of applications of rewrite rules (which can be 0). The solid arrows represent universality ($\forall x, y_1, y_2$) while dashed arrows represent existence ($\exists z$).

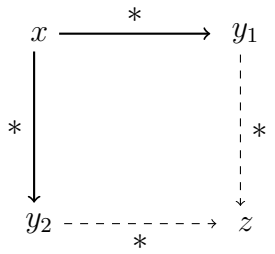


Figure 2.3: Illustration of the confluence property [BN99].

If a TRS is convergent, it means that it always computes canonical forms for its input terms. Those properties can be proven on a specific set of rules, or for some types of rules. For example, we can deduce from [BN99] that if every rule of the set reduces the length of the expression (and the set is finite), then the associated TRS is terminating.

If rules describing the commutativity (e.g. $x + y \to y + x$) and/or associativity of certain operators are taken into account, the termination property will never be achieved. To deal with this difficulty, all basic operations of term rewriting are extended to deal with finite congruence classes of terms. This is called term rewriting *modulo* a finite congruence class (e.g. modulo commutativity or modulo associativity and commutativity).

Very often in the literature, the definition of a rewrite rule $l \to r$ also includes two properties:

1. l is not a variable, and
2. all variables in r also occur in l.

This can be explained by the fact that those properties are almost always necessary to prove termination or confluence for a TRS. In our case, studying the termination or confluence of an obfuscating system is of little interest. This is why we chose a more general definition so that it would also include obfuscation rewriting. Indeed, it is very possible to imagine obfuscating rules that would not respect those properties: for example, one could use $x \to 2x - x$ to increase the size of the expression and frequency of the variable x, or $(x + y) \to (x + y + z - z)$, to add more variables into the expression. In fact, obfuscating rewrite rules are very close to production rules (often called *productions*) used in context-free grammar [HMU06], as their purpose is more to generate than to reduce, and studies on deobfuscation sometimes use the formalism of context-free grammar [RS14].

However, when considering rewrite rules during simplification (see Section 4.3), these two properties are taken into account: this use of rewrite rules is completely in the scope of traditional rewriting, and the notions of termination and convergence are of interest. In Section 4.3.1, we briefly assess the termination of our rewriting system, but the confluence was left for future work.

2.5 DAG Representation

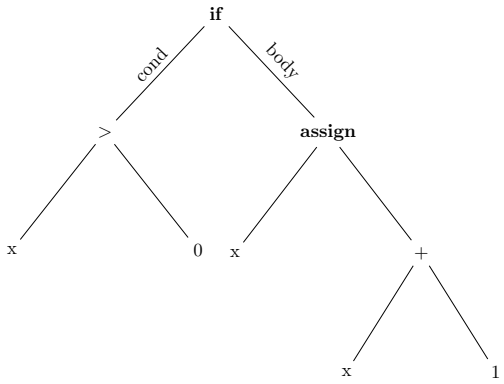
Both in computer algebra [Hec03] and in the field of compilation [ALSU06], trees and graphs are used to represent programs or expressions. One common representation of programs is the *Abstract Syntax Tree* (AST), or *syntax tree*, which is designed to depict the hierarchical syntactic structure of the source program. Thus, each node represents an operator, a function or a structure of the program, while leaves represent operands (namely, variables or constants). Figure 2.4 provides a small pseudo-code example in Figure 2.4a and its corresponding AST in Figure 2.4b.

Syntax trees may also be used to represent expressions. In this case, the nodes just represent classical operators (e.g. $+, -, \oplus, \ldots$). Therefore, one might want to avoid repetition of multiple occurrences of the same subexpression. A *directed acyclic graph* (DAG) for an expression identifies the *common subexpressions* of the expression [ALSU06]. In Figure 2.5, we give an AST representation and a DAG representation of expression $2 \times (x \wedge y) + (x \wedge y)$.

As we use this DAG representation to define complexity metrics for MBA expressions (see Section 3.4), we provide a definition of the DAG representation

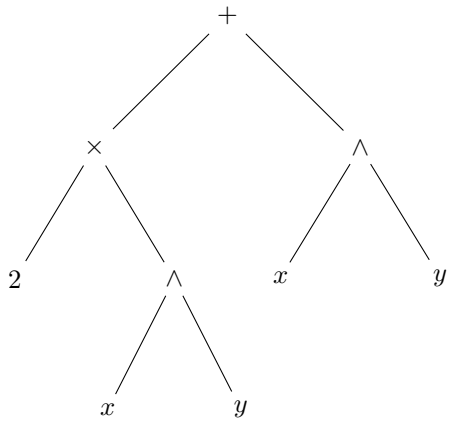
```
if (x > 0)
  x = x + 1
return x
```

(a) Small pseudo-code example.

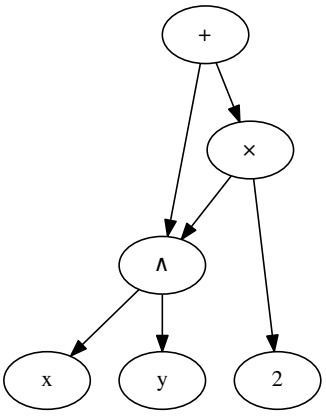


(b) AST representation.

Figure 2.4: An example of the AST representation of source code.



(a) Abstract syntax tree.



(b) Directed acyclic graph.

Figure 2.5: Different representations of $2 \times (x \wedge y) + (x \wedge y)$.

with Definition 5. The interest of the sharing of common expressions is presented in Section 3.4.1.

Definition 5 (DAG representation)**.** *The DAG representation of an MBA expression is an acyclic graph G where:*

- *all leaves represent constant numbers or variables, other nodes represent arithmetic or bitwise operators;*
- *an edge from a node v to a node v' means v' is an operand of v;*
- *there is only one root node;*
- *common expressions are* shared*, which means they only appear once in the graph.*

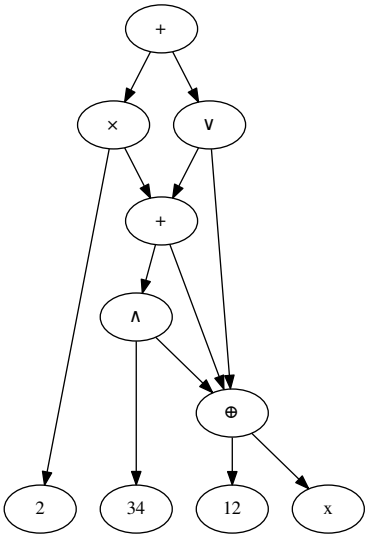
One can note that the DAG representation is equivalent to the compact *term graph* representation [Det99] used in term graph rewriting. It is also worth emphasizing that a list of equalities representing common subexpressions—list that can be represented with an AST—presents in our context the same properties as a DAG representation with the sharing of common subexpressions. For example, we consider the AST of the list of expressions in Figure 2.6a and the DAF in Figure 2.6b as equivalent representations, and use both evenly throughout our work.

2.6 Optimization and Deobfuscation

Program optimization consists in transforming a program to produce a "better" program. Usually better means faster, but other characteristics may be targeted, such as shorter code, or code consuming less power. Compilers often have a machine-independent optimization phase, operating on the intermediate representation of the program. While a more efficient code is not always more readable, optimization sometimes share a common goal with deobfuscation. For example, *dead-code elimination* (removing code not used in the output of a function) or *constant propagation* [ALSU06] (replacing variables by their value if it is a constant) can be easily acknowledged as transformations improving the readability. Some obfuscation techniques may even be non-resistant to the optimization process and deobfuscated with classic optimization passes, or by customizing such passes [GG10, Spa10]. We consider this type of obfuscation to provide little resilience as they can be undone. We present in this section two optimization techniques that could in theory deobfuscate any program, since they are based on the program's behavior and not on its description: *superoptimizers* and *program*

$$a = (x \oplus 12)$$
$$b = (a \wedge 34) + a$$
$$c = 2b + (b \vee a)$$

(a)



(b)

Figure 2.6: List of expressions and DAG representation.

synthesis are worth detailing since they provide at least a partial answer to the question of expression simplification.

2.6.1 Superoptimizers

Superoptimization [Mas87] means finding the optimal code sequence for a single, loop-free assembly sequence of instructions (called the target sequence). Here, the definition of the optimal code sequence is traditionally chosen as the *fastest* code sequence, or also commonly the smallest one.

One of the first approaches to superoptimization was proposed by Massalin [Mas87] and was a *brute-force* approach, enumerating sequences of instructions of increasing length and choosing the lowest cost sequence equivalent to the target sequence. Other approaches later appeared, for example the tool Delani [JNR02] that uses a structure representing all possible equivalent sequences of the target sequence under some equality-preserving transformation rules. The work of Bansal et al. [BA06] uses a training set of programs to build an optimization database in order to gain better performances than [Mas87].

While superoptimization may prove useful on some examples of obfuscated expression, it does not operate in the same context as ours, which leads to two major issues:

- the definition of a better program is based on performances (faster, smaller), which is different from our readability goal;
- the optimized program examples are often small snippets compared to the average size of an obfuscated expression (examples before optimization contain about ten instructions, while assembly version of expression in Figure 2.1 contains more than 35 instructions). Furthermore, those examples are composed of "normal" code that could be naturally encountered in a program, not obfuscated code, which could be a problem for the approach needing a training set of programs.

2.6.2 Program Synthesis

We lately took an interest in program synthesis [Kre98, Gul10], which is the process of automatically discovering an executable program given user intention—this intention being expressed using various forms of constraints such as input-output examples, demonstrations, natural language... Program synthesis has found many practical applications: generating optimal code sequences (superoptimizers can be considered as a specific case of program synthesis), automating repetitive programming tasks, optimizing performance-critical inner loops...

The work of Jha et al. [JGST10] addresses the subject of synthesis for deobfuscation and bit-vector manipulation routines, which is of interest for us since both those cases match our context. From a finite set of input/output (I/O) constraints, they synthesize a satisfying program and check its equivalence with the target program by using distinguishing inputs (i.e. inputs for which the output of the synthesized program is different from the target program's output). If a distinguishing input is found, it is added into the set of I/O constraints and a new program is synthesized.

Despite the lack of definition of what a "simple" program is, the work of Jha et al. would need further study to determine its pertinence for MBA deobfuscation. In the absence of a public implementation of the synthesizer (named BRAHMA), we were not able to test it on our current examples, but we have good confidence that it might prove efficient when the deobfuscated expression contains a small number of operators. The benchmarks given in [JGST10] show that the programs leading to the longest synthesis are the ones containing a mixing of operators, which could be an indication that MBA expressions also add difficulty to this approach. We plan to verify this hypothesis in further work.

Chapter 3

MBA Complexity

In this chapter, we give a few answers about the difficulty of MBA simplification, such difficulty explaining (at least partly) the resilience of the MBA obfuscation technique so far. First, we recall a theoretical study of the incompatibility between arithmetic and boolean operators, and analyze it in the context of MBA expressions. Then, we show how the existing set of simplification tools fails to provide an effective solution for MBA simplification. The third section lists the difficulties that result from the reverse-engineering context, and the final section presents our complexity metrics for MBA expressions.

3.1 Incompatibility of Operators

Bitwise and arithmetic operators do not naturally interact very well, as there are no general rules (e.g. distributivity, associativity...) to cope with the mixing of operators. Though there are some cases where rules analogous to distribution can be used (e.g. Expression (3.1)), the impossibility of generalizing such rules (see Expression (3.2)) supplies additional diversity during the obfuscation process.

$$\forall x, y \in \mathbb{Z}/2^n\mathbb{Z} : \quad 2 \times (x \wedge y) = (2x \wedge 2y) \tag{3.1}$$

$$\exists x, y \in \mathbb{Z}/2^n\mathbb{Z} : \quad 3 \times (x \wedge y) \neq (3x \wedge 3y) \tag{3.2}$$

In this case, the equality of Expression (3.1) is due to the fact that the multiplication by 2^n can actually be seen as a left shift of n bits, which is a bitwise operator. We give some insight about the incompatibility of arithmetic and boolean operators in this section.

IDEA [LM91] is a well-known block cipher of the 90s, famous for its combined use of integer arithmetic and bitwise operators. One of the major characteristics of IDEA at the time of its proposal was its lack of S-boxes. Instead, it relied on

a construction using three key components, carefully interleaved to prevent any easy manipulation of the resulting expressions:

- the multiplication $\odot$ in $(\mathbb{Z}/(2^{16}+1)\mathbb{Z})^*$ (noted $\mathbb{Z}^*_{2^{16}+1}$ for more readability)
- the addition $\boxplus$ in $\mathbb{Z}/2^{16}\mathbb{Z}$,
- the bitwise XOR $\oplus$ in $\mathrm{GF}(2)^{16}$.

Even though the multiplication in $\mathbb{Z}^*_{2^{16}+1}$ is not part of the operators we consider in an MBA expression, IDEA provides us with a detailed example of incompatibility between operators which hopefully helps one understand the usefulness of MBA for obfuscation. The incompatibility study of the operators in IDEA was at the basis of the argument on the *confusion* property a block cipher must fulfill.

There are four main reasons why the three operations are incompatible [LM91]:

- No pair of the three operations satisfies a distributive law.
- No pair of the three operations satisfies a generalized associative law.
- When considering the quasi-groups, $(\mathbb{Z}^*_{2^{16}+1}, \odot)$ and $(\mathrm{GF}(2)^{16}, \oplus)$ are not isotopic, neither are $(\mathbb{Z}/2^{16}\mathbb{Z}, \boxplus)$ and $(\mathrm{GF}(2)^{16}, \oplus)$. The isotopism between $(\mathbb{Z}^*_{2^{16}+1}, \odot)$ and $(\mathbb{Z}/2^{16}\mathbb{Z}, \boxplus)$ is essentially the discrete logarithm, which is not a simple and straightforward bijection.
- It is possible to analyze $\boxplus$ and $\odot$ as acting on the same set. However it means either analyzing a non-polynomial function on $\mathbb{Z}/2^{16}\mathbb{Z}$ to represent $\odot$ or analyzing a high degree polynomial on $\mathbb{Z}^*_{2^{16}+1}$ to represent $\boxplus$.

The idea behind the notion of *confusion* is to make any description of the relation between the ciphertext, the plaintext and the key so involved and complex that it is useless for the attacker. It is a clear link with obfuscation concerns, although the starting point for each context is different: in cryptography, the intention is to gain intrinsic computational complexity, while in obfuscation, one seeks a complexity linked to the form of writing that would prevent simplification.

3.2 Existing Tools and MBA Simplification

We exposed quickly in Section 2.2.4 how common tools for expression simplification are not sufficient to simplify MBA expressions. In this section we choose examples of computer algebra softwares, SMT solvers and optimization processes, and detail strengths and weaknesses of these tools regarding MBA simplification.

3.2.1 Computer Algebra Software

We stated in Section 2.2.4 that common computer algebra programs—also called symbolic computation programs—do not provide the possibility to manipulate MBA expressions: for example, the computer algebra software Maple provides bitwise computation on constants only[1], and so does Wolfram Mathematica[2]. In the same way, SageMath provides simplification for polynomials, but does not support symbolic bitwise computation[3]. SageMath has a `Bitset`[4] class which is equivalent to the bit-vector representation, but it loses the semantics of the word-level expression. We give in Figure 3.1 a few examples of how to use Sage's function `simplify` on arithmetic expressions, and how bitwise symbolic expressions are not supported. One can observe that the `simplify` function fails to reduce Expression (2.6), and the `expand` function needs to be called explicitly. This illustrates the difficulty of having a general simplification routine.

```
sage: simplify(3*(x + 12) - 12 + 5*x)
8*x + 24
sage: simplify((x - 3)^2 - x^2 + 7*x - 7)
(x - 3)^2 - x^2 + 7*x - 7
sage: expand((x - 3)^2 - x^2 + 7*x - 7)
x + 2
sage: x & 98
TypeError: unsupported operand type(s) for &:
'sage.symbolic.expression.Expression' and
'sage.symbolic.expression.Expression'
```

Figure 3.1: Using Sage for simplifying arithmetic and bitwise expressions.

3.2.2 SMT Solvers

We presented in Section 2.3 the bit-vector logic, which is a good theoretical framework to consider MBA expressions, and the tools implementing this logic, which are SMT solvers. We detail here how different SMT solvers (namely Z3 [dMB08], Boolector [NPB15] and Yices [Dut14]) deal with MBA simplification.

Z3[5] provides a `simplify` routine that can achieve polynomial simplification as

[1] `http://www.maplesoft.com/support/help/Maple/view.aspx?path=Bits`

[2] `http://reference.wolfram.com/language/guide/BitwiseOperations.html`

[3] `http://doc.sagemath.org/html/en/reference/logic/sage/logic/boolformula.html`

[4] `http://doc.sagemath.org/html/en/reference/data_structures/sage/data_structures/bitset.html`

[5] All tests using Z3 were performed with version 4.2.2.

well as some bitwise simplification. We illustrate in Figure 3.2 a few examples of what `simplify` succeeds at simplifying. Note that Z3 does not support the power operator (`**` in Python) on bit-vector variables, which could explain the lack of simplification for polynomials of degree greater than one. In this figure, one can also see how to prove the equivalence between an MBA expression and its simplified version. This can be used either to help the designer of new MBA rewrite rules, or the reverser trying to simplify the obfuscated expression—however, this requires to make a "guess" about the simplified expression. This is the rough principle of superoptimization, detailed in Section 2.6.1.

```
In[1]: import z3

In[2]: x,y = z3.BitVecs('x y', 8)
In[3]: z3.simplify(3*(x + 12) - 12 + 5*x)
Out[3]: 24 + 8*x
In[4] z3.simplify((x - 3)*(x - 3) - x*x + 7*x - 7)
Out[4]: 249 + (253 + x)*(253 + x) + 255*x*x + 7*x
In[5] z3.simplify(x & 74 & 21)
Out[5]: 0
In[6] z3.simplify(x & 74 | 95)
Out[6]: 95
In[7] z3.prove(x + y == (x ^ y) + 2*(x & y))
proved
```

Figure 3.2: Using Z3's `simplify` for arithmetic or bitwise simplification.

While MBA expressions can be manipulated using SMT solvers that implement bit-vector logic, they prove little efficiency for our context of MBA expression simplification. Even in the case of a simple bitwise expression equivalent to a constant, Z3's simplification is incomplete, as illustrated in Figure 3.3.

Nevertheless, two steps used in Z3's `simplify` function—and also used during proofs to help check satisfiability—share a common interest with us. The first step is rewriting (see Section 2.4) applied to perform basic simplifications (e.g. $x + 0 = x$) and presents features similar to the simplification solution proposed in Section 4.3. This rewriting step is often part of what is called the *pre-processing* in SMT solvers, as it is used mostly before the proof of satisfiability for performance reasons. The second step is called *bit-blasting* (or *flattening*) and consists in writing explicitly every boolean formula representing a bit of the expression; it is mainly the approach we used for the simplification solution detailed in Section 4.2. Bit-blasting is often used in last resort in SMT solvers, as it increases the number of objects to manipulate (n boolean expressions instead of one word-level expression).

```
In[1]: import z3

In[2]: x = z3.BitVec('x', 8)
In[3]: e = ((x & 87) | 43) ^ ((x & 94) | 138)
In[4]: z3.prove(e == 161)
proved
In[5]: z3.simplify(e)
Out[5]:
Concat(0,
       Extract(6, 6, x),
       1,
       Extract(4, 4, x),
       1,
       Extract(2, 2, x),
       3) ^
Concat(1,
       Extract(6, 6, x),
       0,
       Extract(4, 4, x),
       1,
       Extract(2, 2, x),
       2)
```

Figure 3.3: An example of insufficient simplification by Z3's `simplify`.

When examining the API documentation of Boolector and Yices, one can notice that no function similar to Z3's `simplify` is available, only functions designed to create assertions, check for their satisfiability, and get a model if satisfiable. Adding this to the fact that the pre-processing steps of solvers are rarely documented, this means the internal operating of those solvers is not available to the user. Therefore, Boolector and Yices are not of great relevance in the context of MBA simplification, except for proving equivalence of expressions.

3.2.3 Optimization

We exposed in Section 2.6 that optimization passes may help to deobfuscate some obfuscation techniques. In our case, modern compilers can indeed simplify a few examples of MBA expressions, but fail at simplifying them as soon as several MBA rewrite rules are applied. We used the LLVM compiler, with its front end for C/C++/Objective C languages called `clang`, to perform tests in order to evaluate the efficiency of optimization on MBA expressions. We used the version 3.7 of

`clang`, and studied the optimized code produced in LLVM IR. In Figure 3.4, we show a source code containing an MBA expression in the function `compute_MBA` equivalent to an addition; the rewrite rule and substitution used to obfuscate the function is

$$x + y \to (x \vee y) + (x \wedge y)$$
$$\sigma = \{x \mapsto x, y \mapsto y\}$$

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

int compute_MBA(int x, int y) {
  return (x | y) + (x & y);
}

int main(int argc, char* argv[]) {
  if(argc != 3)
    return(1);
  int x = atoi(argv[1]);
  int y = atoi(argv[2]);
  printf("%ul\n", compute_MBA(x,y));
  return(0);
}
```

Figure 3.4: MBA expression in C source code.

In Figure 3.5, we present the LLVM IR code corresponding to the translation of the function `compute_MBA` (with relevant operators colored), Figure 3.5a being the LLVM IR without optimizations and Figure 3.5b with optimizations (we used the `-O3` option of `clang`). One can note that LLVM's optimizations successfully simplified the MBA expression to an addition.

Nevertheless, when two MBA rewrite rules have been applied, LLVM cannot simplify the expression. We illustrate this in the Figure 3.6, where two rewrite rules have been applied in the source code, both those rules independently being simplified by LLVM optimization if applied alone (both substitutions are trivially $\{x \mapsto x, y \mapsto y\}$):

$$x + y \to (x \vee y) + (x \wedge y)$$
$$x \wedge y \to (x \vee y) - (x \oplus y).$$

The composition of these two rules can be expressed as

$$x + y \to 2 \times (x \vee y) - (x \oplus y),$$

```
; Function Attrs:
;     nounwind uwtable
define i32 @compute_MBA(i32 %x,
                        i32 %y) #0 {
entry:
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  store i32 %y, i32* %y.addr, align 4
  %0 = load i32, i32* %x.addr, align 4
  %1 = load i32, i32* %y.addr, align 4
  %or = or i32 %0, %1
  %2 = load i32, i32* %x.addr, align 4
  %3 = load i32, i32* %y.addr, align 4
  %and = and i32 %2, %3
  %add = add nsw i32 %or, %and
  ret i32 %add
}
```

(a) LLVM IR without optimization.

```
; Function Attrs:
;     nounwind readnone uwtable
define i32 @compute_MBA(i32 %x,
                        i32 %y) #0 {
entry:
  %add = add i32 %y, %x
  ret i32 %add
}
```

(b) LLVM IR with optimization.

Figure 3.5: LLVM IR of `compute_MBA` with and without optimization.

```
int compute_MBA2(int x, int y) {
  return 2*(x | y) - (x ^ y);
}
```

(a) `compute_MBA2` with two MBA rewritings.

```
; Function Attrs: nounwind readnone uwtable
define i32 @compute_MBA2(i32 %x,
                         i32 %y) #0 {
entry:
  %or = or i32 %y, %x
  %mul = shl nsw i32 %or, 1
  %xor = xor i32 %y, %x
  %sub = sub nsw i32 %mul, %xor
  ret i32 %sub
}
```

(b) Optimized LLVM IR.

Figure 3.6: LLVM optimization failing to simplify an MBA expression.

which implementation can be found in Figure 3.6a, in the source code of `compute_MBA`. Figure 3.6b shows that LLVM optimization cannot simplify the consecutive application of two rewrite rules. On the other hand, it is interesting to notice that LLVM's optimization passes can simplify the expression of Figure 3.3 to the constant 161.

Given that the optimization process of LLVM is composed of many transformation passes[6] and is therefore quite complex, it is not trivial to explain those results. By examining the list of LLVM's optimization passes, one can observe that the `InstCombine`[7] is designed to "Combine instructions to form fewer, simple instructions. [...] This pass is where algebraic simplification happens." It is then

[6] `http://llvm.org/docs/Passes.html`

[7] `http://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions`

very possible that this transformation pass provides at least some simplification techniques in our scope of research. Unfortunately, `InstCombine` consists of more than 23000 lines of code; furthermore, a quick research for MBA rewriting rules gives no result—while rules such as $(x \wedge y) \oplus (x \vee y) \rightarrow x \oplus y$ are indeed present in the code. This makes the optimization passes a powerful tool for expression simplification, but quite difficult to adapt to our context of MBA given its complexity and size.

3.3 Reverse Engineering Context

The simplification of MBA expressions commonly arises in the process of reverse engineering obfuscated programs. Consequently, the original expressions of the program have probably been through other transformations in addition to the obfuscation process, mainly the optimization (applied on the intermediate representation) and the translation from intermediate representation to assembly language. Figure 3.7 illustrates the main transformations being applied to change a program's representation back and forth between source code, IR and assembly language.

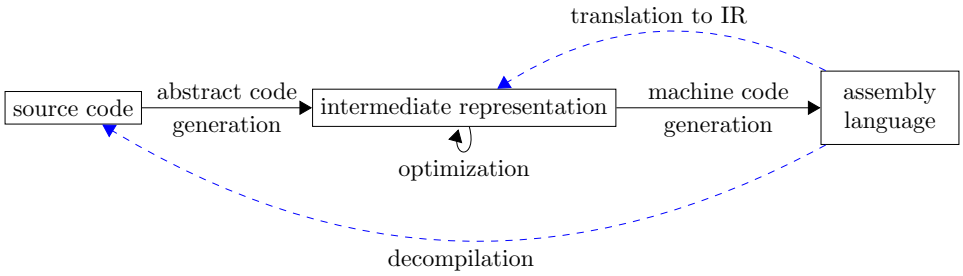


Figure 3.7: Transformation of a program's representation.

The black arrows represent transformations that typically occur in the process of compilation, while the blue dashed arrows represent transformations that could be used by a reverse engineer. As obfuscation is commonly applied on the source code or on the IR, various program transformations are applied on the obfuscated code during compilation. In particular, optimization passes and machine code generation can both induce big changes in the obfuscated code. This means that, in the context of MBA obfuscation, knowing only the obfuscation steps might not be enough to fully understand how the resulting expression was obtained—if the analyst needs such understanding.

We detail in this section how optimization and generation of assembly language can add difficulty to the simplification of MBA-obfuscated expressions.

3.3.1 Impact of Optimization

We saw in Sections 2.6 and 3.2.3 how the optimization process could help deobfuscate programs and even simplify some expressions. However, optimization can also help design obfuscation techniques. Indeed, optimization seeks for an efficient program, often at the expense of the readability of the code. For example, *loop unrolling* and *function inlining* are both standard obfuscation and optimization techniques. While working on MBA-obfuscated expressions, optimization passes may often change the expressions in ways that could be understood by an analyst as steps of the obfuscation technique.

We provide in this section an example of an MBA-obfuscated expression of $(x \vee 2)$, obtained by first applying on $(x \vee 2)$ the identity defined as the composition of function f and f^{-1} on 32 bits:

$$\begin{aligned} f &: x \mapsto 728040545x + 198791817 \\ f^{-1} &: x \mapsto 264282017x - 1538260777, \end{aligned}$$

which produces the intermediate state of obfuscation

$$264282017 \times (728040545(x \vee 2) + 198791817) - 1538260777.$$

And then by using the following rewrite rule and substitution

$$\begin{aligned} &xy \to (x \wedge y) \times (x \vee y) + (x \wedge (\neg y)) \times (\neg x \wedge y) \\ &\sigma = \{x \mapsto (x \vee 2), y \mapsto 728040545\}, \end{aligned}$$

which causes the final MBA-obfuscated expression computing $(x \vee 2)$:

$$\begin{aligned} \texttt{mul1} &= ((x \vee 2) \wedge 728040545) \times ((x \vee 2) \vee 728040545) \\ \texttt{mul2} &= ((x \vee 2) \wedge (\neg 728040545)) \times (\neg(x \vee 2) \wedge 728040545) \\ \texttt{result} &= ((\texttt{mul1} + \texttt{mul2}) + 198791817) \times 264282017 - 1538260777 \end{aligned}$$

The implementation of this MBA-obfuscated expression can be found in Figure 3.8.

Optimizing this function with `clang` and the option `-O3` yields the LLVM IR presented in Figure 3.9. We illustrate the optimization process of the obfuscated expression in the following formulas:

$$\begin{aligned} \texttt{mul1} &= ((x \vee 2) \wedge 728040545) \times ((x \vee 2) \vee 728040545) \\ &= \underbrace{\underbrace{(x \wedge 728040545)}_{\%\texttt{and}} \times \underbrace{(x \vee 728040547)}_{\%\texttt{or1}}}_{\%\texttt{mul}} \end{aligned}$$

```
int compute_MBA3(int x)
{
  int e = (x | 2);

  int mul1 = ((e & 728040545)*(e | 728040545));
  int mul2 = (e & (~728040545))*(~e & 728040545);
  int mul = (mul1 + mul2) + 198791817;
  return mul*264282017 - 1538260777;
}
```

Figure 3.8: MBA-obfuscated source code.

$$
\begin{aligned}
\texttt{mul2} &= ((x \vee 2) \wedge (\neg 728040545)) \times (\neg(x \vee 2) \wedge 728040545) \\
&= ((x \wedge (\neg 728040547)) \vee 2) \times (((x \vee 2) \oplus (-1)) \wedge 728040545) \\
&= ((x \wedge (-728040548)) \vee 2) \times (((x \vee 2) \wedge 728040545) \oplus ((-1) \wedge 728040545)) \\
&= \underbrace{\underbrace{((x \wedge (-728040548)) \vee 2)}_{\%\texttt{and2}} \times \underbrace{((x \wedge 728040545) \oplus 728040545)}_{\%\texttt{and3}}}_{\%\texttt{mul4}} \\
\texttt{result} &= ((\texttt{mul1} + \texttt{mul2}) + 198791817) \times 264282017 - 1538260777 \\
&= \underbrace{264282017 \times \underbrace{(\texttt{mul1} + \texttt{mul2})}_{\%\texttt{add}}.}_{\%0}
\end{aligned}
$$

One can note that the resulting expression is quite different from the obfuscated one of `compute_MBA3`: while the original `mul1` and `mul2` contained 728040545 as their only constant, the optimized version includes new constants such as 728040547 and -728040548—but 198791817 and -1538260777 have disappeared from `result`. The operators are also altered, as the optimized expression introduces a new operator $\oplus$. A simplifying approach based on the identification of the different rewrite rules used for obfuscation would clearly be impeded by the transformations applied during optimization. It would require to analyze optimization in a similar way as obfuscation. In the Section 4.3, we present such a simplification algorithm and detail how to cope with this type of difficulties.

3.3.2 Analyzing Assembly

Reverse engineering an obfuscated program very often requires to analyze the assembly from an executable. This means that in addition to MBA obfuscation and optimization, expressions also went through the translation from the compiler's

```
; Function Attrs: nounwind readnone uwtable
define i32 @compute_MBA3(i32 %x) #0 {
entry:
  %and = and i32 %x, 728040545
  %or1 = or i32 %x, 728040547
  %mul = mul nsw i32 %and, %or1
  %or = and i32 %x, -728040548
  %and2 = or i32 %or, 2
  %and3 = xor i32 %and, 728040545
  %mul4 = mul nsw i32 %and2, %and3
  %add = add nsw i32 %mul4, %mul
  %0 = mul i32 %add, 264282017
  ret i32 %0
}
```

Figure 3.9: Optimized LLVM IR for `compute_MBA3`.

intermediate representation to a target assembly language. While it is possible to analyze and understand the assembly directly, it is a low-level programming language and is often considered less readable than high-level languages.

To gain a more readable representation from an assembly code, it is possible to either *decompile* it to a source code, or translate it back to an intermediate representation—both techniques are illustrated in Figure 3.7. As the decompilation aims at producing a sound source code, it is often far more complicated than the translation to IR, we thus focus our work on this second technique.

Traditionally, a reverse engineering framework is used to symbolically execute parts of the program (see Section 1.3.2) and obtain a high-level description of the instructions from the IR of the framework. There also exist work on the translation of machine code into LLVM IR [CC11]. In this section, we compare the high-level output of three of the most common frameworks: Miasm[8] [Des12], Triton[9] [SS15] and Medusa[10] [Szk]. Each tool influences the form of the resulting expression by its choice of semantics, representation and potentially its own simplification passes. We tested each framework on the same binary, obtained by compiling the function `compute_MBA3` of Figure 3.8.

In Figure 3.10, we present the high-level output of Triton. The `SymVar_0` variable stands for the input of the function. One can see that in order to display the size of the variables, Triton uses bitwise AND masks with $2^n - 1$ (in the

[8]Used version is of commit `2b93bc6682f3a08a5eccccefa135535708434f9e`.

[9]Used version is 0.4.

[10]Used version is of commit `db15edd5eb000c38cdaa52eacc7710a0cce8d932`.

```
ref_0 = SymVar_0
ref_1 = (ref_0 & 0xFFFFFFFF)
ref_3 = ((ref_1 & 0xFFFFFFFF) & 0x2B650461)
ref_10 = (ref_0 & 0xFFFFFFFF)
ref_12 = ((ref_10 & 0xFFFFFFFF) | 0x2B650463)
ref_19 = (((sx(0x20, (ref_12 & 0xFFFFFFFF))
          * sx(0x20, (ref_3 & 0xFFFFFFFF)))
          & 0xFFFFFFFFFFFFFFFF) & 0xFFFFFFFF)
ref_23 = ((ref_0 & 0xFFFFFFFF) & 0xD49AFB9C)
ref_30 = ((ref_23 & 0xFFFFFFFF) | 0x2)
ref_37 = ((ref_3 & 0xFFFFFFFF) ^ 0x2B650461)
ref_44 = (((sx(0x20, (ref_37 & 0xFFFFFFFF))
          * sx(0x20, (ref_30 & 0xFFFFFFFF)))
          & 0xFFFFFFFFFFFFFFFF) & 0xFFFFFFFF)
ref_48 = (((ref_44 & 0xFFFFFFFF) + (ref_19 & 0xFFFFFFFF))
          & 0xFFFFFFFF)
ref_56 = (((sx(0x20, (ref_48 & 0xFFFFFFFF))
          * sx(0x20, 0xFC09FA1)) & 0xFFFFFFFFFFFFFFFF)
          & 0xFFFFFFFF)
```

Figure 3.10: Output of translation to IR with Triton.

example, n equals 32 or 64), which quite reduces the readability of the expression.

In addition to those masks, the presence of functions relative to the semantics of some instructions, here for example `sx` for sign-extension of multiplications, also induces more work for the analyst trying to simplify MBA expressions. Miasm produces a very similar output, with a different way of dealing with sign extensions—Miasm uses `if` conditions to consider the sign of the variables. We display the high-level output of Miasm for `compute_MBA3` in Figure 3.11. For clarity purposes and as they do not affect the final result, we cleaned the output of expressions concerning flags registers (`cf, zf, ...`). In this output, the input variable is in the `EDI` register.

In Figure 3.12, we give the output of Medusa when symbolically executing the same function. We indicate the input variable with `x`. Medusa makes the choice of giving explicitly the size of every component, with indications of the type `bv32(), bv64()`. One can also note the presence of the `sign_extend` and `bcast` functions. Adding this to the fact that the result of the function is given as one rather big expression, it is quite easy to imagine that the analyst would have to process this kind of output before any simplification algorithm.

Both optimization and translation from assembly add complexity to the simplification of MBA-obfuscated expressions in a reverse engineering context. The evaluation of our simplification algorithm in Section 5.2.2 is done in a "clean" mathematical context (MBA-obfuscated expressions are directly considered on the

```
EAX = EDI
EAX = ((EAX & 0x2B650461) & 0xffffffff)
ECX = EDI
ECX = ((ECX | 0x2B650463) & 0xffffffff)
ECX = ((((((EAX & 0xffffffff) << 0) | (((0xFFFFFFFF
      if (((EAX >> 31) & 0x1)) else 0x0) & 0xffffffff) << 32))
      * (((ECX & 0xffffffff) << 0) | (((0xFFFFFFFF
      if (((ECX >> 31) & 0x1)) else 0x0) & 0xffffffff) << 32)))
      & 0xffffffffffffffff) & 0xffffffff)
EDI = ((EDI & 0xD49AFB9C) & 0xffffffff)
EDI = ((EDI | 0x2) & 0xffffffff)
EAX = ((EAX ^ 0x2B650461) & 0xffffffff)
EAX = ((((((EAX & 0xffffffff) << 0) | (((0xFFFFFFFF
      if (((EAX >> 31) & 0x1)) else 0x0) & 0xffffffff) << 32))
      * (((EDI & 0xffffffff) << 0) | (((0xFFFFFFFF
      if (((EDI >> 31) & 0x1)) else 0x0) & 0xffffffff) << 32)))
      & 0xffffffffffffffff) & 0xffffffff)
EAX = ((EAX + ECX) & 0xffffffff)
EAX = ((EAX * 0xFC09FA1) & 0xffffffff)
```

Figure 3.11: Output of translation to IR with Miasm.

```
eax = bcast(
    (sign_extend(
        (bcast(
            (sign_extend(((x & bv32(0x2B650461)) ^ bv32(0x2B650461)),
                         bv64(0x00000040))
             * sign_extend(((x & bv32(0xD49AFB9C)) | bv32(0x0002)),
                           bv64(0x00000040))),
            bv32(0x0020))
         + bcast(
             (sign_extend((x | bv32(0x2B650463)), bv64(0x00000040))
              * sign_extend((x & bv32(0x2B650461)), bv64(0x00000040))),
             bv32(0x0020))),
        bv64(0x00000040)) * bv64(0x0FC09FA1)),
    bv32(0x0020))
```

Figure 3.12: High-level output of symbolic execution with Medusa.

source level, in our case in Python). Nevertheless, we detail the simplification of a compiled MBA-obfuscated expression in Section 4.1.

3.4 Complexity Metrics

Despite the lack of tools and theoretical ground around MBA expressions, we designed three metrics intended to characterize their complexity. We stated in Section 2.2.1 that the definition of *simplicity* (or complexity, since the definitions are clearly linked) was also dependent on the simplification algorithm considered. As we focused our work more on the rewriting approach (see Section 4.3), our complexity measures are more related to rewriting concepts. We argue that decreasing these metrics improves the simplicity of MBA expressions in a general way, both for human understanding and automatic analysis.

We stress the fact that those metrics are not meant to characterize the resilience of the MBA obfuscation technique as presented in Section 2.1.2, but the complexity of an MBA expression in general. Nevertheless, the complexity of the MBA expressions generated by an obfuscation technique can indeed be a factor used to evaluate the resilience of said technique, but we will detail this topic in Chapter 5.

To define our metrics, we use the DAG representation detailed in Section 2.5.

3.4.1 Number of Nodes

We define the *size* of an expression as its number of nodes: operators, variables and constants. The DAG representation (see Section 2.5) and its sharing property present a great advantage, as each occurrence of a subexpression is only taken into account once when computing the size of an expression. For example, expression of Figure 2.1 has 72 nodes if the sharing is used, and 797 nodes if each occurrence of every subexpression is repeated. We consider that every occurrence can be simplified in the same way, which is true because the rewrite rules traditionally used are true whatever the value of the variables. This means that simplifying an occurrence of a subexpression is equivalent to simplifying all its occurrences, and thus the sharing is both sound and interesting for us. With conditional rewrite rules, it would be possible to have occurrences with different simplification, but we do not consider that kind of obfuscation in our context.

Decreasing the number of nodes contributes to reducing the expression size, meaning it will be easier to apprehend and manipulate. It may also be useful to reduce the number of variables for any brute-force approach (by using Z3 to prove the equivalence of expressions for example), as it decreases the size of the input set.

3.4.2 MBA Alternation

When exposing the subject of expression simplification in Section 2.2.1, we stressed the fact that we were studying the complexity of *mixed* expressions. A "complicated" arithmetic expression that can be reduced by a standard computer algebra software is not relevant in our context. With the *MBA alternation*, we intend to design a metric to help quantify the "MBA aspect" of an expression. For example, a purely arithmetic or a purely boolean expression has a null MBA alternation. Likewise, a computer algebra software applying only arithmetic simplifications (like expansion) on an MBA-obfuscated expression should not greatly decrease the MBA alternation metric of a robust MBA obfuscation.

To define the MBA alternation of an expression, we first need to define the *type* of an operator *op*. The type is arithmetic if $op \in \{+, -, \times\}$ and boolean (or bitwise) if $op \in \{\wedge, \vee, \oplus, \neg\}$. The MBA alternation is simply the number of edges linking two nodes that represent operators of different types (nodes representing variables and constants do not have a type, and thus do not affect this metric).

Definition 6 (MBA alternation)**.** *For a graph $G = (V, E)$ with V the set of vertices and E the set of edges, the* MBA alternation $alt_{MBA}(G)$ *is:*

$$alt_{MBA}(G) = |\{(v_1, v_2) \text{ such that } type(v_1) \neq type(v_2)\}|,$$

where $(v_1, v_2) \in E$ represents the edge linking the two vertices $v_1, v_2 \in V$.

These edges represent difficult points in the simplification of MBA expressions. Indeed, subtrees containing edges of the same type—thus representing purely boolean or arithmetic expressions—are likely to be simplified with classical simplification technique, with still the issue of finding the adapted strategy (e.g. expansion or factorisation for polynomials).

One may note that some bitwise operators (e.g. bitwise not $\neg$, left shift $\ll$) can be rewritten as arithmetic expressions quite easily: for example, $\neg x = -x - 1$ and $x \ll n = x \times 2^n$, while there exists no simple equivalence for other bitwise operators ($\oplus, \wedge \ldots$). One may use such rewritings in the simplification process, to reduce MBA alternation for example.

3.4.3 Average Bit-Vector Size

For this metric, we add to the DAG representation a property we call the *bit-vector size*—this bit-vector size was indeed present in the outputs of Triton, Miasm and Medusa in the Section 3.3.2, either with binary masks or explicitly with functions.

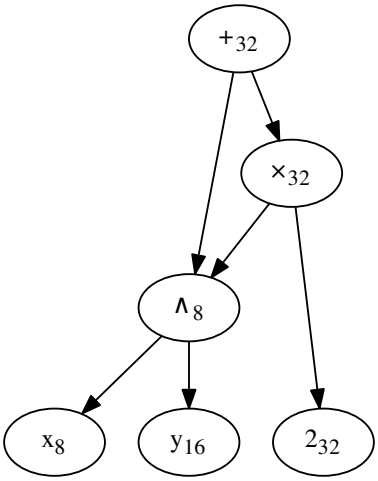


Figure 3.13: DAG for $2 \times (x \wedge y) + (x \wedge y)$ with bit-vector size in subscript.

Definition 7 (Bit-vector size)**.** *For a node v, the bit-vector size* $\text{bvsize}(v)$ *is:*

- *If v is a leaf node, the bit size of the variable or constant it represents. This size can be deduced from the context (e.g. size of the input of a function), or by additional indications (e.g. binary masks). The size of a constant may also be inferred from the actual number of bits of that constant (possibly rounded to the next power of two).*
- *If v represents an operator, the bit size of the output of the operation. This depends on the nature of the operator:*
  - *if v represents a binary operator in $\{+, -, \times, \oplus, \vee\}$ with v_1, v_2 as operands, then* $\text{bvsize}(v) = \max(\text{bvsize}(v_1), \text{bvsize}(v_2))$
  - *if v represents a boolean AND with operands v_1, v_2, then* $\text{bvsize}(v) = \min(\text{bvsize}(v_1), \text{bvsize}(v_2))$
  - *if v represents a unary operator in $\{\neg, -\}$, then* $\text{bvsize}(v) = \text{bvsize}(v_1)$ *for v_1 its operand.*

The DAG example of Figure 2.5b could then be represented as in Figure 3.13, assuming that the variable x is on 8 bits, the variable y on 16 bits, and the constant 2 on 32 bits (this is just an illustration of the definition, and very unlikely to happen in real-life settings).

This definition of the bit-vector size just accounts for a "default" size that would be inferred from the analyzed program. It is possible to later apply transformations to reduce the bit-vector size of specific nodes in some cases. The most common reduction occurs when an operator `AND` ($\wedge$) has operands of different bit-vector size, say op_1 and op_2, with $\text{bvsize}(op_1) < \text{bvsize}(op_2)$; the bit-vector size of the AND operation is then $\text{bvsize}(op_1)$. If op_2 only contains operators in $\{+, -, \times, \wedge, \vee, \oplus, \neg\}$, then we can reduce the bit-vector size of all terms in op_2 to $\text{bvsize}(op_1)$. The restriction concerning operators in op_2 guarantees that there will be no dependencies on the most significant bits in the computation of the least significant bits, thus the transformation is sound. We produce an example of such a reduction in Figure 3.14: the operation `AND` being on 8 bits (which is the bit-vector size of z), it is possible to reduce the expression $x + 259$ on 8 bits only, becoming $x + 3$. We use this reduction to remove the extra obfuscation of bit-vector size extension, presented in Section 4.1.2 during our analysis of a real-life example of MBA-obfuscated expression.

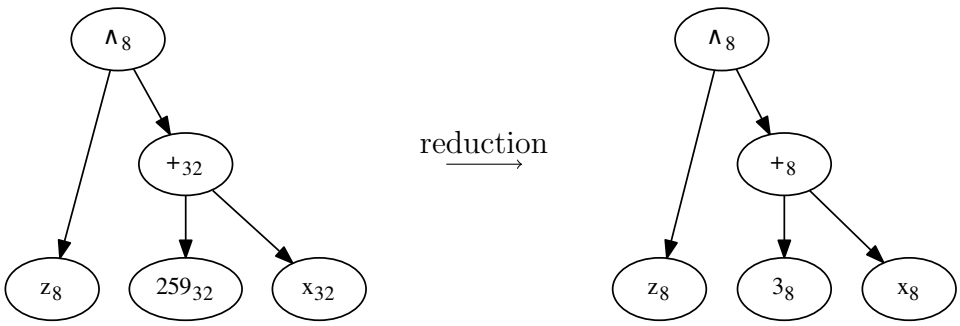


Figure 3.14: An example of the reduction of bit-vector size on $(x + 3) \wedge z$.

When simplifying an expression, one may want a global metric which would decrease when the bit-vector size of subtrees in reduced. Considering only the bit-vector size of the root node would cause us to miss any simplification of a subgraph of our term graph. Therefore, we use an average bit-vector size (on all the nodes: operators, variables and constants); while it does not hold meaning in itself, it accounts for both local and global reductions of the bit-vector size. For now, the examples we analyzed did not require the use of this metric; as seen in Section 4.1.2, the reduction of the bit-vector size is global and all the nodes of the resulting DAG are on 8 bits. If more complicated examples arise, we could design a set of metrics, which would be less global and keep more information than an

average bit-vector size: for example, a list of the subtrees of same bit-vector size and their number of nodes.

Decreasing the bit-vector size of certain nodes might allow an easier recognition of the rewrite rules used for obfuscation. It is also very interesting for simplification approaches that use *bit-blasting*, as the one described in Section 4.2, because their complexity relies mainly on the number of bits of the obfuscated expression.

Chapter 4

Analysis of the MBA Obfuscation Technique

In this chapter, we study the MBA obfuscation technique as described by Zhou et al. in [ZM06, ZMGJ07], i.e. we investigate how to obfuscate and deobfuscate expressions. First, we provide a detailed analysis of an MBA-obfuscated expression found in an obfuscated program, and compare the used obfuscation technique to the one given by Zhou et al. Then, we present the two approaches we propose to simplify MBA expressions, and thus deobfuscate MBA obfuscation by recovering an expression equal (or at least closer) to the original one. The first approach operates at the bit level and computes a canonical form for mixed expressions, while the second approach stays at the word level and aims at inverting the obfuscation transformations.

Both approaches are complementary, as they consider expressions on different levels and present different strengths and weaknesses, thus it would be possible to use them in combination. We only consider them separately in this work, and leave the design of a unified simplification algorithm to future work.

The implementations of both algorithms are open-source and can be found on GitHub: `arybo`[1] is implemented in C and Python and is based on a bit-blasting approach, while SSPAM[2] is implemented in Python and works at the word-level by using pattern matching and other simplifications.

[1] `https://github.com/quarkslab/arybo`
[2] `https://github.com/quarkslab/sspam/`

4.1 Manual Reconstruction of the Obfuscation Process

This section details how we linked the description of MBA obfuscation given in [ZM06, ZMGJ07] to existing examples of expressions extracted from an obfuscated program. As we did not encounter examples of MBA opaque constant in obfuscated programs, we do not address the analysis of the opaque constant technique in this section.

We explained in Section 2.1.2 that MBA obfuscation of expressions is conducted through two main steps: rewritings and insertion of identities. We thoroughly analyzed several examples of MBA-obfuscated expressions in order to determine how these steps were used, and if other obfuscating processes were involved. We present here a detailed analysis of an MBA-obfuscated expression in Figure 4.1 (in assembly language). This analysis is similar to a reverse engineering process because it is based on the examination of a finished product of obfuscation. Nevertheless, the difference lies in the fact that where most analysts would just be interested in recovering a simplified expression, we want to retrace as precisely as possible the steps involved in the creation of the obfuscated expression.

The obfuscated expression we analyze is equivalent to $(x \oplus \texttt{0x5c}) = (x \oplus 92)$ on 8 bits, and is part of an obfuscated HMAC algorithm [MG14].

4.1.1 From Assembly to Source Code

As stated in Section 3.3.2, analyzing assembly is considered rather difficult, and using a high-level representation of this expression facilitates the understanding of the obfuscation technique. Therefore, we use the Miasm reverse engineering framework (any other reverse engineering framework could be used, e.g. Medusa, Triton...) to produce a high-level Python code for this expression, given in Figure 4.2—to enhance readability, we removed the expressions concerning the flags (`af, pf, zf, nf, of` and `cf`), as they do not concern us.

This representation of the Miasm IR can be improved in terms of readability. A few basic transformations can be applied to make the output of Figure 4.2 clearer:

- replacing the data accessed in memory (`memory(...)`) by symbolic variables;
- removing the instructions added by Miasm to display semantically correct code, useless in our case, i.e. `(... << 0) | ((0x0 & 0xffffff) << 8)`;
- considering that any number or variable is at most on 32 bits, and thus removing any non-relevant binary masks, i.e. `(... & 0xFFFFFFFF)`;
- regrouping the common subexpressions in variables.

```
movzx   edi, byte ptr [esi+edx-0B8A5h]
mov     eax, esi
imul    esi, edi, 0EDh
imul    edi, 0FFFFFE26h
add     edi, 55h ; 'U'
and     edi, 0FEh
lea     ebx, [esi+edi+0D6h]
movzx   esi, bl
lea     ebx, [esi+esi]
mov     edi, 0FFh
sub     edi, ebx
and     edi, 0FEh
add     edi, esi
imul    edi, 0E587A503h
add     edi, 0B717A54Dh
imul    esi, edi, 0E09C02E7h
imul    edi, 0AD17DB56h
add     edi, 60BA9824h
and     edi, 0FFFFFF46h
imul    edi, 0A57C144Bh
lea     edi, [esi+edi-4A12D88Ah]
imul    esi, edi, 1DCE1563h
imul    edi, 0C463D53Ah
add     edi, 3C8878AFh
and     edi, 0CC44B4F4h
lea     ebx, [esi+edi-46696D2h]
mov     esi, ebx
and     esi, 94h
add     esi, esi
movzx   edi, bl
sub     esi, edi
imul    esi, 67000000h
add     esi, 0D000000h
sar     esi, 18h
imul    edi, esi, 0FFFFB22Dh
imul    esi, 0AEh
or      esi, 22h
imul    esi, 0E5h
lea     ebx, [edi+esi+0C2h]
mov     [eax+edx+result], bl
```

Figure 4.1: MBA-obfuscated expression in assembly language (x86).

```
EDI = (((memory(((EDX + ESI + 0xFFFF475B) & 0xffffffff), 0x1) & 0xff)
        << 0) | ((0x0 & 0xffffff) << 8))
EAX = ESI
ESI = ((EDI * 0xED) & 0xffffffff)
EDI = ((EDI * 0xFFFFFE26) & 0xffffffff)
EDI = ((EDI + 0x55) & 0xffffffff)
EDI = ((EDI & 0xFE) & 0xffffffff)
EBX = ((EDI + ESI + 0xD6) & 0xffffffff)
ESI = ((((EBX & 0xff) & 0xff) << 0) | ((0x0 & 0xffffff) << 8))
EBX = ((ESI * 0x2) & 0xffffffff)
EDI = 0xFF
EDI = ((EDI + ((- EBX) & 0xffffffff)) & 0xffffffff)
EDI = ((EDI & 0xFE) & 0xffffffff)
EDI = ((EDI + ESI) & 0xffffffff)
EDI = ((EDI * 0xE587A503) & 0xffffffff)
EDI = ((EDI + 0xB717A54D) & 0xffffffff)
ESI = ((EDI * 0xE09C02E7) & 0xffffffff)
EDI = ((EDI * 0xAD17DB56) & 0xffffffff)
EDI = ((EDI + 0x60BA9824) & 0xffffffff)
EDI = ((EDI & 0xFFFFFF46) & 0xffffffff)
EDI = ((EDI * 0xA57C144B) & 0xffffffff)
EDI = ((EDI + ESI + 0xB5ED2776) & 0xffffffff)
ESI = ((EDI * 0x1DCE1563) & 0xffffffff)
EDI = ((EDI * 0xC463D53A) & 0xffffffff)
EDI = ((EDI + 0x3C8878AF) & 0xffffffff)
EDI = ((EDI & 0xCC44B4F4) & 0xffffffff)
EBX = ((EDI + ESI + 0xFB99692E) & 0xffffffff)
ESI = EBX
ESI = ((ESI & 0x94) & 0xffffffff)
ESI = ((ESI + ESI) & 0xffffffff)
EDI = ((((EBX & 0xff) & 0xff) << 0) | ((0x0 & 0xffffff) << 8))
ESI = ((ESI + ((- EDI) & 0xffffffff)) & 0xffffffff)
ESI = ((ESI * 0x67000000) & 0xffffffff)
ESI = ((ESI + 0xD000000) & 0xffffffff)
ESI = ((ESI >> 0x18) & 0xffffffff)
EDI = ((ESI * 0xFFFFB22D) & 0xffffffff)
ESI = ((ESI * 0xAE) & 0xffffffff)
ESI = ((ESI | 0x22) & 0xffffffff)
ESI = ((ESI * 0xE5) & 0xffffffff)
EBX = ((EDI + ESI + 0xC2) & 0xffffffff)
memory(((EAX + EDX + 0xFFFF475B) & 0xffffffff), 0x1) = (EBX & 0xff)
```

Figure 4.2: MBA-obfuscated expression (raw representation of Miasm IR in Python).

Applying all these reductions yields the following representation of the expression (we display numbers in their hexadecimal form so that further reduction of the bit-vector size will be clearer):

$$
\begin{aligned}
a &= (\texttt{0xFF} \wedge (\texttt{0xD6} + ((\texttt{0xED})x + (\texttt{0xFE} \wedge (\texttt{0x55} + (\texttt{0xFFFFFE26})x)))))\\
b &= \texttt{0xB717A54D} + \texttt{0xE587A503} \times ((\texttt{0xFE} \wedge (\texttt{0xFF} - (\texttt{0x2})a)) + a)\\
c &= \texttt{0xB5ED2776} + \texttt{0xA57C144B} \times (\texttt{0xFFFFFF46} \wedge (\texttt{0x60BA9824} + (\texttt{0xAD17DB56})b))\\
&\quad + (\texttt{0xE09C02E7})b\\
d &= \texttt{0xFB99692E} + (\texttt{0x1DCE1563})c + (\texttt{0xCC44B4F4} \wedge (\texttt{0x3C8878AF} + (\texttt{0xC463D53A})c))\\
e &= (\texttt{0x94} \wedge d)\\
f &= ((\texttt{0x0D000000} + \texttt{0x67000000} \times (e + e - (\texttt{0xFF} \wedge d))) \gg \texttt{0x18})\\
R &= (\texttt{0xFF} \wedge (\texttt{0xC2} + \texttt{0xE5} \times (\texttt{0x22} \vee (\texttt{0xAE})f) + (\texttt{0xFFFFB22D})f))
\end{aligned}
$$

4.1.2 Other Obfuscations

While analyzing this example of MBA-obfuscated expression, we had to perform simplifying transformations in order to deobfuscate techniques not directly related to MBA obfuscation.

Bit-vector Size Extension

The result of the computations being on 8 bits (because of the final $\wedge$ `0xFF` in R), it is thus possible to reduce all the expressions on 8 bits. We explained in Section 3.4.3 that such reduction could be performed if all operators belong to $\{+, -, \times, \wedge, \vee, \oplus, \neg\}$, yet the MBA-obfuscated expression contains a right shift $\gg$ in the variable f. Considering the fact that the constants `0x67000000` and `0x0D000000` have their relevant bits only on their 8 most significant bits, that the instruction $\gg$ `0x18` shifts these to the 8 least significant bits, and that the end result is on 8 bits, we can use the equivalence:

$$\underbrace{(\texttt{0x0D000000} + (\texttt{0x67000000})x) \gg \texttt{0x18}}_{\text{on 32 bits}} = \underbrace{(\texttt{0x0D} + (\texttt{0x67})x)}_{\text{on 8 bits}}$$

For the rest of the MBA-obfuscated expression, the bit-vector size reduction consists in keeping only the 8 least significant bits of the constants, yielding the

following formula:

$$
\begin{aligned}
a &= \texttt{0xD6} + (\texttt{0xED})x + (\texttt{0xFE} \wedge (\texttt{0x55} + (\texttt{0x26})x)) \\
b &= \texttt{0x4D} + \texttt{0x03} \times ((\texttt{0xFE} \wedge (\texttt{0xFF} - (\texttt{0x2})a)) + a) \\
c &= \texttt{0x76} + \texttt{0x4B} \times (\texttt{0x46} \wedge (\texttt{0x24} + (\texttt{0x56})b)) + (\texttt{0xE7})b \\
d &= \texttt{0x2E} + (\texttt{0x63})c + (\texttt{0xF4} \wedge (\texttt{0xAF} + (\texttt{0x3A})c)) \\
e &= \texttt{0x94} \wedge d \\
f &= \texttt{0x0D} + \texttt{0x67} \times (e + e - d) \\
R &= \texttt{0xFF} \wedge (\texttt{0xC2} + \texttt{0xE5} \times (\texttt{0x22} \vee (\texttt{0xAE})f) + (\texttt{0x2D})f)
\end{aligned}
$$

Since the computations are actually done on 8 bits, the presence of terms on 32 bits is very likely due to an obfuscation technique being applied after the MBA obfuscation. We further refer to that technique as *bit-vector size extension*.

Once the bit-vector size extension has been deobfuscated, we can represent the constants of the expression in their decimal form, giving the following expression:

$$
\begin{aligned}
a &= 214 + 237x + (254 \wedge (85 + 38x)) \\
b &= 77 + 3 \times ((254 \wedge (255 - 2a)) + a) \\
c &= 118 + 75 \times (70 \wedge (36 + 86b)) + 231b \\
d &= 46 + 99c + (244 \wedge (175 + 58c)) \\
e &= 148 \wedge d \\
f &= 13 + 103 \times (e + e - d) \\
R &= (194 + 229 \times (34 \vee 174f) + 45f)
\end{aligned}
$$

Encodings

We quickly presented in Section 1.4.2 the data-flow obfuscation technique of *encodings*: relevant variables are encoded before being written in memory, and decoded upon reading. The encoding methods are traditionally affine functions. In our case, the MBA-obfuscated expression of Figure 4.1 contains computations on an encoded variable, and produces an encoded result. Therefore, in order to consider an expression working on clear inputs and outputs, we need to apply encoding and decoding functions—those encodings are not included in the assembly code because they are performed in different locations. The encoding of the input is performed by the function $x \mapsto 229x + 247$, while the decoding of the output is computed by the function $x \mapsto 237 \times (x - 247)$. Once this supplementary layer of encoding and decoding is applied, we get the complete MBA-obfuscation of Figure 4.3.

$$
\begin{aligned}
x' &= 229x + 247 \\
a &= 214 + 237x' + (254 \wedge (85 + 38x')) \\
b &= 77 + 3 \times ((254 \wedge (255 - 2a)) + a) \\
c &= 118 + 75 \times (70 \wedge (36 + 86b)) + 231b \\
d &= 46 + 99c + (244 \wedge (175 + 58c)) \\
e &= 148 \wedge d \\
f &= 13 + 103 \times (e + e - d) \\
R &= (194 + 229 \times (34 \vee 174f) + 45f) \\
R' &= 255 \wedge (237 \times (R - 247))
\end{aligned}
$$

Figure 4.3: High-level representation of the MBA-obfuscated expression.

4.1.3 Reversing the MBA-Obfuscated Expression

In order to identify the different obfuscation steps, we manually simplified the expression of Figure 4.3. We remind that we work on 8 bits, so computations are made modulo 2^8 (e.g. $237 \times 229 = 1 \mod 2^8$). Then the first lines can be rewritten as

$$
\begin{aligned}
a &= 214 + 237 \times (229x + 247) + (254 \wedge (85 + 38 \times (229x + 247))) \\
&= 129 + x + (254 \wedge (254x + 255)) \\
&= 129 + x + (254 \wedge (-2x - 1)) \\
&= 129 + x + (254 \wedge (\neg(2x))).
\end{aligned}
$$

In this case, it seems the obfuscation rule $x \oplus y \rightarrow x - y + 2 \times (\neg x \wedge y)$ was used, with substitution $\sigma = \{x \mapsto x, y \mapsto 127\}$. In order to get close to this form, we write a as

$$
\begin{aligned}
a &= x - 127 + ((2 \times 127) \wedge (2 \times (\neg x))) \\
&= x - 127 + 2 \times (127 \wedge (\neg x))
\end{aligned}
$$

Indeed, because of the even binary mask (2×127), we can write $\neg(2x)$ as $2 \times (\neg x)$. Furthermore, as the multiplication by two can also be considered as a boolean operation (a left shift by one bit), we have $(2x \wedge 2y) = 2 \times (x \wedge y)$, whatever the value of x and y. This form of the expression can be explained in two ways: either the obfuscating rule is written as $x \oplus y \rightarrow x - y + (\neg(2x) \wedge 2y)$, or the multiplication by two has moved during optimization—we were able to reproduce this kind of behavior using `clang` optimizations. It is now possible to apply the

inverse of the obfuscating rule, i.e. the reduction rule $x - y + 2 \times (\neg x \wedge y) \rightarrow x \oplus y$, with substitution $\sigma = \{x \mapsto x, y \mapsto 127\}$, and get

$$a = (x \oplus 127).$$

We continue by simplifying b in the same manner:

$$\begin{aligned} b &= 77 + 3 \times ((254 \wedge (255 - 2a)) + a) \\ &= 77 + 3 \times ((2 \times 127 \wedge 2 \times (\neg a)) + a) \\ &= 77 + 3 \times (2 \times (127 \wedge \neg a) + a) \end{aligned}$$

We can see a form quite close to the one in a, leading us to think that the same rewrite rule was used. One could note that we made the choice of decomposing 254 as 2×127 because of the previous simplification of a, but we could also have used $254 = 2 \times 255 \mod 2^8$. This is a plausible possibility, as the constant 127 does not appear (this can be explained by the insertion of identities). We need to continue the simplification further to display a form matching the rewrite rule:

$$\begin{aligned} c &= 118 + 75 \times (70 \wedge (\mathbf{36} + \mathbf{86b})) + 231b \\ 36 + 86b &= 36 + 86 \times (77 + 3 \times (2 \times (127 \wedge \neg a) + a)) \\ &= 2 \times (2 \times (127 \wedge \neg a) + a) - 254 \\ &= 2 \times (2 \times (127 \wedge \neg a) + a - 127) \\ &= 2 \times (a \oplus 127) \\ &= 2 \times ((x \oplus 127) \oplus 127) = 2x \end{aligned}$$

This time, we had to distribute the affine function to the terms of b, and then factorize the expression by 2 in order to rewrite the term (with the same rule used in a). One can note that two rewritings were applied on XOR operations that cancel each other out, which suggests that those operations were part of the obfuscation process. We then try to simplify in the same way $231b$.

$$\begin{aligned} 231b &= 231 \times (77 + 3 \times (2 \times (127 \wedge \neg a) + a)) \\ &= 123 + 181 \times (2 \times (127 \wedge \neg a) + a) \end{aligned}$$

Because of the affine functions used throughout the obfuscation, it is sometimes difficult to display a form that can be rewritten with a known rule. In order to go forward in the simplification process, we "inject" constants to force the apparition

of the right form:

$$\begin{aligned} 231b &= 123 + 181 \times (2 \times (127 \wedge \neg a) + a) \\ &= 123 + 181 \times (2 \times (127 \wedge \neg a) + a - 127) + 127 \times 181 \\ &= 70 + 181 \times (a \oplus 127) \\ &= 70 + 181x \end{aligned}$$

$$\begin{aligned} c &= 118 + 75 \times (70 \wedge (36 + 86b)) + 231b \\ c &= 118 + 75 \times (70 \wedge 2x) + 70 + 181x \\ c &= 188 + 75 \times (70 \wedge 2x) + 181x \end{aligned}$$

To further simplify the affine functions, we need to consider d.

$$\begin{aligned} d &= 46 + 99c + (244 \wedge (\mathbf{175} + \mathbf{58c})) \\ 175 + 58c &= 175 + 58 \times (188 + 75 \times (70 \wedge 2x) + 181x \\ &= 71 - 2 \times (70 \wedge 2x) + 2x \\ &= -2 \times ((70 \wedge 2x) - x) + 71 \end{aligned}$$

We have once again a form that resembles the obfuscating rule $x \oplus y \to x - y + 2 \times (\neg x \wedge y)$. The difficulty here is that we do not have any indication to help us choose between $70 = 2 \times 35$ and $70 = 2 \times 163 \mod 2^8$. By testing the two possibilities, we determined that the simplification keeping us the closest to inverting the obfuscation process is:

$$\begin{aligned} 175 + 58c &= -2 \times ((70 \wedge 2x) - x) + 71 \\ &= -2 \times ((2 \times 163 \wedge 2x) - x) + 71 \\ &= -2 \times ((163 \wedge x) \times 2 - x) + 71 \\ &= -2 \times ((\neg 92 \wedge x) \times 2 - x + 92) + 71 + 2 \times 92 \\ &= -2 \times (x \oplus 92) - 1 \\ &= \neg(2 \times (x \oplus 92)) \end{aligned}$$

Simplifying d, e, f, R and R' is performed with the same principles: identifying the obfuscating rewrite rule used and trying to display a form that allows the rewriting. The main difficulties are to identify said rule, and unravel the different side effects caused by the insertion of identities and the optimization to get an expression on which the rewriting can be applied. This also means that finding the right substitution for the rewrite rule (for example, deciding which number has been multiplied by two) can also be complex.

Once the expression is simplified, we are able to retrace precisely the obfuscation steps used to construct it: mainly, a step we call *duplication*, followed by MBA

rewritings and identities insertions. The duplication repeats the original operator with random constants, by using the property that $(c \oplus c) = 0$ for all $c \in \mathbb{Z}/2^n\mathbb{Z}$. In the example we are analyzing, the duplication was done in the following way:

$$\begin{aligned}(x \oplus 92) &= (((((((x \oplus 127) \oplus 127) \oplus 92) \oplus 122) \oplus 122) \oplus 17) \oplus 17) \\ &= ((((((x \oplus 127) \oplus 127) \oplus 92) \oplus 122) \oplus 107) \oplus 17),\end{aligned}$$

the second equality being obtained by computing $(122 \oplus 17) = 107 \mod 2^8$. We represent this obfuscation step with a function DUPLICATE, taking for inputs:

- an expression of one XOR operation $e = x \oplus y$,
- the number of bits n,
- a number of duplications to perform, that we call the *obfuscation degree*, noted d.

The algorithm of the function as we imagine it (based on the example) is given in Algorithm 1. The notation eval$(c \oplus c')$, with c and c' two constants, indicates that the value of $c \oplus c'$ is computed during execution of the algorithm.

Algorithm 1 Duplication Algorithm for the XOR operator.

Require: expression $e = (x \oplus y)$ a XOR between two terms, number of bits n, degree of obfuscation d
Ensure: Duplicated expression e'

1: **procedure** DUPLICATE(e, n, d)
2: $e_1 \leftarrow x, e_2 \leftarrow (x \oplus y)$
3: Draw d random constants modulo 2^n: $c_1, \ldots, c_d$
4: $e_1 \leftarrow (e_1 \oplus c_1)$
5: **for** i in $1, \ldots, \lfloor \frac{d}{2} \rfloor$ **do**
6: $e_1 \leftarrow e_1 \oplus (\text{eval}(c_i \oplus c_{i+1}))$ ▷ $c_i \oplus c_{i+1}$ is computed
7: **end for**
8: $e_1 \leftarrow (e_1 \oplus c_{\lfloor \frac{d}{2} \rfloor})$
9: $e_2 \leftarrow (e_2 \oplus c_{\lfloor \frac{d}{2} \rfloor + 1})$
10: **for** i in $\lfloor \frac{d}{2} \rfloor + 1, \ldots, d-1$ **do**
11: $e_2 \leftarrow e_2 \oplus (\text{eval}(c_i \oplus c_{i+1}))$
12: **end for**
13: $e_2 \leftarrow (e_2 \oplus c_d)$
14: $e' \rightarrow (e_1 \oplus e_2)$
15: **end procedure**

For example, for $d = 3$, the duplication of $x \oplus y$ on n bits produces

$$(((((x \oplus c_1) \oplus c_1) \oplus y) \oplus c_2) \oplus c_2 \oplus c_3) \oplus c_3,$$

with c_1, c_2 and c_3 are constants on n bits, and where $(c_2 \oplus c_3)$ is evaluated during duplication. Regarding our interpretation, the MBA-obfuscated expression for $(x \oplus 92)$ has been duplicated with $d = 3$.

We only encountered the duplication of the XOR operator and can only imagine different ways of duplicating other operators: for example, $(x + y)$ could be transformed in $(((x + y) + c_1) + c_2)$, with $c_1 + c_2 = 0 \mod 2^n$. It is also possible to always insert new XOR operators with random constants whatever the original operator is; as we did not encounter an MBA-obfuscated expression for other operator, we cannot exactly know what is the general behavior of this obfuscation step.

After the duplication, each XOR operation is rewritten with an obfuscating rule, and identities are inserted:

$$\begin{aligned}(x \oplus c) &= E \qquad \text{with } E \text{ an MBA expression equivalent to } (x \oplus c)\\ &= (a \times E + b) \times a^{-1} - a^{-1}b \qquad \text{with } a, b \in \mathbb{Z}/2^n\mathbb{Z}, \text{ and } a \text{ odd}\end{aligned}$$

The inner affine—with coefficients a, b—is distributed on the terms of the MBA expression E. The distribution could be part of the obfuscation process or a side effect of optimization.

In the example of Figure 4.3, the two rewrite rules used are

$$\begin{aligned}x \oplus y &\to x - y + 2 \times (\neg x \wedge y)\\ x \oplus y &\to 2 \times (x \vee y) - x - y,\end{aligned}$$

and the coefficients of the identities seem to be chosen at random.

We summarize the obfuscation technique as we reverse engineered it in Algorithm 2. The algorithm is designed for the obfuscation of an expression containing a binary operator with two operands (in our example, $(x \oplus 92)$).

We implemented this MBA obfuscation technique in Python, using the Python AST in order to represent expressions. If the optimization of common subexpressions elimination is applied, the AST representation is equivalent to a DAG representation (as we stated in Section 2.5). The `ast` module provides features for tree traversal (either for analysis or modification of the nodes).

4.2 Simplification Using Bit-Blasting Approach

This first simplification method we propose is directly inspired from the bit-vector logic, and more precisely the action of *bit-blasting*—we provided an example of

Algorithm 2 MBA-Obfuscation Algorithm.

Require: expression e an operation between two terms, number of bits n, a list of obfuscating rewrite rules R for the operator in e
Ensure: MBA-obfuscated expression e'

```
1: procedure MBA-Obf(e, n, R)
2:     e' ← Duplicate(e)
3:     for all operator in e' do
4:         Choose a random rule r ∈ R
5:         Rewrite with r
6:         Choose two random coefficients a, b on n bits for the affine (a must be invertible modulo 2^n)
7:         Compute a^-1 and -ba^-1 coefficients of the inverse affine
8:         Insert composition of inner and outer affine functions around the rewritten operator
9:         Distribute the first affine on the rewritten operator
10:    end for
11: end procedure
```

bit-blasting in SMT solvers in Section 3.2.2. Our goal is to represent an MBA expression in $\mathbb{Z}/2^n\mathbb{Z}$ symbolically at the bit level with n boolean expressions, and exploit the existence of canonical forms for these expressions to uniquely identify a mixed expression. This simplification algorithm was presented in [GEV16], and we present its main concepts in this section.

4.2.1 Description

We recall that the set $\{0, 1\}$ with the XOR and AND operations is the finite field of two elements $\mathbb{F}_2$, and that $\mathbb{F}_2^n$ is a vectorial space of dimension n over $\{0, 1\}$.

When manipulating boolean expressions, we use the *Algebraic Normal Form* (ANF), which means that they only contain XOR and AND operators. As the ANF is a canonical form, it provides a unique representation for equivalent expressions, and is defined as such for an expression with a variable $x = (x_0 \dots x_{n-1})^\intercal$ on n bits:

$$\bigoplus_{u \in \mathbb{F}_2^n} c_u \bigwedge_{i=0}^{n-1} x_i^{u_i},$$

where $c_u \in \mathbb{F}_2$, $x_i^{u_i} = x_i$ if $u_i = 1$ and $x_i^{u_i} = 1$ if $u_i = 0$. When using this canonical form, for x_i, y_i two bits of x, y in $\mathbb{Z}/2^n\mathbb{Z}$, $(x_i \vee y_i)$ is written $((x_i \wedge y_i) \oplus x_i \oplus y_i)$, and $\neg x_i$ is written $(x_i \oplus 1)$.

At the bit level, a variable x in $\mathbb{Z}/2^n\mathbb{Z}$ is represented with a symbolic vector

in $\mathbb{F}_2^n$, where x_0 is the LSB (Least Significant Bit) of x and x_{n-1} its MSB (Most Significant Bit). For example, with $n = 4$ bits, we have

$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad y = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}.$$

We can then represent expressions in the following way (for the sake of readability, we write $\cdot$ for the bitwise AND operator in boolean expressions):

$$x \oplus y = \begin{pmatrix} x_0 \oplus y_0 \\ x_1 \oplus y_1 \\ x_2 \oplus y_2 \\ x_3 \oplus y_3 \end{pmatrix}, \quad x \wedge y = \begin{pmatrix} x_0 \cdot y_0 \\ x_1 \cdot y_1 \\ x_2 \cdot y_2 \\ x_3 \cdot y_3 \end{pmatrix}, \quad x \vee y = \begin{pmatrix} (x_0 \cdot y_0) \oplus x_0 \oplus y_0 \\ (x_1 \cdot y_1) \oplus x_1 \oplus y_1 \\ (x_2 \cdot y_2) \oplus x_2 \oplus y_2 \\ (x_3 \cdot y_3) \oplus x_3 \oplus y_3 \end{pmatrix}.$$

Operations with constants can also be represented, for example,

$$((x \oplus 14) \wedge 7) = \begin{pmatrix} x_0 \\ x_1 \oplus 1 \\ x_2 \oplus 1 \\ 0 \end{pmatrix}.$$

One can note that while it is fairly trivial to represent word-level boolean operators, arithmetic operators require more work. In order to represent an addition, a classical one-bit carry adder (also called *full adder*) is used. Let x and y be in $\mathbb{Z}/2^n\mathbb{Z}$ and $s = x + y$, then each bit s_i of s can be expressed as

$$s_i = x_i \oplus y_i \oplus c_i \quad \text{with} \begin{cases} c_0 = 0 \\ c_{i+1} = (x_i \cdot y_i) \oplus (c_i \cdot (x_i \oplus y_i)) \end{cases} \tag{4.1}$$

We provide here the first boolean expressions describing an addition of two variables:

$$x + y = \begin{pmatrix} x_0 \oplus y_0 \\ x_1 \oplus y_1 \oplus (x_0 \wedge y_0) \\ x_2 \oplus y_2 \oplus ((x_1 \wedge y_1) \wedge ((x_1 \oplus y_1) \wedge (x_0 \wedge y_0))) \\ \ldots \end{pmatrix}$$

One may note that the arithmetic addition is quite expensive to translate in expressions on every bits.

The representation of subtraction is based on the fact that $-y = \neg y + 1$, thus $x - y = x + (-y) = x + \neg y + 1$. The drawback of this representation is that it

involves two additions. Another way is to write a real binary subtractor. This is similar to constructing a full adder, only changing the way the carry bit is computed: for a subtractor, $c_{i+1} = ((x_i \oplus 1) \cdot y_i) \oplus (c_i \cdot ((x_i \oplus 1) \oplus y_i))$.

In order to represent the multiplication, we use the fact that

$$y = \sum_{i=0}^{n-1} 2^i y_i.$$

It is therefore possible to perform the multiplication using n additions:

$$\begin{aligned} x \times y &= x \times (\sum_{i=0}^{n-1} 2^i y_i) \\ &= \sum_{i=0}^{n-1} x \times 2^i y_i \\ &= \sum_{i=0}^{n-1} (x \ll i) \times y_i \end{aligned}$$

As y_i is a bit, the multiplication by y_i can be represented with a boolean AND. Then for each value of i between 0 and n, the expression $(x \ll i) \wedge y_i$ must be computed, and added to the overall sum.

Vectorial Decomposition

We denote $\mathcal{B}_{k \cdot n}^n$ the set of all applications from $\mathbb{F}_2^{k \cdot n}$ (k variables of n bits) into $\mathbb{F}_2^n$. We represent the applications in $\mathcal{B}_{k \cdot n}^n$ with a vectorial decomposition, that will be of use for the identification process described in Section 4.2.2. This representation is based on separating the non-affine part and the affine part of the application.

Let us start with the case of an affine application with k input variables of n bits. The classical description of such applications is used:

$$\begin{aligned} F : \mathbb{F}_2^{k \cdot n} &\to \mathbb{F}_2^n \\ V &\mapsto A \times V \oplus B \end{aligned}$$

where V is a vector of size $k \cdot n$ containing all the bits of the input variables, A is a $n \times (k \cdot n)$ matrix and B is a constant vector of $\mathbb{F}_2^n$. The operator $\times$ stands here for classic matrix to vector multiplication, and $\oplus$ for coordinate-by-coordinate XOR between two vectors.

For instance, with $n = 4$ and $k = 1$, $f(x) = (x \oplus 14) \wedge 7$ is equivalent to the following affine application that belongs to $\mathcal{B}^4_{1\cdot 4}$:

$$F : \mathbb{F}_2^4 \to \mathbb{F}_2^4$$
$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Moreover, for non-affine applications, we introduce a purely non-affine part NA that also belongs to $\mathcal{B}^n_{k\cdot n}$. This non-affine part contains all monomials of the boolean expressions that are of degree greater or equal to 2. The vectorial decomposition of any application in $\mathcal{B}^n_{k\cdot n}$ is then

$$F(V) = NA(V) \oplus A \times V \oplus B$$

As an example, we provide both the bit-blasted representation and the vectorial decomposition of f such that $f(x) = x + 1$ on 4 bits.

$$(x+1) = \begin{pmatrix} x_0 \oplus 1 \\ x_0 \oplus x_1 \\ (x_0 \cdot x_1) \oplus x_2 \\ (x_0 \cdot x_1 \cdot x_2) \oplus x_3 \end{pmatrix}$$

In this case, $x_0 \cdot x_1$ and $x_0 \cdot x_1 \cdot x_2$ are the non-affine parts of the boolean expressions. Thus, we have:

$$\begin{aligned} F(V) &= \begin{pmatrix} 0 \\ 0 \\ x_0 \cdot x_1 \\ x_0 \cdot x_1 \cdot x_2 \end{pmatrix} \oplus \begin{pmatrix} x_0 \oplus 1 \\ x_0 \oplus x_1 \\ x_2 \\ x_3 \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} 0 \\ 0 \\ x_0 \cdot x_1 \\ x_0 \cdot x_1 \cdot x_2 \end{pmatrix}}_{NA(V)} \oplus \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\text{affine}(V)} \end{aligned}$$

The unicity of this decomposition results from the fact that we use the ANF canonical form.

4.2.2 Identification

The identification process (noted Id) takes the ANF of an application F in $\mathcal{B}_{k\cdot n}^n$ and returns an equivalent function f in $\mathbb{Z}/2^n\mathbb{Z} \to \mathbb{Z}/2^n\mathbb{Z}$. The latter representation is supposed to be more readable—and therefore more useful—for a human analyst. We present an example of the identification process of an application with one 4-bit input variable in Figure 4.4.

$$\text{If} \quad F : \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \mapsto \begin{pmatrix} x_0 \\ x_1 \oplus 1 \\ x_2 \oplus 1 \\ x_3 \end{pmatrix},$$

$$\text{then} \quad \text{Id}(F) = f : x \mapsto (x \oplus 6).$$

Figure 4.4: An example of the identification process on 4 bits.

One of the main issues of identification is to obtain a "simple" form for f. For now, we consider that the identification is only done on applications of one binary operator, making the simplest form of f easy to decide.

We detail in the next sections how to identify various operators. For both arithmetic and boolean operators, we use the vectorial decomposition presented in Section 4.2.1 to help the identification.

Identification of Boolean Operators

While describing the different identification methods, we distinguish the case where the operator has a variable and a constant as operands (thus the function has one variable), from the case where both operands are variables (the function has two variables).

Let us consider a function F in $\mathcal{B}_{k\cdot n}^n$ of k variables (we focus on the cases where $k \in \{1, 2\}$), then

$$F(V) = NA(V) \oplus A \times V \oplus B,$$

with V the vector of size $k \times 2^n$ containing the bits of the variable(s), A of coefficients $a_{i,j}$ ($0 \leq i < n$ and $0 \leq j < k \times n$), and B of coefficients b_i ($0 \leq i < n$). The constant b in $\mathbb{Z}/2^n\mathbb{Z}$ represented by B is defined as

$$b = \sum_{i=0}^{n-1} 2^i b_i.$$

When identifying a boolean operator (XOR, AND, OR) with a constant operand, the form of F does not display a non-affine part $NA(V)$. From the presence of $NA(V)$ and the form of A and B, one can deduce the identification of F:

XOR: If $F(V) = A \times V \oplus B$, with A the identity matrix and B different from the null vector, then $\textsc{Id}(F) = f : x \mapsto (x \oplus b)$, with b the constant represented by B.

In the case where the operator has two variables as operands, the $n \times 2n$ matrix A is a concatenation of two identity matrices of size $n \times n$, as shown in the following example:

$$\text{if} \quad F : X, Y \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix},$$

$$\text{then} \quad Id(F) = f : x, y \mapsto x \oplus y.$$

AND: If $F(V) = A \times V$, with A a diagonal matrix different from the identity, then $\textsc{Id}(F) = f : x \mapsto (x \wedge a)$, with $a = \sum_{i=0}^{n-1} 2^i a_{i,i}$

In the case where both operands are variables, F is composed only of $NA(V)$, with $NA(V) = (x_i \cdot y_i)_{0 \leq i < n}$.

OR: As we use the ANF for boolean expressions, the OR operator is written as $(x \vee b) = ((x \wedge b) \oplus x \oplus b)$. We use the fact that $(x \wedge y) \oplus x = (x \wedge \neg y)$ for all x, y to write $(x \vee b)$ as $(x \wedge \neg b) \oplus b$.

Thus, if $F(V) = A \times V \oplus B$ with A a diagonal matrix different from the identity, B a non-null vector with corresponding constant b, and $\sum_{i=0}^{n-1} 2^i a_{i,i} = \neg b$, then $\textsc{Id}(F) = f : x \mapsto (x \vee b)$.

When both operands are variables, F has the same affine part as a XOR, and $NA(V) = (x_i \cdot y_i)_{0 \leq i < n}$.

Identification of Additions

Regarding arithmetic operators, there is no trivial formula to describe the form of $NA(V)$. We propose a heuristic technique to identify an application F in $\mathcal{B}_n^n$ performing an addition between a variable and a constant. This technique is based on two observations: firstly, the addition is a T-function [KS03], meaning

that each output bit s_i is depending only on the $x_0, \ldots, x_i$ input bits. The second observation is that $B = F(0)$, and $F(0)$ can be easily computed using the ANF of F—the vector B can also be recovered from the vectorial decomposition. Thus we can determine if a function F is an addition of a variable and a constant by:

1. checking that F is a T-function with a graph representing dependencies between the input and output bits;
2. computing the ANF of $f : x \mapsto x + b$, with b the constant represented by B, and testing if it is equal to F.

If the ANF of f equals F, then $\textsc{Id}(F) = f : x \mapsto x + b$. The first step allows us to rule out non T-functions.

Similar techniques can be used to identify a subtraction, but further work is needed to identify multiplication and division operators.

4.2.3 Implementation

This approach has been implemented by Adrien Guinet in a tool called `arybo`, which is quickly described in this section. For more details, the reader can refer to [GEV16]. The implementation is composed of two parts:

- `libpetanque`, a library used to manipulate boolean expressions and bit-vectors inside $\mathbb{F}_2^n$, written in C++ with Python bindings;
- `arybo`, a Python library that uses `libpetanque` to support MBA expressions.

Both components form a toolkit called `arybo`[3]. For instance, the vectorial decomposition described in Section 4.2.1 is handled directly by `libpetanque`, while the adder is implemented in pure Python inside `arybo`. This allows `libpetanque` to be used as a library for other purposes.

Libpetanque

The `libpetanque` library handles the storage of vectors of symbolic bit expressions (*bit-vectors*) and the *canonicalization* of these expressions.

A boolean expression in $\mathbb{F}_2$ is represented in `libpetanque` with an Abstract Syntax Tree (AST): a node can represent a XOR or an AND operation, a symbol (a 1-bit variable) or an immediate (1 or 0). When an expression is created, it is canonicalized with the following process:

[3] `https://github.com/quarkslab/arybo`

1. Apply elementary rules based on *neutral* and *absorbing* elements: $a \cdot 0 = 0$, $a \oplus 0 = a$, $a \cdot 1 = a$, ...

2. *Flatten*, i.e. change binary nodes in n-ary nodes when possible: $(a \oplus (b \oplus c)) = a \oplus b \oplus c$.

3. Sort operators arguments: an arbitrary order is defined for the operators and the symbols used. Then, for two operators of the same kind, a lexicographical comparison is performed to order them.

4. Apply standard reduction rules, e.g. $a \oplus a = 0$, $a \cdot a = 1$.

5. Expand AND operations on XOR operations: $a \cdot (b \oplus c) = (a \cdot b) \oplus (a \cdot c)$

For each boolean expression, those canonicalization steps are repeated until no more transformation modifies the AST of the expression, leading to a canonical representation.

Arybo

The Python library `arybo` uses `libpetanque` to symbolically work with MBA expressions. It implements word-level addition, subtraction, multiplication and division algorithms. The library can be used in any external Python script. An IPython interactive shell is also provided for quick prototyping.

In Section 5.2.1, we present our evaluation of `arybo` regarding the simplification of MBA-obfuscated expressions.

4.3 Symbolic Simplification

Alongside the bit-blasting method, we worked on another approach for MBA simplification, based on the automation of the manual simplification presented in Section 4.1.

As we discussed in Section 2.2, there exist both theoretical ground and tools to manipulate and simplify arithmetic expressions (e.g. polynomial expansion, factorization) and bitwise expressions (e.g. CNF, DNF). While there is no such things for MBA expressions yet, it is still possible to use existing simplification techniques on subterms of the MBA expression that may contain only one type of operator. To create the missing link between alternating arithmetic and boolean subexpressions (meaning the edges creating MBA alternation, as explained in Section 3.4.2), one may use term rewriting.

4.3.1 Description

We described in Sections 2.1.2 and 4.1 the MBA obfuscation technique of Zhou et al., which is mainly based on rewriting operators with known equivalent MBA expressions, and applying affine functions on parts of the expression. From this fact, one can imagine a simplification algorithm that uses existing arithmetic simplifications to compute the composition of affine functions, and then uses a list of rewrite rules to transform the resulting MBA expression into another simpler and equivalent expression. One can note that regarding this MBA obfuscation technique, we do not need a bitwise simplification step, since rewritings and arithmetic expansions are enough. The useless XOR operations induced in the duplication step can be removed with basic constant folding, which is an optimization technique computing the value of constant subexpressions.

The manual simplification we performed in Section 4.1.3 can be summarized into a succession of the automatic simplification steps described in the following paragraphs.

Step One: Arithmetic Simplification

The first stage is to use classical arithmetic simplification techniques in order to compute the composition of affine functions. In our case, expansion seems to be the best choice to compute this composition. This is in theory possible with any computer algebra software, given that it supports the declaration and manipulation of MBA expressions. This step helps decrease the number of nodes of the obfuscated expression's DAG representation, as the affine functions are chosen so that their composition is equivalent to identity.

We illustrate this simplification on the term a of the obfuscated $(x \oplus 92)$ expression analyzed in Section 4.1.3:

$$\begin{aligned} a &= 214 + 237 \times (229x + 247) + (254 \wedge (85 + 38 \times (229x + 247))) \\ &= 129 + x + (254 \wedge (254x + 255)) \end{aligned}$$

As far as the composition of affine functions is concerned, this operation is mainly composed of the distribution of the coefficients of the variable (here, 237 and 38), then of the constant folding between the different constants parts, i.e. computing (237×229), $(214 + 237 \times 247)$ .... While other arithmetic simplifications are not needed to simplify this specific expression, it might be useful in other cases, for example to simplify terms such as $2x - x$.

Step Two: MBA Rewriting

The second step consists in using known rewrite rules to transform an MBA expression into a simpler expression, by reducing the number of nodes and the MBA

alternation of the DAG representation. The reduction rules are traditionally the inverses of the rules used in obfuscation. A list of common MBA obfuscating rules can be stored, and for every possible reduction rule, one can search if there exists a substitution σ such that the rewriting is possible. As we already pointed out in Section 4.1.3, it is not always trivial to identify the correct substitution. Let us consider a after arithmetic simplification:

$$a = 129 + x + (254 \wedge (254x + 255))$$

We know that the used obfuscating rule is $x \oplus y \rightarrow x - y + 2 \times (\neg x \wedge y)$, which once inverted, cannot be used as it is to simplify a. Even considering the reduction rule $x - y + (\neg(2x) \wedge 2y) \rightarrow (x \oplus y)$ which is closer to the form of a, one still needs to transform $(254x + 255)$ into $\neg(2x)$ and 129 into -127. We expose how we deal with this type of situation in Section 4.3.3, where we detail how we implemented the rewriting—the implementation of term rewriting is often called *pattern matching*—and what are the main issues of it.

Since the obfuscation phases are applied iteratively, those two simplification steps can also be used repeatedly until a fixed point is attained. It is relatively safe to assume that such a fixed point will be reached if all the rewriting rules reduce the size of the expression (see Lemma 2.3.3 in [BN99]), which is the case for our default list of reduction rules—the size can be considered as the number of nodes in the DAG representation for example. Furthermore, the expansion is a canonical form for the set of arithmetic rules, the arithmetic simplification step is thus sure to finish.

4.3.2 Implementation

We implemented this simplification approach in a tool named SSPAM[4], for Symbolic Simplification with PAttern Matching. This tool, fully implemented in Python, uses the `ast` module to represent expressions with their Python AST. It relies on two other Python modules: `sympy` offers symbolic computations and is used as an arithmetic simplifier, and `Z3` is used to help the matching of equivalent terms (see Section 4.3.3).

SSPAM is composed of a simplifier module, a pre-processing module, a pattern matcher, and tools—a Python script performing common subexpression elimination, and several classes to analyze and transform an AST.

The simplifier module processes strings or files as input, parsing them into AST and applying the main simplification loop. The input can be either an expression or a list of assignments; in the latter case, SSPAM simplifies each assignment and replaces the variables with their value in further expressions.

[4] `https://github.com/quarkslab/sspam/`

The main simplification loop is composed of:

- arithmetic simplification (performed by `sympy`),
- pre-processing,
- pattern matching: SSPAM contains a list of default rewrite rules and every single one is tested on the input expression. The first rewriting that can be applied is used—this clearly needs improvement, as the order of the patterns should not impact the simplification.

The arithmetic simplification can be applied before or after pattern matching, and the simplification will give similar results. Nevertheless, it proved to be often quicker when the arithmetic simplification was applied first.

The pre-processing is composed of transformations that aim at "normalizing" the expression. For now, only two transformations are performed: left shifts ($\ll$) are transformed into multiplications by a power of two, and subtractions by additions of inverse in order to favor commutative operators ($x - y = x + (-1) \times y$).

The interface of SSPAM is quite minimalist, as the tool is intended to perform the simplification with as little user involvement as possible. Figure 4.5 illustrates a typical call to the main simplification function of SSPAM.

```
In[1]: from sspam import simplifier

In[2]: a = "214 + 237*(229*x + 247) + (254 & (85 + 38*(229*x + 247)))"
In[3]: simplifier.simplify(a)
Out[3]: '(x ^ 127)'
In[4]: rules = [("A - B + 2*(~A & B)", "A ^ B")]
In[5]: simplifier.simplify(a, custom_rules=rules, use_default=False)
Out[5]: '(x ^ 127)'
```

Figure 4.5: A few common uses of the simplification tool SSPAM.

The function `simplify` provides the opportunity for the users to add their own reduction rules (here, `A` and `B` are *placeholders* for any term), and to prevent the use of SSPAM's default rules. The simplified expression is returned as a string.

4.3.3 Pattern Matching

Pattern matching is a field that can be viewed as the implementation of term rewriting. While term rewriting mainly describes the action of rewriting, in our case pattern matching also includes the matching of the pattern (i.e. finding a substitution so that the rewrite rule can be used). The left-hand side of the

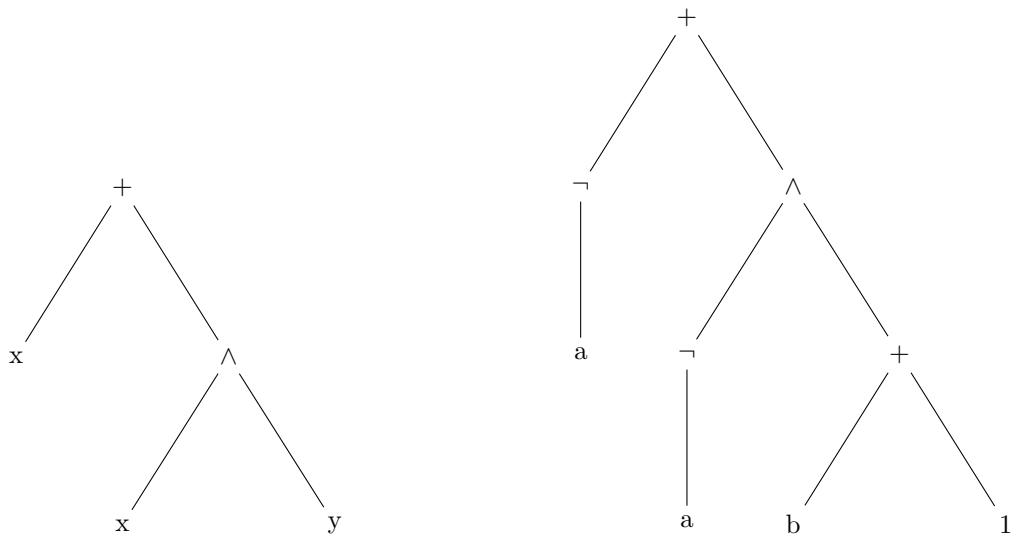
rewrite rule is then a pattern to be matched on any expression (that we usually call the *target* expression). For example, considering the rewrite rule

$$(x \wedge y) + (x \vee y) \to x + y,$$

one would want to match $(x \wedge y) + (x \vee y)$, with x and y *placeholders* for any term (also called *wildcards*). Finding values for the placeholders that make the pattern match is the same idea as finding the correct substitution to apply the rewriting. If a pattern is matched on the target expression, the target is replaced by the right-hand side of the rule, with corresponding values for the placeholders.

In SSPAM, patterns are represented with their AST, in the same way as the expressions. The pattern matching is performed by visiting two ASTs simultaneously—pattern and target expression. If the nodes are of the same type and if their children match, the matching is positive. When a wildcard is encountered, its value is either created (on the first encounter), or checked for equality with the target subexpression (on following encounters).

For example, let us try to match the pattern $x + (x \wedge y)$ (with x and y being any term) on the expression $\neg a + (\neg a \wedge (b + 1))$. The corresponding ASTs are presented in Figure 4.6.



(a) Pattern AST. (b) Target expression AST.

Figure 4.6: Illustration of ASTs for pattern and target expressions.

The matching process starts at the root node of both ASTs (in this case, an addition), and proceeds to visit and compare each node in the following way:

- root nodes are both additions, we check each child for a match:
  - left child: x is a wildcard and has no value yet, we thus associate x with the expression $\neg a$ and this is a match
  - right child: both nodes are AND operations, we check each child for a match:
    - left child: x is a wildcard and is already associated to a value, we check that this value is the left child of the AND operation in the target expression. As both the expression associated to x and the target are $\neg a$, this is a match.
    - right child: y is a wildcard and has no value yet, we thus associate y with the expression $b + 1$ and this is a match

    $\Rightarrow$ Both operands of the AND operation match, thus it is a match

  $\Rightarrow$ Both operands of the root addition match, thus it is a match

The pattern $x + (x \wedge y)$ therefore matches on the expression $\neg a + (\neg a \wedge (b+1))$, with the substitution (associated expressions to the wildcards) $\{x \rightarrow (\neg a), y \rightarrow (b + 1)\}$.

The classical issues of term rewriting and pattern matching, that we mentioned in Section 2.4, are commutativity and associativity of operators. For example, when matching the pattern $(x \wedge y) + x + y$, one would want the following expressions to produce a positive match:

$$(b \wedge a) + a + b$$
$$(a \wedge b) + b + a$$
$$b + (a \wedge b) + a$$
$$\vdots$$

Commutativity of operators can be dealt with pretty easily: if an operator is commutative and the left operands of the pattern and target are not matching, then a match is tested between the left operand of the pattern and the right operand of the target (and vice versa). While it does not bring complexity in implementing the pattern matching, it does increase the number of matching tests.

The traditional way of handling associativity is to *flatten* associative operators [KL91]. This consists in transforming a binary operator $\star$ into a m-ary ($m \geq 2$) operator such that no operand of $\star$ is an operator $\star$. We illustrate this

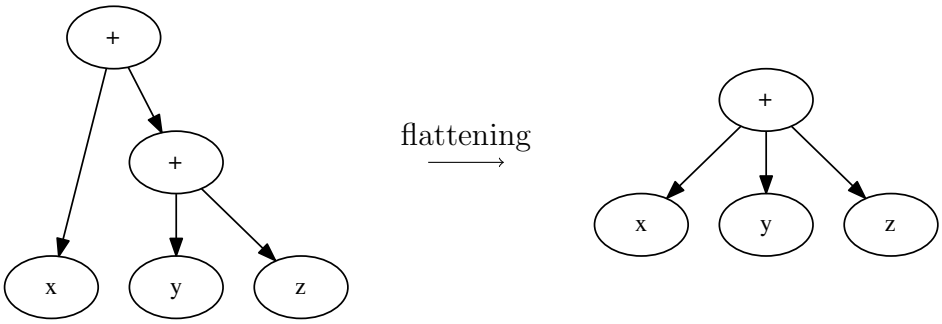


Figure 4.7: An example of flattening of $x + (y + z)$.

process on an addition of three operators in Figure 4.7. The matching is then done on sets of operands, yielding even more matching tests to perform.

Another issue more specific to our case is the multiple equivalent ways of writing a given expression. For example, let us match the pattern $2 \times (x \vee y) - (x \oplus y)$ on the target expression $2 \times (x \vee 92) - (x \oplus 92)$. It matches with the substitution $\sigma = \{x \mapsto x, y \mapsto 92\}$. Nevertheless, the target expression can appear in multiple equivalent forms on 8 bits:

$$\begin{aligned} 2 \times (x \vee 92) - (x \oplus 92) &= 2 \times (x \vee 92) + (\neg(x \oplus 92)) + 1 && (4.2)\\ &= 2 \times (x \vee 92) + (\neg x \oplus 92) + 1 && (4.3)\\ &= 2 \times (x \vee 92) + (x \oplus 163) + 1 && (4.4)\\ &= (2x \vee 184) + (x \oplus 163) + 1 && (4.5) \end{aligned}$$

It is possible to derive new patterns from the initial one in order to match expressions (4.2) and (4.3) in the following way:

$$\begin{aligned} 2 \times (x \vee y) - (x \oplus y) &= 2 \times (x \vee y) + (\neg(x \vee y)) + 1\\ &= 2 \times (x \vee y) + (\neg x \oplus y) + 1 \end{aligned}$$

However, this cannot be done for expressions (4.4) and (4.5), because of the evaluation of $2 \times 92 = 184 \mod 2^8$ and $\neg 92 = 163 \mod 2^8$. In order to deal with these situations, we use an SMT solver—in our case, Z3—to prove that both instances of subexpressions are equivalent (e.g. $\neg(x \oplus 92)$ and $(x \oplus 163)$). This is possible when a part of the pattern has matched and given possible values for x and y. For example, when matching $2 \times (x \vee y) + \neg(x \oplus y) + 1$ on the target expression $2 \times (x \vee 92) + (x \oplus 163) + 1$, the subexpression $2 \times (x \vee 92)$ gives a positive

match with $2 \times (x \vee y)$, with substitution $\sigma = \{x \mapsto x, y \mapsto 92\}$. When the other subexpression $(x \oplus 163)$ yields a negative match, SSPAM constructs the supposed matching expression according to the substitution $(2 \times (x \vee 92) + \neg(x \oplus 92) + 1)$ and asks Z3 to prove the equivalence of the two expressions (see Figure 4.8 for the corresponding Python code).

```
x = z3.BitVec('x', 8)
supposed_expr = 2*(x | 92) + (~(x ^ 92)) + 1
target_expr = 2*(x | 92) + (x ^ 163) + 1
z3.prove(supposed_expr == target_expr)
```

Figure 4.8: Proving equivalence of expressions with Z3.

Queries to Z3 could be done for any type of expressions, but while it can sometimes provide better simplification, it makes the matching very slow if several patterns are possible. In SSPAM, those queries are made only when encountering multiplications by two, negations and negative terms; as they are the most common problem-prone operations in typical MBA rewrite rules. For example, simplifying the MBA-obfuscated expression for $(x \oplus 92)$ of Figure 4.3 takes about 8 seconds with selective queries to Z3 (when provided the exact list of needed patterns), while with systematic queries to Z3 it takes around 27 seconds. We call *flexible matching* the principle of rewriting equivalent expressions with the help of an SMT Solver.

In Section 5.2.2, we present our evaluation of SSPAM on the simplification of MBA-obfuscated expressions.

Chapter 5

Resilience of the MBA Obfuscation Technique

In Chapter 3, we presented a few elements explaining the difficulty to deobfuscate MBA-obfuscated expressions. In this chapter, we assess the resilience of the MBA obfuscation technique—based on the description of [ZMGJ07] and our own reconstruction described in Section 4.1.3—meaning its resistance to deobfuscation algorithms, both known (using a black-box approach) and presented in Chapter 4 (white-box approaches). As some public work already exists on the resilience of the MBA opaque constant technique (see Section 2.2.4), we mainly focus our work on expression obfuscation, but still present a weakness of the MBA opaque constant technique. Then we suggest several improvements to improve the resilience of the MBA obfuscation for expressions.

On Expression Complexity and Resilience

A first point of interest is the difference between the *complexity* of the obfuscated expressions (however we define it), and the *resilience* of the MBA obfuscation technique. Indeed, even if one can contribute to the other, these notions are not equivalent.

The different obfuscation steps of the Algorithm 2 we reconstructed in Section 4.1—namely, duplication, rewritings and insertion of identities—increase the MBA complexity metrics we defined in Section 3.4:

- *Number of nodes:* both the duplication step and the insertion of identities increase the number of nodes of the expression (and also introduce random constants). The rewriting step often increases this metric as well.
- *MBA alternation:* the obfuscating rewrite rules are chosen so that they increase the MBA alternation.

- *Bit-vector size:* while the MBA obfuscation in itself does not change the bit-vector size, the additional obfuscation of *bit-vector size extension* we exhibited in Section 4.1.2 clearly aims at artificially increasing the bit-vector size of the obfuscated expression.

Therefore, the MBA obfuscation actually produces "more complex" expressions in terms of understandability—at least by our standards. However, this does not imply that the obfuscated expressions are truly difficult to deobfuscate, or in other words, that the obfuscation technique is *resilient*.

The main difficulty when evaluating the resilience of MBA obfuscation is the lack of simplification solution specific to MBA expressions: to our knowledge, there are no public attacks on MBA obfuscation for expression that would allow for a quantification of the obfuscation's resistance to these attacks. We state that there are two main types of attacks to apply to this obfuscation: those using a *black-box* approach, and those using a *white-box* approach. What improves the resilience against black-box approaches may or may not be the same as for white-box approaches. We first give a few elements to evaluate the resilience of the MBA obfuscation technique regarding black-box approaches; although there are no public solutions, general techniques can be used as they consider the deobfuscation without studying the form of the obfuscated code. Regarding white-box approaches, as there are none to base our study, we treat the subject in regard to the two simplification algorithms we propose in Chapter 4.

5.1 Resilience Against Black-Box Approaches

Black-box approaches, as their name suggests, infer information about a program by considering its inputs and outputs. There are several ways to deobfuscate an expression in black-box, from very basic brute-force testing of all solutions, to reconstruction of functions.

Different domains take an interest in the reconstruction of functions from a set of inputs and outputs, a few examples being program synthesis (see Section 2.6.2), the learnability of boolean functions [Ant10], or program self-testing and self-correcting [BLR90]. Two issues are commonly of concern: how many input/output pairs are needed to find a good approximation of a function, and what algorithm should be used to build that approximation.

One can note that the difficulty of reconstructing a function in black-box can depend on the function itself. It is probably easier to deobfuscate a function computing $(x \oplus 92)$ than one computing $(x \oplus 92) + (x \wedge 45)$ (a more complex function), or $(x_1 \oplus x_2) \ll 8 + (x_3 \oplus x_4)$ (containing more variables). One thing to keep in mind when considering a deobfuscation context is that the program

being analyzed has been transformed to lead the analysis into false presumptions and dead-ends. For example, an MBA obfuscated expression might compute a complicated expression of several operators, but later in the program, only the 8 less significant bits of the obfuscated expression would be used, leading to the actual computation of a $(x \oplus 92)$. Encodings are typically a good way to make the demarcation of the function to be analyzed more difficult.

Reconstructing a function is mostly based on having a candidate for said function (chosen at random, or from different hypotheses), and validate that candidate, which can be done by testing every input/output pairs exhaustively, with a proof from an SMT solver, or with a probabilistic model...
Analyzing a program as a black-box implies that the actual aspect of the expression, including the mixing of operators, does not affect the efficiency of the attack—except when computing input/output pairs. Nevertheless, it is possible to increase the resilience of the obfuscation technique by trying to make the reconstruction difficult, or slowing down the computation of input/output pairs.

Several tactics can be considered:

- adding more variables that do not affect the output of the computations, as it leads the effort of the analyst towards reconstructing functions of more variables than the actual function computed.

- increasing the bit-vector size, as it increases the space of possible functions for the reconstruction (for example, there are more functions computing $(x \oplus c)$ with c a constant modulo 2^n as n increases). Furthermore, a higher bit-vector size can also slow down validation of the candidate, especially if it is done by exhaustive test or by proving the equivalence. For example, testing all input/output pairs for the obfuscated $(x \oplus 92)$ on 8 bits (see Figure 4.3) in a Python script takes around 0.2 ms, while the same test on the 32-bit version (without simplifying the bit-vector size extension) takes more than three minutes. The proof of equivalence with Z3 between the MBA-obfuscated expression on 8 bits takes about 0.01s, while on 32 bits it takes about 0.04s.

- increasing the number of nodes of the expression, as it can also slow down the computation of input/output pairs.

We can thus conclude that the mixing of operators does not seem to bring much resilience against black-box approaches. Nevertheless, we want to stress the fact that some MBA expressions seem to cause a heavy slow down when proving equivalence with Z3. For example, proving the equivalence of the MBA Equation 5.1 on 32 bits in Z3 takes 0.02 seconds, while proving the equivalence of

Equation 5.2 takes 63 seconds.

$$(x \wedge y) = (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \tag{5.1}$$

$$\begin{aligned}(x \wedge y) = &-3 \times (\neg y) - \neg(x \vee y) - 5 \times (\neg x \wedge y) + 2 \times (\neg x) \\ &+ 2 \times (\neg(x \wedge y)) + (x \vee y)\end{aligned} \tag{5.2}$$

It is still subject to further work to analyze the reasons for these poor performances, whether it is because of the overall size of the MBA, or a reason specific to its form.

5.2 Resilience Against our Simplification Tools

In this section, we provide the evaluation of both our tools on real examples of MBA-obfuscated expressions, and use the results of these evaluations to conclude on the resilience of the MBA obfuscation technique regarding our simplification solutions. Unlike the previous attacks using black-box approaches, our tools use the actual description of the expression, and are thus using a white-box approach.

5.2.1 Bit-Blasting

The big strength of the bit-blasting approach is to transform the problem of MBA simplification into boolean expression simplification, which is a domain where a lot of work has been done. When using the bit-blasted representation, we have several canonical forms available that give a unique representation for any expression.

The main drawback of this approach is that the canonicalization of boolean expressions can be very expensive in memory and time, especially when arithmetic operators are involved. For example, experimental results show that canonicalizing the representation of $x + y$ on 16 bits takes about 5.7 seconds, which is the time to canonicalize around 700 000 XOR operations. The size of the boolean expressions representing each bit also grows exponentially when considering arithmetic operators: for example, we showed in [GEV16] that the i-th bit expression of $x+y$ contains $2^i + 1$ monomials in its ANF form. For an expression on 16 bits, this means that the most significant bit will be a XOR operation between 65537 terms.

Another issue is that identification from boolean expressions to word-level expressions is not trivial. We gave a few techniques when trying to identify one operator, but the question of identifying more complex expressions containing several operators is clearly something to work on in the future.

Evaluation

We tested `arybo` on examples of MBA-obfuscated expressions that we generated, in order to show the increase in execution time with the number of bits. For each number of bits n, we generated 10 examples of MBA-obfuscated expressions for a XOR operation between a variable and a random constant (the execution was performed on a Core i7-3520M processor). The MBA-obfuscated expressions were generated with the obfuscation degree ($d = 3$) as the obfuscated $(x \oplus 92)$ of Figure 4.3.

For each expression, `arybo` was successful at identifying the original expression. The execution times are synthesized in Table 5.1.

n	execution time (s)
8	0.09
9	0.39
10	1.74
11	8.81
12	61.32

Table 5.1: Execution time of Arybo for the canonicalization of $(x \oplus c)$ with c constant and n the number of bits.

Canonicalization for other obfuscated operators between a variable and a constant (AND, OR, addition) presents similar execution time. One can see that the number of bits of the expression greatly influences the execution time. In real settings, an MBA-obfuscated expression on 32 bits could not be handled by `arybo` in a reasonable time. The number of variables of the expression is also of importance: for example, the canonicalization of an obfuscated expression (with the same parameters as previously) for $(x \oplus y)$ takes more than two minutes on 8 bits.

In conclusion, the simplification using bit-blasting is quite efficient on expressions with a low number of bits (e.g. 8 bits). The resilience of the MBA obfuscation technique—regarding this technique—is thus very dependent on the number of bits of the obfuscated expression. It is very probable that the obfuscation of bit-vector size extension (see Section 4.1.2) is designed to improve the resilience of obfuscated expressions with a low number of bits. Nevertheless, this kind of obfuscation can be easily simplified with the reduction method we exposed in Section 3.4.3 (implementation of such reduction is currently being developed in both `arybo` and SSPAM).

5.2.2 Symbolic Simplification

Because the algorithm using symbolic simplification works at the word-level, the simplification is not impeded by an increasing number of bits. The ASTs representing the expressions are also far smaller than the representation in the bit-blasting approach. For example, $x + y$ has 3 nodes at the word-level for any number of bits; when bit-blasted on 4 bits it is represented by four ASTs of respectively 3, 6, 17 and 41 nodes.

The main drawback of this approach is that it is highly dependent on the chosen set of rewrite rules. Indeed, if only one obfuscation rule is unknown, the simplification algorithm is not able to reduce the expression as much as it would with knowledge of that rule. Even when having the list of all used obfuscating rules, it is not guaranteed to have a satisfying deobfuscation. As the obfuscation process usually has a constraint of not deteriorating greatly the performances of the program being obfuscated, we can assume that the set of MBA rewrite rules will be of "reasonable" size. We show in Section 5.4.5 that the most common linear MBA rewrite rules are composed of 4 boolean expressions, which induces a low number of possible rules. Eventually, all the obfuscating rules can be recovered by analysts.

In this section, we provide tests to evaluate the efficiency of the tool SSPAM[1].

Methodology

We tested SSPAM on both public examples of MBA-obfuscated expressions [ZM06, MG14], and examples we generated with our own Python obfuscator (based on what we reconstructed, see Algorithm 2).

Here, we consider that the output of the simplification tool is fully simplified when it returns the original expression, which we know to be only one operator for the public examples, and is of course known for our generated expressions.

In the following experiments, we only try to determine the resilience of the MBA obfuscation as defined in [ZMGJ07], and used alone without other control flow or data flow obfuscation techniques. If several layers of obfuscation were to be used, the difficulty of simplifying the expression would greatly increase, and the analyst would very probably need to deobfuscate each layer separately.

We only tested our solution in a "clean" context, where MBA-obfuscated expressions are generated and simplified directly from the source code, instead of a reverse-engineering context where the analyzed MBA would be in their assembly form. This is due to the fact that automation of the generation, compilation, and especially the translation from assembly to a high-level semantics is not trivial. Nevertheless, we implemented transformations in our MBA obfuscator designed

[1] All tests were realized with version 0.2.0.

to simulate certain aspects of optimization: mainly, the distribution of the multiplication by two in boolean expressions (e.g. $2 \times (x \wedge y) = (2x \wedge 2y)$), and the distribution of the inner affine on the rewritten expression.

Simplifying Available Examples

We applied SSPAM on the few public obfuscated expressions available in the literature:

- All examples of obfuscated operators of Zhou et al.'s work [ZM06] were fully simplified by our tool. A comparison of obfuscated inputs and simplified outputs can be found in Figure 5.1. Simplifying each example takes less than two seconds with our tool.

$$
\begin{aligned}
t_1 &= (4211719010 \oplus 2937410391x) + 2 \times (2937410391x \vee 83248285) + 4064867995 \\
t_2 &= (2937410391x \vee 3393925841) - ((2937410391x) \wedge 901041454) + 638264265y \\
z &= 519915623t_1 - ((3383387769t_2 + 129219187) \oplus 2756971371) \\
&\quad - 2((911579527t_2 + 4165748108) \vee 2756971371) + 4137204492
\end{aligned}
$$

(a) Obfuscated expressions [ZM06].

$$
\begin{aligned}
t_1 &= ((2937410391x) + 4148116279) \\
t_2 &= ((638264265y) + 3393925841) \\
z &= (x + y)
\end{aligned}
$$

(b) Outputs of SSPAM.

Figure 5.1: Simplification of some state of the art examples.

- A larger example of an MBA-obfuscated expression found in a real-life obfuscated DRM was given in [MG14]. We analyzed this example in Section 4.1, and used the version of Figure 4.3 to test the simplification. SSPAM is able to retrieve the original expression ($x \oplus 92$) in about 8 seconds given the exact 6 rewrite rules needed (and about 12 seconds when all 23 default rules are available).

Generating New MBA-obfuscated Expressions

Using the obfuscation algorithm detailed in Algorithm 2, we chose four expressions to obfuscate: $(x + y)$, $(x \oplus y)$, $(x \wedge 78)$ and $(x \vee 12)$ on 8 bits, and four rewriting rules from [War02] (given in Figure 5.2), one for each operator $+, \oplus, \wedge, \vee$.

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

Figure 5.2: Rewriting rules used to obfuscate our sample expressions.

The duplication of the XOR operation is done as described in Algorithm 1. We use the same principle for the duplication of the addition, for some obfuscation degree d:

$$x + y = (((x + c_1) - c_1 + c_2) - c_2) + y + \cdots + c_d$$

where $-c_1 + c_2$ is evaluated. We designed a duplication algorithm for the AND and OR operators, based on the idea of duplicating the original operator:

$$
\begin{aligned}
(x \wedge y) &= (x \wedge (y \wedge c_1)) + \cdots + (x \wedge (y \wedge c_d)) \\
(x \vee y) &= (x \vee (y \wedge c_1)) \vee \cdots \vee (x \vee (y \wedge c_d))
\end{aligned}
$$

Where $c_1 \oplus \cdots \oplus c_d = -1 \mod 2^n$, and $(c_i \wedge c_j) = 0$ for all $1 \leq i, j \leq d, i \neq j$. One can notice that if y is a constant, $y \wedge c_i$ can be evaluated. The duplication of $(x \wedge y)$ can also be done with OR operations instead of additions, but we prefer to use additions in order to increase the MBA alternation of the expression.

For each expression to be obfuscated, for each degree from 1 to 6, we generated 20 obfuscated expressions to be used as input for SSPAM. We then computed the number of fully simplified expressions and the average number of nodes reduction (from obfuscated expressions to simplified expressions), as well as the average MBA alternation reduction.

The average number of node reduction can be seen in Figure 5.3. The 100% ratio (e.g. for $x+y$ when $d = 1$) means that all 20 obfuscated expressions were fully simplified. When the ratio is different from 100%, the fully simplified expressions are not considered in the average computation, in order to get a better idea of the behavior of not completely simplified expressions. The results for average MBA alternation reduction were very similar, thus we do not provide the corresponding graph.

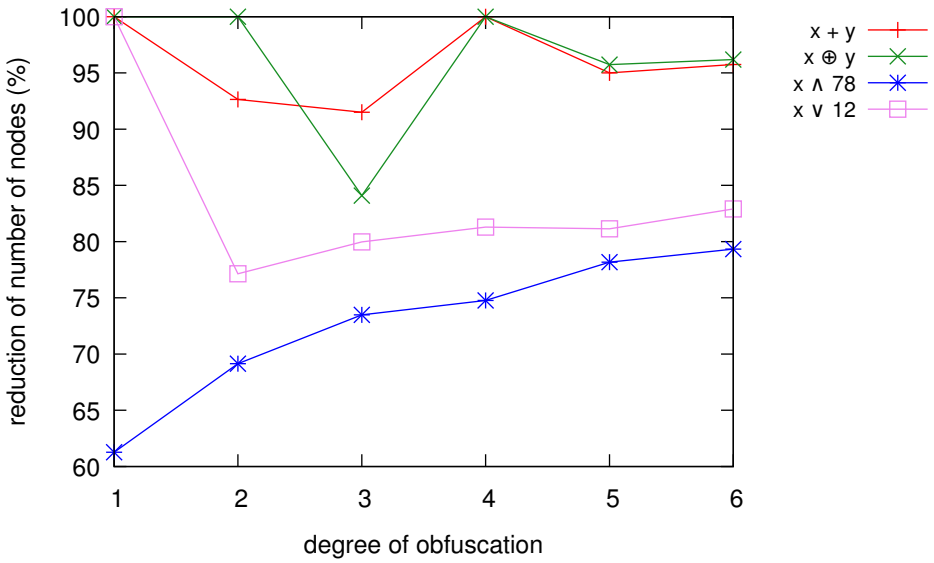


Figure 5.3: Average Number of Nodes Reduction.

For expressions $x + y$ and $x \oplus y$, the number of fully simplified expressions is between 18 and 20 (on 20 expressions) when $d \leq 4$. On the other hand, no expression was fully simplified for expressions $x \wedge 78$ and $x \vee 12$, whatever the value of d. When the degree of obfuscation equals 5, no obfuscated expression was fully simplified.

Globally, the simplification method seems to be quite efficient: on the worst case, the size of the expression is still reduced by more than 60%, and a fair amount of expressions have been fully simplified. By analyzing more precisely the graph, it appears that obfuscated additions and XOR operations are better simplified than AND and OR operations. In our example, it could be due either to the choice of operator (addition/XOR or AND/OR), or the operands (variables or constants). Indeed, $x + y$ and $x \oplus y$ do not contain any constants, which usually bring more randomness to the simplification. We thus performed the same tests on two types of obfuscated expressions, $x \oplus 135$ and $x \wedge y$, to decide whether the presence of constants or the operator was decisive. As previously, for each original expression and for each degree of obfuscation $1 \leq d \leq 6$, we generated 20 obfuscated examples. The results are shown in Figure 5.4.

We add to this graph the information that, for $x \oplus 135$, when the ratio is not 100% (all expressions fully simplified), only one or two expressions were not fully simplified. For $x \wedge y$, whatever the value of d, no expressions were fully simplified on the 20 examples of obfuscated expressions.

This confirms that the simplification of obfuscated additions and XOR oper-

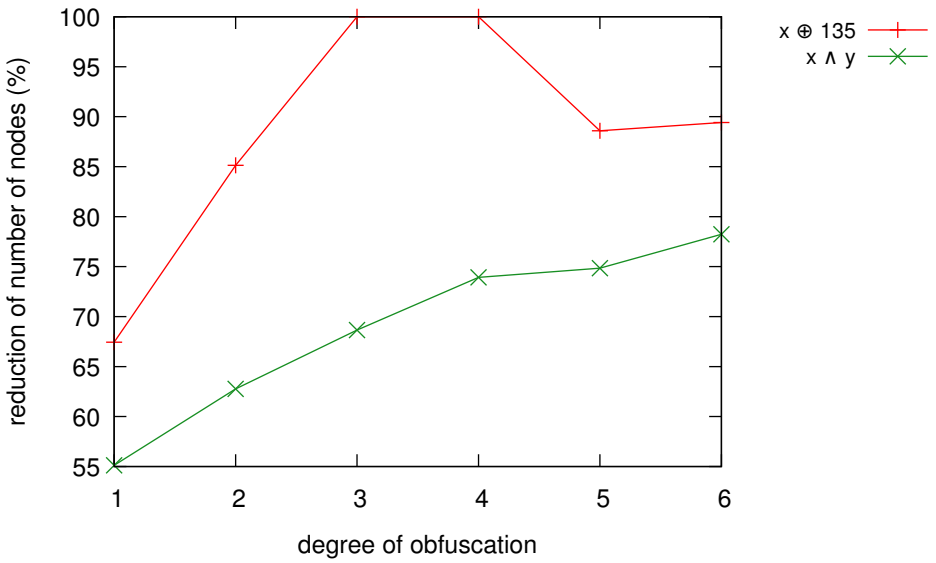


Figure 5.4: Average Number of Nodes Reduction for $x \oplus 135$ and $x \wedge y$.

ations is much more efficient than the simplification of obfuscated AND and OR operations. Considering that we implemented our own duplication method for those two operators, we cannot really deduce anything about the MBA obfuscation technique as it exists publicly. When quickly analyzing the expressions that failed to be simplified, it seems that a bitwise simplifier would quite improve the result of the simplification. SSPAM does not possess such a simplifier because to our knowledge, there is no public bitwise simplifier that would be easily integrable in our tool, and we have left for future work the implementation of one.

We also tested the simplification of expressions with an increasing number of bits: for $d = 3$, and for a number of bits of 8, 16, 32, and 64, we generated 10 instances of MBA-obfuscated expressions for $x+y$, and computed the average time of simplification on those 10 examples. For every number of bits, all expressions were fully simplified. We reproduce the times of simplification in Table 5.2.

n	execution time (s)
8	9.80
16	10.98
32	11.69
64	12.99

Table 5.2: Simplification times of an obfuscated $(x + y)$, with increasing number of bits.

One can see that increasing the number of bits does not greatly augment the execution time, as only the queries to Z3 might be slowed down because of n. What does increase the simplification time is rather the size (the number of nodes of the DAG representation) of the obfuscated expression. We produce in Table 5.3 a comparison of the different execution times for an increasing degree of obfuscation (tested on 10 examples for each degree). The second column contains the average sizes in number of nodes of the obfuscated expression.

d	average size	execution time (s)
1	42	0.86
2	70	3.97
3	84	9.22
4	96	18.39

Table 5.3: Simplification times of an obfuscated $(x + y)$, with increasing degree of obfuscation.

SSPAM thus successfully simplifies most of the generated expressions (when the duplication algorithm is the one we reconstructed). The simplification is not impeded by the number of bits, but rather by the size of the obfuscated expression, in terms of number of nodes. It is safe to assume that, for performance reasons, the obfuscated expressions will less likely be of great size. Thus the MBA obfuscation technique offers little resilience to this type of simplification.

5.3 Algebraic Weakness of the Opaque Constant Technique

The resilience of the opaque constant technique presented in [ZMGJ07] was already assessed by Biondi et al. in [BJLS15], thus we did not focus our research on this technique. Nevertheless, we do point a weakness of the obfuscation: depending on the form of the null MBA expression used for obfuscation, the expanded form of the opaque constant displays clearly the constant to be hidden. We detail this weakness in Lemma 1.

Lemma 1. *Let*

- $P(X) = a_0 + a_1X + a_2X^2 + \cdots + a_dX^d$ *be a polynomial of degree* d*,*
- Q *be a polynomial of degree* d*, such that* $P(Q(X)) = X$ *for all* $X \in \mathbb{Z}/2^n\mathbb{Z}$*,*
- *and* $E = \sum a_i \left(\prod e_{i,j}(x_0, \ldots, x_{t-1})\right)$ *be a null MBA expression, with no* $e_{i,j}$ *such that* $e_{i,j}(x_0, \ldots, x_{t-1}) = 1$ *whatever the values of* $x_0, \ldots, x_{t_1}$ *(i.e. the sum composing the MBA does not contain any constant).*

Then the constant monomial of $P(E + Q(K))$ is equal to K.

Proof:

$$\begin{aligned} P(E + Q(K)) &= a_0 + a_1(E + Q(K)) + \cdots + a_d(E + Q(K))^d \\ &= a_0 + a_1 Q(K) + a_2 Q(K)^2 + \cdots + a_d Q(K)^d + \varphi(E) \end{aligned}$$

with $\varphi(E) = \sum_{k=1}^{d} a_k \Big(\sum_{i=0}^{k-1} E^{k-i} Q(K)^i \Big)$, a polynomial in variables $x_0, \ldots, x_{t-1}$, with no constant monomial (every monomial of $\varphi(E)$ is multiplied by a positive power of E). This means that the constant part of $P(E + Q(K))$ is

$$a_0 + a_1 Q(K) + a_2 Q(K)^2 + \cdots + a_d Q(K)^d = P(Q(K)) = K. \quad \square$$

Thus, the expanded form of the opaque constant will reveal the constant K. We provide an example with the following formulas.

$$\begin{aligned} P(X) &= 91 + 195X + 192X^2 \mod 2^8 \\ Q(X) &= 55 + 107X + 192X^2 \mod 2^8 \\ P(Q(X)) &= 7077888X^4 + 7888896X^3 + 6290688X^2 + 2280705X + 591616 \\ &= X \mod 2^8 \\ E &= x - y + 2 \times (\neg x \wedge y) - (x \oplus y) = 0 \end{aligned}$$

$$\begin{aligned} P(Q(\mathbf{26}) + E) = {} & 192x^2 + 128xy + 128x \times (x \oplus y) + 235x + 192y^2 + 128y \times (x \oplus y) \\ & - 235y + 214 \times (\neg x \wedge y) + 192 \times (x \oplus y)^2 - 235 \times (x \oplus y) + \mathbf{26} \end{aligned}$$

However, if E contains a constant part, we have $E = E' + c$, and

$$\begin{aligned} P(E + Q(K)) &= a_0 + a_1(E' + c + Q(K)) + \cdots + a_d(E' + c + Q(K))^d \\ &= P(c + Q(K)) + \varphi(E') \end{aligned}$$

The constant monomial of the opaque constant is then $P(c + Q(K))$. From this observation, we strongly advise avoiding the use of this obfuscation technique with a null MBA expression not containing a constant part.

5.4 Suggested Improvements

In Sections 5.1 and 5.2, we show that several ways—using public algorithms or our contributions—exist to design an attack on MBA-obfuscated expressions. While

no perfect simplification solution exists, these attacks are efficient enough to consider that the MBA obfuscation technique offers little resilience as it is. In this section, we suggest a few improvement in order to improve the resilience against deobfuscation, whether they are general improvements or specific to our simplification tools.

5.4.1 Producing Less Common Subexpressions

In the current version of the obfuscation algorithm (see Algorithm 2), the XOR operators to be rewritten are chosen from inner operator to outer operator. We illustrate this process with the first obfuscation steps of the expression $(x \oplus 92)$. Let us recall that the duplication (see Algorithm 1) produces

$$(x \oplus 92) = ((((((x \oplus 127) \oplus 127) \oplus 92) \oplus 122) \oplus 107) \oplus 17).$$

We use e_0 to represent the obfuscated expression corresponding to $(x \oplus 127)$; the second XOR operation to be obfuscated is thus $(e_0 \oplus 127)$, with an MBA rewrite step producing

$$(e_0 \oplus 127) \to e_0 - 127 + 2 \times (\neg e_0 + 127).$$

One can notice that e_0 now appears twice in the overall obfuscated expression, meaning it is a common subexpression that can be shared in the DAG representation. As most rewrite rules increase the number of occurrences of terms, each rewrite step yields more common subexpressions representing the same obfuscated subexpression.

If the obfuscation is performed from the outer operator to the inner one, then each occurrence of a XOR operator is obfuscated differently. This produces a bigger expression with very few common subexpressions, at the expense of a longer obfuscation time (far more obfuscation steps are required), and a much bigger expression—which can be a problem depending on the context.

We implemented this improvement and tested it to obfuscate an expression with the same duplicated form as seen before. While the DAG of the original obfuscated $(x \oplus 92)$ has 72 nodes and an MBA alternation of 12, the new obfuscated one (with obfuscation from outer operator to inner operator) produces a DAG of 719 nodes and an MBA alternation of 230.

We also studied the impact of LLVM optimizations on the size of obfuscated expressions. In this case, the size is defined as the number of words, because computing the DAG of an expression from assembly code is hard to automate. We obfuscated the expression $(x \oplus 92)$ with random affine functions and a small list of MBA rewrite rules, with the original obfuscation and the improved one. On 50 examples generated with the original obfuscation, LLVM optimizations (using option `-O3` of `clang`) reduce the size of the code by 75% in average. On 50 examples

generated with the improved obfuscation, LLVM optimizations reduce the size of the code only by 40% in average.

In terms of readability, an obfuscated expression with fewer common subexpressions is more difficult to process, as the analysis of smaller parts cannot be reused at other positions in the expression. This can also increase the resilience against black-box attacks, as the expression contains much more computations—making it longer to gather input/output pairs, as well as slowing down proofs with SMT solvers.

5.4.2 Using New Identities

The MBA obfuscation technique currently uses affine functions in order to insert identities around rewritten expressions. An improvement would be to use different invertible functions, such as polynomials. Invertible polynomials in $\mathbb{Z}/2^n\mathbb{Z}$ were characterized by Rivest in [Riv99], but no inversion algorithm was provided. A subclass of those invertible polynomials with a closed-form formula to compute a polynomial's inverse is described in [ZMGJ07] in order to construct opaque constants. This subclass is only composed of the polynomials sharing the same degree as their inverse.

We propose in [BERR16] an algorithm to invert all invertible polynomials in $\mathbb{Z}/2^n\mathbb{Z}$, based on Newton's inversion algorithm. This provides a greater number of polynomials to use for obfuscation, as well as more resilience to algebraic attacks. Indeed, one of the attacks proposed in [BJLS15] regarding the MBA opaque constant technique is based on certain properties of the polynomials. Using a larger class of invertible polynomials makes the opaque constant resistant to this attack.

Permutation polynomials could thus be used either as identities (instead of affine functions) in the expression obfuscation, or as polynomials for the opaque constants.

5.4.3 Improving the Duplication

One can note that the duplication step (see Algorithm 1) keeps the initial constant of the expression if there is one (e.g. 92). A black-box approach might choose its candidate by giving priority to constants included in the obfuscated expression, thus it seems a good idea to avoid displaying any original constants in the obfuscated result. An example of how to use duplication to hide the original constant could be to evaluate an operation with another constant (here, $127 \oplus 92 = 35 \mod 2^8$):

$$\begin{aligned}(x \oplus 92) &= ((((((x \oplus 127) \oplus 127) \oplus 92) \oplus 122) \oplus 107) \oplus 17) \\ &= (((((x \oplus 127) \oplus 35) \oplus 122) \oplus 107) \oplus 17)\end{aligned}$$

Another improvement of this step could be to insert new variables that do not change the result of the computations, but increase the search space for function reconstruction. For example, DUPLICATE could also introduce a variable y with $((x \oplus y) \oplus y)$ instead of constants.

5.4.4 Increasing the Resilience Against our Tools

In order to improve the general resilience of the obfuscation technique against simplification using bit-blasting, the expression produced should have a high number of bits; then a different location in the obfuscated code could be used to gather the bits actually needed for the computation. Hiding the number of used bits can also be done by using traditional "bit hacks" such as the *interleaving* of bits. The obfuscation could also use more arithmetic operators, especially multiplication, as they are more difficult to process for this simplification approach.

The key to improving the resilience of the obfuscation against symbolic approaches such as SSPAM is to increase the *diversity*—of the obfuscation steps, of the rewriting rules... For example, the technique could rewrite random nodes instead of the ones introduced by the duplication step. We refer to our work in [EGV16], where we tested SSPAM on this type of obfuscation: the deobfuscation is then globally less efficient. Different types of rewrite rules would also improve the resilience. For example, if a method were found to create rules for specific expressions involving constant (e.g. $x \oplus 42$), then one could create constant-dependent rewrite rules during obfuscation, for example the obfuscating rule:

$$x \oplus 42 \rightarrow ((x \vee 191) \wedge (x \oplus 106)) + (x \wedge 64).$$

This would easily increase the number (and diversity) of obfuscating rules, intensifying as much the efforts of the analyst in order to recover said rules.

Rewrite rules with constraints (for example, a range on the variables) would also greatly increase the difficulty of simplification: in addition to know the list of rewrite rules, the simplification tool would also have to recover the corresponding constraints and verify them to apply the rule. This verification is often easier when obfuscating, as more information about the program is available, than when reverse-engineering a binary.

In the next section, we show how the parameters used in the generation of MBA rewrite rules (as described in [ZMGJ07]) can determine the number, and thus the diversity, of available obfuscating rules during obfuscation.

5.4.5 Expanding the Pool of Available Rewrite Rules

In this section, we study the method described in [ZMGJ07] to create equivalent MBA expressions, and thus MBA rewrite rules. We recall very briefly this method (please refer to Section 2.1.3 for more details).

In their paper, Zhou et al. state that for any $\{0,1\}$-matrix of size $2^t \times s$ with linearly dependent column vectors, one can generate a null linear MBA expression $E = \sum_{i=0}^{s-1} a_i e_i$. Here, for all i, a_i is in $\mathbb{Z}/2^n\mathbb{Z}$, e_i is a boolean expression of t variables, and s is the number of boolean expressions of the MBA. From this null MBA, several MBA rewrite rules can be inferred by choosing different left-hand sides (LHS) and right-hand sides (RHS) from the null MBA expression. For example, the null MBA

$$x + y - (x \wedge y) - (x \vee y) = 0$$

can lead to multiple rewrite rules, such as

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

Given this method to generate linear MBA rewrite rules, we want to enumerate all possible rewritings of traditional operators (XOR, AND, addition...) for a given s. The operator to be rewritten (the LHS of the rewrite rule) has to be a sum of boolean expressions represented by $\{0,1\}$-vectors $v_1, \dots, v_k$ of size 2^t, with $k < s$, and coefficients $a_1, \dots, a_k$. For example, when considering expressions of two variables x and y ($t = 2$), x is represented with the truth-table vector $v_1 = (0\ 0\ 1\ 1)^\intercal$, y with $v_2 = (0\ 1\ 0\ 1)^\intercal$, and thus $x+y$ is represented by $a_1v_1 + a_2v_2$ with $a_1 = a_2 = 1$ (here, $k = 2$).

In order to enumerate all MBA rewrite rules for the expressions represented by $v_1, \dots, v_k$, one must find all vectors $v_{k+1}, \dots, v_s$ such that there exist some coefficients $a_{k+1}, \dots, a_s$ verifying

$$a_1v_1 + \dots + a_kv_k + a_{k+1}v_{k+1} + \dots a_sv_s = 0.$$

The LHS of the rewrite rule being represented by $\sum_{i=1}^{k} a_i v_i$, the RHS is obtained with by $\sum_{i=k+1}^{s} -a_i v_i$.

We illustrate this with the enumeration of all MBA rewrite rules of $x + y$ with this method, for $s = 5$. We choose $s = 5$ because in most examples of MBA rewrite

rules we encountered, the RHS of the rewrite rule was composed of three boolean expressions (i.e. $s - k = 3$). As $x + y$ contains two boolean expressions x and y, we have $k = 2$, and thus $s = 5$.

Let us consider the $2^t \times s$ matrix A, with $t = 2$ and $s = 5$:

$$A = \begin{array}{c} \begin{array}{ccccc} x & y & u & v & w \end{array} \\ \begin{pmatrix} 0 & 0 & u_0 & v_0 & w_0 \\ 0 & 1 & u_1 & v_1 & w_1 \\ 1 & 0 & u_2 & v_2 & w_2 \\ 1 & 1 & u_3 & v_3 & w_3 \end{pmatrix} \end{array}$$

We want to enumerate all triplet of vectors (u, v, w) such that there exists a linear combination of all columns of A that is equal to a null vector. We consider every non-null $\{0, 1\}$-vector of size 4, which makes a total of 15 possible vectors for u, v, w. For each triplet in those possible vectors, we have to check if a solution $\{\alpha, \beta, \gamma\}$ exists in $\mathbb{Z}/2^n\mathbb{Z}$ to the following system:

$$\begin{aligned} 0 + 0 + \alpha u_0 + \beta v_0 + \gamma w_0 &= 0 \\ 0 + 1 + \alpha u_1 + \beta v_1 + \gamma w_1 &= 0 \\ 1 + 0 + \alpha u_2 + \beta v_2 + \gamma w_2 &= 0 \\ 1 + 1 + \alpha u_3 + \beta v_3 + \gamma w_3 &= 0 \end{aligned}$$

This system must be solved for every combination of three vectors among the 15 available ones, which makes $\binom{15}{3} = 455$ possibilities. In this case, the system is overdetermined, as they are more equations than unknowns. To bypass this issue, we solve the system composed of the last three equations, which gives us every solution when $u_0 = v_0 = w_0 = 0$. When a solution is found, we try to verify the first equation on all possible values of u_0, v_0 and w_0. We discard any solution where α, β or γ is null, as we want to have exactly five boolean expressions in the resulting MBA equation.

In order to solve the system, we use the Python module `numpy` and the function `numpy.linalg.solve`. From each three vectors u, v, w with solutions α, β, γ, we are able to construct a MBA rewrite rule for $x + y$. A first noteworthy remark is that some rules that seem different are actually the same. For example, the rules

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

are equal (one just needs to swap x and y to get the same expressions). When two equal rules are found, only one is kept for the enumeration. This gives us twelve

distinct obfuscating rules for $x + y$:

$$
\begin{aligned}
&x + y \rightarrow (x \vee y) + y - (\neg x \wedge y)\\
&x + y \rightarrow (x \vee y) + (\neg x \vee y) - (\neg x)\\
&x + y \rightarrow -1 + y - (\neg x)\\
&x + y \rightarrow 2 \times (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)\\
&x + y \rightarrow 2 \times (-1) - (\neg x) - (\neg y)\\
&x + y \rightarrow (x \oplus y) + 2y - 2 \times (\neg x \wedge y)\\
&x + y \rightarrow (x \oplus y) + 2 \times (\neg x \vee y) - 2 \times (\neg x)\\
&x + y \rightarrow -(x \oplus y) + 2y + 2 \times (x \wedge \neg y)\\
&x + y \rightarrow 2y - (\neg x \wedge y) + (x \wedge \neg y)\\
&x + y \rightarrow 2y - (\neg x) + (\neg y)\\
&x + y \rightarrow y + (x \wedge \neg y) + (x \wedge y)\\
&x + y \rightarrow (\neg x \wedge y) + (x \wedge \neg y) + 2 \times (x \wedge y)
\end{aligned}
$$

We use the same method to enumerate all rewrite for $(x \oplus y)$, $(x \wedge y)$, $(x \vee y)$. As all these are boolean expressions equivalent to one vector, we have $k = 1$ and still aim for $s - k = 3$, so $s = 4$. We present the number of distinct rewrite rules for each operator in Table 5.5, as well as the enumeration when the RHS of the rewrite rule is composed of four boolean expressions (i.e. $s - k = 4$). We provide in Appendix A the list of rewrite rules for $s - k = 3$ for all operators, and in Appendix B, some examples of rewrite rules for $s - k = 4$.

	$s - k = 3$	$s - k = 4$
$x + y$	12	167
$x \oplus y$	12	79
$x \wedge y$	7	48
$x \vee y$	6	84

Figure 5.5: Enumeration of all possible MBA rewrite rules.

One can notice that for most common cases when $s - k = 3$, the number of rewrite rules for each operator is not very high (it is very possible to store and try to identify less than 40 rules). We thus strongly recommend generating MBA rewrite rules with $s - k \geq 4$.

Conclusion

Code obfuscation intends to protect programs by making their analysis more costly, in terms of skills, time and tools. It operates mainly by either mutating code and data, or adding bogus information into the code. Our work has been focused on the study of one particular data obfuscation using Mixed Boolean-Arithmetic (MBA) expressions, to obfuscate constants and expressions. This study was conducted in two parts, mainly the reconstruction of the obfuscation algorithm and the design of two deobfuscation solutions. The information we gathered about the technique, together with the simplification algorithms we provided, allowed us to assess the resilience of the MBA obfuscation.

We provided the context to fully apprehend the notions around obfuscation in Chapter 1. We gave the traditional definition and properties of obfuscation, as well as some basic concepts of cryptographic obfuscation. Our work is focused mainly on practical obfuscation, which intends to resist the action of *reverse engineering*, mainly by modifying the control flow and the data flow of a program. Therefore, we listed a few common techniques of reverse engineering, along with classical obfuscation techniques. We also presented a variety of methods and metrics used to quantify the quality of an obfuscation (usually called *resilience*), even though this issue is still being investigated.

In Chapter 2, we detailed the different concepts that we used in our study of the MBA obfuscation. We recalled the various contributions of Zhou et al. [ZMGJ07]: the definition of a *polynomial* MBA expression, two obfuscation algorithms (one to create opaque constants, the other to obfuscate expressions), as well as a method to generate MBA equivalences. We showed how the issue of simplifying an expression is recognized as complex, even when considering arithmetic or boolean expressions. There are nevertheless fields of research that can be of use in our context, namely bit-vector logic and term rewriting, and we presented how they relate to our problematic.

There is a strong lack of literature on the subject of MBA expressions: while the concept can be found in cryptography (without the label "MBA"), it is not used for the same reasons as in obfuscation. We thus built our own theoretical background, and initiated explanations on the difficulty of MBA deobfuscation in

Chapter 3. In our case, we believe that the former resilience of the obfuscation was mainly due to the absence of tools to manipulate and simplify MBA expressions, as well as the fact that optimization passes add transformations to the initial obfuscation. We proposed three metrics to help characterize what is a *complex* MBA expression, with the help of the DAG representation: the number of nodes, the MBA alternation, and the bit-vector size.

In order to study the MBA obfuscation technique, we first had to reconstruct its algorithm: from the method described in the work of Zhou et al. and examples of expressions found in a commercial obfuscated program, we were able to fully identify the MBA obfuscation technique used on those samples. In the process, we also encountered other obfuscation techniques applied in conjunction with MBA obfuscation—bit-vector extension and encodings. From this study, presented in Chapter 4, we designed two deobfuscation tools for MBA-obfuscated expressions. The first one, `arybo`, is based on the principle of bit-blasting, i.e. explicitly writing and canonicalizing every bit expression of every operator, and then finding the corresponding word-level expression. The second solution, SSPAM, manipulates expressions at word-level, and intends to invert the obfuscation transformations applied on the expression. It uses classical arithmetic expansion and term rewriting (also called *pattern matching* when referring to the implementation).

With this work of analysis and deobfuscation of the MBA obfuscation technique, we were able to give elements to assess its *resilience*, meaning the obscuring capacity of the technique, in Chapter 5. To our knowledge, there exists no public solution for MBA simplification, and thus we assessed the resilience against solutions not specific to MBA (i.e. black-box approaches), as well as against our own solutions. Regarding black-box approaches, the impact of the mixing of operators would need further investigations, but it seems classical techniques such as *program synthesis* (see Section 2.6.2) could probably be used to deobfuscate "simple" expressions—e.g. an operation on 8 bits between a variable and a constant. Then we evaluated our two simplification tools on MBA expressions. The bit-blasting solution, `arybo`, is quite efficient on a small number of bits (e.g. expressions on 8 bits), but exhibits performance issues when the number of bits augments—mainly caused by the exponentially increasing number of expressions and the cost of canonicalization. On the other hand, SSPAM successfully deobfuscates public examples of MBA-obfuscated expressions, and gives positive results when tested on our own samples. From the results of our evaluations, we concluded that in its current state, the MBA obfuscation technique does not provide great resistance to our simplification algorithms. Finally, we suggested several improvements in order to increase the resilience.

Future Work

There are still many ways to improve our two simplification algorithms. We stated that `arybo` had performance issues when the number of bits rises, which could be solved with a change of representation for the bit expressions (e.g. the use of symmetric functions [Weg87]). We also need to find new heuristics or methods to identify expressions containing several operators from bit-level to word-level.

Regarding SSPAM, the absence of a real bitwise simplifier could be problematic: instead of fully implementing one, it would be possible to use `arybo` in situations where the bit expressions are not of great complexity. The implementation of strategies in the choice of the simplification to be applied is also a future milestone, as it could greatly improve the results. For example, we noticed that some expressions require arithmetic factorization instead of expansion in order to be matched to known patterns. We could thus imagine different simplification strategies guided with our MBA complexity metrics.

These metrics are not enough, and could be complemented with other concepts or quantities to assess the resilience of MBA obfuscation. For example, designers of obfuscations would be interested in characterizing what is a *good* MBA-obfuscating rewrite rule: do we need to check for a great MBA alternation, or to verify that the rewrite rule cannot be expressed as the successive applications of simpler rewrite rules? Another interesting approach could be to study the termination and confluence of the induced rewrite systems used for simplification, and try to choose obfuscating rules that would impede those properties. We would also like to design a metric describing the "distance" between the original expression and the obfuscated one: are the original constants still appearing in the obfuscated expression, are the operands that were close in the original DAG still close in the obfuscated DAG...

For now, our main lead in order to increase the resilience of the MBA obfuscation technique is to drastically increase the variety of the rewrite rules, for example by generating specific rules during the obfuscation (e.g. using a rule transforming $x + 5$ instead of $x + y$ with $\{y \mapsto 5\}$). Our first attempts at creating such expressions, by testing exhaustively all combinations of operators and operands of a given size, lead us to think that it is not trivial to find such rewrite rules, especially under a restricted time. Using rewrite rules that only apply with some condition (for example, a range of values for the variables), could also impede any approach using pattern matching. In particular, conditional rules would mean that different occurrences of the same subexpression could be rewritten differently, thus minimizing the interest of the sharing in the DAG representation.

In order to increase the resilience to bit-blasting approach, rewrite rules containing more arithmetic operators, especially multiplication, should be preferred. The resilience of the MBA obfuscation technique could also depend on the purpose of the analyst, whether it is identifying a distinctive feature of a standard algorithm, or extracting a formula.

In conclusion, we were able to show that the current MBA obfuscation technique does not offer great resilience and can be defeated with our two public tools. Nevertheless, there is still a lot of possibilities in order to improve both MBA obfuscation and our deobfuscation algorithms, and it is still not clear which side—between obfuscation and deobfuscation—will eventually "win" in the end, or if a balance will be found where the difficulty of MBA deobfuscation would be linked to our metrics.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Ant10] Martin Anthony. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, chapter 6: Probabilistic Learning and Boolean Functions, page 197–220. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[BA06] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A Decision Procedure for Bit-vector Arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 522–527, New York, NY, USA, 1998. ACM.

[BERR16] Lucas Barthelemy, Ninon Eyrolles, Guenaël Renault, and Raphaël Roblin. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 51–59, New York, NY, USA, 2016. ACM.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, 2001. Springer-Verlag.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 215–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[BJLS15] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of Synthesis in Concolic Deobfuscation. Preprint, December 2015.

[BL82] Bruno Buchberger and Rüdiger Loos. Algebraic Simplification. In Bruno Buchberger, George Edwin Collins, and Rüdiger Loos, editors, *Computer Algebra*, volume 4 of *Computing Supplementa*, pages 11–43. Springer, 1982.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 73–83, New York, NY, USA, 1990. ACM.

[BN99] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Aug. 1999.

[BS14] Vasily Bukasof and Dmitry Schelkunov. Deobfuscation and beyond. ZeroNights conference, 2014.

[Car04] Jacques Carette. Understanding Expression Simplification. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 72–79, New York, NY, USA, 2004. ACM.

[CC11] Vitaly Chipounov and George Candea. Enabling Sophisticated Analyses of x86 Binaries with RevGen. In *7th Workshop on Hot Topics in System Dependability (HotDep)*, Hong Kong, China, June 2011.

[CEJVO03] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. *White-Box Cryptography and an AES Implementation*, pages 250–270. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[CKWG16] Joel Coffman, Daniel M. Kelly, Christopher C. Wellons, and Andrew S. Gearhart. Rop gadget prevalence and survival under compiler-based binary diversification schemes. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 15–26, New York, NY, USA, 2016. ACM.

[CN09] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[Coh93] Frederick B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565 – 584, 1993.

[Cor16] Marie-Angela Cornelie. *Implantations et protections de mécanismes cryptographiques logiciels et matériels*. PhD thesis, Université Grenoble Alpes, April 2016.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, University of Auckland, 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[Des12] Fabrice Desclaux. Miasm: Framework de reverse engineering. In *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications*. SSTIC, 2012.

[Det99] Detlef Plump. Term Graph Rewriting. In *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, volume 2, pages 3–61. World Scientific Pub Co Inc, 1999.

[DGBJ14] Bruce Dang, Alexandre Gazet, Elias Bachaalany, and Sebastien Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, chapter 5: Obfuscation. Wiley Publishing, 2014.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Handbook of Theoretical Computer Science (Vol. B). chapter Rewrite Systems, pages 243–320. MIT Press, Cambridge, MA, USA, 1990.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer, 2008.

[DPG05] Mila Dalla Preda and Roberto Giacobazzi. *Semantic-Based Code Obfuscation by Abstract Interpretation*, pages 1325–1336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[DYJAJ14] Apon Daniel, Huang Yan, Katz Jonathan, and Malozemoff Alex J. Implementing Cryptographic Program Obfuscation. Cryptology ePrint Archive, Report 2014/779, 2014.

[EGV16] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 27–38, New York, NY, USA, 2016. ACM.

[FCMCI12] A.J. Farrugia, B. Chevallier-Mames, M. Ciet, and T. Icart. Performing boolean logic operations using arithmetic operations by code obfuscation, August 9 2012. US Patent App. 13/024,258.

[FSA97] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[GBD05] Vijay Ganesh, Sergey Berezin, and David L. Dill. A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, Stanford University, 2005.

[GEV16] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *Proceedings of GreHack 2016*, GreHack 2016, Grenoble, France, November 2016.

[GG10] Yoann Guillot and Alexandre Gazet. Automatic Binary Deobfuscation. *Journal in Computer Virology*, 6(3):261–276, 2010.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, pages 40–49, Washington, DC, USA, 2013. IEEE Computer Society.

[GLZ12] Yuan Xiang Gu, Clifford Liem, and Yongxin Zhou. System and method providing dependency networks throughout applications for attack resistance. App. PCT/CA2011/050157, Publication Number WO2012126083 A1, Sept. 2012. Irdeto Canada Corporation.

[GR07] Shafi Goldwasser and Guy N. Rothblum. On Best-possible Obfuscation. In *Proceedings of the 4th Conference on Theory of Cryptography*, TCC'07, pages 194–213, Berlin, Heidelberg, 2007. Springer-Verlag.

[Gul10] Sumit Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Operating and programming systems series. Elsevier, New York, 1977. Elsevier computer science library.

[Hec03] Heck, Andre. *Introduction to Maple*, chapter 6: Internal Data Representation and Substitution. Springer, 2003.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[JGZ08] Harold Joseph Johnson, Yuan Xiang Gu, and Yongxin Zhou. System and method of interlocking to protect software-mediated program and device behaviors. US Patent App. 11/980,392, Publication Number US20080208560 A1, Aug. 2008.

[JJN+08] Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei (Nick) Saw, and Ramarathnam Venkatesan. The Superdiversifier: Peephole Individualization for Software Protection. In Kanta Matsuura and Eiichiro Fujisaki, editors, *Proceedings of the Third International Workshop on Security*, IWSEC '08, pages 100–120, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, pages 3–9. IEEE, 2015.

[JSV09] Mariusz H Jakubowski, Chit Wei Saw, and Ramarathnam Venkatesan. Iterated Transformations and Quantitative Metrics for Software Protection. In *SECRYPT*, pages 359–368, 2009.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KL91] E. Kounalis and D. Lugiez. Compilation of Pattern Matching with Associative-commutative Functions. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Colloquium on Trees in Algebra and Programming (CAAP '91): Vol 1*, TAPSOFT '91, pages 57–73, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Klo92] J. W. Klop. Handbook of Logic in Computer Science (Vol. 2). chapter Term Rewriting Systems, pages 1–116. Oxford University Press, Inc., New York, NY, USA, 1992.

[KMM+06] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A Qualitative Analysis of Java Obfuscation. In *proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA*, 2006.

[KN10] Dmitry Khovratovich and Ivica Nikolić. Rotational Cryptanalysis of ARX. In *Fast Software Encryption*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010.

[Kre98] Christoph Kreitz. *Automated Deduction — A Basis for Applications: Volume III Applications*, chapter 5: Program Synthesis, pages 105–134. Springer Netherlands, Dordrecht, 1998.

[KS03] Alexander Klimov and Adi Shamir. A New Class of Invertible Mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470–483. Springer, 2003.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[Kup96] Tony R. Kuphaldt. Lessons in Electric Circuits, Vol. IV - Digital, chapter 7: Boolean Algebra. `http://www.allaboutcircuits.com/textbook/digital/#chpt-7`, 1996.

[KW13] Dhiru Kholia and Przemyslaw Wkegrzyn. Looking Inside the (Drop) Box. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[KZ05] Arun Kandanchatha and Yongxin Zhou. System and method for obscuring bit-wise and two's complement integer computations in software. US Patent App. 11/039,817, Publication Number US20050166191 A1, Jul. 2005. Cloakware Corporation.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LGJ08] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. A Compiler-based Infrastructure for Software-protection. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 33–44, New York, NY, USA, 2008. ACM.

[LM91] Xuejia Lai and James L. Massey. A Proposal for a New Block Encryption Standard. In *Advances in Cryptology — EUROCRYPT '90*,

volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 1991.

[Map] Maple (Release 12.0). Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. `http://www.maplesoft.com/`.

[Mar14] Mariano Ceccato. On the Need for More Human Studies to Assess Software Protection. Technical report, November 2014.

[Mas87] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[McD12] J. Todd McDonald. Capturing the Essence of Practical Obfuscation. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management, ICISTM 2012*, volume 285 of *Communications in Computer and Information Science*, pages 451–456. Springer, 2012.

[MG14] Camille Mougey and Francis Gabriel. DRM obfuscation versus auxiliary attacks. Recon conference, 2014.

[MP15] Rabih Mohsen and Alexandre Miranda Pinto. Algorithmic Information Theory for Obfuscation Security. Technical Report 793, 2015.

[Mui13] James A. Muir. *A Tutorial on White-Box AES*, pages 209–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[NPB15] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *JSAT*, 9:53–58, 2015.

[PF06] Philippe Biondi and Fabrice Desclaux. Silver Needle in the Skype. In *BlackHat Europe*, 2006.

[Riv99] Ronald L. Rivest. Permutation Polynomials Modulo 2^w. *Finite Fields and Their Applications*, 7:2001, 1999.

[RS14] Dusan Repel and Ingo Stengel. Grammar-based transformations: attack and defence. *Information Management & Computer Security*, 22(2):141–154, 2014.

[S+15] W. A. Stein et al. *Sage Mathematics Software (Version 6.5)*. The Sage Development Team, 2015. `http://www.sagemath.org`.

[Spa10] Branko Spasojevic. Code Deobfuscation by Optimization. In *27th Chaos Communication Congress*, 2010.

[SS15] Florent Saudel and Jonathan Salwan. Triton: A Dynamic Symbolic Execution Framework. In *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications*, pages 31–54. SSTIC, 2015.

[Szk] Kévin Szkudłapski. (lead developper). `https://github.com/wisk/medusa/`.

[Vui03] Jean Vuillemin. Digital Algebra and Circuits. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 733–746. Springer, 2003.

[War02] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[WHdM13] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently Solving Quantified Bit-Vector Formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

[WS06] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[ZM06] Yongxin Zhou and Alec Main. Diversity Via Code Transformations: A Solution For NGNA Renewable Security. Technical report, The NCTA Technical Papers, 2006.

[ZMGJ07] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *8th International Workshop in Information Security Applications*, WISA '07, pages 61–75, 2007.

Appendix A

MBA Rewrite Rules for $s - k = 3$

A.1 Addition

$$
\begin{aligned}
x + y &\rightarrow (x \vee y) + y - (\neg x \wedge y) \\
x + y &\rightarrow (x \vee y) + (\neg x \vee y) - (\neg x) \\
x + y &\rightarrow -1 + y - (\neg x) \\
x + y &\rightarrow 2 \times (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\
x + y &\rightarrow 2 \times (-1) - (\neg x) - (\neg y) \\
x + y &\rightarrow (x \oplus y) + 2y - 2 \times (\neg x \wedge y) \\
x + y &\rightarrow (x \oplus y) + 2 \times (\neg x \vee y) - 2 \times (\neg x) \\
x + y &\rightarrow -(x \oplus y) + 2y + 2 \times (x \wedge \neg y) \\
x + y &\rightarrow 2y - (\neg x \wedge y) + (x \wedge \neg y) \\
x + y &\rightarrow 2y - (\neg x) + (\neg y) \\
x + y &\rightarrow y + (x \wedge \neg y) + (x \wedge y) \\
x + y &\rightarrow (\neg x \wedge y) + (x \wedge \neg y) + 2 \times (x \wedge y)
\end{aligned}
$$

A.2 XOR

$$
\begin{aligned}
&x \oplus y \rightarrow (x \vee y) - y + (\neg x \wedge y)\\
&x \oplus y \rightarrow (x \vee y) - (\neg x \vee y) + (\neg x)\\
&x \oplus y \rightarrow (-1) - (\neg x \vee y) + (\neg x \wedge y)\\
&x \oplus y \rightarrow 2 \times (x \vee y) - y - x\\
&x \oplus y \rightarrow 2 \times (-1) - (\neg x \vee y) - (x \vee \neg y)\\
&x \oplus y \rightarrow -y + 2 \times (\neg x \wedge y) + x\\
&x \oplus y \rightarrow -(\neg x \vee y) + 2 \times (\neg x \wedge y) + (x \vee \neg y)\\
&x \oplus y \rightarrow y + x - 2 \times (x \wedge y)\\
&x \oplus y \rightarrow (\neg x \vee y) + (x \vee \neg y) - 2 \times (\neg(x \oplus y))\\
&x \oplus y \rightarrow y + (x \wedge \neg y) - (x \wedge y)\\
&x \oplus y \rightarrow y + (\neg y) - (\neg(x \oplus y))\\
&x \oplus y \rightarrow (\neg x \vee y) + (x \wedge \neg y) - (\neg(x \oplus y))
\end{aligned}
$$

A.3 AND

$$
\begin{aligned}
&x \wedge y \rightarrow -(x \vee y) + y + x\\
&x \wedge y \rightarrow 1 + y + (x \vee \neg y)\\
&x \wedge y \rightarrow (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)\\
&x \wedge y \rightarrow (-1) - (\neg x \wedge y) - (\neg y)\\
&x \wedge y \rightarrow -(x \oplus y) + y + (x \wedge \neg y)\\
&x \wedge y \rightarrow -(\neg(x \wedge y)) + y + (\neg y)\\
&x \wedge y \rightarrow -(\neg(x \wedge y)) + (\neg x \vee y) + (x \wedge \neg y)
\end{aligned}
$$

A.4 OR

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

Appendix B

Some MBA Rewrite Rules for $s - k = 4$

B.1 Addition

$$
\begin{aligned}
&x + y \to 3 \times (x \vee \neg y) + (\neg x \vee y) - 2 \times (\neg y) - 2 \times (\neg (x \oplus y)) \\
&x + y \to -(x \vee \neg y) - (\neg x) + (x \wedge y) + 2 \times (-1) \\
&x + y \to (x \vee \neg y) + (\neg x \wedge y) - (\neg (x \wedge y)) + (x \vee y) \\
&x + y \to 2 \times (\neg (x \oplus y)) + 3 \times (\neg x \wedge y) + 3 \times (x \wedge \neg y) - 2 \times (\neg (x \wedge y)) \\
&\qquad \vdots
\end{aligned}
$$

B.2 XOR

$$
\begin{aligned}
&x \oplus y \to -(x \vee \neg y) + (\neg x \vee y) - 2 \times (\neg (x \vee y)) + 2 \times (\neg y) \\
&x \oplus y \to (x \vee \neg y) - 3 \times (\neg (x \vee y)) + 2 \times (\neg x) - y \\
&x \oplus y \to -(x \vee \neg y) + (\neg y) + (x \wedge \neg y) + y \\
&x \oplus y \to (x \vee \neg y) + (\neg x \vee y) - 2 \times (\neg (x \vee y)) - 2 \times (x \wedge y) \\
&\qquad \vdots
\end{aligned}
$$

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Obfuscation par Expressions Mixtes Arithmético-Booléennes : Reconstruction, Analyse et Outils de Simplification

Mots clefs : obfuscation, expressions MBA, protection de code

Résumé : L'obfuscation, également appelée obscurcissement ou offuscation, est une technique de protection logicielle contre la rétro-conception. Elle transforme du code afin de rendre son analyse plus difficile. Les expressions mixtes arithmético-booléennes (MBA) sont une technique d'obfuscation du flot de données introduite en 2007 et qu'on rencontre dans des applications réelles. Cette technique est présentée comme robuste alors même que le domaine de l'obfuscation MBA étant assez jeune, il bénéficie de peu de littérature sur la conception et l'analyse de telles expressions obfusquées. Nous nous sommes attachés à effectuer une étude approfondie de cette technique qui soulève d'intéressantes questions à la fois théoriques et pratiques, autant sur l'obfuscation que sur la désobfuscation (ou simplification) d'expressions MBA.
Durant nos recherches, nous avons structuré le sujet de l'obfuscation MBA, et l'avons relié à d'autres domaines, principalement la cryptographie, la réécriture ou la logique des vecteurs de bits. Nous avons également reconstruit une implémentation d'obfuscation MBA à partir d'échantillons publics. Nous avons étudié ce que signifie simplifier une expression obfusquée, et défini nos propres métriques de simplicité pour les expressions MBA. Cette étude nous a permis de concevoir deux outils de désobfuscation, dont l'implémentation a simplifié avec succès plusieurs exemples publics d'expressions obfusquées. Enfin, nous avons évalué la résilience de l'obfuscation MBA par rapport à nos algorithmes de simplification (ainsi que d'autres techniques de désobfuscation), et nous avons conclu que la technique d'obfuscation MBA offrait peu de résilience en l'état. Nous avons donc proposé des pistes pour améliorer ce type d'obfuscation.

Title : Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools

Keywords : obfuscation, MBA expressions, software protection

Abstract : Software obfuscation is a software protection technique that transforms code in order to make its analysis more difficult by reverse-engineering. Mixed Boolean-Arithmetic (MBA) expressions are an obfuscation technique introduced in 2007 and used in real life products. They are presented as a strong data flow obfuscation technique, even though there is little literature on the design and analysis of such obfuscated expressions. We have performed an in-depth study of this technique which raises many theoretical and practical questions both around the obfuscation and deobfuscation (or simplification) of MBA expressions.
In our study, we structured the subject of MBA obfuscation and linked it to other topics, mainly cryptography, rewriting and bit-vector logic. We also reconstructed an MBA obfuscation implementation from public samples. We studied the meaning of simplifying an obfuscated expression, and defined our own simplicity metrics for MBA expressions. Our study of MBA simplification yielded the implementation of two deobfuscation tools that successfully simplified several public examples of obfuscated expressions. Finally, we assessed the resilience of the MBA obfuscation with respect to our simplification algorithms (as well as other deobfuscation techniques), concluding that the MBA obfuscation technique offers little resilience as it is, and we proposed new ideas to improve it.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France