MBASED: Practical Simplifications of Mixed Boolean-Arithmetic Obfuscation

Nitin Krishnaswamy
nitinkrish101@gmail.com

Sanjana Mandadi
sanjanamandadi@gmail.com

Micah Nelson
micahanainelson@gmail.com

Timothy Slater
t.slater0001@gmail.com

Benson Liu*
bensonhliu@gmail.com

New Jersey's Governor's School of Engineering and Technology
July 26, 2024

*Corresponding Author

***Abstract*—Mixed Boolean-arithmetic obfuscation is a technique that transforms simple Boolean expressions into complex expressions by combining arithmetic and Boolean operations. It serves the purpose of making code more difficult to analyze, protecting software from reverse engineering and tampering. Conversely, mixed Boolean-arithmetic deobfuscation aims to simplify obfuscated Boolean expressions. In this paper, we present a Binary Ninja plugin called MBASED (Mixed Boolean-Arithmetic Simplification Engine for Deobfuscation) that performs mixed Boolean deobfuscation within C programs to assist reverse engineers in understanding them. We first created a system that processes Boolean expressions and converts them into a parse tree. We then employed the Python library, SymPy, and the SMT Solver, Z3, to simplify the Boolean expressions, and print the results to the Binary Ninja console. MBASED substantially decreases the number of variables and operations within Boolean expressions, indicating Boolean simplification.**

I. Introduction

Reverse engineering is the process by which systems, such as computer malware, are understood. It is generally used to protect against future malware attacks as well as to assess the vulnerabilities of software. This is especially important because the nature and methods of cyber attacks are continuously evolving to bypass evolving security measures. This perpetual "arms race" between malware writers and cybersecurity experts has opened up many new areas of research [1].

One important tool that is used in reverse engineering is decompilers. Decompilers are programs that translate machine code (binary 1s and 0s) back into source code (in this case, C), so that it can be read by humans. Although it is useful for reverse engineers, the process of decompilation is particularly difficult, and decompilers rarely give the complete picture of what the original source code looked like.

However, as progress has been made in reverse engineering techniques, malware developers have also made the process more difficult with new anti-debugging practices. One notable practice is obfuscation, in which programs are made more difficult to understand through complicated logic. It is important to note that the added complexity does not change the performance or behavior of the programs. The obfuscation technique that we will be addressing in this paper is known as mixed Boolean-arithmetic (MBA) obfuscation, which is the practice of making Boolean expressions more complicated.

Deobfuscating these algorithms has been an ongoing challenge for reverse engineers, although progress has recently been made through tools like MBA-Blast, in which researchers proved that arithmetic reduction rules can apply to MBA expressions [2].

This paper discusses the development of the first practical implementation of MBA deobfuscation in an industry-grade decompiler — MBASED. In this case, the plugin we have created will be used for a decompiler known as Binary Ninja, and it will be easy to integrate into this platform because no parts of the decompilation process are modified. Much of this plugin's system design matches that of a normal compiler, and it is benchmarked against examples used by prior works like Obfuscator LLVM [3].

II. Background

A. Reverse Engineering

Reverse engineering is the process of analyzing a system to understand what it does. In the context of software, an important application of reverse engineering is malware analysis, which is what developers use to improve the protection of their

programs. There are two different types of analysis and each has its uses.

1) Static Analysis: Static analysis is a way of assessing systems or programs for errors, vulnerabilities, or simply to understand their function, as is done with reverse engineering. In the context of computer science, static analysis refers to the examination of code without any actual execution [4]. It is particularly useful when assessing code for risks before deployment or for finding issues early on in development. One of the drawbacks of static analysis, however, is the fact that it can either be applied to very specific cases, or it must use approximations for more general cases. The reason for this limitation is what is known as the Halting Problem. The Halting Problem is a concept that essentially looks for a way for a program to predict whether another program will stop (i.e. "halt") or continue infinitely [5]. It was proved undecidable by Alan Turing, as there is no possible algorithm that will always correctly be able to do this for any given program and inputs. Static analysis tools attempt to do just this, but the Halting Problem limits them from being completely accurate.

2) Dynamic Analysis: Dynamic analysis is the process of assessing code during runtime to check for errors during the code's performance. This is done by monitoring the program as it runs, using many different test cases with different outputs, and profiling, which is the use of specific tools to take in the characteristics of a program and optimize them. This type of analysis is useful for understanding how a program operates in real-world scenarios and may also help in the debugging process if runtime errors occur. The major drawback with dynamic analysis is that it is often impractical to generate inputs that cover the range required to properly test a program, especially if the data provided to generate inputs is limited. Both types of analysis have their advantages and can be used in different contexts.

Despite its limitations, the reason that static analysis is still used both in this paper and in the real world is because, at very large scales, dynamic analysis is simply not practical or cost-effective. It is far easier to get a comprehensive view of a program without executing it than it is to cover all of the necessary test cases during execution. Additionally, although it is challenged by the Halting Problem, static analysis still provides good approximations for our purposes. We employ many methods to improve the accuracy of our static analysis as well.

B. Compiled and Interpreted Languages

In computer science, compiled languages refer to programming languages that are converted from human-readable code to machine code (i.e. "compiled") directly in the environment where they are executed. There are two different "times" during which code is generally compiled. They are known as Ahead of Time (AOT) compilation and Just in Time (JIT) compilation. As the names suggest, AOT compilation occurs before the program is executed, typically during software installation, and JIT occurs right before the program is executed [6]. Two examples of compiled languages are C and C++. Interpreted languages, conversely, are executed line by line without being converted to machine code. Python is an example of an interpreted language.

The main focus of this paper will be in the context of compiled languages, as decompilers are the tools that are used to translate the machine code back to source code. The problem with decompilers is that it is extremely difficult to properly represent the source code in a human-readable way after it has been compiled. A lot of the information from the original code is lost in translation, which is part of what makes reverse engineering so difficult. There will always be some information missing that may paint a different picture of the program, so reverse engineers are left to make educated assumptions and interpretations about the code.

C. Compilers

The function of a compiler is to convert high-level source code into low-level machine code, which is defined by a specification called the Instruction Set Architecture (ISA).

The general structure of the compilation process can be separated into passes. A pass occurs when the compiler reads and transforms the entire program.

The first pass is called the analysis phase, or the front end of the compiler. During this phase, the compiler checks that the program follows the correct grammar rules for the programming language and verifies that the code has logical sense, as well as correct type definitions for variables.

The beginning of the analysis phase focuses on lexical analysis. The lexer, also called a scanner, reads the input code and creates a list of tokens that identify each data type defined in the programming language. For example, the lexer identifies which parts of the program are operators ($=$, $!=$, $<$, and $>$) or identifiers (which represent variable names) and returns a list of these basic components, that make up the program. Then the lexer, or another device called a screener, removes tokens that are unnecessary for compiling the program and transforms some tokens into different representations. For instance, whitespace would be removed, while identifiers could be changed into a sequence of numbers for efficiency. Information about the identifiers, such as the values that the identifiers represent, is recorded in a symbol table.

The next parts of the analysis phase are syntactical and semantic analysis. During syntactical analysis, a parser analyzes the program's syntactic structure and recognizes potential errors within the program. Semantic analysis verifies the correctness of the program by ensuring that valid types are used with their corresponding expressions, variables are declared before use, and the string's expression contains valid control flow. For example, the compiler checks that the type of an expression is consistent with the operations the expression is associated with. Both analysis steps rely on a rule set known as a context-free grammar. As our system utilizes context-free grammars during the parsing phase as well, this concept is explained in greater detail in the *System Design* section.

The tokens are then mapped to an intermediate data structure known as a parse tree.

From here, the compiler may perform a pass that focuses on code optimization, called the middle-end. The purpose of optimization may focus on increasing program execution speed or decreasing the consumption of memory and energy. However, the optimization phase may be omitted for simplicity or performance.

The final pass is code generation, which is performed by the back end. The compiler traverses the structure of the program recursively as it converts the intermediate representation into machine code or assembly, which is the human-readable representation of machine code. The code generator must allocate a limited number of registers to variables and intermediate values within the program since registers make accessing values more efficient. The code generator also works with instruction selection; it forms instruction sequences within the program, based on the ISA.

D. Decompilers

Conversely, decompilers attempt to lift source code from machine code. They function similarly to compilers but with the lexical analysis phase omitted.

After the decompiler creates an intermediate code representation of the program, it develops a control-flow graph of subroutines within the program. The control-flow graph assists the decompiler with determining the high-level control structures defined in the program. It also eliminates intermediate jumps left by the compiler (jumps are responsible for making the program execute a different section of code). Then, data flow analysis is conducted to improve upon the intermediate code, so that high-level language expressions can be found. The control flow analyzer uses the control flow graph to structure the subroutines within the program into high-level language constructs. [7] Finally, the high-level program code is generated.

Decompilers are used for multiple purposes, such as recovering lost source code, reformatting unstructured code, and malware analysis. Reverse engineers often read disassembly, which is shown in assembly language. However, since assembly masks the semantics of the original high-level program, decompilers can provide more information about the program's control flow and data flow.

E. Binary Ninja

Binary Ninja is an industry-standard reverse engineering platform that includes a suite of tools to aid reverse engineers in interpreting code through its interactive decompiler. One of its unique features that is important for this project is its powerful plugin interface. Binary Ninja has a Plugin Manager that is both easy to use and accessible across different platforms, meaning that MBASED can be efficiently implemented by anyone who needs it. Additionally, Binary Ninja applies a feature called Single Static Analysis (SSA) form, which forces variables to only be assigned values one time. If that value changes at any point, a new version of the variable is created. This is helpful for data-flow analysis, as it becomes easier to keep track of the variables. Binary Ninja also employs what is known as Binary Ninja Intermediate Language (BNIL) [8].

1) BNIL: In the process of deobfuscating mixed Boolean-arithmetic, the first transformation of the input takes the machine code and "lifts" it to an intermediate representation of the code known as BNIL. This is an in-between form of the code that Binary Ninja uses in the process of moving from machine code to source code, which in this case is C. Within BNIL, there are multiple different "levels" to choose from, each with their own degree of complexity. Low-level IL is the intermediate representation that is closest to machine code. It is mainly used for simple debugging purposes. Medium-level IL is easier for humans to understand and provides a more simplified representation of the code. High-level IL is closest to source code, with even more abstraction. It is often the most convenient level for reverse engineers to use, although the other levels also provide unique information that may be important, depending on the objective of using the intermediate languages.

F. Code Obfuscation and LLVM

Code obfuscation is the process of rewriting code in a form that is functionally identical to its source with the intent of preventing reverse engineers from interpreting the altered code's function. Many types of commercial software are often obfuscated to deter potential threat actors from finding vulnerabilities by reading the source code. For example, while a financial technology such as Paypal may want to obfuscate its code to prevent criminals from hijacking funds from other users, a video game development company such as Epic Games may want to obfuscate their game's code to prevent cheating software from interacting with the game's function. Obfuscation is also used by threat actors for malicious operations. Malware is often obfuscated to inhibit security researchers' ability to assess the function and scale of its effect. Additionally, obfuscated malware is often more difficult for anti-virus and other security tools to detect [9].

As it stands, obfuscation is an easier and more researched field than deobfuscation, which is the process of simplifying code that has been obfuscated.

One important development in obfuscation is the Obfuscator-LLVM (Low-Level Virtual Machine) tool. Obfuscator-LLVM works by building an intermediate representation of the code, which is then given to a middle-end optimizer that performs the obfuscation mechanisms. This can involve several different methods:

1) Instruction Substitution: LLVM takes simple, standard instructions within the intermediate language and replaces them with more complicated instructions that do the same thing (MBA falls in this category).

2) Data Obfuscation: It is very common for data within programs to be hidden from viewers with LLVM. Typically, this data includes variable names and their values, strings, constants, etc.

3) Dead Code Insertion: Oftentimes, extra code that changes nothing about the execution of a program is added to make it harder to read. The dead code tends to be very complex, despite not impacting the program's function.

4) Control Flow Obfuscation: The control flow of a program is oftentimes altered through iteration and conditionals, and it is made more readable by using modularity. LLVM can make the control flow of a program more confusing by doing things like "flattening" the control flow chain, or making it linear, and thus less useful [3].

G. Mixed Boolean-Arithmetic Obfuscation & Boolean Simplification

Mixed Boolean-Arithmetic expressions are hybrid expressions that include both Boolean operators (e.g., `AND`, `OR` and `NOT`), and arithmetic operators (e.g., addition, subtraction, and multiplication). Using Boolean Algebra, the form of mixed Boolean-arithmetic expressions can be altered [10].

Boolean Algebra has numerous identities that can be used to reformulate an expression into functionally equivalent alternatives. Rules for altering a function in this way plenty: DeMorgan's law, the distributivity law, and the associativity laws are all examples of these identities. While respecting the precedence of each part of a Boolean expression, we can repeatedly expand or reduce expressions by replacing their terms with functionally equivalent sub-expressions. The characteristics of these rules make Boolean algebra an intuitive choice for an obfuscation method. Thus, mixed Boolean-arithmetic identities can be applied to obfuscate code. In this method of obfuscation, Boolean expressions within compiled code are expanded to alternative representations to obscure the function and meaning of an expression to reverse engineers. For example, given that the expression in Figure 1 is within a source code file, following the obfuscation process, it will appear as the expression in Figure 2 through a traditional decompiler.

The problem of simplifying mixed Boolean-arithmetic expressions into their simplest form is considered NP-Hard (nondeterministic polynomial time-hard), meaning that there is no single algorithm that can solve this problem in polynomial time. Fully simplifying a mixed Boolean-arithmetic expression yields an exponential number of possibilities. As a result, simplifying Mixed Boolean-Arithmetic expressions remains an open research issue.

```
!A & !B
```

Fig. 1: Original expression

Boolean simplification algorithms are methods of reducing Boolean expressions to their simplest form. In practice, this is important for digital logic, where circuits must use the fewest possible logic gates to maintain efficiency. Some of the fundamental Boolean simplification identities are as follows:

1) The Identity Law:
   - $A \mid true = A$
   - $A \mid false = A$
2) The Idempotent Law:
   - $A \,\&\, A = A$
   - $A \mid A = A$
3) The Inverse Law:
   - $A \,\&\, !A = false$
   - $A \mid !A = true$
4) The Commutative Law:
   - $A \,\&\, B = B \,\&\, A$
   - $A \mid B = B \mid A$
5) The Associative Law:
   - $(A \,\&\, B) \,\&\, C = A \,\&\, (B \,\&\, C)$
   - $(A \mid B) \mid C = A \mid (B \mid C)$
6) The Distributive Law:
   - $A \mid B \,\&\, C = (A \mid B) \,\&\, (A \mid C)$
   - $A \,\&\, (B \mid C) = A \,\&\, B \mid A \,\&\, C$

*Note that "and" (&) takes precedence over "or" (|) in the order of operations for Boolean expressions.

```
(!A&!B|C&D&(E&(!F^(!G^(!H&!I|J&K&L
&M^(!N&!O&!P^Q)&(!R^(!S^T|U)^(!V^!W
&X^(Y|!Z&A)|!AB&A
&!AD)|!AE&(!AF|AG&(AH|(AI&(AJ^AK^(
!AL^AM)^(AN|!AO^!AP)^AQ|!AR^(!AS^!
AT& AU)^(AV|!AW|AX))))))))))
```

Fig. 2: Obfuscated expression

Two traditional and well-known algorithms that also help in the simplification process are Karnaugh Maps and the Quine McCluskey Algorithm.

1) Karnaugh Maps: Karnaugh Maps, also called K-Maps, are a graphical representation of Boolean simplification. They make it easier to visualize the simplification process and help prevent the human error produced by classical Boolean algebraic simplification. Typically, K-Maps are best utilized for expressions with a small number of variables, as handling larger graphical representations requires more sophisticated software. Additionally, it is easier to manually use K-Maps when the number of variables is kept to a minimum, as the graphs become increasingly labor-intensive as they grow more complex. Despite this, as long as the number of variables is kept under 5, K-Maps are relatively easy to use for simplification by hand.

2) Quine McCluskey: Identified by Willard V. Quine and further developed and optimized by Edward J. McCluskey in 1956, the Quine McCluskey algorithm is a tabular representation of Boolean simplification. While K Maps are preferred for handling smaller expressions, Quine McCluskey is used for more complex expressions with a greater number of variables [11]. This makes them more difficult to use by hand, but

they tend to be easier to implement on a computer. The only drawback to this algorithm is that its computational time grows exponentially as the size of the expression grows, slowing it down significantly.

III. System Overview

This section provides some of the general reasoning behind the decisions we made for MBASED. The full system is explained in the System Design.

A. Threat Model

Within the threat model scope is a deobfuscation plugin designed to reduce the complexity of mixed Boolean-arithmetic (MBA) expressions found within the obfuscated malware code by using several reverse engineering aid tools. The main assumptions made are that cyber threats continue to evolve, current tooling is reasonably effective, and the development environment can be considered secure. The potential threats here are obfuscation, data breaches, insider threats, exploitation of vulnerabilities, reverse engineering, and phishing attacks. The mitigations are by regular updates of the plugin, being strongly encrypted with access control, conducting background checks with continuous monitoring, vulnerability assessment, obfuscation and anti-reversing techniques, phishing awareness training, and MFA. This will be validated and verified by regular reassessment of threats, checking with real-world data, security audits, penetration tests, and feedback from users.

B. Using MLIL

For our plugin, the Medium Level IL (MLIL) was used as the intermediate representation between the machine code and source code. MLIL is the best option because Low-level IL does not represent the Boolean expressions in a way that is similar to source code, and High-level IL has too much abstraction. MLIL is a good middle-ground to extract the Boolean expressions and separately simplify them.

C. Visitor Design Pattern

A design pattern is a concept in engineering used to describe generic, reusable solutions to common problems. They provide a good template that makes it easier to solve these problems in different applications. A design pattern that we used in our program is the Visitor Design Pattern. The Visitor Design Pattern is a type of algorithm that allows us to traverse through objects of different classes without adding significant extra code to those particular classes. Instead, a separate `Visitor` class is created with methods that correspond to each object type that must be ”visited.” When an object is visited, it will have an “accept” method in its class that calls on the corresponding method in the `Visitor` class. Although this design pattern still technically alters the code within the objects’ classes, the implementation for an “accept” method typically is far simpler than any of the `Visitor` methods and likely won’t cause major issues. The main benefit of using the Visitor Design Pattern is that it works as a “template” that makes our code extensible, meaning it can be easily improved upon by others and easily implemented for practical purposes [12].

IV. System Design

This is an outline of the different components of our deobfuscation plugin as well as an explanation of what each part does. Essentially, the obfuscated machine code is run through Binary Ninja to produce the Medium Level IL for the program. This intermediate language is then encoded to restrict all of the expression representations to capital letters for variables and single characters for operations. The encoded expression is then sent to the LL(1) Parser, where it is converted into a parse tree based on our Context-Free Grammar for the expressions. Next, the simplifiers, are used as the tree is traversed to do the main deobfuscation. Finally, the expression is decoded back to its original representation, except now simplified, and it is given back to Binary Ninja.

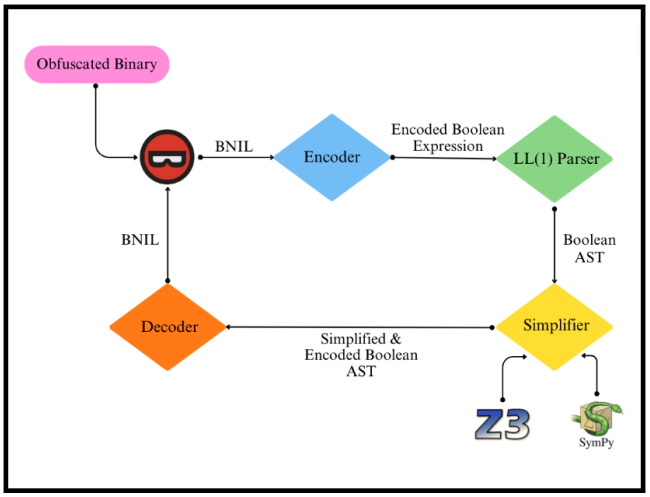

Fig. 3: A general overview of MBASED components

A. Encoder

1) Defining the Encoder: Encoders are integral parts of digital systems as they provide the ability to transform information into a coded format for processing, transformation, and storage. They ease the process of interpreting information by compacting code and preventing repetitions. The primary form of encoders our research will be addressing is a branch of dictionary encoding. Dictionary encoding facilitates the simplification and representation of logical expressions in a more readable form. To accomplish this, encoders replace all expressions with a simple generated key. The end product is a vastly shorter expression. These encoders are particularly useful in situations where multiple conditions need to be represented concisely, such as in state machines, control systems, and decision-making algorithms. In addition to aiding with brevity, encoders are vital in reducing errors to a minimum. Logical expressions often contain multiple operators and variables making them prone to misinterpretations. By converting all logical expressions to a simple key-value, it mitigates the

risk of overlooking these key elements, a costly error, when being run through an algorithm. Encoders allow for modularity in our software. Instead of treating our expressions as one entity, we can instead isolate them. This makes it easier to test and validate the expressions thus reducing possible errors. Encoders provide a highly efficient way to manage logical expressions. By shortening them far beyond what they were initialized as, processing becomes streamlined. This increased speed is vital in critical systems that require quick responses.

2) Initial Setup and Class Definitions: We initiated our research by defining the essential classes and data structures required to handle and manipulate Boolean expressions. This foundational step involved setting up the `NameGenerator` and `DictionaryEncoder` classes, which provided the necessary framework for our subsequent work.

3) Structured Input Processing of Boolean Expressions: To facilitate the systematic handling of Boolean expressions, we created the Boolean class, which stored both the raw and encoded forms of these expressions. Inputs were given in the form of a Medium Level IL (MLIL) string and passed through the class.

4) Streamlining Boolean Expressions by Removing Redundant Information: We identified and removed unnecessary information from the input strings to standardize the Boolean expressions. This would involve analyzing the MLIL input and eliminating all irrelevant data. Only the logical expressions would be extracted.

5) Parsing Boolean Expressions Using Regular Expressions: Utilizing regular expressions, we effectively parsed the Boolean expressions to isolate logical operators and conditions. More specifically, the regex pattern " `(\|\||&&|!(?!\=)|\(|\)|\^)` " was designed to identify operators such as "||", "&&", "!", and "(", ")", and separate the values into a list. This parsing step allowed us to decompose the expressions into their constituent elements, making it possible to handle each component individually. For example, a list of split operators and conditions could look like: ['(', '[ebp_1 + 0x14].b == 0', '||', '[ebp_1 + 0x14].b != 0', ')']. This list would then be used for the encoding process.

6) Generating Unique Identifiers for Boolean Conditions: We developed a method to generate unique identifiers for each condition within the Boolean expressions using the `NameGenerator` class. This class employed the *itertools* and *string* libraries to create unique, recognizable key values. The generate_unique_uppercase_string method produced a sequence of uppercase strings, ensuring that each condition was assigned a distinct identifier. Additionally, each generated code was sequentially created, beginning with "A" and progressing in order (e.g., "B", ..., "AA"). This approach provided a clear and systematic way to encode the conditions.

7) Mapping Conditions to Generated Identifiers: We methodically assigned the generated unique identifiers to their corresponding conditions, storing these mappings in a dictionary, generated_dictionary_keys, within the `NameGenerator` class. This dictionary served as a reference that linked original conditions to their encoded forms, maintaining consistency throughout the encoding process. The generate_name method ensured that each condition was either assigned a new key or matched with an existing one from the dictionary, thereby standardizing the encoding process.

8) Checking Dictionary for Existing Values: To ensure the uniqueness and consistency of our encoded expressions, we implemented a mechanism within the `NameGenerator` class to check whether a given condition had already been assigned a key value. The generate_name method would traverse through the dictionary containing the mappings. If the condition was previously encountered, we reuse the existing key; otherwise, a new key is generated. The dictionary lookup feature enabled efficient and accurate retrieval of keys.

9) Rigorous Testing of the Encoding Process: In the final step of our research, we rigorously tested the encoding process to verify its accuracy and reliability. We created comprehensive unit tests to compare the encoded expressions against expected results.

B. Parser

1) Parsing Rules: We have developed an LL(1) context-free grammar to parse a given Boolean expression and manipulate it in later steps of our process. Context-free grammars are defined by the 4-tuple G, where $G = V_T, V_N, S, P$. V_T represents a set of terminal symbols, each of which is a fundamental term in the defined language (such as an operator like + in a grammar for arithmetic in mathematics). V_N represents non-terminal symbols, each of which can be replaced by sequences of terminal symbols; S is the starting symbol that represents the parent non-terminal of all other terminal or non-terminal symbols (though when written, can be denoted by any non-terminal symbol). P represents the grammar's production rules, the set of rules that assigns a non-terminal symbol to each valid sequence of terminal symbols. To be considered LL(1), a grammar must abide by the following conditions: 1) the grammar must support left to right scanning during parsing, 2) the grammar must support leftmost derivation of its non-terminal symbols, and 3) must be parsable by actively scanning just one symbol ahead at a time to decide what to parse it to. These traits streamline our parsing process by simplifying the steps needed to apply each rule while still maintaining the complexity needed for our scope. Our context-free grammar for Boolean expressions is depicted in Fig. 4.

The non-terminal symbol `Expr` represents binary operators (`AND` and `OR`); the non-terminal symbol Term represents unary the operator `NOT` and also the represents precedence distinguishment through the use of parentheses. The Var non-terminal symbol represents irreducible symbols within an expression which include `True`, `False`, or variable symbols that are determined during the encoding process. The `Expr` is the result of factoring the production rule to eliminate left-recursion, essential for representing Boolean expressions using a valid LL(1) grammar [13]. These rules are then represented and confirmed to be LL(1) through a parse table (See Table 1 in Appendix). If any cell in the parse table contains more than one production rule, the grammar is not considered LL(1).

```
Expr ::= Term Expr′

Expr′ ::= & Term Expr′
        | | Term Expr′
        | ^ Term Expr′
        | ϵ

Term ::= (Expr)
       | ! Term
       | Var

Var ::= [A–Z]+
      | t
      | f
```

Fig. 4: LL(1) grammar rules in Extended Backus-Naur Form (EBNF)

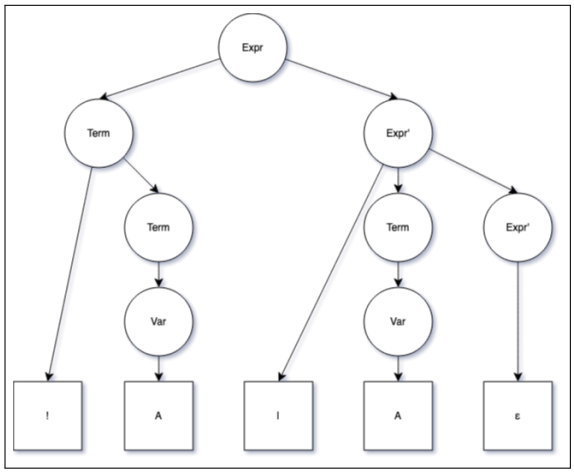

Fig. 5: Parse Tree Visualization

2) Top-Down and Recursive Descent Parsing: Top-down parsers derive a representation of a given text by evaluating from the ”top” of the grammar structure. This ”top”, also known as the ”start symbol”, is the most simplified representation of an input string defined by the grammar. For our grammar for Boolean expressions, this start symbol is the outermost `Expr` non-terminal that encompasses the entire expression string. We have implemented a recursive descent parser, a type of parser that utilizes the top-down methodology. Recursive descent parsers recursively evaluate non-terminal values until expanding them to their terminal value representation. Once the input string is parsed according to a valid LL(1) grammar, it can be converted to an intermediate hierarchical structure called a parse tree.

3) Parse Tree Conversion: Syntax trees represent terminal and non-terminal symbols as nodes that are connected by their common parent nodes. As the recursive descent parser traverses the encoded Boolean-arithmetic expression, it translates it to this intermediate hierarchy-like data structure to represent the anatomy of the expression. The root node – often represented by a “start symbol” non-terminal – begins the tree as the parent node of all nodes representing non-terminal symbols. These non-terminal symbols can also be evaluated to more non-terminal nodes as the parser continues to recurse. Upon expanding a non-terminal node to all terminal nodes, the branch of recursion halts. According to the production rules of the LL(1) grammar, each recursive expansion performed by the parser is represented by a new node in the parse tree. For example, the expression $!\,A \mid A$ can be represented as such:

The expansion of a `Expr` node to `Term` and `ExprPrime` node can be represented by the corresponding classes within our code (See Appendix). This representation of a mixed Boolean-arithmetic expression can be easily manipulated by subsequent components of our process.

C. Simplifier

To simplify a parsed Boolean expression, we pass the representation of the obfuscated expression to a `Solver` object. Upon initialization, the solver object is given the names of the different passes that are to be utilized to simplify the expression. We have implemented two passes that utilize distinct methods to simplify the expression in series. After the first pass completes, the second pass operates on the first’s output, further simplifying the expression into its final form.

1) SymPy: The first pass utilizes SymPy, a free, BSD-licensed Python library capable of manipulating symbolic mathematical operations. SymPy features an implementation of the Quine-McCluskey algorithm, which we have utilized for our results. The resulting output is expressed in either conjunctive normal form or disjunctive normal form.

The conjunctive normal form (CNF) of a Boolean expression is written as

$$E_0 \mid E_1 \mid E_2 \mid \ ... \mid E_{n-1}$$

where each sub-expression E_i contains only the `NOT` or `OR` logical operators.

Conversely, Boolean expressions in disjunctive normal form (DNF) as written as

$$E_0 \,\&\, E_1 \,\&\, E_2 \,\&\, ...\&\, E_{n-1}$$

where each sub-expression E_i contains only the `NOT` (!) or `AND` (&) logical operators [14].

The SymPy module’s simplify_logic function applies the Quine-McCluskey algorithm in two passes: one in CNF and the other in DNF [14] [15]. Upon simplifying the expression to both CNF and DNF, it returns the simplified version of the expression with the fewest symbols.

D. SMT Solvers and Z3

The second pass makes use of a powerful tool called an SMT Solver. SMTs, or Satisfiability Modulo Theories,

refer to the problem of whether or not logical formulas are "satisfiable". What this means is that at least one "model" must exist that satisfies the "constraints" of the logical formula [16]. In other words, the formula must be true in at least one instance. A mathematical example of a satisfiable formula is $x+1=5$. Here, the instance in which this is true is x = 4. On the other hand, an unsatisfiable formula would look like this: $y+5=y$. In no instances can this formula be proven true. SMT Solvers are what is used to determine this satisfiability. Part of the Solver's implementation involves simplifying the formulas to make them easier to evaluate. As such, SMT Solvers can be used to convert complex Boolean expressions into their simplest form.

We utilized an SMT Solver known as Z3 for our deobfuscation program due to its efficiency and accuracy with Boolean simplification. The relevant features of Z3 are as follows [17]:

1) Simplifier: Z3's simplifier applies all of the simplification rules discussed above with the addition of contextualized simplification. This means that it finds equational definitions and reduces formulas using those definitions. An example of this is: x = 4 q(x) $\rightarrow$ x = 4 q(4).
2) Compiler: The compiler for Z3 is the tool that takes the parse tree created by the parser and converts it to a separate data structure composed of congruence-closure nodes, which assert expression equality.
3) SAT Solver: SAT Solvers are tools that determine the satisfiability of the question of whether there is a set assignment of truth values that makes a given Boolean expression true. SMT Solvers typically use the functions of SAT Solvers for their purposes, and in this case, Z3 uses an SAT Solver for things like pruning methods, which reduce search space algorithms.

E. Decoder

1) Defining the Decoder: Decoders are essential components of our system as they perform the inverse function of encoders by converting coded information back into its original format. This conversion process is crucial for interpreting, displaying, or processing the data accurately. Decoders interpret encoded information and convert it back into a more comprehensible form. They are fundamental in systems where data is encoded for efficient transmission, storage, or processing, and subsequently needs to be decoded for use. In our use case, the decoder takes the generated keys and reverts them back to their original logical expression. Decoders ensure that the original information is accurately retrieved, maintaining data integrity and usability. Decoders are necessary to convert the information that our algorithms were processing in the back-end, back into user-readable code. The use of decoders also ensures accuracy as they can consistently replace generated keys with their associated values. This accuracy is critical in applications where precise information is necessary, such as malware analysis. Additionally, the concision and clarity of the decoder make it easier to identify and correct errors in the encoding or transmission processes. Decoders allow our system to continue to simplify expressions while simultaneously preventing the issue of user readability. Decoders simplify system maintenance by providing clear mappings between encoded inputs and their corresponding outputs. This simplicity makes it easier to update and maintain the system, ensuring long-term efficiency.

2) Initial Setup and Class Definitions: We developed the `DictionaryDecoder` class to decode encoded Boolean expressions back to their original form. This class was responsible for reversing the encoding process, using the dictionary of mappings created during the encoding stage.

3) Structured Input Processing of Boolean Expressions: We developed the `DictionaryDecoder` class to decode encoded Boolean expressions back to their original form. This class was responsible for reversing the encoding process, using the dictionary of mappings created during the encoding stage.

4) Parsing Boolean Expressions Using Regular Expressions: Utilizing regular expressions, we effectively parsed the simplified Boolean expressions to isolate logical operators and conditions. We then created a regex expression, `"(\w+|\|\||&&|[!()&|^])"`, to decompose the encoded expressions into individual tokens, making it easier to match them with their original conditions. It would identify Boolean operators such as, "|" and "&" and separate the values into a list. This process was simpler than for the encoder as there are less possible values (due to it being a simplified Boolean expression). For example, a list of split operators and simplified conditionals could look like ['(', '(', 'A', '|', '!', 'A', '&', '!', 'B', ')', '|', '!', 'C', ')']. This list would be used to decode into the original format.

5) Mapping Simplified Conditionals to Original Values: Our `DictionaryDecoder` class takes the dictionary of values that map each simplified conditional to its corresponding MLIL expression and stores it in the mapping dictionary. As our decoder parses through the list of split operators, we locate each identifier's corresponding MLIL in the mapping dictionary and store them in a list, decoded_parts, for later use.

6) Handling the Not Exception: The `NOT` operator, represented by "!", requires special handling as it modifies the conditions it precedes. For instance, in the expression !(x == 3), the result should be x != 3. To manage this, our decoder function first identifies all instances of the "!" operator. Once located, it isolates the condition associated with each `NOT` operator. Using the re library, the function then replaces "==" with "!=" within these conditions, ensuring accurate transformation of the expression.

7) Outputted Expression: After mapping all encoded values to their original conditions, the decoder reconstructed the original Boolean expression. This involved assembling the decoded tokens, including logical operators and conditions, in the correct order to form the complete expression. The end result is the decoded expression which is once again in MLIL format. An example of how this equation would look is: `([ebp_1+0x14].b==0 || [ebp_1+0x14].b!=0)`. This is the user experience after incorporating our plugin.

8) Testing the Decoder: Similar to the encoder, we comprehensively tested our decoder by creating unique test cases that address all possible edge cases. We verified that our decoder works in all required scenarios.

V. Experimental Procedures

The goal of this experiment was to evaluate the impact of obfuscation and subsequent simplification on Boolean expressions by benchmarking the counts of operations and variables at each stage.

A. Approach

We started by first preparing our environment by importing the necessary modules and libraries. We defined utility functions to count the number of operations and variables in a parse tree. These were used in the experiment at different stages in the expression analysis.

We then defined a main function, `run_experiment`, which takes as an argument a list of passes to be used during simplification and the maximum number of variables, n, for generating expressions. We then iterated through values from 2 to n. For each run, we produced a Boolean expression having a number of variables at that iteration through the `BooleanGenerator` class. We then lexed and parsed this expression to recover the original parse tree and count the number of operations and variables in this parse tree.

Afterward, we applied obfuscation to the original parse tree using the `MBAObfuscator` class. We counted and recorded the operations and variables in the obfuscated parse tree. This would increase the complexity of the expression, in turn increasing the difficulty in its interpretation.

Then we simplified the obfuscated parse tree using our `Solver` class that we had initialized with the passes in the provided. The number of operations and variables in the simplified parse tree was counted and recorded again. In the ideal case, the obfuscation should be reversed by simplifying the tree back to a state close to the original expression.

For each iteration, we recorded the number of variables, the original, obfuscated, and simplified expressions, and the counts for operations and variables at each stage. The results of each iteration were collated into a list and returned at the end of the function.

B. Data Collection

For each i (the number of variables), the original Boolean expression, the number of operations and variables in the original expression, the obfuscated expression, and its operation and variable counts against the original number, were all recorded, along with the simplified expression and its operation and variable counts.

Counters `OpCounter` and `VarCounter` were also used for the quantitative measurement of the complexity of a Boolean expression at each stage of the experiment. These counters traversed the parse tree of an expression and counted the number of operations and variables present.

- Operation Counter (`OpCounter`): This counter traversed the parse tree and counted the number of operations (such as `AND`, `OR`, `NOT`, etc.) in the expression. By comparing the operation count before and after obfuscation and simplification, we gauged the changes in expression complexity.
- Variable Counter (`VarCounter`): Similarly, this counter traversed the parse tree and counted the number of distinct variables in the expression. This helped in understanding how the obfuscation and simplification processes affected the number of variables in the expression.

For example, an original expression might have had a low count of operations and variables. Obfuscation was expected to increase these counts significantly, making the expression more complex and harder to interpret. The simplification process should then reduce these counts, ideally bringing them back closer to the original values.

C. Example Output

For a small number of variables (n=2), the output included:

- Original Expression: $(A \,\&\, !B)$
- Original Operation Count: 2
- Original Variable Count: 2
- Obfuscated Expression: $((A \wedge !!B) \,\&\, A)$
- Obfuscated Operation Count: 4
- Obfuscated Variable Count: 3
- Simplified Expression: $A \,\&\, !B$
- Simplified Operation Count: 2
- Simplified Variable Count: 2

This process was repeated for each value up to the specified maximum n.

We ensured that the Boolean expressions generated were syntactically valid for the lexer and parser. The obfuscation step significantly increased the complexity of the expressions, as measured by the operation and variable counts. The simplification step aimed to reduce this complexity, ideally restoring the expression to a form that was close to the original.

D. Applicability Testing

Another benchmark was testing our plugin's applicability in the real world, as this is what sets MBASED apart from other deobfuscators. A test was conducted between the authors of this paper analyzing the length of time it took to match a mixed Boolean-arithmetic expression to its corresponding program, given a set of programs with different control flows. This test aims to emulate part of what reverse engineers do, as they must be able to understand both the programs and the expressions. Half of the subjects in this test were given an obfuscated Boolean expression to match to a program, while the other half was given the deobfuscated version of the same expression. The goal was to determine the average time each group took to find the answer and compare the results. In this way, we can understand whether MBASED speeds up the analysis process for reverse engineers.

VI. Results

In this project, we aimed to simplify mixed Boolean-arithmetic expressions using powerful SMT Solvers and Python libraries. One of our benchmarks for checking if the Boolean expressions were reduced was the Operation Counter and the Variable Counter, which counted the number of operations and the number of variables, respectively, for both the obfuscated and deobfuscated versions of each expression. Based on the data collected, MBASED significantly lowers the number of operations, with the best case tested being 93 operations in the obfuscated expression and 18 operations in the deobfuscated expression. On average our plugin reduced the number of operations by about 50% (Figure 6). Although less consistent with its reduction, the number of variables also tended to be lowered by MBASED (with exceptions). In the best case tested, an expression with 44 variables was simplified to an expression with just 13 variables. On average, there was approximately a 30% reduction in the variable count (Figure 7).

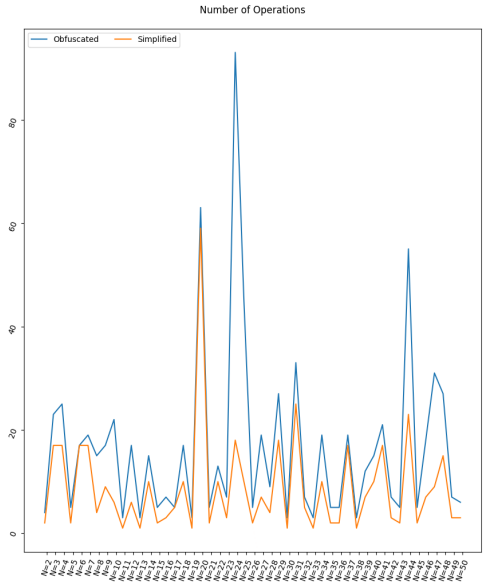

Fig. 6: Graph of the operation count for obfuscated and simplified expressions

Our second benchmark, which was a test to determine the applicability of the plugin, was given to two separate groups of subjects consisting of the authors. The group that worked with the obfuscated expression, `(A^!B)&A`, and matched it to the correct program found that the average time it took was 76 seconds. On the other hand, the group that worked with the deobfuscated expression, `A&B`, found that it took an average of 27 seconds to match the correct program. Despite the simplicity of these particular expressions, the same ideas can be applied to larger-scale reverse engineering.

VII. Conclusion

In this paper, we presented a plugin, MBASED, for Binary Ninja that is designed to simplify mixed Boolean-arithmetic

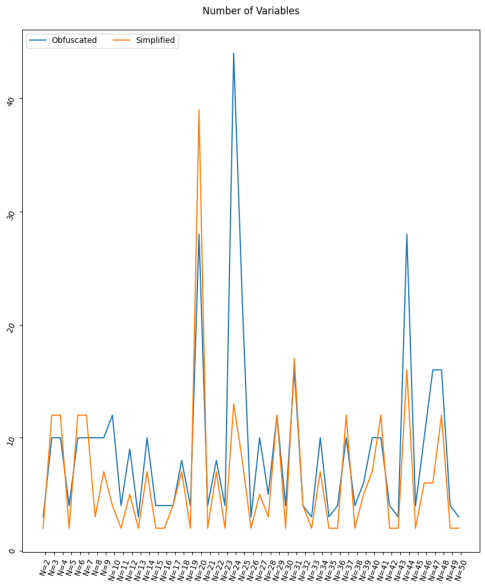

Fig. 7: Graph of the variable count for obfuscated and simplified expressions

expressions using the Z3 SMT solver and SymPy library. Our program effectively reduces complex Boolean expressions into simpler forms, enhancing the readability of the decompiled code. Again, what sets this program apart from other deobfuscators is the fact that it is the first application of MBA deobfuscation within an industry-grade decompiler. MBASED does this without changing anything about Binary Ninja's decompilation structure. With this plugin added to Binary Ninja's decompiler, the process of reverse engineering in the real world is shortened and made easier for those working with malware analysis.

A. Limitations and Future Works

Some of the limitations and potential for further development with MBASED is as follows:

1) Speed: Compared to other MBA deobfuscators that have been developed in recent years, MBASED is relatively slow in its operation, as runtime was not an aspect that we focused on in our research. Improvements can be made to make the deobfuscation process more efficient, as this is important if the program is given a large number of inputs.

2) Variable Count: Unlike the operation count, the variable count for our program is not always successfully reduced. One possible explanation for this is our use of Obfuscator LLVM as a benchmark, which, as the number of variables in the input expression increases, tends to greatly reduce the variable count on its own during obfuscation. Then, in the deobfuscation process, our plugin may be expanding the expressions slightly to make them simpler. Regardless of what the issue may be, it would be beneficial for the plugin to be improved upon so that the variable count is consistently reduced.

3) Other Forms of Obfuscation: This plugin only addresses Mixed Boolean-Arithmetic Deobfuscation. However, other

forms of obfuscation are also important to address for reverse engineering, such as variable renaming. MBASED could potentially be expanded upon to include other deobfuscation techniques.

Because of the plugin's use of the Visitor Design Pattern it is relatively easy for the program to be expanded upon, as users would just have to create their own `Visitor` class and add their own implementation to make changes. We have linked our GitHub repository for this project here: https://github.com/bliutech/mbased.

	Expr	Expr′	Term	Var
[A-Z]+	Expr → Term Expr′		Term → Var	Var → [A-Z]+
t	Expr → Term Expr′		Term → Var	Var → t
f	Expr → Term Expr′		Term → Var	Var → f
!	Expr → Term Expr′		Term → ! Term	
^		Expr′ → ^ Term Expr′		
&		Expr′ → & Term Expr′		
|		Expr′ → | Term Expr′		
(	Expr → Term Expr′		Term → (Expr)	
)		Expr′ → ϵ		
$		Expr′ → ϵ		

TABLE I: LL(1) Boolean Expression Parse Table

```
class TermExpr(Expr):
    """
    A class to represent an expression containing only a Term node.

    Attributes
    ----------
    first : Term
        The variable contained by the TermExpr.

    second : Optional[ExprPrime]
    """

    def __init__(self, first: Term, second: Optional[ExprPrime] = None):
        self.first = first
        self.second = second

    def __str__(self) -> str:
        if self.second is not None:
            return f"{self.first} {self.second}"

        return str(self.first)

    . . .
```

Fig. 8: TermExpr class definition for the Expr to Term-ExprPrime node expansion

ACKNOWLEDGEMENTS

The authors of this paper gratefully acknowledge the following: Project Mentor Benson Liu for his valuable knowledge of computer science and hands-on involvement; Residential Teaching Assistant Aashi Mishra for her guidance and assistance; Dean Jean Patrick Antoine, the Director of the Governor's School of Engineering and Technology (GSET) for his management and guidance; Rutgers University and Rutgers School of Engineering for the chance to advance knowledge, explore engineering, and open up new opportunities; and lastly Lockheed Martin, the New Jersey Office of the Secretary of Higher Education, and NJ GSET Alumni, for their continued funding and support of this research.

REFERENCES

[1] C. Treude, F. Figueira Filho, M.-A. Storey, and M. Salois, "An Exploratory Study of Software Reverse Engineering in a Security Context," Oct. 2011, pp. 184–188.

[2] 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

[3] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM: software protection for the masses," in *Proceedings of the 1st International Workshop on Software Protection*, ser. SPRO '15. IEEE Press, 2015, pp. 3–9, place: Florence, Italy.

[4] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, Nov. 2004, place: USA Publisher: IEEE Educational Activities Department. [Online]. Available: https://doi.org/10.1109/MSP.2004.111

[5] L. Burkholder, "The halting problem," *SIGACT News*, vol. 18, no. 3, p. 48–60, apr 1987. [Online]. Available: https://doi.org/10.1145/24658.24665

[6] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "Aot vs. jit: impact of profile data on code quality," *SIGPLAN Not.*, vol. 52, no. 5, p. 1–10, jun 2017. [Online]. Available: https://doi.org/10.1145/3140582.3081037

[7] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," vol. 25, pp. 811–829, 1995. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250706

[8] "Binary Ninja User Documentation." [Online]. Available: https://docs.binary.ninja/index.html

[9] J. Singh and J. Singh, "Challenge of malware analysis: Malware obfuscation techniques," *International Journal of Information Security Science*, vol. 7, no. 3, p. 100–110, 2018.

[10] 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*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 276–292. [Online]. Available: https://doi.org/10.1007/978-3-030-86890-1_16

[11] E. J. McCluskey, "Minimization of boolean functions," *The Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[12] B. C. Oliveira, M. Wang, and J. Gibbons, "The visitor pattern as a reusable, generic, type-safe component," *SIGPLAN Not.*, vol. 43, no. 10, pp. 439–456, Oct. 2008, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/1449955.1449799

[13] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. USA: Cambridge University Press, 2003.

[14] D. Gries and F. Schneider, *A Logical Approach to Discrete Math*, ser. Monographs in Computer Science. Springer New York, 1993. [Online]. Available: https://books.google.com/books?id=ZWTDQ6H6gsUC

[15] "SymPy 1.13.1 documentation." [Online]. Available: https://docs.sympy.org/latest/index.html

[16] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st ed. Springer Publishing Company, Incorporated, 2018.

[17] L. De Moura and N. 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*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.