Inspecting Compiler Optimizations on Mixed Boolean Arithmetic Obfuscation

Rachael Little
University of New Hampshire
rachael.little@unh.edu

Dongpeng Xu
University of New Hampshire
dongpeng.xu@unh.edu

***Abstract*—Software obfuscation is a form of code protection designed to hide the inner workings of a program from reverse engineering and analysis. Mixed Boolean Arithmetic (MBA) is one popular form that obscures simple arithmetic expressions via transformation to more complex equations involving both boolean and arithmetic operations. Most prior works focused on developing strong MBA at the source code or expression level; however, how many of them are resilient against compiler optimizations still remain unknown. In this work, we carefully inspect the strength of MBA obfuscation after various compiler optimizations. We embed MBA expressions from several popular datasets into C programs and examine how they appear post-compilation using the compilers GCC, Clang, and MSVC. Surprisingly, we discover a notable trend of reduction in MBA size and complexity after compiler optimization. We report our findings and discuss how MBA expressions are impacted by compiler optimizations.**

I. Introduction

Mixed Boolean Arithmetic (MBA) is a form of software obfuscation which has been a prominent focus of research in software security over the last two decades[1][2][3]. Software obfuscation is the process of transforming program instructions into a semantically equivalent but much harder to analyze form; depending on the tools used, it can be applied at the binary or source code level. MBA specifically involves rewriting algebraic instructions by inter-mixing boolean operators so that SMT solvers are unable to solve them or greatly slowed down in their analysis. MBA have been in use both in commercial DRM and protection scheme[4] and by the authors of malware[5].

Numerous MBA simplification techniques have been devised to attempt to break MBA obfuscation and measure their strength, including those employing pattern matching [6], program synthesis[7] [8], machine learning [9], and adaptation for SMT solving[4]. The majority of these work at the expression or C source code level, with some support for binary analysis, with the exception of the recent tool GooMBA. GooMBA uses a combination of heuristics, solving and program synthesis to solve MBA at the binary level [10]. In response to several of these techniques, recent works have also focused on improving the strength, size, and complexity of MBA[3][11]. In particular, non-linear (also known as polynomial) MBA, which heavily increase the semantic complexity of MBA expressions, have been shown to thwart a number of popular tools[5] [12].

Problem. Interestingly, a number of MBA expressions embedded at source code level can be removed by compiler optimizations. Although a few works have discussed this possibility in general detail[1], none have comprehensively examined the persistence of popular MBA datasets post-compilation. In this paper, we test this using three popular industry compilers, GCC, MSVC, and Clang, and how they affect MBA at various optimization levels. We give an overview of these compilers and their optimizations, and show the size reduction and simplification of both polynomial and linear MBA from a diverse set of recent works.

Findings. We find that compiler optimizations are able to reduce a considerable number of MBA, at times even entirely reducing them to their original expressions. Across all datasets we tested, over 50% of the MBA were reduced in size post-compilation using aggressive optimization settings. Compilers are able to simplify lengthy sub-expressions generated by term rewriting rules. Many MBA datasets are created using term-rewriting, which leaves redundancies that compilers can easily identify and remove. Our dataset and analysis scripts are available at https://github.com/mbacompilation/inspecting-compiler-opts.

II. Background

Mixed Boolean Arithmetic(MBA) expressions are a form of software obfuscation applied to mathematical operations to obscure their semantics and deter analysis. Specifically, they feature boolean (or bitwise) operations intermixed with arithmetic (addition, division, multiplication, and subtraction). Typical mathematical solvers such as Z3 are unable to process them correctly, and as a result, several software protection companies have employed them for protection.[13].

MBA appear majorly in two forms, linear and polynomial. Linear MBA is typically generated by exchanging arithmetic operations (such as plus and minus) with equivalent boolean expressions. Fig. 1 shows an example of some MBA transformations provided by the Tigress obfuscator[14].

Over the last two decades, numerous works have been developed to advance the state of knowledge of MBA, both

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23007
www.ndss-symposium.org

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

Fig. 1: Linear MBA from the Tigress Obfuscator.

from both defending and attacking sides; initial survey works detail the construction and means of breaking early forms of MBA[1][2]. Later works including MBA-Obfuscator[3] and Loki[11] improve upon these and boast powerful advancements in the complexity of available MBA formula.

One area which has received less study is the application of typical compiler optimizations upon MBA. Modern compilers apply transformations to code which improve its performance and, optionally, tremendously reduce its size. Although notable works such as Eyrolles' thesis[1] notes that compiler optimizations are able to remove some MBA, and some build upon compiler optimization theory to reduce MBA[15], no works focus specifically on the effects of standard compiler optimization passes on MBA datasets at a widespread level. In the following sections, we describe some MBA formula in more detail, and give an overview on typical compiler optimizations.

A. MBA Formula

1) Tigress: The Tigress obfuscator, which offers a number of software obfuscation types, includes MBA as part of its data protection options. Tigress takes arithmetic expressions at the C source code level and then transforms them to several possible MBA equivalents. At its most basic, default settings, it applies a transformation to each operation it encounters (addition or subtraction) but does not support multiplication or division. Tigress also includes options for increasing the size and complexity of its MBA outputs using depth level (effectively, the number of times it applies transformations on each operation), splitting the output into multiple pieces, and adding opaque predicates, among others[14]. Its MBA transformations are derived from a set of boolean identities detailed in the book Hacker's Delight[14][16].

Tigress' transformations have been used in several works on MBA, including Syntia[7], a synthesis-based tool for software deobfuscation, QSynth, a later work which refines and builds upon the concept of oracle-driven synthesis for simplifying MBA expressions[8], and Saturn, a deobfuscation tool which lifts binaries to LLVM and applies compiler optimizations on them for deobfuscation[15]. However, it has a limited number MBA rewriting rules, containing 16 unique MBA expressions which are reapplied, making it prone to pattern matching attacks[7].

2) MBA Obfuscator: MBA-Obfuscator[3] attempted to expand the complexity and resilience of MBA obfuscations by introducing non-linearity. Where linear MBA is defined as a sum of terms consisting of a boolean operation combined with a constant, polynomial MBA extend this definition to include the multiplication of a boolean expression with each term. Polynomial MBA are much harder to factor than linear MBA, and several prominent MBA deobfuscation tools are unable to solve them. Fig. 2 presents a polynomial MBA as shown by [4]. Unlike the linear MBA shown in Fig. 1, the polynomial MBA may include multiplied boolean terms, for instance, "$(x \wedge y) * (x \vee y)$".

$$(x \wedge y) * (x \vee y) + (x \wedge \neg y) * (\neg x \wedge y) - 41$$

Fig. 2: An example of Polynomial MBA.

MBA Obfuscator describes methods to generate unlimited polynomial MBA which are difficult to factor and reverse, including term rearranging, recursive rewriting, the use of linear combinations. Many MBA tools including NeuReduce and MBA-Blast are unable to simplify the resulting poly MBA, and can only reduce the linear MBA in the MBA Obfuscator dataset[3][5]. Newer works which build upon the MBA-solving and program synthesis techniques in [4] [7][8], SiMBA[12] and GAMBA[5] are able to solve both the polynomial MBA and linear MBA within the MBA Obfuscator dataset. The dataset includes over one thousand poly MBA samples and several linear ones.

3) Loki: The 2023 obfuscation tool Loki, which primarily focuses on virtualization obfuscation, also uses MBA to diversify its handlers. It generates a large number of semantically equivalent arithmetic expressions using synthesis to verify their accuracy, and then randomly selects from them to recursively rewrite expression terms up to a given complexity level. The dataset includes 30 "depth" levels, where the depth indicates the number of times an expression has had recursive rewriting applied[11]. In total, Loki's MBA dataset includes approximately thirty-thousand individual samples; of these, the authors find that MBA-Blast is only able to simplify 0.5% of these.

B. The Use of Optimization Techniques in MBA Deobfuscation

In her thesis' thorough exploration of MBA formula, Eyrolles observes several MBA which may be removed post-compilation, and discusses some changes which might strengthen them[1]. However, the work does not perform an in-depth survey of the effects of compilation on MBA, and since publication, new MBA datasets have been created. As far as we find, no work has performed a comprehensive survey of industry compiler effects on MBA. Eyrolles also discusses works which incorporate compiler optimization theory to reduce MBA, but these are highly specific tools which are constrained by several limitations which make their general use infeasible. For example, Souper, a superoptimizer which can synthesize MBA expressions, requires prior knowledge about the MBA expressions[1][17].

Another work which takes advantage of LLVM optimization is Saturn, which performs general deobfuscation covering MBA, virtualization, and ROP-chain obfuscation. Its authors find that existing compiler optimizations are quite effectively able to simplify obfuscated code by removal of junk instructions.[15]. However, this work similarly does not include a comprehensive evaluation of MBA which survive compilation, as it is somewhat outside of its scope as a general deobfuscation tool using LLVM.

Aside from Saturn, Souper, and MBA-Blast, most MBA generation and evaluation tools work at the expression or C source code level, rather than at the binary level and do not test the survival rate of MBA post-compilation.

We thus identify a hole in the state of MBA research concerning the real-world reliability in MBA works. Although a range of complex, dense MBA samples are available, their practicality of use in executables is less clear. Evidently, a work examining this is needed, as it has major security implications for the use of MBA as a protective mechanism in software.

C. Compiler Optimizations

Popular industry compilers typically include optimizations to speed the execution of generated programs by default, as well as additional settings for reducing the size or speed of compiled code. Optimizations can perform arithmetic simplifications, function inlining, and compressing functions of small size into main code when the functions are smaller than typical assembly prologue and cleanup code[18][19].

Particular optimizations such as constant folding, dead code removal, and constant propagation are especially effective at simplifying arithmetic sequences. These have been an established practice for several decades and the focus of several works on compiler theory[20][21]. They are so ingrained that the GCC, a widely-used compiler initially released in 1987[22], applies them by default, and Clang, another popular compiler, applies them during its first pass of its compilation[19]. Additional "aggressive" levels, enabled by special compiler settings, apply additional rounds of these optimizations at higher levels of intermediate representation(IR), which enable much more efficient optimization.

These optimizations and additional options vary across compiler type; we describe them in more detail below.

D. GCC

By default, GCC performs basic arithmetic optimizations such as constant folding and dead code elimination with no additional optimizations enabled. GCC, as part of its compilation process, also transforms code to GIMPLE, its intermediate representation (IR), and its backend representation Register Transfer Language (RTL). At more aggressive settings, GCC performs arithmetic optimizations on these representations in order to achieve maximum efficiency.[18]

GCC provides optimization options for both size and general performance improvements, typically at the cost of higher compilation time. Most of these settings concern program structure, including compressing functions and skipping function stack setup where possible, optimizing loop structures, inlining code, and many others; however, they also enable arithmetic simplification on its intermediate representations.

E. Clang

Like GCC, Clang includes basic arithmetic optimizations by default, and only enables more intensive transformations at higher levels. Although most of Clang's size and speed settings also concern general code reduction not relevant to MBA simplification, there are several enabled arithmetic optimizations, including those performed on Clang's intermediate representation, "LLVM-IR"[19].

F. MSVC

The MSVC compiler includes two families of optimizations which separately optimize for speed and size. By default, when optimizations are enabled MSVC tends towards speed-improving optimizations. Conversely, MSVC can also optimize for smaller binaries (fewer instructions), sometimes at the cost of compilation time and overall program performance, using /O1.

III. Implementation

For our MBA compilation analysis, we pass a selection of MBA through three industry compilers, GCC, MSVC, and Clang, with and without additional optimizations, and then examine the compiled MBA expressions for comparison.

In this section, we detail our process for compiling the MBA and then lifting the result from the binary level for comparison. We also describe some of our techniques for measuring complexity and size reduction.

A. Complexity Metrics

Measuring MBA complexity varies across usage examples and generally has no singular definition[1]. The authors of MBA-Blast, which can also lift MBA from the binary level, [13] describe measuring the semantic complexity of MBA expressions by representing them as trees or as directed acyclic graphs (DAG)s and then examining the number of leaf nodes(constants) and operations.

Eyrolles defines a measure of complexity using DAGs where a DAG node is an operation between two other nodes and does not appear more than once in any given graph. Instead, duplicate operations are replaced with directed links which point to a single instance of that operation. This provides a more precise definition of the MBA's complexity, as once a single term is solved, identical terms may easily be rewritten as the solved term, without requiring repeat analysis[1].

We use a hybrid approach and record several metrics across our samples: the overall number of operations, the number of DAG nodes as defined by Eyrolles, the number of DAG node reductions across MBA datasets, and the proportion of DAG nodes to total operations, as this gives insight to how many operations within the MBA are unique or repeated.

Fig. 4 shows a sample AST representation of the MBA expression `(A + B) ^ (A - (A + B))`. The total number

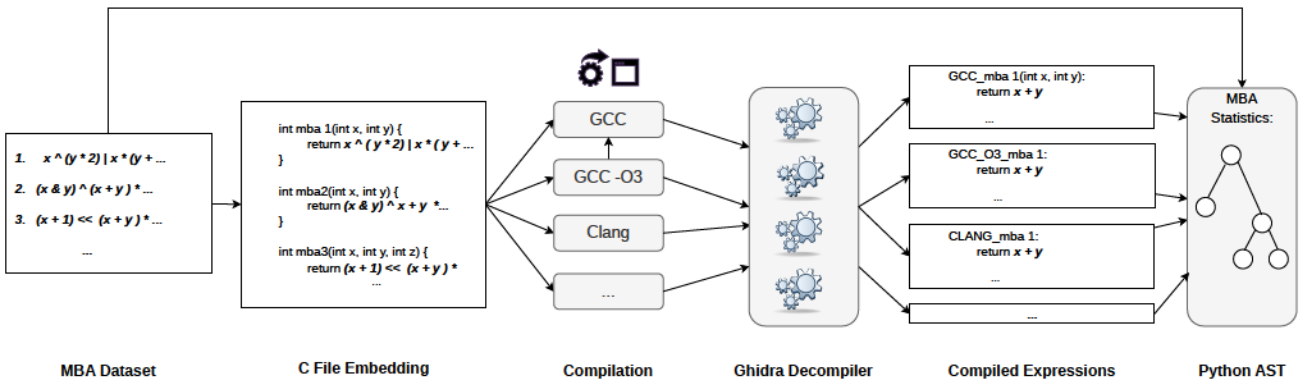

Fig. 3: Overview of MBA Analysis Pipeline

of operations for this expression is 4, but the number of unique sub-trees, or DAG nodes, is 3. Boxed operations represent duplicate appearances of a DAG node.

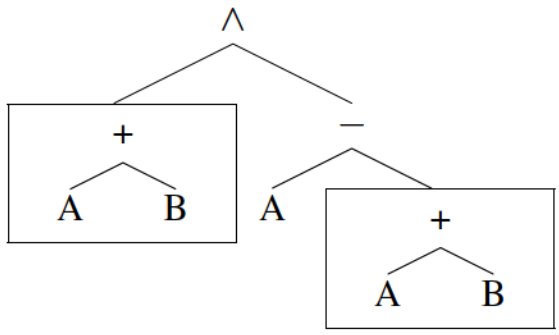

Fig. 4: An MBA expression represented as an Abstract Syntax Tree (AST). Duplicate nodes, which comprise a single DAG node, are indicated by the boxed branches.

B. Pipeline

Our method is as follows: First, we create a C file for the MBA dataset in which each MBA is returned from a function. We compile this C file using the GCC, Clang, and MSVC compilers, first with no optimizations enabled, and then with aggressive optimization and size-minimization settings. Then, we use the Ghidra decompiler[23] to lift each function back to expression level representation and extract the post-compilation MBA for analysis using the Python AST library. Fig. 5 presents an example of a decompiled MBA expression in Ghidra's format and in our lifted format as it is stored in our dataset. Once we have the expression represented as an AST, we perform additional processing to calculate the number of DAG nodes, total operations, and other metrics.

```
int mba_5(void)
{
  int in_R9D;
  int param_8;

  return (param_8 | in_R9D) * 2 -
    (in_R9D ^ param_8);
}


mba_5:
    (e | c) * 2 - (c ^ e)
```

Fig. 5: An example of Ghidra decompilation output for an MBA expression from the QSynth dataset, and our representation post-conversion its original terms.

At higher optimization levels, some expressions are split up into variables which are reused in the returned expression. This affects how unique terms and the total number of operations may be calculated within an expression compared to its pre-compiled form, and so to ensure the best comparison, we combine these variables into a single expression to be lifted into the Python AST library. Fig. 3 illustrates our overall process.

We used compiler versions: MSVC 19.32 for x86 on Windows 10, Clang 14.0.0 on Ubuntu, and GCC version 11.4.0 on Ubuntu. For our analysis, we used Python version 3.12 and Ghidra 11.03.

IV. Evaluation: Compiler Optimization on MBA

For our evaluation, we compile expressions from three large datasets using GCC, MSVC, and Clang and detail the changes in expression complexity. We explore traits of the most resilient MBA and discuss those which are most likely to affect resistance to compilation optimizations.

A. Dataset

We test on three MBA datasets from notable works, including the full datasets from the greybox synthesis deobfuscator

TABLE I: MBA Removal and Reduction Post-Compilation by Number of Operations
(No. Removed – No. Reduced)

MBA (No. Samples)	GCC	GCC - O3	Clang	Clang - O3	MSVC	MSVC - O1
Loki D1 (1250)	1019 – 1215	1160 – 1230	78 – 723	1182 – 1229	266 – 843	772 – 1127
Loki D30 (1250)	150 – 1250	286 – 1250	1 – 1245	637 – 1249	10 – 1247	100 – 1250
QSynth (501)	8 – 496	10 – 497	1 – 416	20 – 496	3 – 441	7 – 485
Poly MBA (1008)	1 – 481	1 – 496	0 – 197	1 – 926	1 – 276	1 – 876

Qsynth[8], which is generated using Tigress, the polynomial MBA from MBA-Obfuscator[3], and a selection of MBA from the obfuscation tool Loki[11]. For the Loki dataset, which consists of expressions ranging from 1 to 30 levels in complexity for 5 operations, we randomly selected 250 MBA expressions from all operation types at levels 1 and 30 for each available operation, totaling 2500 expressions. Across all datasets, we test 5007 MBA expressions.

B. Compiler Settings

We evaluated the effects of compilation using popular industry compilers GCC, Clang, and MSVC.

For compiler optimizations, we select GCC and Clang's default and aggressive arithmetic 7optimizations (O3). For MSVC, we test at default settings and with O1, which favors for size reduction and employs aggressive optimizations similar to that of GCC and Clang's O3 settings.

C. Results

Reduction and Removal of MBA Overall, we find a remarkable reduction in MBA across most datasets when compiled with or without optimizations, with a notable exception in the Poly MBA dataset.

Table I shows the number of complete simplifications and reductions across datasets, where the compiled number of operations is the same as that of the ground truth, and partial reductions, respectively. The Loki dataset exhibits the greatest number of reductions, with a majority being removed entirely by Clang's aggressive optimization settings. All of Loki's expressions at the highest complexity level are reduced to a degree. Although only a small number of QSynth expressions are fully reduced to the original size of the ground truth, the majority of expressions experience some reduction in size. The Poly MBA exhibits the least reduction, with only one expression fully reduced and less than half of the dataset being partially reduced by two of the compilers at default settings.

Some QSynth expressions were smaller after compilation than the original ground truth. This happened in instances where the ground truth expressions had some terms which could be easily simplified, for instance, $((a - a) + a)$. The smallest number of expressions simplified in this way was seen with Clang at default settings with three expressions, and the largest with Clang at aggressive optimizations, which reduced 73 expressions to smaller than their ground truth.

TABLE II: Average Percent Size Decrease in Total Operations

Dataset	GCC - O3	Clang - O3	MSVC - O1
Loki D1	2.1%	1.5%	10.8%
Loki D30	69.9%	46.7%	73.9%
QSynth	62.6%	65.1%	45.9%
Poly MBA	4.3%	17.3%	12.6%

D. Examining Compiled MBA Traits

Table II shows the average percent change in DAG nodes for samples which were reduced in size, excluding samples which were fully reduced to ground truth size. MBA datasets which use repeated expression rewriting, including Loki and QSynth, appear to be more susceptible to reduction. Conversely, Polynomial MBA, which allows for the multiplication of boolean expressions across terms, appears to be the most resilient of all datasets. Loki Depth 1 at a glance experiences a lower reduction rate, but this is likely because only a few samples are measured.

Negated Terms In a number of datasets, we found sets of operations applied to inputs which essentially "cancel out", or more specifically, are easily identified as being equivalent to zero or to one of the original input variables. An example of this is shown in Fig. 6, which contains a portion of an MBA expression from the QSynth dataset. In this expression, $d*(d\oplus d)$ is equivalent to zero, which renders the overall term equal to zero. These terms, which we call "negated" terms as they essentially self-cancel, add to the overall size of an expression, but ultimately are easily removed by compiler optimizations.

$$(a * d) \oplus (a - d)) * ((d * (d \oplus d))$$

Fig. 6: An example of an MBA term which is easily simplified.

We measured the proportion of DAG nodes for each expression which constitutes a negated term and show them in Table III, to determine how much this factor may contribute to the overall size reduction of each dataset. Overall, the Loki and QSynth datasets exhibit the highest number of negated terms, which are greatly reduced by aggressive compiler optimizations. The Poly MBA contains the lowest number of negated terms and shows a very small reduction. Recall that DAG node measurements do not include duplicate terms, and

TABLE III: Average Proportion of Negated Terms

Dataset	MBA	GCC O3	Clang O3	MSVC O1
Loki D1	15.7%	6.9%	5.6%	5.8%
Loki D30	27.6%	15.0%	9.4%	16.8%
QSynth	15.6%	5.0%	4.4%	5.6%
Poly MBA	6.6%	5.6%	6.4%	5.9%

so the overall total proportion of negated terms for expressions is likely higher.

Unique vs. Repeated Terms A trend that we found among MBA samples which demonstrated high resistance to compiler optimizations is a low proportion of repeated terms. We measure this by measuring the number of DAG nodes compared to total operations. For example, the term "(x + y) — a + (x + y)" has four total operations, but only two unique ones; "(x + y)" is repeated twice and makes up a significant proportion of the expression. Table IV shows the average proportion of unique operations across MBA datasets pre-and post-compilation, calculated by averaging the ratio of DAG nodes to total operations per expression. With the exception of the Loki Depth 1 dataset, which contains very small expressions, the datasets range from averaging 27% to 78% unique terms per expression. Post compilation, the proportion of unique terms tends to be higher, suggesting that compiler optimizations are effective at identifying and removing duplicated expressions. The Poly MBA dataset's overall low reduction by optimizations may be partly explained by its relatively high proportion of unique terms.

TABLE IV: Average Proportion of Unique Terms

Dataset	MBA	GCC O3	Clang O3	MSVC O1
Loki D1	98.0%	100.0%	100.0%	99.8%
Loki D30	29.8%	77.2%	90.0%	60.6%
QSynth	34.6%	64.5%	65.7%	54.3%
Poly MBA	82.6%	84.6%	87.2%	86.8%

Average Operation Types We also look at the average proportion of operation type (arithmetic and boolean) across each dataset. Because the Loki and QSynth datasets contain little multiplication, while the Poly MBA dataset employs it heavily due its polynomial nature, we list multiplication separately. Fig. 7 and Fig. 8 show the average proportions of operation type across each post-aggressive compilation. In general, arithmetic (addition and subtraction) operations go through the greatest reduction after compilation, with multiplication being second; for the most part, the greatest percentage of remaining terms consists of boolean operations. This aligns with existing research stating that boolean operations themselves are difficult to simplify.

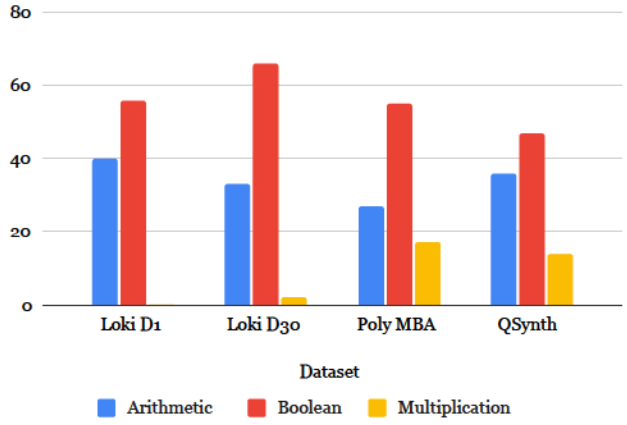

Fig. 7: Average Proportions of Operation Types Before Compilation

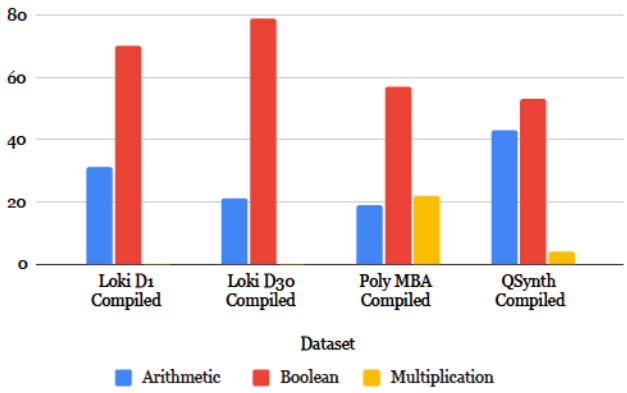

Fig. 8: Average Proportions of Operation Types After Compilation

E. Summary

Overall, we find that many popular MBA expressions can be effectively reduced by compiler optimizations, which has tremendous implications for their usage as a security feature. Some factors which likely contribute to susceptibility include duplicated or easily negated terms. Based on these findings, we theorize that MBA generation may be strengthened by rigorous term rewriting, and suggest this as potential future works.

V. Conclusion

We collected the most popular MBA datasets in recent years and surveyed their resilience through compilation. We used GCC, MSVC, and Clang at both default and aggressive optimization settings and discovered that many MBA expressions may be dramatically reduced in size or even removed entirely, even with no optimizations enabled. Aggressive optimizations are quite effective in simplifying MBA expressions.

Acknowledgments

We would like to thank the reviewers for their insightful feedback. This research was supported by NSF grants 2211905 and 2022279.

REFERENCES

[1] N. Eyrolles, "Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools," Theses, Université Paris Saclay (COmUE), Jun. 2017. [Online]. Available: https://theses.hal.science/tel-01623849

[2] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *Information Security Applications*, S. Kim, M. Yung, and H.-W. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75.

[3] B. Liu, W. Feng, Q. Zheng, J. Li, and D. Xu, "Software obfuscation with non-linear mixed boolean-arithmetic expressions," in *Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I 23*. Springer, 2021, pp. 276–292.

[4] D. Xu, B. Liu, W. Feng, J. Ming, Q. Zheng, J. Li, and Q. Yu, "Boosting smt solver performance on mixed-bitwise-arithmetic expressions," ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 651–664. [Online]. Available: https://doi.org/10.1145/3453483.3454068

[5] B. Reichenwallner and P. Meerwald-Stadler, "Simplification of general mixed boolean-arithmetic expressions: Gamba," in *2023 IEEE European Symposium on Security and Privacy Workshops (EuroSP;PW)*. IEEE, Jul. 2023. [Online]. Available: http://dx.doi.org/10.1109/EuroSPW59978.2023.00053

[6] A. Guinet, N. Eyrolles, and M. Videau, "Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions," in *GreHack 2016*, 2016.

[7] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[8] R. David, L. Coniglio, and M. Ceccato, "Qsynth - a program synthesis based approach for binary code deobfuscation," 01 2020.

[9] W. Feng, B. Liu, D. Xu, Q. Zheng, and Y. Xu, "NeuReduce: Reducing mixed Boolean-arithmetic expressions by recurrent neural network," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 635–644. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.56

[10] A. Petrov, "Hands-free binary deobfuscation with goomba," 2023. [Online]. Available: https://hex-rays.com/blog/deobfuscation-with-goomba/

[11] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening code obfuscation against automated attacks," in *USENIX Security Symposium*, 2022.

[12] B. Reichenwallner and P. Meerwald-Stadler, "Efficient deobfuscation of linear mixed boolean-arithmetic expressions," in *Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, ser. Checkmate '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 19–28. [Online]. Available: https://doi.org/10.1145/3560831.3564256

[13] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and simplifying mixed Boolean-Arithmetic obfuscation," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1701–1718. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/liu-binbin

[14] C. Collberg, "Tigress - encode arithmetic," 2023, [Online: accessed 10-May-2024]. [Online]. Available: https://tigress.wtf/encodeArithmetic.html

[15] P. Garba and M. Favaro, "Saturn – software deobfuscation framework based on llvm," 2019.

[16] H. Warren, *Hacker's Delight*. Addison-Wesley, 2003.

[17] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," 2017. [Online]. Available: https://arxiv.org/abs/1711.04422

[18] A. Petrov, "A gnu manual," [Online: accessed 11-May-2024]. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[19] "Clang user manual," [Online: accessed 05-19-2024]. [Online]. Available: https://clang.llvm.org/docs/UsersManual.html

[20] J. Knoop, O. Rüthing, and B. Steffen, "Optimal code motion: theory and practice," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, p. 1117–1155, jul 1994. [Online]. Available: https://doi.org/10.1145/183432.183443

[21] J. Cocke, "Programming languages and their compilers," 1969. [Online]. Available: https://api.semanticscholar.org/CorpusID:61034705

[22] "Gcc releases," [Online: accessed 29-May-2024. [Online]. Available: https://www.gnu.org/software/gcc/releases.html

[23] "Ghidra," [Online: accessed 25-May-2024]. [Online]. Available: https://ghidra-sre.org/