

FUZZING A SOFTWARE VERIFIER

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Student id: 10844993

Department of Computer Science

Contents

Abstract	7
Declaration	8
Copyright	9
Acknowledgements	10
1 Introduction	11
1.1 Motivation	11
1.2 Research Question, Aim and Objectives	12
1.3 Contribution.	13
1.4 Dissertation Structure.	13
2 Background and Theory	15
2.1 SAT and SMT Solver	15
2.2 Bounded Model Checking	17
2.2.1 Incremental Bounded Model Checking	18
2.2.2 Bounded Model Checker	18
2.3 Fuzzing	28
2.3.1 Black-box, White-box and Grey-box Fuzzing	28
2.3.2 Generation-based, Mutation-based and Hybrid Fuzzing	29
3 Methodology and Implementation	36
3.1 Entry Point	36
3.2 Goto Grammar	38
3.3 Initial version of GotoFuzz	44
3.4 Improved version of GotoFuzz	51
3.5 Summary	55

4	Evaluation	58
4.0.1	Vulnerability Detection	58
4.0.2	Coverage Improvement	63
4.0.3	Summary	71
5	Conclusion and Further Work	72
5.1	Deliverables and Key Achievements	72
5.2	Reflection	73
5.3	Limitations and Future Work	74
	Bibliography	75

Word Count: 12353

List of Tables

3.1	Coverage Data with PRNG-based Mutator	55
4.1	Sample with the Most Coverage Improvement	66
4.2	Average Improvement Based on Individual Sample	66
4.3	Accumulated Improvement of Line Coverage	67
4.4	Accumulated Improvement of Function Coverage	67
4.5	Improvement with Extra Unwinding Times	70

List of Figures

2.1	Inference Rules for Arithmetic Resolution	17
2.2	Examples of using Linear Arithmetic Theory Solver	17
2.3	Example of Constraint and Property	20
2.4	ESBMC Commands	21
2.5	Example of Using ESBMC in incremental mode	22
2.6	The Overview of ESBMC	23
2.7	Example of Symbol Table	24
2.8	Example of Control Flow Chart	25
2.9	Example of Loop Unwinding	26
2.10	Example C Code	27
2.11	Goto Program	27
2.12	Program in SSA Form	27
2.13	Examples of Using Csmith	30
2.14	Examples of Using LibFuzzer	33
2.15	Example of Hybrid Fuzzing for Zlib Compress Library	35
3.1	The Grammar of Goto Program	43
3.2	Examples of a Goto Program	44
3.3	Mutation on Sequential and Non-Sequential Structure	46
3.4	Example of CFG Mutation. The mutated blocks/edges are illustrated in red.	47
3.5	Mutation algorithm for Sequential code structure	48
3.6	Mutation algorithm for Non-Sequential code structure	49
3.7	Result of mutation execution	50
3.8	The Overview of Initial GotoFuzz. White rectangles represent the components of ESBMC; grey rectangles represent the components of the GotoFuzz	51
3.9	The Result of the GotoFuzz	52

3.10	Mutation algorithm for Non-Sequential code structure	54
3.11	The Overview of Improved GotoFuzz. White rectangles represent the components of ESBMC; grey rectangles represent the components of the GotoFuzz	56
3.12	The Length of The Seeds Grows Rapidly	56
4.1	Unwinding is Not Terminated	59
4.2	The Difference Between Goto Programs Before and After Output . . .	61
4.3	The Difference when Using Incremental BMC Mode	62
4.4	Example of Coverage Report	65
4.5	Example of Reaching Maximal Line/Function Coverage Improvement	68
4.6	Example of PRNG Generated Corpus	69
4.7	Example of Running Out of Memory	70
5.1	The Commitment to ESBMC	73

Abstract

FUZZING A SOFTWARE VERIFIER

Chenfeng Wei

A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2022

The complexity of dynamic testing increases with the growth of the project. This results in the increasing usage of static analysis tools, such as bounded model checkers, which do not require compilation and act automatically. As a Satisfiability modulo theories(SMT) based bounded model checker, ESBMC has been successfully applied into a variety of areas due to its high efficiency and adaptability to multi-programming language and multi-platform. However, the construction of a coverage-guided mutation-based fuzzer for this verifier is a challenge, as directly generated random input cannot be “understood” by ESBMC and will be excluded beyond validating. Despite modern fuzzers such as Csmith and libFuzzer having been applied to ESBMC, in-depth exploration of uncovered codes and vulnerabilities is still a challenge.

In this paper, we present the *GotoFuzz*, a coverage-guided hybrid fuzzer that generates inputs according to the pre-defined grammar of the ESBMC’s intermediate representation(IR). The goal is to perform structure mutation on this IR. To demonstrate the design and improvement, two different implementations are proposed: an initial version based on the combination of libFuzzer and Csmith, and the second version with an additional pseudo-random number generator introduced. The comparison and testing result confirms that our GotoFuzz has the ability to dig out in-depth implementation and more improvement in the code coverage and vulnerabilities. At last, we will discuss the limitation of current work and the potential of extending our GotoFuzz to other verifier.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.library.manchester.ac.uk/about/regulations/>) and in The University’s policy on presentation of Theses

Acknowledgements

I would like to thank my supervisor Dr Lucas Cordeiro. He gave me very detailed guidance throughout the research of the project. With his encouragement, I managed to contribute code to ESBMC. I would also like to thank PhD student Rafael Menezes. Whenever I encountered problems with coding, he will offer some help, guiding me step by step through the design of the final program. ESBMC has opened up a new world for me, I can't wait to continue to use it to explore the field of security.

Chapter 1

Introduction

1.1 Motivation

The bounded model checking(BMC) technique has been widely used in practice to detect software vulnerabilities due to its ability to handle projects with high complexity. As a state-of-art BMC tool, ESBMC [1] has shown high efficiency and extraordinary error detection ability by winning first place in the ReachSafety-XCSP subcategory and second place in the SoftwareSystems-AWS-C-Common-ReachSafety, ReachSafety-ECA, and ReachSafety-Arrays subcategories [2]. In contrast to other model checkers, according to Fink’s study [3], ESBMC has generally less CPU time consumption than other modern verifiers, including CPAchecker[4], PeSCo[5], Symbiotic[6] Ultimate Automizer[7], only second to 2LS[8]. However, despite being an efficient verifier, there are problems regarding the ESBMC robustness. This can be reflected by the ”Filler Code” and ”Variable Wrapping” benchmarks set in Fink’s work, where ESBMC could be ”buggy” when all global variables in the executed program got randomly wrapped during fuzzing. Numerous reports of ESBMC errors are another piece of evidence. According to ESBMC’s GitHub repository, [9], approximately 30 vulnerabilities were claimed in 2021. Among these vulnerabilities three of which were formally confirmed and corrected by developers. Common vulnerabilities are segmentation overflows and undefined behaviour.

1.2 Research Question, Aim and Objectives

Research Question.

Given that fuzzers like Csmith[10] and libFuzzer[11] have already been applied to ESBMC, a natural thought is whether it is possible to construct a fuzzer based on a hybrid combination of these two fuzzers.

Aim.

This project aims to answer this question by using ESBMC as a vehicle for research. The goal is to construct a hybrid grey-box fuzzer which can

1. generate random test cases that not violating the grammatical rules of IR which is called Goto program. This can be done by the combination of Csmith and ESBMC's frontend parser.
2. perform mutation on the Goto programs based on the its grammar with the help of libFuzzer. These will only alter its semantic property while keeping syntactical correction.
3. fuzz ESBMC by performing symbolic execution and verification on mutated Goto programs. The result will be reflected by the coverage information and the error messages.

Objectives.

The objectives of this project are listed in the following:

1. summarise the principle workflow of ESBMC and the syntax grammar of its Goto program.
2. study the working principle of the fuzzer. Csmith and libFuzzer will be used as typical research subjects. The advantages and disadvantages of these mainstream generation-based and mutation-based fuzzer will be demonstrated.
3. based on 2, propose our structure-aware coverage-guided fuzzer GotoFuzz.

Deliverables.

Our work makes three major deliverables, which can be listed as follows:

1. A summary of the syntax grammar associated with Goto programs. Despite the slight differences in overall language feature supported. This can be useful for understanding the inner workings of the BMC black box. In addition, the summary can be valuable for understanding other CProver-based[12] BMC tools like ESBMC, including CBMC[13] and JBMC[14].
2. Our proposed structure-aware coverage-guided GotoFuzz.
3. Final dissertation to summarize the work done for deliverables 1 and 2.

1.3 Contribution.

The hybrid combination of generation-based and mutation fuzzer has been explored and introduced to various areas. For example, CONFUZZION [15] is the first fuzzer able to detect Java Virtual Machine(JVM) type confusion vulnerabilities. This mutation-based feedback-guided black-box performs on the syntactically valid Java programs generated by other JVM fuzzer. Zest [16] is another hybrid fuzzer which automatically guides random input generators to better explore the semantic analysis stage of the test program, leveraging program feedback via code coverage and performing feedback-directed parameter search. Despite having many successful cases, this hybrid approach has not yet been used for fuzzing bounded model checkers. To the best of our knowledge, GotoFuzz could be the first fuzzer that applies this hybrid fuzzing approach into this area. GotoFuzz demonstrates its feasibility and performance benefits through the results, including the expected improvement in code coverage and the discovery of some unintended vulnerabilities. In addition, hybrid fuzzers like zest focus on the mutation of the variable's value, whereas our implementation, focuses on structural mutation. This gives a new way of thinking about the variation approach.

1.4 Dissertation Structure.

This dissertation contains five chapters including the introduction. The layout of the remaining sections can be shown as follows. In chapter 2, we first introduce the

background knowledge related to the model checker and satisfiability modulo theories (SMT), leading to an SMT-based bounded model checker. In addition, ESBMC will be used as a use case to explain the working principles and software architecture of this type of verifier, including its frontend, middleware and backend, which is useful for understanding the design rationale of the new fuzzer construction in chapter 3. In addition, we categorise the mainstream fuzzer into different types and present their respective advantages and disadvantages. In chapter 3, we propose the GotoFuzz. The syntactical grammar of Goto language will first be summarised. The entry point and the format of the generated inputs will then be demonstrated. Next, two approaches for constructing the fuzzer will be proposed. The implementation based directly on libFuzeer and Csmith is presented firstly. The benefit and drawbacks of this prototype will be demonstrated, leading to the improved version where a pseudo-random number generator (PRNG) will be introduced to. In chapter 4, we discuss the design of the test suite and explain the test results. Chapter 5 summarises the project's deliverables and key achievements, ending with the dissuasion of the limitation and the possibility of applying out GotoFuzz to a wider area.

Chapter 2

Background and Theory

This chapter lays the theoretical foundation that will become essential in later chapters. Firstly, this chapter introduces the background to SAT and SMT solvers and lists the tools that are based on these theoretical implementations. This provides the theoretical foundation that will become essential in the later description of the BMC technique. Next, we will introduce the general model checking technique, followed by the bounded model checking and the incremental bounded model checking which can handle programs with higher complexity. Depending on the type of solver on which their backend is based, we introduce an SMT-based BMC called ESBMC. The demonstration of the software architecture of ESBMC will be divided into four components and illustrative examples will be provided to explain its workflow. Finally, this section will describe the background of fuzzing. Depending on the principle of operation, the fuzzer will be categorised and the basis for the categorisation, the principles of operation, and the advantages and disadvantages will be discussed in turn. This will inform us when designing our fuzzer GotoFuzz in the next chapter.

2.1 SAT and SMT Solver

A solver is used to solve the formula satisfiability problem. Broadly speaking, these problems can be divided into two categories: Boolean satisfiability problems(SAT) and satisfiability modulo theories(SMT) problems. The SAT problem asks whether a given Boolean formula is satisfiable and the tool used to solve such problems is called SAT solver. The two decision procedure algorithms used by SAT solvers to determine the satisfiability of formulas are Davis-Putnam-Logemann-Loveland (DPLL) and Conflict Driven Clause Learning (CDCL), the latter of which is widely used for its lower time

complexity.

SAT solvers are automatic and efficient. However, systems are usually designed and modelled at a higher level than the Boolean level and the translation to Boolean logic can be expensive. SMT solver is, therefore, introduced to create verification engines that can reason natively at a higher level of abstraction, while still retaining the speed and automation of today's Boolean engines. An SMT solver can be seen as a combination of the theories solver and the CDCL-based SAT solver. In mathematics, a theory is a consistent set of first-order formulas, including theories of Equality, Bit-vector, Linear-arithmetic, Arrays, etc. The theory provides inference rules for reasoning and simplifying complex sets of formulas. Figure 2.1 shows inference rule *AR* based on the Theory of Linear Arithmetic, where x is maximal in $ax + t$ and $-bx + s$; $a, b, g, h > 0$, $h \in \mathbb{N}$ and $ga = hb$. Figure 2.2 provides an example of the application of the inference rule to a first-order formula, where the set of inequalities yields an unsatisfiable result after inference. After conversion by theories solver, the high order formulae are transformed into Boolean formulae and can therefore be further processed by SAT solver. Examples of the modern SMT solver can be listed as follows:

1. Z3 [17] is an SMT solver developed by Microsoft Research. It supports a wide range of theories, including several non-standard extensions, such as tuples, lists, sets, and recursively defined sorts.
2. Yices[18] is an SMT solver developed at SRI international. It supports several theories, including e decides the satisfiability of formulas containing uninterpreted function symbols with equality, real and integer arithmetic, bit-vectors, scalar types, and tuples while not supporting floating-point.
3. MathSAT [19] is a general-purpose SMT solver developed by FBK-IRST and the University of Trento. The solver supports several SMT theories, including floating-points, but also supports the creation of Craig-interpolants [20] and partial assignment enumeration. It is available under a non-commercial, academic, free license.
4. CVC4 [21] is a theorem prover with support for for satisfiability modulo theories (SMT) problems. It can be used to prove the validity of first-order formulas in a large number of built-in logical theories and their combination. The theorem prover is a collaboration between the New York University and the University of Iowa, and it is released under a BSD license. CVC4 is an efficient open-source automatic theorem prover

$$\frac{ax + t \geq 0 \quad -bx + s \geq 0}{gt + hs \geq 0}$$

Figure 2.1: Inference Rules for Arithmetic Resolution

1. $5x - 3y - 3 \geq 0$ input
2. $-3x + 2y + z + 1 \geq 0$ input
3. $-2y - 10z + 1 \geq 0$ input
4. $y + 5z - 4 \geq 0$ AR(1,2)
5. $-7 \geq 0$ AR(3,4)
6. \perp

Figure 2.2: Examples of using Linear Arithmetic Theory Solver

5. Boolector[22] is an SMT solver for quantifier-free theories of bit-vectors and arrays. The solver uses bit-blasting for bit-vectors and lemmas on demand for arrays, converting the formulae to SAT formulae; it supports several SAT solvers as back-ends, including PicoSAT[23], MiniSAT, and CryptoMiniSAT[24]. The solver, however, does not support floating-point. Boolector focuses on continuous refinement of an initial abstraction of the SMT formula, based on the SAT solver's satisfying assignments. It is released under an MIT-like license with non-commercial and no-competition-use clauses

2.2 Bounded Model Checking

Model checking or property checking is a technique for automatically verifying correctness properties of finite-state systems, abstracting concrete procedures into clauses and using SAT/SMT solver for verification [25]. Model checking algorithms explicitly

enumerated the reachable states of the system to check the correctness of a given specification. This restricted the capacity of model checkers to systems with a few million states. Since the number of states can fill dramatically in the number of variables, early implementations were simply ready to deal with projects of small size and didn't scale to models with industrial complexity. Bounded model checking is, therefore, proposed. Instead of checking for all states, this technique only checks if a property holds for a subset of states. Bounded model checking does not solve the complexity problem of model checking, since it still relies on an exponential procedure and hence is limited in its capacity. But experiments have shown that it can solve many cases that cannot be solved by BDD-based techniques. The basic idea is to search for a counterexample in executions whose length is bounded by some integer k . If no bug is found then one increases k until either a bug is found, the problem becomes intractable, or some pre-defined upper bound is reached.

2.2.1 Incremental Bounded Model Checking

Incremental Bounded Model Checking was proposed to solve the incremental satisfiability problem. This technique aims to verify if satisfiability is preserved for a set of formulas that are satisfiable when a new set is inserted. During an incremental BMC process, the program is unstoppably unrolled until the completeness threshold is reached, or any error and bugs are reported. This ensures that smaller problems are solved sequentially instead of guessing an upper bound for the verification. This method maximises his value by finding counterexamples to the property while additionally proving the correctness of the procedure. In addition, Incremental Bounded Model Checking proved to be especially efficient in the presence of incremental SAT-solvers [26].

2.2.2 Bounded Model Checker

Traditionally, Bounded Model Checkers were based on Boolean Satisfiability (SAT). A typical example of an SAT solver is CBMC. CBMC implements BMC for ANSI-C/C++ programs using SAT solvers like MiniSat, verifying the absence of violated assertions under a given loop unwinding bound. CBMC was created over a decade ago and has been maintained ever since. Many BMCs have been developed on this basis. In fact, the SMT-based verifier ESBMC described below was first developed based on CBMC v2.9 (2008)[2]. After several iterations, the CBMC backend now also adds

support for some SMT solvers[27].

To cope with increasing software complexity, the SMT (Satisfiability Modulo Theories) solvers have to be used as back-ends for solving the generated verification conditions. A typical example of an SAT solver is ESBMC. ESBMC is an SMT-based contextual boundary model checker that has been widely used to verify multi-language programs, including C/C++, Java and Solidity. ESBMC can automatically find memory safety and assertion violations. The basic working principle of an SMT-based bounded model checker begins with a transition system M , a property ϕ , and a bound k . ESBMC unwinds the system k times and converts it into a verification condition (VC) ψ . [28] ESBMC checks the negation of this VC so that ψ is satisfiable if and only if ϕ has a counterexample of depth k or less. To cope with increasing software complexity, sorts of SMT (Satisfiability Modulo Theories) solvers have been used as the back-end of ESBMC for solving the generated VCs. The VC ψ is a quantifier-free formula in a decidable subset of first-order logic, which is then checked for satisfiability by an SMT solver. model checking problem is formulated by constructing the following logical formula $\psi_k = I(s_0) \wedge \bigvee_{i=0}^k \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i)$, where I is the set of initial states of M and $\gamma(s_j, s_{j+1})$ represents the transition relation of M between time steps j and $j+1$. If the formula is satisfiable, the SMT solver will provide a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. If it is unsatisfiable, we can conclude that no error state is reachable in k steps or less. Note that this approach can be used only to find violations of the property up to the bound k .

Figure 2.3 provides an example of this transformation. $store(a, i, v)$ means to write the value of v in position i of array a . $select(a, i)$ means to read the value at position i of array a . C corresponds to the constraint part $I(s_0) \wedge \bigvee_{i=0}^k \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ and P represents the property $\phi(s_i)$. ESBMC performs satisfiability check of formula $C \wedge \neg P$.

ESBMC is used as a command-line tool as shown in Figure 2.4. Despite being a general BMC, ESBMC also implements the k -induction algorithm to add support for the incremental bounded model checking. Figure 2.5 provides a working example with incremental BMC, where the option ”-k-induction” is used to select the k -induction proof rule. For this particular C program *test1.c*, ESBMC provides the ”VERIFICATION SUCCESSFUL” prints as the result, meaning in this situation ESBMC does not find any bugs.

Next, we will take a deeper look at the software architecture of ESBMC, which can be helpful for understanding the robustness and potential vulnerability within ESBMC.

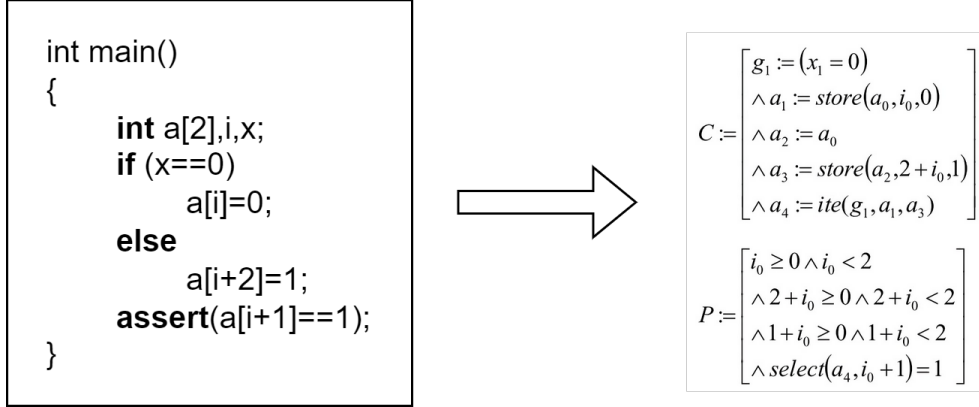


Figure 2.3: Example of Constraint and Property

Figure 2.6 provides a general overview, where ESBMC has been divided into four components: the front-end, the Goto Converter, the Symbolic Engine and the back-end. Given that ESBMC supports a variety of front-ends, for the sake of the discussion that follows, the Clang-based C/C++ front-end has been used as our case study.

Frontend

The frontend is an important piece of technology which facilitates the transition between the program under verification and a format the tool can work upon. The main goal is to translate source code into the control flow chart (CFG) representation called the symbol table, an auxiliary data structure storing the meaning and range of variable names. Despite several parsers having been constructed corresponding to different programming languages in ESBMC, the result will always be this uniform intermediate representation. This conversion is achieved by a front-end. This component shares mostly the same structure with a typical compiler including a pre-processor, scanner, parser and type checker. Pre-processor handles special operations that will be performed according to the preprocessor instructions, such as replacement or expansion macros. A lexical analysis of the scanner and a syntax score of the parser resulted in the Abstract Syntax Tree (AST). To simplify the analysis, ASTs have been converted into a simpler form, called an intermediate representation (IR), by a type checker in which a symbol table is generated simultaneously. At this point, the source code as input is converted into symbol tables[12]. Figure 2.7 shows an example of a symbol table. The symbol table interprets the parsed program as symbol information and stores it, including the name, type and value of each symbol.

```

→ bin ./esbmc --help

* * *          ESBMC 6.10.0          * * *

Main Usage:
  --input-file file.c ...             source file names

Options:
  -? [ --help ]                       show help

Printing options:
  --symbol-table-only                 only show symbol table
  --symbol-table-too                  show symbol table and verify
  --parse-tree-only                   only show parse tree
  --parse-tree-too                   show parse tree and verify
  --goto-functions-only               only show goto program
  --goto-functions-too               show goto program and verify
  --program-only                     only show program expression
  --program-too                      show program expression and verify
  --ssa-symbol-table                 show symbol table along with SSA
  --ssa-guards
  --ssa-sliced                       print the sliced SSAs
  --ssa-no-location
  --smt-formula-only                 only show SMT formula {not supported by all
  --smt-formula-too                 solvers},
  --smt-model                       show SMT formula (not supported by all
  --smt-model                       solvers), and verify
  --smt-model                       show SMT model (not supported by all
  --smt-model                       solvers), if the formula is SAT

Trace:
  --quiet                           do not print unwinding information during
  --quiet                           symbolic execution
  --compact-trace
  --symex-trace                      print instructions during symbolic execution
  --ssa-trace                        print SSA during SMT encoding
  --ssa-smt-trace                   print generated SMT during SMT encoding
  --symex-ssa-trace                 print generated SSA during symbolic
  --symex-ssa-trace                 execution
  --show-goto-value-sets            show value-set analysis for the goto
  --show-goto-value-sets            functions
  --show-symex-value-sets           show value-set analysis during symbolic
  --show-symex-value-sets           execution

```

Figure 2.4: ESBMC Commands

```

> bin ./esbmc test1.c --k-induction
ESBMC version 6.10.0 64-bit x86_64 linux
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
file test1.c: Parsing
test1.c:10:5: warning: implicit declaration of function 'assert' is invalid in C99 [-Wimplicit-function-declaration]
    assert(x>0);
    ^
test1.c:13:3: warning: implicit declaration of function 'assert' is invalid in C99 [-Wimplicit-function-declaration]
    assert(x>0);
    ^
Converting
Generating GOTO Program
GOTO program creation time: 0.334s
GOTO program processing time: 0.013s
*** Checking base case, k = 1
Starting Bounded Model Checking
Not unwinding
Symex completed in: 0.002s (24 assignments)
Slicing time: 0.000s (removed 12 assignments)
Generated 2 VCC(s), 2 remaining after simplification (12 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.003s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.003s
Runtime decision procedure: 0.104s
BMC program time: 0.110s
No bug has been found in the base case
*** Checking forward condition, k = 1
Starting Bounded Model Checking
Not unwinding
Symex completed in: 0.002s (22 assignments)
Slicing time: 0.000s (removed 14 assignments)
Generated 1 VCC(s), 1 remaining after simplification (8 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.001s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.001s
Runtime decision procedure: 0.001s
The forward condition is unable to prove the property
*** Checking base case, k = 2
Starting Bounded Model Checking
Unwinding loop 2 iteration 1 file test1.c line 8 function main
Not unwinding
Symex completed in: 0.002s (30 assignments)
Slicing time: 0.001s (removed 14 assignments)
Generated 3 VCC(s), 3 remaining after simplification (16 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.003s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.003s
Runtime decision procedure: 0.221s
BMC program time: 0.228s
No bug has been found in the base case
*** Checking forward condition, k = 2
Starting Bounded Model Checking
Unwinding loop 2 iteration 1 file test1.c line 8 function main
Not unwinding
Symex completed in: 0.003s (27 assignments)
Slicing time: 0.000s (removed 18 assignments)
Generated 1 VCC(s), 1 remaining after simplification (9 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.001s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.001s
Runtime decision procedure: 0.001s
The forward condition is unable to prove the property
*** Checking inductive step, k = 2
Starting Bounded Model Checking
Unwinding loop 2 iteration 1 file test1.c line 8 function main
Not unwinding
Symex completed in: 0.003s (32 assignments)
Slicing time: 0.000s (removed 13 assignments)
Generated 2 VCC(s), 2 remaining after simplification (19 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.004s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.004s
Runtime decision procedure: 0.227s
BMC program time: 0.235s

VERIFICATION SUCCESSFUL

Solution found by the inductive step (k = 2)

```

Figure 2.5: Example of Using ESBMC in incremental mode

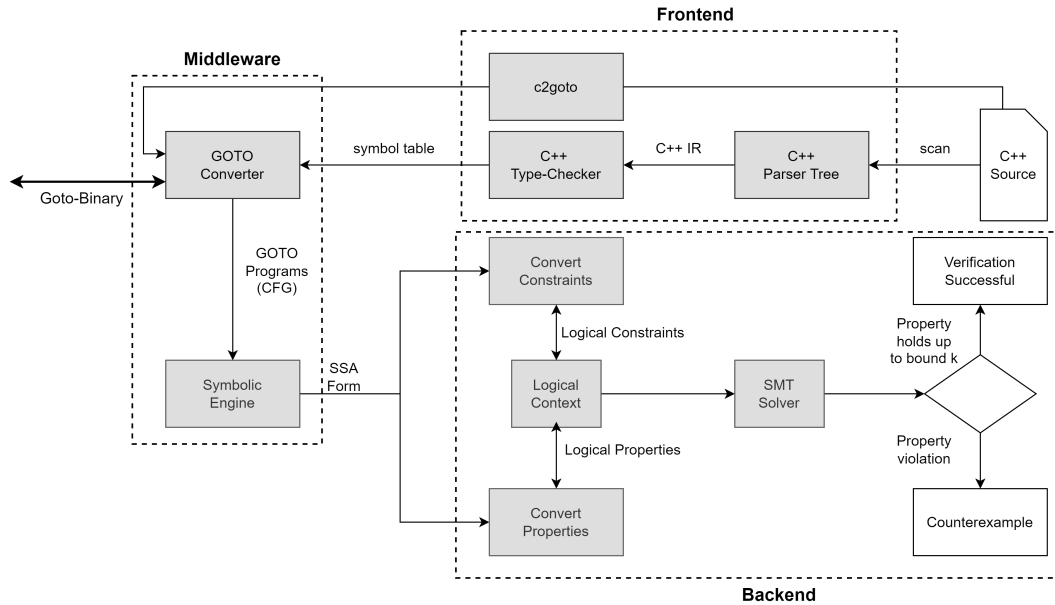


Figure 2.6: The Overview of ESBMC

Another implementation of this conversion is achieved through special compilers like goto-cc for CBMC or c2goto for ESBMC. These compilers work as a substitution for common C compilers like GCC and, on top of having the same functionality, compile programs given in C and C++ into Goto programs. Functionally, this type of compiler is more like an integrated convertor for Goto programs on top of the existing compilers, resulting in files stored in the goto program in binary format. The Goto-binaries are not meant to be executable but are typically given to the verifier.

Goto Generator

The Goto Converter converts symbol table to the Goto programs. Goto programs are the intermediate representation of the ESBMC tool chain. This representation is similar to the control flow graph (CFG) representation used by conventional compilers. Figure 2.8 provides a example of a CFG generated from a C program.

During this process, the Goto program is simplified, and new property checks and instructions can be added. The Goto program is a simplified version of the program: a branch and a backward Goto replace for and while loops. It is very similar to a C program, containing assignments, function calls and returns, and location information. Depending on the ESBMC option directive set, some additional code may be inserted.

```

Symbol.....: c:@F@__VERIFIER_nondet_schar
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_nondet_schar
Mode.....: C (0)
Type.....: signed char ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 67

Symbol.....: c:@F@__VERIFIER_nondet_bool
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_nondet_bool
Mode.....: C (0)
Type.....: _Bool ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 68

Symbol.....: c:@F@__VERIFIER_nondet_float
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_nondet_float
Mode.....: C (0)
Type.....: float ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 69

Symbol.....: c:@F@__VERIFIER_nondet_double
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_nondet_double
Mode.....: C (0)
Type.....: double ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 70

Symbol.....: c:@F@__VERIFIER_error
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_error
Mode.....: C (0)
Type.....: void ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 72

Symbol.....: c:@F@__VERIFIER_assume
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_assume
Mode.....: C (0)
Type.....: void (signed int)
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 73

Symbol.....: c:@F@__VERIFIER_atomic_begin
Module.....: esbmc_intrinsics
Base name....: __VERIFIER_atomic_begin
Mode.....: C (0)
Type.....: void ()
Value.....:
Flags.....: lvalue
Location....: file esbmc_intrinsics.h line 74

```

Figure 2.7: Example of Symbol Table

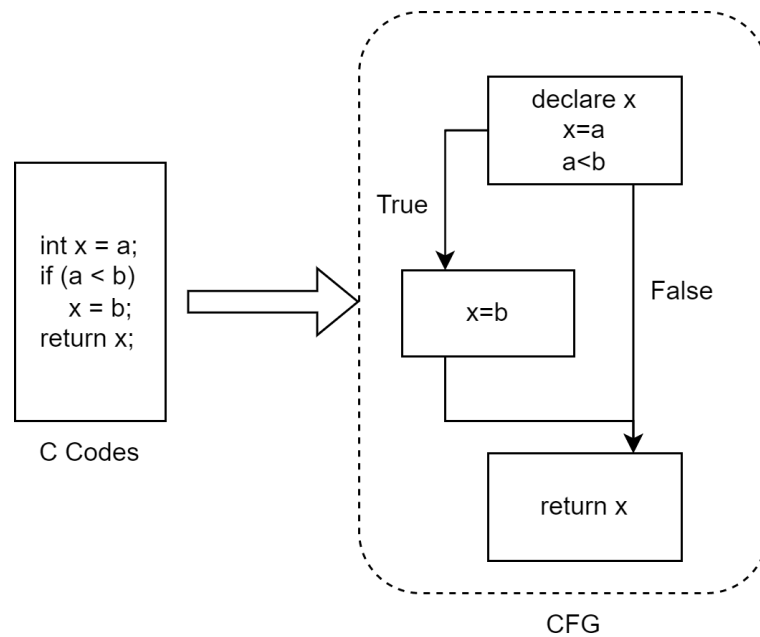


Figure 2.8: Example of Control Flow Chart

For example, concurrency instructions can be inserted if the program is multi-threaded, and k -induction specific instructions are introduced if k -induction verification is enabled.

Symbolic Engine

The parsing stage translates the raw input into an internal data structure or abstract syntax tree that can be easily processed by the rest of the program. Next, the semantic analysis driven by the symbolic engine checks if an input satisfies certain semantic constraints (e.g. if an XML input fits a specific schema), and executes the core logic of the program. The symbolic engine unwinds the GOTO program from the previous step, symbolically executed, unwinding loops and unfolding recursive function calls up to a given bound. Specifically, the Symbolic Engine will firstly convert the variables from program text to a single static assignment (SSA) form. New variables are created to identify branch and loop entry conditions. These variables will guard the assignments based on the branch taken. Symbolic execution will be performed after the conversion to perform semantic analysis, including dynamic memory checks (bounds, memory alignment, offset pointer-free, and double-free) and unwinding assertions. The point of this step is to make sure each assignment is independent[1]. An illustrative example

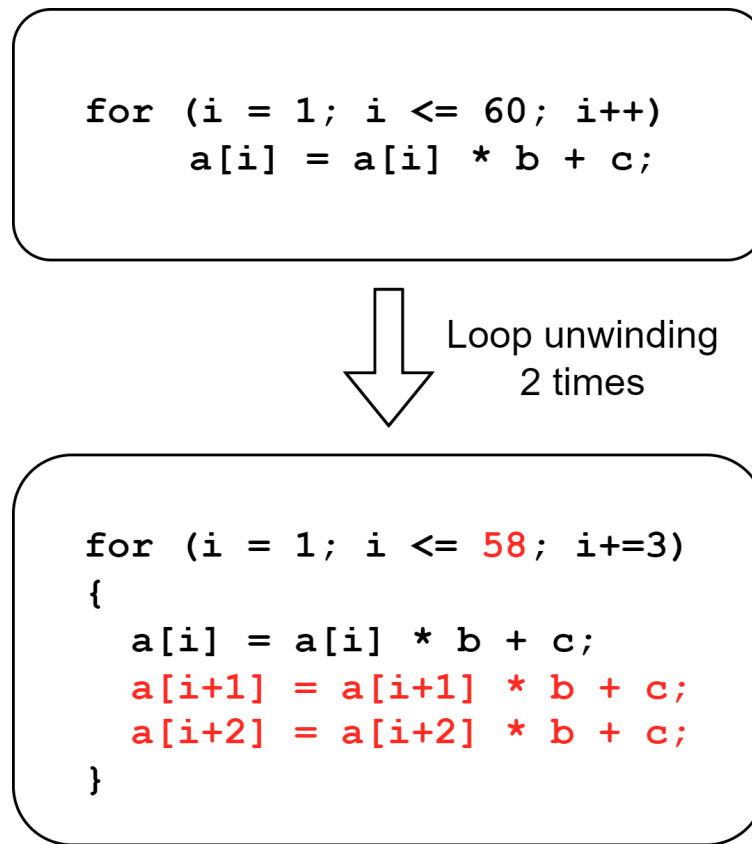


Figure 2.9: Example of Loop Unwinding

of loop unwinding is provided in Figure 2.9. After unrolling the loop two times, the cycle count changed from 60 to 58 while two expressions are added to the loop block.

Figures 2.10 to 2.12 show the differences between a source C program and its Goto program and SSA form. This can be reflected by the Goto program the original for-loop structure is replaced by an IF-THEN-GOTO structure. This representation makes it very easy to interpret the program, keeping just two pieces of state: the current position which basic block and which line, and the values of all variables. Meanwhile, in the SSA form program, the expression gets unwinded and variables get cloned and renamed. This ensures that each variable appearing in the resulting program is written to exactly once. In this way, we can refer to earlier values of the variable by just referencing the name of an older incarnation of the variable.

```

unsigned long factorial(unsigned n) {
    unsigned long fac = 1;
    for (unsigned int i = 1; i <= n; i++) {
        fac *= i;
    }
    return fac;
}

```

Figure 2.10: Example C Code

```

    unsigned long fac.1 = 1;
    unsigned int i.1 = 1;
for_loop_start:
    if (i.1 <= n.1) goto for_loop_entry else goto
        for_loop_end;
for_loop_entry:
    fac.1 *= i.1;
    i.1++;
    goto for_loop_start;
for_loop_end:
    return fac.1;

```

Figure 2.11: Goto Program

```

    unsigned long fac.1 = 1;
    unsigned int i.1 = 1;
for_loop_start:
    if (i.1 <= n.1) goto for_loop_entry else goto
        for_loop_end;
for_loop_entry:
    fac.1.2 = fac.1.1 * i.1.1;
    i.1.2 = i.1.1 + 1;
    goto for_loop_start;
for_loop_end:
    return fac.1;

```

Figure 2.12: Program in SSA Form

Backend

After the SSA formula is generated, the next step is to encode every non-sliced assignment in SMT and check for satisfiability. During this workflow, two sets of SMT formulae are created based on the SSA expressions. We denote C for the constraints and P for the properties. These quantifier-free formulae will be used as input for the SMT solver, a counterexample will be created if there exists a violation of a given property, or an unsatisfiable answer if the property holds. In ESBMC, Several SMT solvers are currently supported, including Z3, Bitwuzla, Boolector, MathSAT, CVC4 and Yices.

2.3 Fuzzing

Fuzz, also known as fuzz testing, is a method that focuses on finding software security vulnerabilities. A typical fuzzing process involves repeatedly driving the target software through an automated or semi-automated method, providing it with specially constructed input data. During the process, any security vulnerabilities or slow units will be monitored and recorded based on the anomaly results and input data [29]. This technique has been proven to be straightforward, yet successful. With the decrease in computational expenses, fuzzing has become progressively helpful for both programmers and testers, who use it to find new bugs/weaknesses in programming.[30] Companies such as Google, Microsoft and Cisco all use fuzz testing as part of their software security development process.

2.3.1 Black-box, White-box and Grey-box Fuzzing

In terms of knowledge of the test subject's internal structure, there are white box, black box and in-between grey box testing. White Box Fuzz usually analyses the application before testing to obtain certain information to assist in the creation of inputs that will find crashes in the application[31]. The internal structure of the program is analysed to assist in the generation of appropriate input values. The primary systematic white-box fuzzing technique is a dynamic symbolic execution. The key idea is that by turning the input into a symbolic value, the program computes an output value that is a function of the symbolic input value, the fuzzer can make sure that these input values will drive the program to a different execution path, thus improving coverage. Unlike the white-box approach, black-box Fuzz does not care about the state of program execution and

considers the program to be a black box. It generates input directly and then tries to find the result. Black-box Fuzz is also often referred to as data-driven Fuzz, and the most traditional Fuzzer is black-box. The generation of values depends on the program input/output behaviour, and not on its internal structure. In contrast to the white-box and black-box, grey-box Fuzz combines both black-box and white-box approaches. This most popular fuzzing strategy combines a black-box fuzzer for execution while using a lightweight white-box approach to provide feedback to the black-box[32].

2.3.2 Generation-based, Mutation-based and Hybrid Fuzzing

In terms of generation strategies for new test inputs, fuzzers can be divided into generation-based and mutation-based and the hybrid fuzzer, which combines the first two modes of operation.

Generation-based Fuzzing

Generation-based fuzzers usually target a single input type, generating inputs according to a pre-defined grammar. Good examples of such fuzzers include Csmith, which generates valid C programs, and Peach, which generates inputs of any type, but requires such a type to be expressed as a grammar definition[33]. Csmith is a tool that can generate random C programs that statically and dynamically conform to the C99 standard. It is useful for stress-testing compilers, static analyzers, and other tools that process C code. Csmith developer team claims to have found bugs in every tool that it has tested, and we have used it to find and report more than 400 previously unknown compiler bugs[10]. As An expressive generator, Csmith has also maximised his expressiveness, supporting many language features and combinations of features. Considering that the C++-based frontend ESBMC is essentially a translation of c++ programs into SMT formulas, more language features will improve code coverage.

Csmith is used as a command-line tool. Figure 2.13 illustrates its basic command to create a random C file *random1.c*, along with the content of the generated program.

Mutation-based Fuzzing

Mutation-based fuzzer creates inputs by randomly mutating analyst-provided or randomly-generated seeds. Guided strategies such as coverage-based or patch-Based are often used to enable more efficient Fuzzing during mutation. Guided Fuzz was first proposed by AFL[34], which counts the coverage of samples after execution, finds the bytes that

```

- bin csmith - random.c
- bin cat random.c
/*
 * This is a RANDOMLY GENERATED PROGRAM.
 *
 * Generator: csmith 2.3.0
 * Git version: 30dcd7
 * Options: (none)
 * Seed: 854357598
 */

#include "csmith.h"

static long __undefined;

/* --- Struct/Union Declarations --- */
/* --- GLOBAL VARIABLES --- */
static volatile uint16_t g_2 = 65532UL; /* VOLATILE GLOBAL g_2 */
static int32_t g_6[8] = {(-8L),(-8L),(-8L),(-8L),(-8L),(-8L),(-8L),(-8L)};
static int32_t g_8 = 0x73672071L;

/* --- FORWARD DECLARATIONS --- */
static int64_t func_1(void);

/* --- FUNCTIONS --- */
/* ----- */
/*
 * reads : g_2 g_6 g_8
 * writes: g_6 g_8
 */
static int64_t func_1(void)
{ /* Block id: 0 */
    uint32_t l_3 = 0x989226C3L;
    int32_t l_4 = 0x68C3EC3DL;
    int32_t *l_5 = &g_6[5];
    int32_t *l_7 = &g_8;
    (*l_7) &= ((*l_5) &= (l_4 = (g_2 <= l_3)));
    return (*l_5);
}

/* ----- */
int main (int argc, char* argv[])
{
    int i;
    int print_hash_value = 0;
    if (argc == 2 && strcmp(argv[1], "1") == 0) print_hash_value = 1;
    platform_main_begin();
    crc32_gentab();
    func_1();
    transparent_crc(g_2, "g_2", print_hash_value);
    for (i = 0; i < 8; i++)
    {
        transparent_crc(g_6[i], "g_6[i]", print_hash_value);
        if (print_hash_value) printf("index = %d\n", i);
    }
    transparent_crc(g_8, "g_8", print_hash_value);
    platform_main_end(crc32_context ^ 0xFFFFFFFFUL, print_hash_value);
    return 0;
}

/*----- statistics -----*/
XXX max struct depth: 0
breakdown:
depth: 0, occurrence: 3
XXX total union variables: 0

XXX non-zero bitfields defined in structs: 0
XXX zero bitfields defined in structs: 0
XXX const bitfields defined in structs: 0
XXX volatile bitfields defined in structs: 0
XXX structs with bitfields in the program: 0
breakdown:
XXX 64bit-bitfields structs in the program: 0
breakdown:
XXX times a bitfields struct's address is taken: 0
XXX times a bitfields struct on LHS: 0
XXX times a bitfields struct on RHS: 0
XXX times a single bitfield on LHS: 0
XXX times a single bitfield on RHS: 0

XXX max expression depth: 4
breakdown:
depth: 1, occurrence: 2
depth: 4, occurrence: 1

XXX total number of pointers: 2

XXX times a variable address is taken: 2
XXX times a pointer is dereferenced on RHS: 1
breakdown:
depth: 1, occurrence: 1
XXX times a pointer is dereferenced on LHS: 2
breakdown:
depth: 1, occurrence: 2
XXX times a pointer is compared with null: 0
XXX times a pointer is compared with address of another variable: 0
XXX times a pointer is compared with another pointer: 0
XXX times a pointer is qualified to be dereferenced: 4

XXX max dereference level: 1
breakdown:
level: 0, occurrence: 0
level: 1, occurrence: 6
XXX number of pointers point to pointers: 0
XXX number of pointers point to scalars: 2
XXX number of pointers point to structs: 0
XXX percent of pointers has null in alias set: 0
XXX average alias set size: 1

XXX times a non-volatile is read: 3
XXX times a non-volatile is write: 5
XXX times a volatile is read: 1
XXX times read thru a pointer: 0
XXX times a volatile is write: 0
XXX times written thru a pointer: 0
XXX times a volatile is available for access: 0
XXX percentage of non-volatile access: 88.9

XXX forward jumps: 0
XXX backward jumps: 0

XXX stmts: 2
XXX max block depth: 0
breakdown:
depth: 0, occurrence: 2

XXX percentage a fresh-made variable is used: 75
XXX percentage an existing variable is used: 25
FYI: the random generator makes assumptions about the integer size. See platform.info for more details.
***** end of statistics *****/

```

Figure 2.13: Examples of Using Csmith

have a greater impact on coverage after mutation, and tests them more often to achieve more efficient mutation. LibFuzzer is a fuzzing tool which comes from LLVM[35] compiler infrastructure which to fuzz each function separately. It has acquired a gigantic measure of popularity nowadays for its viability in tracking down basic bugs and security weaknesses in broadly programming frameworks[16]. Next, the fuzzer keeps track of which code regions have been tested and then performs variants on the corpus of input data to maximise code coverage, whose information is provided by LLVM's SanitizerCoverage instrumentation. The Mutations algorithm includes operations such as flipping randomly chosen bits or inserting/deleting random chunks of bytes. The mutated input is saved if and only if it results in the coverage of a program branch that was not covered by any previously saved input. If any generated input leads to the program crashing, then the input is recorded as a potentially new bug to triage [36]. LibFuzzer can generate entirely random input and work without any initial seeds but will be less efficient if the library under test accepts complex, structured inputs. a corpus of sample inputs can significantly improve efficiency. This implies practically no code coverage at the start, and many hours of work before the first input passing initial integrity checks is produced. Therefore, it is worth having even a few/odd test cases to start with. It has acquired a gigantic measure of popularity nowadays for its viability in tracking down basic bugs and security weaknesses in broadly programming frameworks.

Figure 2.14 shows an example of libFuzzer. In this case, *fuzz.c* is a C source code containing bugs. LibFuzzer is already built into LLVM. In order to work, it needs to be compiled with the source code and enabled with `"-fsanitize=fuzzer"`. `"-fsanitize=address"` is an optional parameter to enable the AddressSanitizer in LLVM, which will help libFuzzer to find bugs. LibFuzzer prints the run to the command line. Take the output line of the second round `"#19"` for example, where each of the parameters is explained below:

1. `"#19"` is used to mark the current round
2. `"NEW"` indicates that the fuzzer has created a test input that covers new areas of the code under test
3. `"cov"` shows total number of code blocks or edges covered
4. `"ft"` stands for `"feature(s)"`, meaning the evaluation of the current code coverage.

5. "corp" represents the number of entries in the current in-memory test corpus and its size in bytes
6. "lim" represents the current limit on the length of new entries
7. "exec/s" stands for the run speed, in particular, the number of fuzzer iterations per second
8. "rss" represents the current memory consumption
9. "L" represents the size of the new input in byte. This signal is only seen in "NEW" and "REDUCE" events
10. "MS" represents the count and list of the mutation operations used to generate the input

After 6 rounds of testing, libFuzzer finds the bug and prints an error message. This error message is also simultaneously recorded as a crash log file.

Hybrid Fuzzing

Both generation-based and mutation-based fuzzers are considered to have inherent flaws. On one hand, a generation-based fuzzer lacks a coverage-guided trace-feedback mechanism, which leads to possible duplication of the generated test cases. Another common flaw in generation-based testing occurs when the input is required to fulfil sophisticated semantic validity criteria that are not explicitly evaluated by the generator. On the other hand, a mutation-based fuzzing lacks a generator's understanding of which inputs are well-formed, thus the lack of an input grammar can also result in inefficient fuzzing for complicated input types, where any traditional mutation (e.g. bit flipping) leads to an invalid input rejected by the target API in the early stage of parsing. For instance, they may struggle with checks against magic numbers, whose value is unlikely to be generated with random mutations. As these checks may appear early in the execution, fuzzers may soon get stuck and stop producing interesting inputs.

For these reason, works[37] [16] [30] [38] have explored the hybrid fuzzing method. The general goal is to combine both fuzzer to perform together, generating random samples while performing mutation accordingly to obtain better Fuzz efficiency. The hybrid fuzzer is more of a grey-box fuzzer than the two previously mentioned black-box fuzzer, leveraging knowledge about the fuzz target to generate inputs that are more


```

    bin clang++ -g -std=c++11 -fsanitize=address,fuzzer fuzz.c -o fuzz
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated [-Wdeprecated]
./bin ./fuzz
INFO: Seed: 297038849
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Loaded 1 modules (7 inline 8-bit counters): 7 [0x5563b9233ed0, 0x5563b9233ed7],
INFO: Loaded 1 PC tables (7 PCs): 7 [0x5563b9233ed8, 0x5563b9233f48],
INFO: maxLen is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#0  INITED cov: 3 ft: 3 corp: 1/1b exec/s: 0 rss: 33Mb
#19 NEW cov: 4 ft: 4 corp: 2/5b lin: 4 exec/s: 0 rss: 33Mb L: 4/4 MS: 2 CopyPart-CopyPart-
#20 REDUCE cov: 4 ft: 4 corp: 2/4b lin: 4 exec/s: 0 rss: 34Mb L: 3/3 MS: 1 EraseBytes-
#1234 NEW cov: 5 ft: 5 corp: 3/7b lin: 14 exec/s: 0 rss: 34Mb L: 3/3 MS: 4 CMP-CrossOver-ChangeBit-ShuffleBytes- DE: "000F"-
#12877 NEW cov: 6 ft: 6 corp: 4/12b lin: 128 exec/s: 0 rss: 35Mb L: 5/5 MS: 3 ShuffleBytes-CrossOver-InsertRepeatedBytes-
#13218 REDUCE cov: 6 ft: 6 corp: 4/11b lin: 128 exec/s: 0 rss: 35Mb L: 4/4 MS: 1 EraseBytes-
#13584 REDUCE cov: 6 ft: 6 corp: 4/10b lin: 135 exec/s: 0 rss: 35Mb L: 3/3 MS: 1 EraseBytes-
#13785 REDUCE cov: 7 ft: 7 corp: 5/14b lin: 333 exec/s: 0 rss: 37Mb L: 4/4 MS: 1 InsertByte-
=====
==19046==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000d1173 at pc 0x5563b91f0f3b bp 0x7ffc0215bd38 sp 0x7ffc0215bd28
==19046==
#0 0x5563b91f0f3a in VulnerableFunction(unsigned char const*, unsigned long) /root/workspace/ESBMC_Commit/release/bin/fuzz.c:10:14
#1 0x5563b91f0fa4 in LLVMFuzzerTestOneInput /root/workspace/ESBMC_Commit/release/bin/fuzz.c:17:3
#2 0x5563b9117323 in fuzzer:Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3e323) (BuildId: 33f2751e
f9b9bd5f697fb42149b0f36a4bdde296)
#3 0x5563b9116a79 in fuzzer:Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) (/root/workspace/ESBMC_Commit/release/bin/fu
zz+0x3da79) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#4 0x5563b9118269 in fuzzer:Fuzzer::MutateAndTestOne() (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3f269) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#5 0x5563b9118de5 in fuzzer:Fuzzer::Loop(std::vector<fuzzer::SizedFile, std::allocator<fuzzer::SizedFile> >>) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3fde
5) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#6 0x5563b910ef22 in fuzzer:FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x2df22) (Buil
dId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#7 0x5563b9138c12 in main (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x57c12) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#8 0x7fcf92925b0f (/lib/x86_64-linux-gnu/libc.so.6+0x29b0f) (BuildId: 60380485a9793d0e873f8ea2c93e02efaa9aa3d)
#9 0x7fcf92925b3f in _libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29b3f) (BuildId: 60380485a9793d0e873f8ea2c93e02efaa9aa3d)
#10 0x5563b90fb964 in _start (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x22964) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)

0x6020000d1173 is located 0 bytes to the right of 3-byte region [0x6020000d1170,0x6020000d1173)
allocated by thread T0 here:
#0 0x5563b91ee85d in operator new[](unsigned long) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x11585d) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#1 0x5563b9117323 in fuzzer:Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3e323) (BuildId: 33f2751e
f9b9bd5f697fb42149b0f36a4bdde296)
#2 0x5563b9116a79 in fuzzer:Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) (/root/workspace/ESBMC_Commit/release/bin/fu
zz+0x3da79) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#3 0x5563b9118269 in fuzzer:Fuzzer::MutateAndTestOne() (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3f269) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#4 0x5563b9118de5 in fuzzer:Fuzzer::Loop(std::vector<fuzzer::SizedFile, std::allocator<fuzzer::SizedFile> >>) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x3fde
5) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#5 0x5563b910ef22 in fuzzer:FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x2df22) (Buil
dId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#6 0x5563b9138c12 in main (/root/workspace/ESBMC_Commit/release/bin/fuzz+0x57c12) (BuildId: 33f2751ef9b9bd5f697fb42149b0f36a4bdde296)
#7 0x7fcf92925b0f (/lib/x86_64-linux-gnu/libc.so.6+0x29b0f) (BuildId: 60380485a9793d0e873f8ea2c93e02efaa9aa3d)

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/workspace/ESBMC_Commit/release/bin/fuzz.c:10:14 in VulnerableFunction(unsigned char const*, unsigned long)
Shadow bytes around the buggy address:
 0x0c048080121d0: fa fa fd fa fa fd fa fa fd fa fa fd fa fa fd fa
 0x0c048080121e0: fa fa fd fa fa fd fa fa fd fa fa fd fa fa fd fa
 0x0c048080121f0: fa fa fd fa fa fd fa fa fd fa fa fd fa fa fd fa
 0x0c04808012200: fa fa fd fa fa fd fa fa fd fa fd fa fd fa fa fd fa
 0x0c04808012210: fa fa fd fa fa fd fa fa fd fa fd fa fd fa fa fd fa
==0x0c04808012220: fa fa fd fa fa fd fa fd fa fa fd fa fa fd fa [03] fa
 0x0c04808012230: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c04808012240: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c04808012250: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c04808012260: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c04808012270: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: f6
Poisoned by user: f7
Container overflow: fc
Array cookie: ac
Intra object redzone: bb
ASan internal: fe
Left alloca redzone: cc
Right alloca redzone: cb
==19046==ABORTING
MS: 4 CrossOver-ShuffleBytes-CopyPart-CrossOver; base unit: adc83b19e793491b1c6ea0f0b46cd9f32e52fc
0x46,0x55,0x5a,
FUZ
artifact_prefix='./; Test unit written to ./crash-8eb8e4ed029b774d80f2b6640d203801cb982a69
Base64: RLVA

```

Figure 2.14: Examples of Using LibFuzzer

likely to be interesting. A simple implementation would be to interpret the bytes provided by mutation-based, stored in the buffer as required, without using any other source, and apply them to either variable or structural mutation, where a structural parameter sequence to be consumed by structural choice points, and a value parameter sequence to be consumed by value choice points.[38]

Take the CFG in Figure 2.8 as an example. the nodes (rectangular blocks) represent the values and the edges (arrows) represent the structures. We can construct parameterised sequences based on the random bytes: (1) A structural parameter sequence to be consumed by structural choice points. (2) A value parameter sequence to be consumed by value choice points. Mutating the structural parameters changes the boolean decisions on the generation of child nodes. On the other hand, by mutating the value parameters only, the control-flow behaviour of the generator is preserved, yet different value choices are sampled. In the case of the CFG generator, mutating value parameters results in mutated choices for the node values, while keeping the shape of the binary tree unmodified. Overall, access to these structure-changing and structure-preserving mutations allows us to explore the input space in a more controlled manner.

Figure 2.15 demonstrates an example of a libFuzzer-based hybrid fuzzer for the Zlib compress library. The fuzzer accepts every random data and tries to uncompress it. The program crashes if the first two bytes of the uncompressed input are 'F' and 'U'. To avoid corruption of the original compression format due to mutations, a custom mutator (LLVMFuzzerCustomMutator) is added to provide a user-defined function with a fixed signature. Thus, the input data will be according to the specified language grammar and if it fails, return a syntactically correct input which, in this case, is 'H' and 'I'. The inputs then get mutated and compressed. Hence, every input that is received by the target function (LLVMFuzzerTestOneInput) is valid compressed data and successfully uncompressed.

```

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data
    , size_t Size)
{
    uint8_t Uncompressed[100];
    size_t UncompressedLen = sizeof(Uncompressed);
    if(Z_OK != uncompress(Uncompressed , &UncompressedLen ,
        Data , Size))
        return 0;
    if(UncompressedLen < 2) return 0;
    if(Uncompressed[0] == 'F' && Uncompressed[1] == 'U')
        abort(); // Error Triggered
    return 0;
}

extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data ,
                                           size_t Size ,
                                           size_t MaxSize ,
                                           uint8_t Seed)
{
    uint8_t Uncompressed[100];
    size_t UncompressedLen = sizeof(Uncompressed);
    size_t CompressedLen = MaxSize;
    if(Z_OK != uncompress(Uncompressed , &UncompressedLen ,
        Data , Size)) {
    }
    UncompressedLen =
        LLVMFuzzerMutate(Uncompressed , UncompressedLen ,
            sizeof(Uncompressed));
    if(Z_OK != compress(Data , &CompressedLen , Uncompressed ,
        UncompressedLen))
        return 0;
    return CompressedLen;
}

```

Figure 2.15: Example of Hybrid Fuzzing for Zlib Compress Library

Chapter 3

Methodology and Implementation

The aim of this Section is to provide an overview of the methods guiding the implementation of the new hybrid fuzzer. In Section 3.1, We will briefly explain the reasons for the construction of GotoFuzz by analysing the structure of ESBMC, in other words, using the ESBMC intermediate language Goto program as the object of fuzz tests. In order to maintain the correctness of the syntax of the Goto program when mutating it, in the following section 3.2, the features and syntax of the Goto language will be introduced and summarised. A visual example of a Goto program will be provided to explain the specific interpretation of the Goto instructions. In section 3.3, two methodologies are presented: the initial approach simply combines libFuzzer with Csmith. This will be the starting point of our design. The refined approach additionally introduces an improved version of the pseudo-random number generator. The design principles of both will be discussed and the limitations of the first approach will be illustrated, explaining why the second was chosen, which will be evidenced later by the benchmark in chapter 4.

3.1 Entry Point

As a first step in constructing a fuzzer, we first consider its possible outputs. Theoretically, this output can be any temporary variable passed in the ESBMC workflow. According to the structure we have described above, Although there is a variety of options for ESBMC's frontend, the parsers for different languages are all based on the APIs provided by the respective compilers. For example, ESBMC-Solidity which parses the smart contracts in the blockchain is constructed on the Solidity compiler, and the ESBMC-Java frontend uses the JVM. In addition, according to section 2.2.2,

c2Goto, which is another customised frontend parser, is also based on extensions to existing compilers clang. It is the same situation for the backend which also directly makes use of the existing and mature tool—SMT solvers. Both components are widely used and tested, which can be assumed to a certain extent of robustness. The middleware, however, appears to be the least tested part. This is due to several reasons. Firstly, middleware is seen as a black box and the complexity of its operational logic makes it difficult to design a white box fuzzer to test it. Secondly, because the variables input and output to the middleware have specific formatting requirements, it is tedious and difficult to summarise the syntax before constructing the input directly or performing mutation. Finally, as most of the work performed by the middleware is translation and transformation, its errors may be masked and left to the backend, where they are erroneous during execution, which can be misleading in terms of discovering the true source of the errors.

Therefore, we propose the GotoFuzz. We have chosen the Goto program as the output for several reasons:

1. Given the target of our fuzzing is the middleware, and the start of the internal passing process of the middleware is the Goto program, it is natural to construct a Goto-objective fuzzer.
2. The Goto program is both context-free and language-independent, reflected in no dependency on the front-end
3. Goto program is a demarcation between syntax and semantics. As described in Section 2.1.2, symbolic execution represents the input to a program symbolically, based on constraints obtained by analysing the semantics of the program, whilst the semantics in the program will be translated and executed. The correctness of both syntax and semantics is required in and beyond this component, yet previously only the syntax was required.
4. It's much easier to perform mutation on intermediate representation than on source code, as the code has already been parsed and stored in the uniform unit of symbol table/ Goto instructions. The vulnerable point is therefore the middleware, that is, the Goto converter and the symbolic engine. The main goal, therefore, is to create semantically arbitrary, in most cases incorrect, but syntactically correct Goto programs.

5. ESBMC provides internal development instructions to output intermediate expressions such as the Goto program and SSA formula, which allows the running progress to be easily traced.

3.2 Goto Grammar

To introduce a hybrid fuzzer to these component, the syntax grammar of the Goto program in ESBMC should firstly be summarised. The Goto program is a list of instructions, each of which has the type of instruction (one of 19 instructions), a code expression, a protection expression and possibly some target for the next instruction. A Goto instruction is defined by three properties, as the meaning of an instruction depends on the "instruction type" field while different kinds of instructions make use of the fields "guard" and "code" for different purposes. Instruction type describes the action performed by this instruction. Specifically speaking, the Instructions Type is an enum value describing the action performed by this instruction. Guard is an (arbitrarily complex) expression (usually an expert) of Boolean type. And Code represents a code statement, which can be seen as a unit in the symbol table. These are the properties that need to be considered when initialising a Goto-instruction data structure, any other property could be set default or generated afterwards. The type field determines the meaning of an instruction node, while the guard and code fields are used for a variety of purposes by different types of instructions [12]. Most importantly, Goto language is a context-free language, as no matter which symbols surround it, the single nonterminal on the left-hand side can always be replaced by the right-hand side. This is what distinguishes it from context-sensitive grammar, allowing to mutate it without violating its syntax.

The formal grammar of the Goto program language used in ESBMC can be shown in Figure 3.1. For simplification, we use the "id" of an expression directly to represent the corresponding expression, and "*" is denoted to represent a list of expressions/instructions.

1. `NO_INSTRUCTION_TYPE`: Instruction will be set to. `NO_INSTRUCTION_TYPE` if it is not explicitly defined.
2. `GOTO`: `GOTO` targets if and only if guard is true. The guard will be set to `TRUE` if it is not explicitly given.

3. ASSUME: This thread of execution waits for guard to evaluate to TRUE, which performs a non-failing guarded self loop.
4. ASSERT: An assertion is TRUE / safe if guard is TRUE in all possible executions, otherwise it is FALSE / unsafe. This instructions is used to express properties to be verified
5. OTHER: Represents an expression that gets evaluated, but does not have any other effect on execution, i.e. doesn't contain a call or assignment.
6. SKIP: Just advance the PC.
7. LOCATION: Semantically like SKIP.
8. ATOMIC_BEGIN, ATOMIC_END: Marks/ Ends a block without interleavings. When a thread executes ATOMIC_BEGIN, no thread other will be able to execute any Instruction until the same thread executes ATOMIC_END.
9. RETURN: Set the value returned by code (which shall be either nil or an instance of code_return_id) and then jump to the end of the function.
10. ASSIGN: Update the left-hand side of code (an instance of code_assign_id) to the value of the right-hand side.
11. DECL: Introduces a symbol denoted by the field code (an instance of code_decl_id). Semantically, the life-time of which is bounded by a corresponding DEAD instruction.
12. DEAD: Ends the life of the symbol denoted by the field code.
13. FUNCTION_CALL: Invoke the function denoted by field code (an instance of code_function_call_id).
14. THROW: Throw an exception. Throw *exception_1, ..., exception_N* where the list of exceptions is extracted from the code field
15. CATCH: Catch an exception.
16. THROW_DECL: List of throws that a function can throw.
17. THROW_DECL_END: End of throw declaration.

An example of a simple Goto program is shown in Figure 3.2, where we can make the assignment to `x` precede its definition by changing the order between instructions; in addition, we can break the contextual relationship between the target of the Goto instruction and the LABEL instruction by making their values no longer equal, thus changing the semantics without violating the syntax grammar rules. that the changes of names or values of variables would predictably be unhelpful.

<i>Program</i> ::=	<i>Instruction</i> * + <i>END_FUNCTION</i>
<i>Instruction</i> ::=	
	(<i>instruction_type</i>)
	(<i>instruction_type</i> , <i>guard</i>)
	(<i>instruction_type</i> , <i>code</i>)
<i>Instruction_type</i> ::=	
	<i>NO_INSTRUCTION_TYPE</i>
	<i>GOTO</i>
	<i>ASSUME</i>
	<i>ASSERT</i>
	<i>OTHER</i>
	<i>SKIP</i>
	<i>LOCATION</i>
	<i>END_FUNCTION</i>
	<i>ATOMIC_BEGIN</i>
	<i>ATOMIC_END</i>
	<i>RETURN</i>
	<i>ASSIGN</i>
	<i>DECL</i>
	<i>DEAD</i>
	<i>FUNCTION_CALL</i>
	<i>THROW</i>
	<i>CATCH</i>
	<i>THROW_DECL</i>
	<i>THROW_DECL_END</i>

Code ::=

	<i>code_block_id</i>
	<i>code_assign_id</i>
	<i>code_init_id</i>
	<i>code_decl_id</i>
	<i>code_dead_id</i>
	<i>code_printf_id</i>
	<i>code_expression_id</i>
	<i>code_return_id</i>
	<i>code_skip_id</i>
	<i>code_free_id</i>
	<i>code_goto_id</i>
	<i>code_asm_id</i>
	<i>code_function_call_id</i>
	<i>code_comma_id</i>
	<i>code_cpp_del_array_id</i>
	<i>code_cpp_delete_id</i>
	<i>code_cpp_catch_id</i>
	<i>code_cpp_throw_id</i>
	<i>code_cpp_throw_decl_id</i>
	<i>code_cpp_throw_decl_end_id</i>

Guard ::=

	<i>constant_bool_id</i>
	<i>constant_int_id</i>
	<i>constant_floatbv_id</i>
	<i>constant_fixedbv_id</i>

Instruction ::=

	<i>NO_INSTRUCTION_TYPE</i>
	<i>GOTO</i>
	<i>GOTO</i> + <i>guard</i>
	<i>ASSUME</i> + <i>guard</i>
	<i>ASSERT</i> + <i>guard</i>
	<i>OTHER</i> + <i>code_expression_id</i>
	<i>OTHER</i> + <i>code_free_id</i>
	<i>OTHER</i> + <i>code_print_f_id</i>
	<i>OTHER</i> + <i>code_asm_id</i>
	<i>OTHER</i> + <i>code_cpp_del_array_id</i>
	<i>OTHER</i> + <i>code_cpp_delete_id</i>
	<i>SKIP</i>
	<i>LOCATION</i>
	<i>ATOMIC_BEGIN</i> + <i>Instruction</i> * + <i>ATOMIC_END</i>
	<i>RETURN</i>
	<i>RETURN</i> + <i>code_return_id</i>
	<i>ASSIGN</i> + <i>code_assign_id</i>
	<i>DECL</i> + <i>code_decl_id</i>
	<i>DEAD</i> + <i>code_dead_id</i>
	<i>FUNCTION_CALL</i> + <i>code_function_call_id</i>
	<i>THROW</i> + <i>code_cpp_throw_id</i> *
	<i>CATCH</i> + <i>code_cpp_catch_id</i> *
	<i>THROW_DECL</i> + <i>code_cpp_throw_decl_id</i> *
	<i>THROW_DECL_END</i> + <i>code_cpp_throw_decl_end_id</i> *

Figure 3.1: The Grammar of Goto Program

```

DECL  x;
x = 1;
IF (x == 1) THEN GOTO 2;
...
LABEL 2;
...

```

Figure 3.2: Examples of a Goto Program

3.3 Intial version of GotoFuzz

We first consider the construction of the GotoFuzz prototype. According to our previous description of the hybrid fuzzer, structure-aware mutations look for interesting input structures in the space of valid inputs, which is mirrored in the Goto program’s randomization of the structure between instructions. We also use structure-preserving mutations to create distinct mutations of the same input structure to investigate alternative execution trails. This is accomplished by altering the values of context-sensitive variables[38].

In order to refer the random numbers generated by libFuzzer to the Goto program in a structural mutation, we first introduce the Durstenfeld Shuffle algorithm. It’s a modern version of Knuth shuffle aka Fisher-Yates shuffle. The original algorithm effectively puts all the elements into a hat; it continually determines the next element by randomly drawing an element from the hat until no elements remain, producing an unbiased permutation: every permutation is equally likely. The modern version has some slight improvement which reduces the algorithm’s time complexity to $O(n)$ compared to $O(n^2)$. The procedure can be demonstrated in the Algorithm. Function *Rand*(*i*, *j*) randomly creates number between *i* and *j* − 1. Array *target* stored the mutation object. Based on Durstenfeld’s method, we proposed our mutation algorithm for GotoFuzz shown in Algorithm 2. To cope with the seeds generated by libFuzzer, we replace function *Rand*() by array *seeds*[], such that the seeds can be used for structure-aware mutation.

Algorithm 1 Durstenfeld Shuffle Algorithm**Require:** target**for** $i \leftarrow 0$ to $n - 2$ **do** $j \leftarrow \text{Rand}(i, n)$ $\triangleright j$ is a random integer such that $i \leq j < n$ $(\text{target}[j], \text{target}[i]) \leftarrow (\text{target}[i], \text{target}[j])$ \triangleright exchange $a[j]$ and $a[i]$ **end for****Algorithm 2** GotoFuzz Mutation Algorithm**Require:** seeds, target**for** $i \leftarrow 0$ to $n - 2$ **do** $j \leftarrow (\text{seeds}[i] \% (n - 2))$ \triangleright replace function $\text{Rand}()$ by array $\text{seeds}[]$ $(\text{target}[j], \text{target}[i]) \leftarrow (\text{target}[i], \text{target}[j])$ **end for**

Next, we describe the implementation in detail. Given that Csmith and libFuzzer have been successfully used in the ESBMC project, it is natural to consider combining the two to construct our GotoFuzz.

Csmith acts as a generator of Goto programs, although this generation works indirectly. As Csmith claims to have comprehensive support for the language features under the C99 standard, there is almost no need to add extra code to ensure that all features of the Goto program are covered—If it is really necessary, we can simply add some manually-written test cases. In addition, Since the values of the variables in the C code generated by Csmith are randomised, we do not need to think about mutating them.

LibFuzzer acts as a mutator of Goto programs. Figure 3.3 shows the mutation workflow of sequential and non-sequential (branch) structures. First, libFuzzer takes the generated random data and stores it as the byte variable Data. Then we interpret and transform Data and store it as an 16-bit unsigned integer array. These random numbers are then used in a mutation on the structure of the Goto program, or CFG, which can be divided into Sequential and Non-Sequential (branch) structures depending on the type of structure between its instructions/nodes. The main difference between these two is the presence or absence of a selective branch structure, which is reflected in the actual code as a node containing a collection of IF-THEN-GO instructions. During the transformation from source code to the instruction GOTO, the Goto converter creates a variables "target" to indicate to target instruction after the instruction GOTO is performed. In Figure 3.3, the Goto program in *main.goto* uses the target number 1 to mark the instruction to be executed when the guard holds, otherwise, the next instruction will be executed in sequence. In the flowchart on the right, this selection

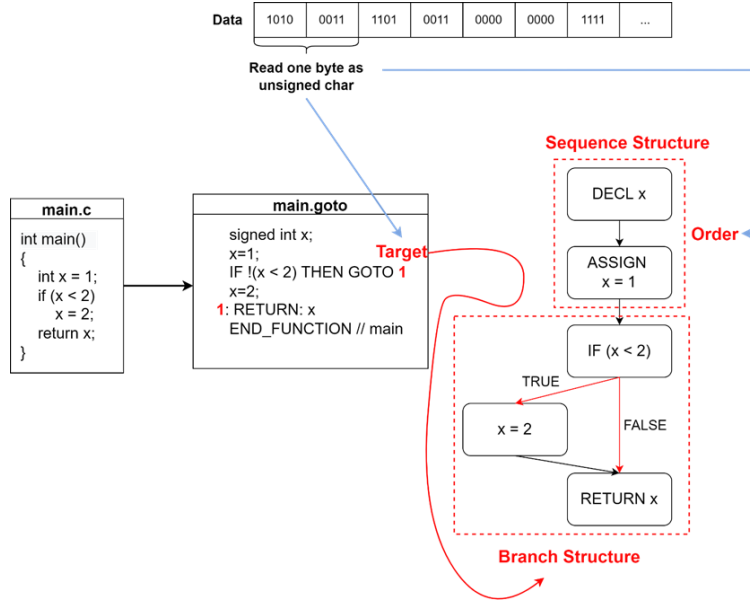


Figure 3.3: Mutation on Sequential and Non-Sequential Structure

structure is illustrated by the red edge. Therefore, in addition to considering changes to the order between nodes, extra mutation on the target needs to be considered. Figure 3.4 provides a detailed example which shows how the CFG changed after mutation in a round. For the reason of simplification, only one exchange occurs for each mutation. The mutated blocks/edges are illustrated in red. For a start, the Sequential mutation performed, resulting in the order of two blocks, indicating two instructions, got swapped. During the later branch mutation, the order of the two edge's targets got swapped. This performs a complete round of our GotoFuzz's mutation.

Figures 3.5 and 3.6 show the code implementations of the mutation for these two different structures. The first step common to both algorithms is to find the main function where the aimed Goto program is located. Once found, the sequence of random numbers used to guide the mutation is checked to see if it has been initialised. If not the randomly generated Data array from libFuzzer will be read. With the random number seed array in hand, we also need to extract the object array for the mutation algorithm. For sequential mutations, the object array will be the set of all instructions in the Goto program except `END_FUNCTION`. For non-sequential variants, the object array will be the set of target instructions. After the initialisation of the random number array and the object array have been completed, both functions will perform the actual mutation operation by the GotoFuzz mutation algorithm. The result of the mutated Goto

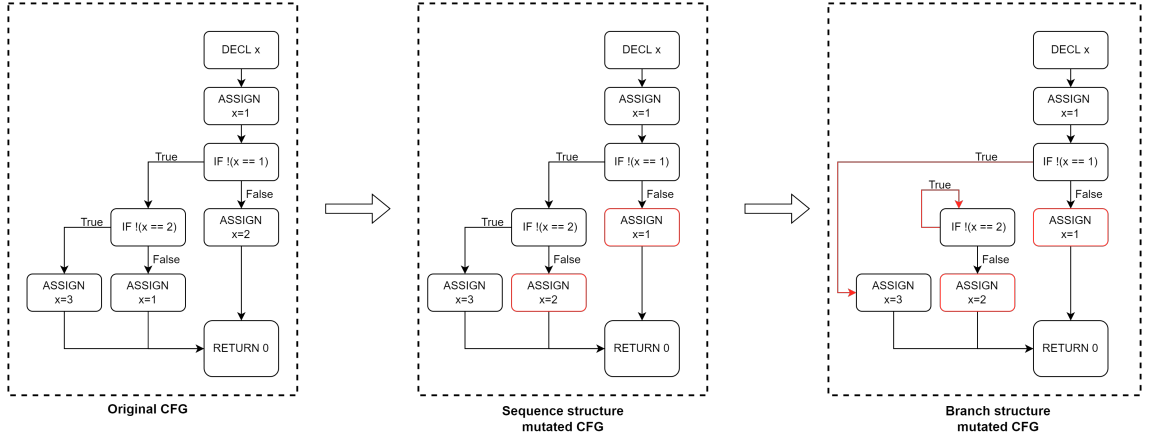


Figure 3.4: Example of CFG Mutation. The mutated blocks/edges are illustrated in red.

Program will be printed out.

Figure 3.7 provides an example result of these mutations in GotoFuzz, where GotoFuzz performed one round of sequential and non-sequential mutation on the Goto program. In the first transformation, the instructions "float y;" and "dead x;" were swapped randomly, while in the second transformation, the order of Goto's target labels changed from [1,2,3,4] to [3,1,2,4], and the target numbers in the Goto instructions changed accordingly.

Based on the discussion above, the construction for GotoFuzz can now be described. The general architecture has shown in Figure 3.8. Before testing. To start testing, GotoFuzz will first generate one or several random C/C++ files using Csmith (or provide its own test cases if required). These files will then be passed through the parser (in this case, clang) and goto converter or directly through c2goto to obtain the goto program. The Goto program will be read by the mutator. A random seed is needed in order to work. Hence, libFuzzer will generate a random length string of bytes and store it in a buffer. We will mutate the target with the help of the mutation algorithm and the byte string will be interpreted as a uint16_t array. Considering that the maximum value of uint_16 reaches 65535, this is sufficient in the case of structure mutation. The mutations are performed by first performing a round of sequential structural mutations followed by a round of non-sequential structural mutations, as shown in Figures 3.5 and 3.6. The generated and mutated Goto program will be further fed into the subsequent components for symbolic execution, loop expansion, the addition of safety properties and SMT solving. We will print the goto program in the process

```

bool mutateSequence(
    message& &msg,
    goto_functionst &func) //const uint8_t *data, size_t
    size,
{
    std::ostringstream os;
    if(m_it != func.function_map.end())
    {
        goto_programt &mmain = m_it->second.body;
        int program_len = mmain.instructions.size();
        if(!hasSeeds())
        {
            if(Data == NULL)
            {
                setPseudoSeeds(mmain);
            }
            else
            {
                setSeeds(mmain);
            }
        }
        std::vector<goto_programt::instructiont::targett>
            instructions;
        os << "Show_original_structure.\n";
        output(mmain, os, msg);
        Forall_goto_program_instructions(it, mmain)
        {
            if((*it).type == END_FUNCTION)
            {
                continue;
            }
            instructions.push_back(it);
        }
        Shuffle(instructions, seeds);

        os << "Show_mutated_sequence_structure.\n";
        output(mmain, os, msg);
    }

    msg.status(os.str());
    return false;
}

```

Figure 3.5: Mutation algorithm for Sequential code structure


```

bool mutateNonSequence(message &msg, goto_functionst &
    func)
{
    std::ostringstream os;
    if (m_it != func.function_map.end())
    {
        goto_programt &mmain = m_it->second.body;
        int program_len = mmain.instructions.size();
        if (!hasSeeds())
        {
            if (Data == NULL)
            {
                setPseudoSeeds(mmain);
            }
            else
            {
                setSeeds(mmain);
            }
        }
        std::vector<goto_programt::instructiont::targett>
            targets;
        Forall_goto_program_instructions(it, mmain)
        {
            if ((*it).has_target())
            {
                targets.push_back((*it).get_target());
            }
        }
        Shuffle(targets, seeds);
        os << "Show_mutated_non-sequence_structure.\n";
        output(mmain, os, msg);
    }

    msg.status(os.str());
    return false;
}

```

Figure 3.6: Mutation algorithm for Non-Sequential code structure

```

Show original structure.
    signed int x;
    x=1;
    float y;
    y=2.200000e+0f;
    IF !(x == 1) THEN GOTO 1
    x=2;
    GOTO 3
1: IF !((double)y == 2.200000e+0) THEN GOTO 2
    y=2.000000f;
    GOTO 3
2: x=(signed int)y;
3: IF !(x > 0) THEN GOTO 4
    x=x - 1;
    GOTO 3
4: dead y;
    dead x;
    RETURN: NONDET(signed int)
    END_FUNCTION // main

Show mutated sequence structure.
    signed int x;
    x=1;
    dead x;
    y=2.200000e+0f;
    IF !(x == 1) THEN GOTO 1
    x=2;
    GOTO 3
1: IF !((double)y == 2.200000e+0) THEN GOTO 2
    y=2.000000f;
    GOTO 3
2: x=(signed int)y;
3: IF !(x > 0) THEN GOTO 4
    x=x - 1;
    GOTO 3
4: dead y;
    float y;
    RETURN: NONDET(signed int)
    END_FUNCTION // main

Show mutated non-sequence structure.
    signed int x;
    x=1;
    dead x;
    y=2.200000e+0f;
    IF !(x == 1) THEN GOTO 3
    x=2;
    GOTO 4
3: IF !(x > 0) THEN GOTO 2
    y=2.000000f;
    GOTO 4
1: IF !((double)y == 2.200000e+0) THEN GOTO 1
4: dead y;
    x=x - 1;
    GOTO 4
2: x=(signed int)y;
    float y;
    RETURN: NONDET(signed int)

```

Figure 3.7: Result of mutation execution

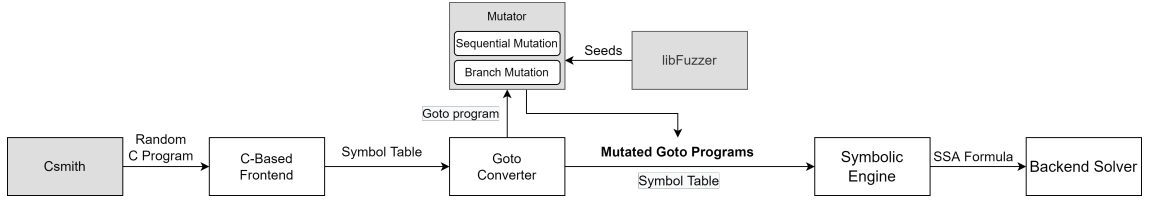


Figure 3.8: The Overview of Initial GotoFuzz. White rectangles represent the components of ESBMC; grey rectangles represent the components of the GotoFuzz

and monitor whether it works as expected, such as whether it remains syntactically correct.

We will then discuss how to stabilise and reproduce the results. First, as the Csmith generated C file is naturally preserved, we can generate the same initial Goto program. We then fix the initial seed of libFuzzer, which in our implementation has a value of 1. Since libFuzzer will only keep the seeds it finds interesting samples based on code coverage improvement, some seeds that might also be potentially valuable are ignored. Therefore, after each round of mutation, GotoFuzz needs to output the Goto program as a binary (this is in the same file format as the Goto binaries output by c2goto). When the same environment needs to be reproduced, the goto binary file can simply be re-read. This also tests the Goto input and output modules of ESBMC at the same time. Finally, we will save the seeds that are automatically output by libFuzzer as well as those whose output causes errors or blocking. These seeds can be imported into libFuzzer and the results reproduced.

Figure 3.9 shows the results of GotoFuzz’s procedure. Here we set the maximum run time of GotoFuzz’s internal libFuzzer to 30 seconds. The fuzzer will loop through the process of ”generating a mutated Goto program and sending it to ESBMC for execution”. At the end of the run (and without any errors causing termination), GotoFuzz will output the interesting seeds of the round and print a short summary, including the run time 31 seconds and the number of rounds executed which is 5368.

3.4 Improved version of GotoFuzz

The previous design seemed to work fine, but an important issue was overlooked, as there was no corpus provided for the initialization of libFuzzer. For the general case, libFuzzer can still find interesting inputs quickly with the help of coverage bootstrapping when not provided with test samples, which, however, is not the case for fuzzing

```

VERIFICATION SUCCESSFUL
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
Starting Bounded Model Checking
Symex completed in: 0.001s (12 assignments)
Slicing time: 0.000s (removed 11 assignments)
Generated 0 VCC(s), 0 remaining after simplification (1 assignments)
BMC program time: 0.002s

VERIFICATION SUCCESSFUL
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
Starting Bounded Model Checking
Symex completed in: 0.001s (12 assignments)
Slicing time: 0.000s (removed 11 assignments)
Generated 0 VCC(s), 0 remaining after simplification (1 assignments)
BMC program time: 0.001s

VERIFICATION SUCCESSFUL
#5368 DONE cov: 11919 ft: 22667 corp: 261/2322b lim: 11 exec/s: 173 rss: 106Mb
##### Recommended dictionary. #####
"\0007" # Uses: 97
"\377\377\377\377\377\377\377D" # Uses: 19
"\001\000\000\001" # Uses: 31
"\000\000\000>" # Uses: 29
"\002K" # Uses: 33
"\377\377\377\010" # Uses: 29
"\001\000\000\000" # Uses: 9
"6\000" # Uses: 7
".\000" # Uses: 9
"\0004" # Uses: 17
"\030\000\000\000\000\000\000\000" # Uses: 8
"\001\000\000\000\000\000\000\017" # Uses: 3
"\000\000\000\005" # Uses: 8
"\000\000\001@" # Uses: 8
"\3774" # Uses: 5
"\377\000" # Uses: 10
"\375\377\377\377" # Uses: 2
"\001\000\000\000\000\000\000\001" # Uses: 0
##### End of recommended dictionary. #####
Done 5368 runs in 31 second(s)

```

Figure 3.9: The Result of the GotoFuzz

ESBMC. In fact, after testing, libFuzzer's seed values only increased slowly and linearly when working with ESBMC. This results in that, although libFuzzer inside GotoFuzz still kept finding new seeds, they failed to reach the length of the sample Goto program, leading to no improvement of the general code coverage. This makes the seed less capable of mutation when its length is significantly less than the goto program length. Besides, this problem cannot be simply solved by padding or repeating, as both strategies do not work for swap-based mutation algorithms. What's even worse, constructing corpus directly for libFuzzer is difficult, as the sample of seeds corresponds to a specific C-source or Goto program, and providing large seeds to smaller programs would compromise efficiency while providing small seeds to larger programs limits the ability of fuzzer mutation.

Based on the discussion above, two improvement strategies are proposed:

- Strategy-1.** We introduce a filter. Since libFuzzer only provides the option to set the maximum length of the seed and not the minimum length, our algorithm sets a minimum threshold for the length of libFuzzer and will return directly when the seed length is less than the number of instructions in the Goto program (the number of targets is always less than or equal to the number of instructions). Skipping the code execution of the section allows libFuzzer to avoid the effort of exploring on small seeds.
- Strategy-2.** We introduce a pseudo-random number generator. A pseudo-random number generator (PRNG) is an algorithm that generates sequences of numbers with properties that approximate those of a sequence of random numbers. The idea is to generate corpora with the required size via PRNG so that libFuzzer can generate seeds with similar lengths based on these corpora.

The code implementation of PSNR is shown in Figure 3.10. First the length of the Goto program, in other words, the number of Goto instructions it contains, is counted. Based on the program length, the data array with length $program_len + 1$ is declared, which will play the role of a random rand array in the mutation algorithm. Next, since the END_FUNCTION instruction is excluded, the subscript of the array of exchanged instructions has a maximum value of $program_len - 1$. The initialisation of the data array is completed by randomly assigning each cell of the data array a value in the range from 0 to $program_len - 1$.

The PRNG is thought to have the advantage of high generation speed. In addition, the PRNG works independently, meaning that the results of the last round of fuzzing do

```

void goto_mutationt::setPseudoSeeds(goto_programt &mmain)
{
    int program_len = mmain.instructions.size();
    uint16_t data[program_len + 1];
    std::random_device os_seed;
    const u32 seed = os_seed();

    engine generator(seed);
    std::uniform_int_distribution<u32> distribute(0,
        program_len - 1);
    for(auto d : data)
    {
        d = uint16_t(distribute(generator));
    }
    seeds = data;
}

```

Figure 3.10: Mutation algorithm for Non-Sequential code structure

not affect the next. However, the random generation of the seed performed by PRNG cannot be guided by coverage information. To check its differences with libFuzzer, we ran tests specifically to allow PRNG to fully initialise GotoFuzz's random arrays to test whether it could replace the role of libFuzzer. The steps can be listed as follows:

- We generated 10 random C programs via Csmith
- we repeatedly performed 100 rounds of fuzzing via GotoFuzz based on each C program
- Given that the results of each execution are random, and to increase statistical validity and readability of the statistics, we counted the impact on code coverage in groups of 10
- We take the average of the data from the 10 programs as the final result

The result has been shown in Table 3.1. The average code coverage from mutation-based fuzzing increases at the beginning, yet this growth soon stopped after 30 rounds. This is because the PRNG-based mutator runs completely in a black box, without tracking and sensing the running state of the test software, making it impossible to find new interesting seeds in the later stages of mutation. Several conclusions can be drawn shown as below. Based on the conclusions above, we had the idea of using PRNG to generate seeds for the corpus sample.

	0	10	20	30	40	50	...	100
Lines	40.30%	42.00%	42.00%	42.10%	42.10%	42.10%	...	42.10%
Functions	43.60%	44.40%	44.40%	44.50%	44.50%	44.50%	...	44.50%

Table 3.1: Coverage Data with PRNG-based Mutator

- PRNG cannot be a replacement for libFuzzer
- PRNG-based seeds produce a significant increase in code coverage with only a few initial mutations
- Although the initial mutation based on PRNG can quickly lead to an increase in code coverage, such an increase is bottlenecked within a few rounds of execution.

Figure 3.11 shows the architecture of the improved version of GotoFuzz. Based on the first version, two new components, PRNG and Corpus, have been introduced. Compared to the previous approach, After the C program has been transformed into a Goto program by parser and generator, the PRNG-Mutator additionally performs 10 rounds of loops. The resulting pseudo-random numbers are seeded and output as integers to Corpus. With the help of Corpus, the length of the seeds generated by the fuzzer grows rapidly, as shown in Figure 3.12. After the first 10 rounds of corpus execution, libFuzzer started initialization in round 11, and in round 12 created a test input that covers new areas of the code under test. Note that the length of the seed is directly close to the set "max-len" limit and not filtered according to the **Strategy 1** mentioned above (if filtered libFuzzer will print "pulse" indicating an empty round, rather than "New"). Finally, libFuzzer-Mutator reads the seed sample and performs structure-aware coverage-guided mutation. The mutated Goto program is then re-fed to ESBMC for the following operations.

3.5 Summary

This section describes two methodologies to implement the GotoFuzz. The first method simply combines libFuzzer with Csmith. The drawbacks of this implementation were discussed and an additional enhancement component PRNG was introduced. We also examined the possibility of PRNG as a mutator, and it proved difficult for PRNG-based

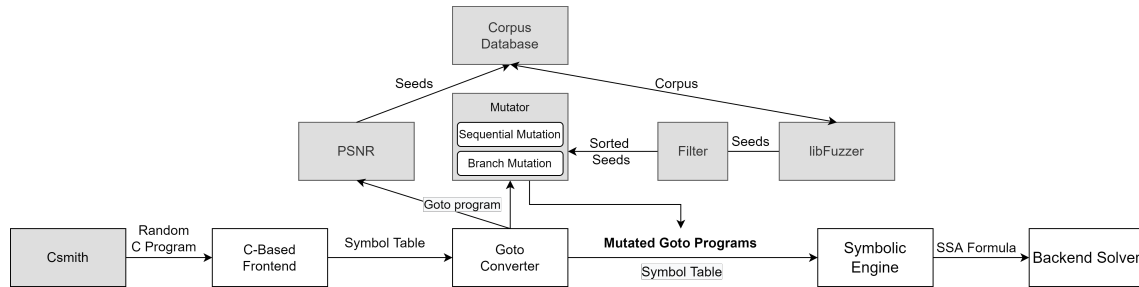


Figure 3.11: The Overview of Improved GotoFuzz. White rectangles represent the components of ESBMC; grey rectangles represent the components of the GotoFuzz

```
Symex completed in: 0.002s (22 assignments)
Slicing time: 0.000s (removed 19 assignments)
Generated 0 VCC(s), 0 remaining after simplification (3 assignments)
BMC program time: 0.002s

VERIFICATION SUCCESSFUL

Solution found by the forward condition: all states are reachable (k = 1)
#11 INITED cov: 23373 ft: 53875 corp: 10/480b exec/s: 0 rss: 132Mb
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
*** Checking base case, k = 1
Starting Bounded Model Checking
Not unwinding
Not unwinding
Symex completed in: 0.020s (125 assignments)
Slicing time: 0.000s (removed 55 assignments)
Generated 45 VCC(s), 35 remaining after simplification (70 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.005s
Solving with solver Boolector 3.2.2
Encoding to solver time: 0.005s
Runtime decision procedure: 0.052s
Building error trace

Counterexample:

State 3 thread 0
-----
  argv[2] = 0

State 5 file random.c line 50 function main thread 0
-----
Violated property:
  file random.c line 50 function main
  dereference failure: array bounds violated

VERIFICATION FAILED

Run found (k = 1)
#12 NEW cov: 23374 ft: 53983 corp: 11/528b lim: 52 exec/s: 0 rss: 132Mb L: 48/52 MS: 1 InsertByte-
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
*** Checking base case, k = 1
Starting Bounded Model Checking
```

Figure 3.12: The Length of The Seeds Grows Rapidly

GotoFuzz to consistently find new interesting seeds. Therefore, an improved fuzzer is proposed and significantly more efficient than the former, as the generated seeds have a closer length required by the mutator.

Chapter 4

Evaluation

This section summarizes our benchmarks and deliverables. The project can be evaluated from the two aspects. First, as the primary purpose of fuzzing is to use it to find hidden vulnerabilities in the program, thus the ability of error detection should be evaluated. One possible approach is to quantify the statistics of distinct crash errors found, rates of the crash and wrong-code errors from different versions of ESBMC, as well as the statistics of Bug-Finding Performance as a function of test-case size. Secondly, the improvement of code coverage will be measured. Tools for statistical code coverage are provided in ESBMC and can be output visually.

The information of the experiment environment can be listed as follows:

- System: Ubuntu 22.04
- Clang: version 14.0.0-1 Ubuntu
- ESBMC: version 6.10.0, 64-bit, x86_64
- CPU: AMD 5800H

4.0.1 Vulnerability Detection

A potential vulnerability was discovered when ESBMC was stress tested with GotoFuzz. The mutated Goto program was verified by ESBMC with unterminated unwinding, as shown in Figure 4.1. This means that there is an infinite loop in the source program and no limit on the number of unwinds is set, resulting in an infinite number of times of loop unwinding. Hence we manually limit the unwinding times by option "--unwind 10", as shown in Figure 4.1. After 9 iterations, the message "Not unwinding" appears on the 10th and the loop starts again with another round. This means that

[illegible]

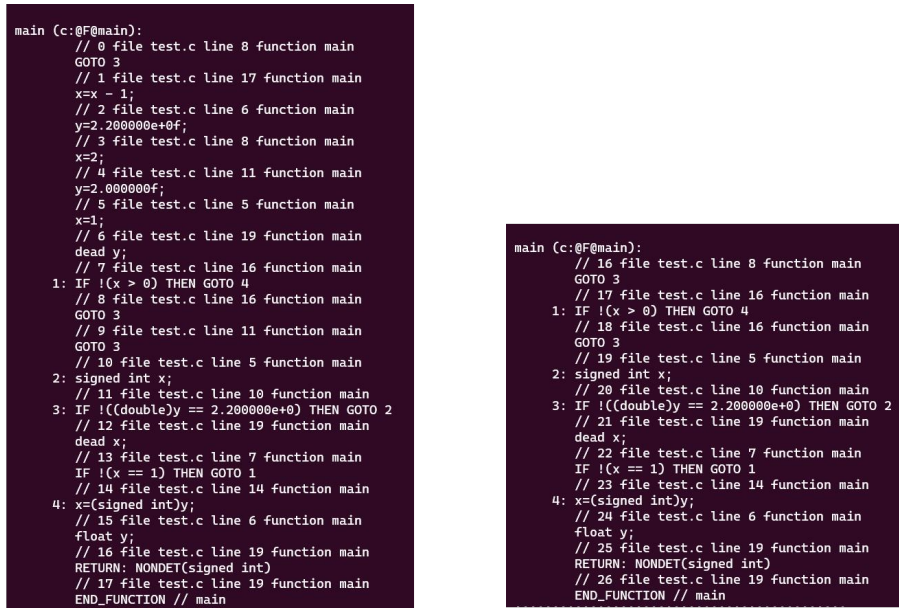
Figure 4.1: Unwinding is Not Terminated

unwinding is not working correctly. To exclude the effects of the different backend, we repeated our test in various SMT solvers, including Boolector, Z3 and CVC4, yet showed the same results. This error will cause the ESBMC verification process to get stuck until GotoFuzz reaches the timeout set by GotoFuzz.

Another abnormal phenomenon was observed. As described in section 3.2, a Goto binary file will be outputting during the mutation. After re-reading this binary file into ESBMC, we found that the Goto program was not identical to the previous Goto program. Specifically speaking, some instructions were "missing" from the Goto program after input. Figure 4.2 shows an example of this phenomenon. We have set the option "--unwind 10" to limit the times of unwinding. The Goto program on the left stands for the original mutated program and the the right represents the re-read program. Ignoring the difference in the positional information in the annotations, we can see that the program on the right is missing some instructions compared to the program on the left, including the ASSIGN and DEAD instructions. Another important difference is that the program on the right can be verified by ESBMC and get the correct result, compared to the program on the left which will block ESBMC.

To exclude the effect of the different operating modes, we also tested in incremental BMC mode with the "--k-induction" option enabled. The result has been shown in Figure 4.3. Compared to the original mutated program, it can be seen that the re-read program is also missing some instructions, including an IF-Then-GOTO instruction. Similarly, the procedure on the right can be verified by ESBMC with no error occurred. Comparing this to Figure 4.2, it appears that these instructions are missing at random. Furthermore, we find that in both Figures 4.2 and 4.3 there is an infinite loop structure in the program on the left. For example, in Figure 4.3 there is a loop of "3:NONDET(signed int)" (which is additionally inserted by ESBMC in Incremental BMC mode) and "GOTO 3". This loop is not unwinding correctly and this leads to the error in Figure 4.1. A final observation is that the probability of such errors occurring in actual testing is quite low.

The reasons for this exception were analysed. Firstly, we analyse whether the error is caused by a bug in GotoFuzz itself. The original parts of GotoFuzz include the mutator that drives the fuzzer and the PRNG that generates random seeds for corpus database. Moreover, both Csmith and libFuzzer can be considered robust. The rest of the feather, including the instruction swapping and Goto binary I/O, are all implemented by using the API provided by ESBMC. Therefore, this is largely possible to rule out a bug in GotoFuzz itself. Secondly, we deduce that the problem is related to



```

main (c:@F@main):
// 0 file test.c line 8 function main
GOTO 3
// 1 file test.c line 17 function main
x=x-1;
// 2 file test.c line 6 function main
y=2.200000e+0f;
// 3 file test.c line 8 function main
x=2;
// 4 file test.c line 11 function main
y=2.000000f;
// 5 file test.c line 5 function main
x=1;
// 6 file test.c line 19 function main
dead y;
// 7 file test.c line 16 function main
1: IF !(x > 0) THEN GOTO 4
// 8 file test.c line 16 function main
GOTO 3
// 9 file test.c line 11 function main
GOTO 3
// 10 file test.c line 5 function main
2: signed int x;
// 11 file test.c line 10 function main
3: IF !((double)y == 2.200000e+0) THEN GOTO 2
// 12 file test.c line 19 function main
dead x;
// 13 file test.c line 7 function main
IF !(x == 1) THEN GOTO 1
// 14 file test.c line 14 function main
4: x=(signed int)y;
// 15 file test.c line 6 function main
float y;
// 16 file test.c line 19 function main
RETURN: NONDET(signed int)
// 17 file test.c line 19 function main
END_FUNCTION // main

```

```

main (c:@F@main):
// 16 file test.c line 8 function main
GOTO 3
// 17 file test.c line 16 function main
1: IF !(x > 0) THEN GOTO 4
// 18 file test.c line 16 function main
GOTO 3
// 19 file test.c line 5 function main
2: signed int x;
// 20 file test.c line 10 function main
3: IF !((double)y == 2.200000e+0) THEN GOTO 2
// 21 file test.c line 19 function main
dead x;
// 22 file test.c line 7 function main
IF !(x == 1) THEN GOTO 1
// 23 file test.c line 14 function main
4: x=(signed int)y;
// 24 file test.c line 6 function main
float y;
// 25 file test.c line 19 function main
RETURN: NONDET(signed int)
// 26 file test.c line 19 function main
END_FUNCTION // main

```

Figure 4.2: The Difference Between Goto Programs Before and After Output

the symbol table, which exported at the same time as the Goto program is exported as a binary file. Although the mutator is not expected to make any changes to the symbol table, it is possible that changes are actually made, resulting in symbols in some instructions not being recognised and ignored. This is consistent with the previously described "instructions missing" scenario. The danger of this vulnerability is that it may cause ESBMC to run out of memory and report errors in endless unwinding. In practice, it is unlikely that the structure of a Goto program will be modified directly in the way that Fuzzer is. However, the potential triggers can be the insertion of code when adding security verification attributes, or parallel execution. Furthermore, we find that in both Figures 4.2 and 4.3 there is an infinite loop structure in the program on the left. For example, in Figure 4.3 there is a infinite loop from "3:NONDET(signed int)" (which is additionally inserted by ESBMC) to "GOTO 3". This loop is not unwinding correctly, leading to the error in Figure 4.1.

The effectiveness of the GotoFuzz can also be demonstrated by comparing it to other structured mutation approaches. A similar test method was used in Fink's study of the BMC performance benchmark, where the structure of the sample's code was mutated, more specifically, "functionalizing post-processor randomly replaces arbitrary sub-expressions by function calls to newly created functions encapsulating the cut sub-expression, where constants and variables within are randomly either used as

```

main (c:@F@main):
  // 0 file test.c line 19 function main
  dead x;
  // 1 file test.c line 7 function main
  IF !(x == 1) THEN GOTO 1
  // 2 file test.c line 6 function main
  float y;
  // 3 file test.c line 5 function main
  x=1;
  // 4 file test.c line 6 function main
  y=2.200000e+0f;
  // 5 file test.c line 11 function main
  y=2.000000f;
  // 6 file test.c line 10 function main
  IF !((double)y == 2.200000e+0) THEN GOTO 2
  // 7 file test.c line 16 function main
1: ASSUME x > 0
  // 8 file test.c line 16 function main
  GOTO 4
  // 9 file test.c line 8 function main
  x=2;
  // 10 file test.c line 5 function main
2: signed int x;
  // 11 file test.c line 16 function main
3: x=NONDET(signed int);
  // 12 file test.c line 11 function main
  GOTO 3
  // 13 file test.c line 19 function main
4: dead y;
  // 14 file test.c line 14 function main
  x=(signed int)y;
  // 15 file test.c line 8 function main
  GOTO 3
  // 16 file test.c line 16 function main
5: IF !(x > 0) THEN GOTO 5
  // 17 file test.c line 17 function main
  x=x - 1;
  // 18 file test.c line 19 function main
  RETURN: NONDET(signed int)
  // 19 file test.c line 19 function main
  END_FUNCTION // main

```

```

main (c:@F@main):
  // 16 file test.c line 19 function main
  dead x;
  // 17 file test.c line 7 function main
  IF !(x == 1) THEN GOTO 1
  // 18 file test.c line 6 function main
  float y;
  // 19 file test.c line 5 function main
  x=1;
  // 20 file test.c line 6 function main
  y=2.200000e+0f;
  // 21 file test.c line 11 function main
  y=2.000000f;
  // 22 file test.c line 10 function main
  IF !((double)y == 2.200000e+0) THEN GOTO 2
  // 23 file test.c line 16 function main
1: ASSUME x > 0
  // 24 file test.c line 16 function main
  GOTO 4
  // 25 file test.c line 5 function main
2: signed int x;
  // 26 file test.c line 16 function main
3: x=NONDET(signed int);
  // 27 file test.c line 11 function main
  GOTO 3
  // 28 file test.c line 19 function main
4: dead y;
  // 29 file test.c line 14 function main
  x=(signed int)y;
  // 30 file test.c line 8 function main
  GOTO 3
  // 31 file test.c line 19 function main
  END_FUNCTION // main

```

Figure 4.3: The Difference when Using Incremental BMC Mode

function parameters or kept as is”[3]. It can be seen that their approach to structural changes differs from our design, but both affect the structure of the code while ensuring that the syntax is correct and the values of the variables remain the same. The fact that no error was found in Finks’ research demonstrates the superiority of GotoFuzz, which does better in following aspect:

1. The Goto program outputs the mutation process and the result in its entirety, which not only facilitates our observation but also facilitates reproduction.
2. In Fink’s benchmark, ”Error” and ”TimeOut” is divided into two separate categories. This division led to potential vulnerabilities being obscured, as timeouts can be caused not only by performance bottlenecks but also by bugs leading to infinite loops. This may be because that Fink’s experiments focus more on testing performance and less on the analysis of errors. In contrast, the GotoFuzz succeeded in discovering hidden vulnerabilities in ESBMC and provided conjectures and explanations for the possible causes.

4.0.2 Coverage Improvement

The improvement in coverage is another test of Fuzzer's ability. To collect the coverage information, Lcov[39] and genhtml[40] are applied with both version 1.14. Information on line coverage and function coverage is collected and represents different test metrics. Line coverage checks if each executable line of code is executed, while function coverage checks if each function gets invoked. Figure 4.4 shows a report of the coverage information. The following steps is needed to generate this report:

1. enabling the "--coverage" option when compiling ESBMC with Clang
2. use one or more programs to have ESBMC validated
3. generate the report via command-line tools Lcov and genhtml

The report is divided into two tables, the top left table shows the overall line coverage and function coverage information, with additional 'Hit' and 'Total' information for each type of coverage. The large table below shows the coverage information by source files' directory representing corresponding ESBMC components. For example, the "c2goto" directory represents the c2goto component, while "big-int" represents the container for handling large numbers.

In theory, GotoFuzz works on top of parsed IR and therefore has no impact on the code coverage of the front-end. Furthermore, GotoFuzz focuses on testing ESBMC's middleware, so changes in code coverage of middleware-related components before and after fuzzing should be monitored. Finally, as the goto program gets changed, this makes it possible for the translated SMT formula to change as well, so backend-related coverage information will also be collected. We have summarised some of the information that we will be collecting and listed it below.

1. esbmc: This reflects the overall ESBMC situation
2. goto-program: This reflects the use of the goto converter and the APIs associated with the manipulation of goto programs. These APIs include exchanging goto program instructions, exporting and reading Goto binaries.
3. goto-symex: This reflects the execution of SSA formula translations and semantic execution by the Symbolic engine.
4. irep2: Irep stands for interpretation representation. he is the prototype and the underlying layer of goto programs.

5. solvers: This reflects the back-end validation of the SMT formula
6. util: This represents the methods that are shared and common to all ESBMC components

Here, we look at GotoFuzz from four perspectives. Firstly we will look at the improvement in code coverage when using a test suite consisting of a single file. Next, we will look at the improvement in code coverage when using a test suite made up of multiple files. The comparison between the two will reflect the percentage increase in code coverage relative to a single file. Next, we will look at the effect of individual file size on code coverage. We guess that File size and coverage are positively correlated, yet the time cost will be significantly higher with the file size growth. Finally, we collect and analyse the results when different command line arguments are set. For example, when using the “–unwind n ” option to limit the times of unwinding, the code coverage can be compared in different n set.

coverage growth based on individual sample

According to the description in Section 3.2, GotoFuzz accepts a C file, mutates and verifies it via ESBMC. Thus, there are two objects for comparison. One is the code coverage with the C file verified directly using ESBMC, and the other is the code coverage after applying GotoFuzz. In order to collect the data, the following steps were carried out:

1. construct samples via Csmith. To reduce the error due to randomness, we generated 20 samples. Each sample was generated with the same Csmith command and with default parameters
2. feed each sample to ESBMC and collect the information. The backend is set uniformly to Z3. Each round was run with the same command with an additional setting “–unwind 1”, which limits the unwinding times to 1. This was done to avoid the bugs mentioned in Section 4.1 as much as possible by reducing loop unrolling.
3. feed each sample to GotoFuzz and collect the information. The backend is set uniformly to Z3. Additionally, we manually set libFuzzer to run for a maximum of 15 minutes and a maximum of 2 GB of memory, while running the same command for each round of ESBMC and setting the unwinding times to 1.

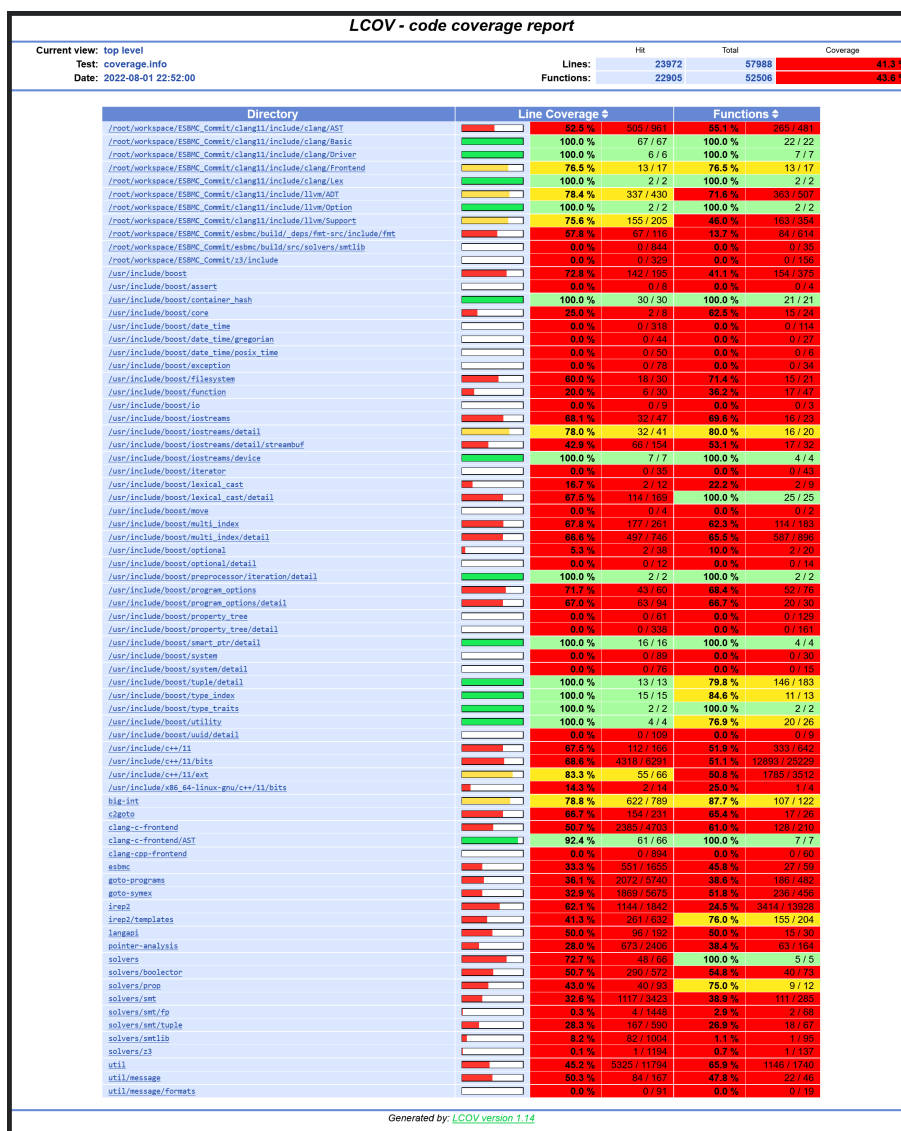


Figure 4.4: Example of Coverage Report

	line	function	esbmc	goto- program	goto- symex	irep2	solvers/ z3	util
Pre	39.20%	40.30%	31.30%	35.30%	30.40%	63.20%	26.30%	44.90%
Post	42.50%	43.30%	37.00%	38.80%	33.50%	65.50%	34.80%	48.70%
Improve	3.00%	2.10%	5.70%	3.60%	2.80%	1.80%	7.50%	3.10%

Table 4.1: Sample with the Most Coverage Improvement

	line	function	esbmc	goto- program	goto- symex	irep2	solvers/ z3	util
Pre	38.79%	39.89%	31.31%	34.88%	30.20%	62.80%	26.35%	44.03%
Post	41.78%	41.98%	37.01%	38.38%	33.05%	63.80%	34.00%	47.40%
Improve	2.99%	2.09%	5.70%	3.50%	2.85%	1.00%	7.65 %	3.38%

Table 4.2: Average Improvement Based on Individual Sample

4. calculate the average code coverage of the 20 samples generated in steps 2 and 3.

Table 4.1 shows the samples with the most line/function coverage improvement. Table 4.2 illustrates the average results for the 20 samples. "Pre" stands for Pre-fuzzing mentioned in step 2. "Post" stands for Post-fuzzing mentioned in step 3. "Improve" stands for the difference between "Post" and "Pre". This value represents the improvement in code coverage brought about by GotoFuzz. Some conclusions can be drawn here. For the overall code coverage, the improvement in line coverage is about 3%, while the improvement in function coverage is about 2%. Furthermore, the largest improvement of the components is 7.65% (solvers/z3) and the smallest is 1% (irep2). Lastly, comparing Tables 4.1 and 4.2, we can see that although the sample in Table 4.1 has a greater line/function code coverage and improvement, it does not mean that this corresponds to the case of every component (solvers/z3).

coverage growth based on multiple samples

GotoFuzz has proved his improvement in code coverage based on individual random C files. Thus, a natural thought is whether these improvements will be accumulated when using multiple files as test suites. Therefore, we generated seven random files using Csmith and fed them into ESBMC and GotoFuzz consecutively and synchronously.

	1	2	3	4	5	6	7
Pre	38.90%	38.90%	39.00%	39.00%	39.00%	39.01%	39.01%
Post	41.90%	42.00%	42.20%	42.20%	42.20%	42.40%	42.70%
Improve	3.00%	3.10%	3.20%	3.20%	3.20%	3.30%	3.60%

Table 4.3: Accumulated Improvement of Line Coverage

	1	2	3	4	5	6	7
Pre	40.10%	40.10%	40.20%	40.20%	40.20%	40.40%	40.40%
Post	42.20%	42.20%	43.20%	43.30%	43.30%	43.40%	43.80%
Improve	3.00%	3.10%	3.20%	3.20%	3.20%	3.30%	3.60%

Table 4.4: Accumulated Improvement of Function Coverage

Tables 4.3 and 4.4 show the datasheet in Line and Function Coverage respectively. The first row of each table represents the number of samples fed into ESBMC and GotoFuzz in each round. The pattern that can be observed is that as the number of samples used increases, the improvement from GotoFuzz is greater, which confirms our suspicion.

Effect of file size

Next, we focus on the impact of the file size of the samples. To do this we generate samples of different sizes. Since Csmith does not provide options to set the file size, we need to use the option "--max-block-size n " indirectly. This option limits the number of non-return statements in a block to n . By experience, a smaller n reflects a smaller file size. In our benchmarks, this number has been set to 1 to 5 respectively. The conclusions can be listed as follows:

1. the small file takes fewer times to reach the upper limit of the coverage improvement. This conclusion can be drawn from Figure 4.5, where the sample with a maximal block size of 1 and size of 29KB. The X-axis represents the number of rounds executed. To reduce the effect of randomness, we take each 250 rounds as a unit. The Y-axis represents the percentage value of code coverage. It can be seen that all code coverage ramps up until 500 rounds are reached, but not after that. This implies that there is an upper limit to the amount of improvement that GotoFuzz can deliver.

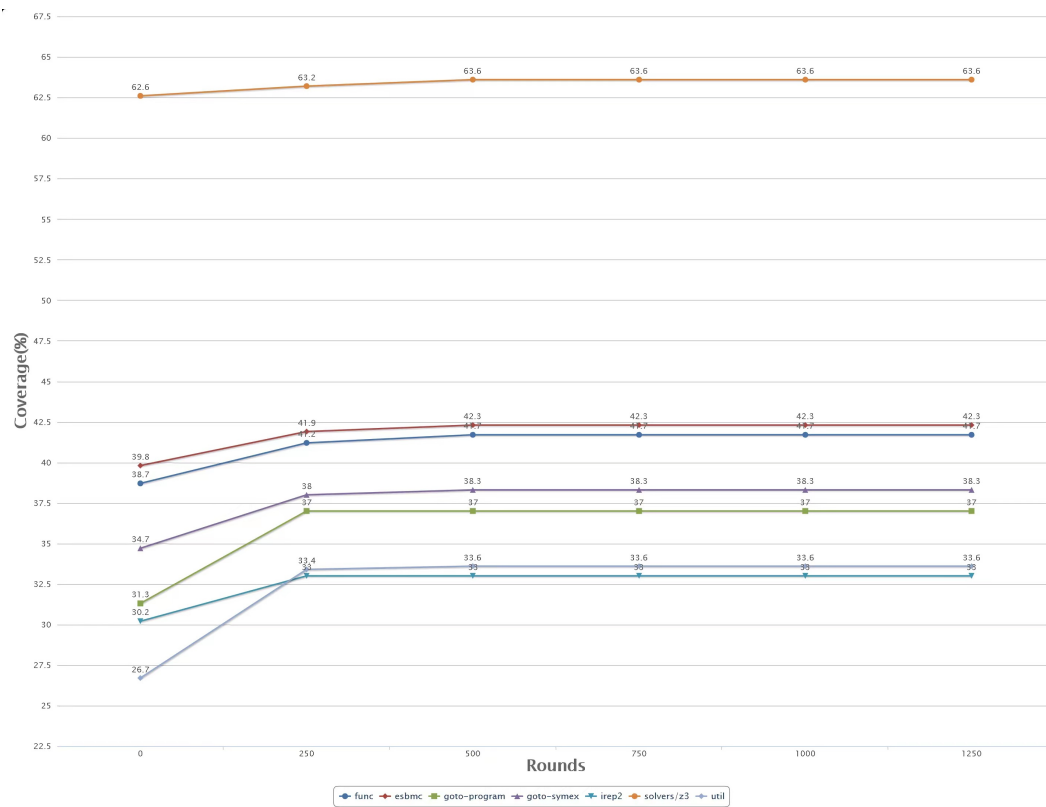


Figure 4.5: Example of Reaching Maximal Line/Function Coverage Improvement


```

VERIFICATION FAILED
#121 NEW cov: 19471 ft: 37063 corp: 13/175Kb lim: 15533 exec/s: 1 rss: 1310Mb L: 15374/15533 MS: 1 ShuffleBytes-
Target: 64-bit little-endian x86_64-unknown-linux with esbmc libc
Starting Bounded Model Checking
Unwinding loop 270 iteration 1 file random6.c line 4504 function main
Unwinding loop 0 iteration 1 file random6.c line 4901 function main
Unwinding loop 0 iteration 2 file random6.c line 4290 function main
Unwinding loop 225 iteration 1 file random6.c line 3659 function main
Unwinding loop 188 iteration 1 file random6.c line 2853 function main
Unwinding loop 291 iteration 1 file random6.c line 4861 function main
Unwinding loop 285 iteration 1 file random6.c line 4803 function main
Unwinding loop 279 iteration 1 file random6.c line 4672 function main
Unwinding loop 0 iteration 1 file random6.c line 5159 function main
Symex completed in: 1.729s (3985 assignments)
Slicing time: 0.013s (removed 3413 assignments)
Generated 2268 VCC(s), 259 remaining after simplification (572 assignments)
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.290s
Solving with solver Z3 v4.8.12
==4356== ERROR: libFuzzer: out-of-memory (used: 2369Mb; limit: 2048Mb)
To change the out-of-memory limit use -rss_limit_mb=<N>

MS: 1 CrossOver-; base unit: d53d52700fbcddd9b329e838fab8a26e7f186fd3
artifact_prefix='./'; Test unit written to ./oom-248fe64977c88114b7afbb1d09a38bd4a613529e
SUMMARY: libFuzzer: out-of-memory

```

Figure 4.7: Example of Running Out of Memory

	line	function	esbmc	goto- program	goto- symex	irep2	solvers/ z3	util
unwind 1	41.90%	42.20%	37.00%	38.50%	33.00%	65.10%	33.80%	47.20%
unwind 5	42.10%	42.20%	37.00%	38.50%	33.20%	65.10%	33.80%	47.80%
Improve	0.20%	0.00%	0.00%	0.00%	0.20%	0.00%	0.00%	0.60%

Table 4.5: Improvement with Extra Unwinding Times

4.0.3 Summary

The GotoFuzz succeeded in discovering hidden vulnerabilities in ESBMC and provided conjectures and explanations for the possible causes. At the same time, we demonstrated that GotoFuzz can improve the code coverage of ESBMC. We also found the relevant exchange rate for code coverage. The code coverage increase is large when using single test samples. This increase will accumulate when fuzzing with multiple files. The conclusion from experiments comparing file sizes is that the size of test cases needs to be constrained. Lastly, different command line arguments can also have an impact on the performance of GotoFuzz.

Chapter 5

Conclusion and Further Work

In this chapter, we review the results of our project to testify whether the goals were achieved and reflect on the project, assessing what went well and what could be improved. The chapter concludes with a discussion of limitations and recommendations for future work.

5.1 Deliverables and Key Achievements

Our project's deliverables can be outlined from three aspects. First, an intelligent hybrid fuzzer GotoFuzz has been designed and developed. This fuzzer allows ESBMC's middleware to be directly tested for the first time. In practice, two approaches were proposed and implemented, leading to 3 new program files created and 9 files altered in ESBMC. The total amount of coding is about 274 lines of code in C++ and 23 lines in Shell code. This code will be committed to the ESBMC GitHub repository in the future.

Another contribution is a summary of the Goto grammar. This can be valuable as the lack of Goto syntax made it difficult to understand the internal logic of the ESBMC middleware. In addition, bounded model checkers, such as CBMC and JBMC, share a similar structure to ESBMC, as they are based on the same platform called Cprover. Hence, this summary is also useful for understanding other Cprover-based BMCs. This document will also be committed to the ESBMC GitHub repository in the future.

The key achievement of this project is a bug-fixing commitment made to ESBMC, as shown in Figure 5.1. As mentioned in section 3.2, the output interface for the Goto program is essential for GotoFuzz, yet this ESBMC API has been broken for a long time. So I spent a lot of time finding the source of the bug and writing code to fix it.



Figure 5.1: The Commitment to ESBMC

These changes were eventually merged into ESBMC and I was able to read/write Goto binaries properly.

5.2 Reflection

What went well?

First, applying the dynamic verification technique Fuzzing to the static verifier BMC was a challenge. Different test methods and tools were understood in detail and successfully applied to the Fuzzer's construction. Secondly, We have succeeded in analysing and summarising the structure of the black box that is ESBMC, leading to the discovery of possible vulnerabilities. Last but not least. The results of the evaluation show the value of our GotoFuzz.

What could be improved?

This project also contains development tasks that consume more time than the original estimate. This is due to several reasons. Firstly, to the best of our knowledge, there is no current work or research on fuzzing verifiers. Therefore, finding relevant and valuable literature can be a challenge. Due to the neglect of the importance of finding and reading papers in related fields, a big amount of mistakes have been made. For example, the architecture design of GotoFuzz got constant refactoring whenever a new idea was found after accidentally finding out a relative paper. The lesson is to seek out and read as much relevant literature and methodology as possible before designing. Secondly, difficulties have been encountered during the coding period. Due to the lack of expectations of the outcome, agile development methods like test-driven mode

cannot be used when developing on ESBMC. Another mistake is neglecting to pay attention to the formatting of the code. Since ESBMC is a well-known open-source project, submissions are scrutinised by reviewers. The lack of attention to the code style resulted in my commits being rejected several times, so a lot of time was spent on code changes.

The result of this timeout is troubling. For example, according to the previous plan, the coding job should be done by the end of July, yet in reality, this work continues until August and is still ongoing during the writing of the dissertation. This is due to poor programming skills and unreasonably time planning.

5.3 Limitations and Future Work

Some future work could be done to apply GotoFuzz to other areas. Firstly, although the vulnerabilities in Section 4.1 were identified, we were neither able to identify the root cause nor fix it. This is due to the fact that GotoFuzz is still a grey-box fuzzer and does not fully understand the internal structure of its testing subject. Hence, One possible work is to change GotoFuzz to a white box fuzzer. In addition, since GotoFuzz only focuses on structural mutation, the randomisation of variables relies entirely on Csmith; however, Csmith does not always find interesting values, especially zero values. Therefore, GotoFuzz could be enhanced by adding control over the variables' value, for example by manually setting some boundary values. Lastly, some future work could be done by applying this fuzzer to a broader area. Given that the core of GotoFuzz is a modification of the Control flow chart, a possible solution is to draw up an abstraction of the target program into a control flow chart form such that it can be fuzzed by GotoFuzz.

Bibliography

- [1] Mikhail R Gadelha, Rafael Menezes, Felipe R Monteiro, Lucas C Cordeiro, and Denis Nicole. Esbmc: scalable and precise test generation based on the floating-point theory:(competition contribution). *Fundamental Approaches to Software Engineering*, 12076:525, 2020.
- [2] Esbmc. <http://www.esbmc.org/>, 2020.
- [3] Xaver Fink, Philipp Berger, and Joost-Pieter Katoen. Configurable benchmarks for c model checkers. In *Lecture Notes in Computer Science*, pages 338–354. Springer International Publishing, 2022.
- [4] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.
- [5] Cedric Richter and Heike Wehrheim. Pesco: Predicting sequential combinations of verifiers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–233. Springer, 2019.
- [6] Marek Chalupa, Jakub Novák, and Jan Strejček. Symbiotic 8: Parallel and targeted test generation. *Fundamental Approaches to Software Engineering*, pages 368–372, 2021.
- [7] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. Ultimate automizer and the search for perfect interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451. Springer, 2018.

- [8] Viktor Malík, Peter Schrammel, and Tomáš Vojnar. 2ls: heap analysis and memory safety. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 368–372. Springer, 2020.
- [9] Issues · esbmc/esbmc. <https://github.com/esbmc/esbmc/issues>, 2022.
- [10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *ACM SIGPLAN Notices*, 46(6):283–294, jun 2011.
- [11] libfuzzer. <https://llvm.org/docs/LibFuzzer.html>, 2020.
- [12] Cprover. <http://cprover.diffblue.com/>, 2022.
- [13] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.
- [14] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. Jbmc: A bounded model checking tool for verifying java bytecode. In *International Conference on Computer Aided Verification*, pages 183–190. Springer, 2018.
- [15] William Bonnaventure, Ahmed Khanfir, Alexandre Bartel, Mike Papadakis, and Yves Le Traon. Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 586–597. IEEE, 2021.
- [16] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.
- [17] Z3. <https://github.com/Z3Prover/z3>, 2022.
- [18] Yices. <https://yices.csl.sri.com/>, 2022.
- [19] Mathsats. <https://mathsat.fbk.eu/>, 2022.
- [20] Stefan Kupferschmid and Bernd Becker. Craig interpolation in the presence of non-linear constraints. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 240–255. Springer, 2011.

- [21] Cvc4. <https://cvc4.github.io/>, 2022.
- [22] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.
- [23] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.
- [24] Mate Soos, Armin Biere, M Heule, M Jarvisalo, and M Suda. Cryptominisat 5.6 with yalsat at the sat race 2019. *Proc. of SAT Race*, pages 14–15, 2019.
- [25] Peled Doron, P Pelliccione, and Paola Spoletini. Model checking. 2009.
- [26] Siert Wieringa. On incremental satisfiability and bounded model checking. *Design and implementation of formal tools and systems*, pages 46–54, 2011.
- [27] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.
- [28] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011.
- [29] Huning Dai, Christian Murphy, and Gail Kaiser. Configuration fuzzing for software vulnerability detection. In *2010 International Conference on Availability, Reliability and Security*, pages 525–530. IEEE, 2010.
- [30] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
- [31] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [32] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM workshop on artificial intelligence and security*, pages 37–47, 2018.

- [33] Structure-aware fuzzing with libfuzzer. <https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md>, 2022.
- [34] Michal Zalewski. American fuzzy lop, 2017.
- [35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [36] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.
- [37] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 711–722. IEEE, 2021.
- [38] Hoang Lam Nguyen and Lars Grunske. Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing. *arXiv preprint arXiv:2202.13114*, 2022.
- [39] Lcov. <https://wiki.documentfoundation.org/Development/Lcov>, 2022.
- [40] genhtml. <https://manpages.ubuntu.com/manpages/bionic/man1/genhtml.1.html>, 2022.