Chapter 1

Introduction

The complexity of manually and dynamic testing increases with the growth of the project. This results in the increasing usage of static analysis tools, such as bounded model checker, which do not require compilation and act automatically [1]. As an SMT-based model checker for program static analysis and verification, ESBMC has been successfully used in a variety of environments due to its adaptability to multi-language, multi-platform and high efficiency[2]. However, these features have led to the difficulty of fuzzing this software verifier. Firstly, the construction of a generation-based fuzzer for ESBMC is a challenge. The adaptability for multiple source programming languages means that the fuzzer needs to consider certain source language's grammar, in the worst case, constructing different inputs for corresponding front-ends. Secondly, the construction of a coverage-guided mutation-based fuzzer is a challenge, as directly generated random input cannot be "understood" by ESBMC and will be excluded beyond validating. These deficiencies will result in fuzzy tests that do not detect errors hidden in the deep execution path of the program and insufficient code coverage. Hence, we propose a fuzzer construction on intermediate representation, which refers to Goto programs. Instead of converting from the source file, this generic fuzzer would generate syntactically correct Goto programs directly. The introduction of mutation will further dynamically modify the properties in the Goto programs by tracking and updating the corpus. These Goto programs will finally be used as input to fuzz test the ESBMC.

The achievement of this goal can be divided into the following steps: (1) Summarise the syntax grammar for constructing Goto intermediate representation, including the symbol table and Goto instructions, i.e. each statement in the Goto program, as well as the Goto programs itself. (2) Investigate a mutation approach for the generated

Goto programs, which will be achieved by introducing libFuzzer[3]. (3) Design and implement the program architecture of the Goto-fuzzer. (4) Debugging and testing, including error detection and performance testing. (5) Fuzz the ESBMC by Goto-fuzzer. Collect the error logs and coverage information.(6) Analyse the data and draw conclusions.

Chapter 2

Background

2.1 ESBMC

ESBMC is a licensed open-source 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 [2]. The basic workflow of this 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. 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 [4].

From an architectural point of view, ESBMC can be divided into three components: front-end, middleware and back-end. A C++ Oriented ESBMC structure can been shown by Figure 2.1.

2.1.1 Front-end

Front-end is an important piece of technology that should facilitate the transition between the program under verification and a format the tool can work upon. Despite that several front-ends have been constructed corresponding to different programming languages in ESBMC, the final output will always be a symbol table, an auxiliary data structure storing the meaning and range of variable names. A general front-end framework includes a pre-processor, scanner, parser and type checker. **Pre-processor**

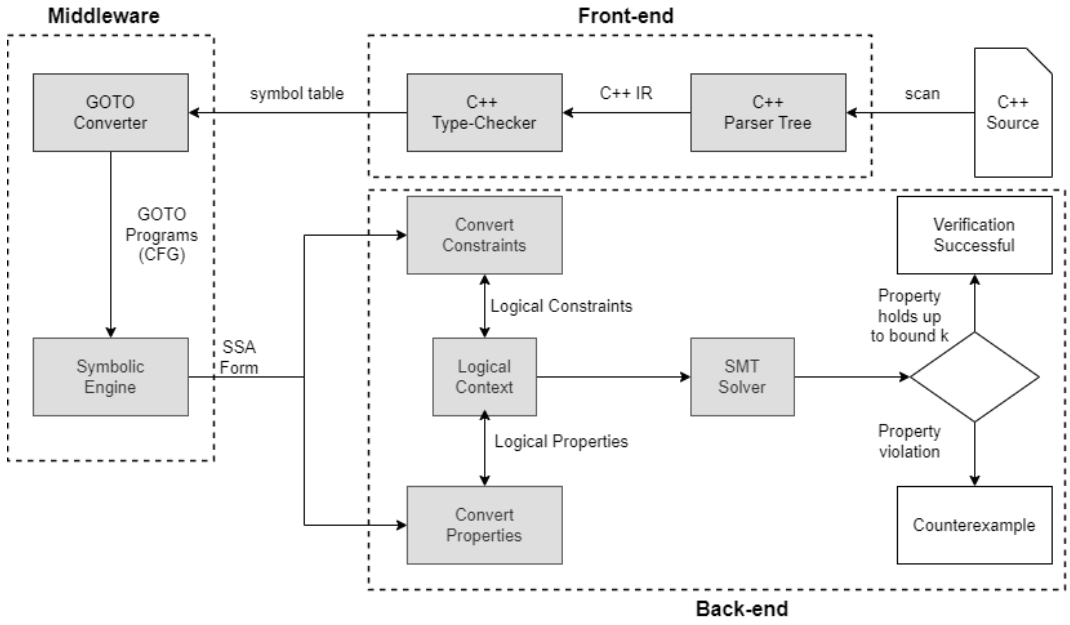

Figure 2.1: C++ oriented ESBMC's architecture overview. White rectangles represent input and output; grey rectangles represent the steps of the verification

handles special operations that will be performed according to the preprocessor instructions, such as replacement or expansion macros [5]. A lexical analysis of the **scanner** and a syntax score of the **parser** resulted in the Abstract Syntax Tree (AST) [6]. To simplify the analysis, ASTs have been converted into a simpler form, called an intermediate representation (IR), by **type checker** in which a symbol table is generated simultaneously. At this point, the source code as input is converted into symbol tables [7].

2.1.2 Middleware

In ESBMC, the generated symbol table enables the middleware to perform further actions, including the conversion from the original program into the equivalent Goto program, an intermediate representation generated from the symbol table, the generation of the single static assignment (SSA) form and the symbolical execution of the program.

Goto Converter firstly AST into its Control flow chart form, which refers to Goto programs. These programs are language independent and similar to the intermediate languages of many compilers. Each 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. An **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. 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 [7].

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 [2]. 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 [7].

2.1.3 Back-end

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 [2].

2.2 Fuzzing

Fuzz testing (fuzzing) is a software testing technique. The core idea is to feed random data generated automatically into a program by a fuzzer. Traditionally, fuzzers can be classified into two categories based on how they are input, including generation-based fuzzers and coverage-guided mutation-based fuzzers.

2.2.1 Generation-Based Fuzzer

A **generation-based** fuzzer construct inputs according to some provided format specification, especially the grammar of a language [8]. One of the good examples of such fuzzers is **Csmith** [9] which is already applied in ESBMC. Csmith works as a generator for C programs. These programs will be fed to the target software, in most cases

compilers, run the executables and compare the outputs. In practice, it is used as a stress testing tool to check the stability of ESBMC.

2.2.2 Mutation-Based Fuzzer

A **mutation-based** fuzzer generates inputs by randomly altering analyst-supplied or created seeds. These programmes do not require syntactic definitions and are not limited to a single input type [8]. These coverage-guided fuzzers use coverage information as feedback to alter existing inputs into new ones, attempting to maximise the amount of code covered by the overall input corpus.

LibFuzzer is a powerful mutation-based Fuzzer chosen as our research object. LibFuzzer is linked to the library under test and feeds fuzzed inputs to it through a defined fuzzing target function. The fuzzer records which branches of the code were accessed and creates mutations on the corpus of input data to optimise code coverage. LLVM's SanitizerCoverage instrumentation provides code coverage information for libFuzzer[3].

2.2.3 Structure-Aware Fuzzer

Both fuzzer types mentioned above are inherently flawed. 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[10]. On the other hand, the presence of checksums in the input format or the complexity of the input format itself can easily trip up a mutation-based fuzzer, as virtually all generated inputs can be invalid for the target data structure.

We, therefore, consider a **structure-aware** fuzzer proposed by Google [11]. This fuzzer is a combination of the two fuzzing approaches described above: as a generation-based fuzzer in essence, the generator will now accept random value generated by the mutation-based fuzzer as seed. This mutator, in this case, libFuzzer, will mutate and optimise the random bytes by tracking and feedback.

Chapter 3

Research Methodology

This section is originally divided into three subsections, with Section 3.1 discussing the introduction and application of structure-aware fuzzer to ESBMC, after which two approaches are proposed. We will first introduce the first approach in Section 3.2, which uses Csmith as a generator, and then mutates the original program via libFuzzer. We will discuss this method and explain why the second approach was chosen. In Section 3.2, we will t describe the second approach that will be used in development.

3.1 Introduce Structure-Aware fuzzer to ESBMC

First, we need to decide the output form of the fuzzer, in this case, the generator. In theory, the potential object can be any temporary variable passed in the ESBMC workflow. We have chosen the Goto program as the output for several reasons: (1) the Goto program is language-independent, reflected in no dependency on the front-end (2) 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. The main goal, therefore, is to create semantically arbitrary, in most cases incorrect, but syntactically correct goto programs.

Next, we explore how to achieve randomness via mutation. We use structure-aware mutations to 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.

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

Figure 3.1: A simple Goto program

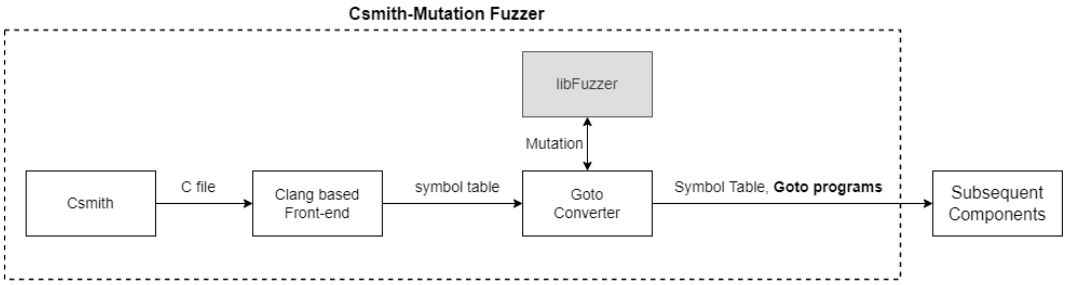

Figure 3.2: Csmith-mutation fuzzer architecture. White rectangles represent components reused from ESBMC; grey rectangles represent new created components

We also use structure-preserving mutations to create distinct mutations of the same input structure to investigate alternative execution trails. This is accomplished by making the values of context-sensitive variables unequal [10].

As shown in Figure 3.1, we take a simple goto program as an example, 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.

3.2 First Approach: Csmith-Mutation Fuzzer

We propose the first implementation approach, whose architecture is shown in Figure 3.2. First, Csmith will create random C files, which will be passed through a front-end

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
        size_t Size, string filepath)
{
    // Convert C codes to Goto programs
    Goto_program=c2goto(filepath);
    // randomise Goto programs by libFuzzer
    Goto_program=mutateor(Goto_program, Data, size)
    // Apply the Goto program to the subsequent process
    do_bmc(Goto_program);
    return 0;
}
```

Figure 3.3: Programming Logic of Csmith-Mutation Fuzzer

and a converter to form Goto programs. Here we introduce a mutator which semantically randomizing Goto programs by libFuzzer. Randonmized goto programs will finally be deliverred to the subsequent process. A simplified version of the programming logic is shown in Figure 3.3.

The advantage of the Csmith-Mutation Fuzzer is that it can reuse the existing generator Csmith and only need to consider constructing a mutator; however, it also has several drawbacks: (1) We cannot guarantee that all language features supported within ESBMC are covered, as Csmith's C standard support is c99 while that of ESBMC is c11. This will result in some of the statements not being covered during fuzz testing; (2) The fuzzer cannot be refined to test for specific statements (3) The approach of Csmith fuzzing is still front-end related as C file can only be read in via C-based front-end (in this case clang). We would like to have a more generic, front-end-independent fuzzer.

3.3 Second Approach: Goto-Fuzzer

Here we propose a second approaches, the architecture of which is shown in Figure 3.4. The core difference between two methods is that the construction output of the generator has changed from a source file to an intermediate file, which refers to the symbol. The choice of the symbol table over the goto program as the output of the generator is more a matter of development convenience, as no direct functions to construct a Goto program are provided in ESBMC. In contrast, the construction of a code

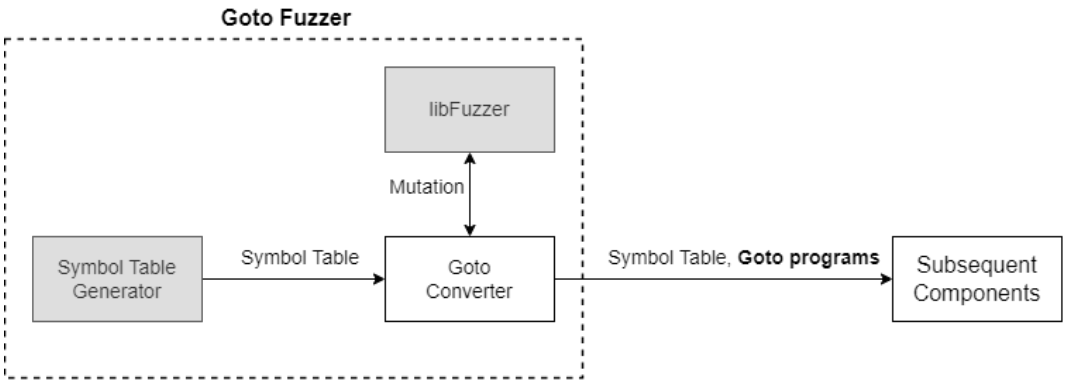

Figure 3.4: Goto-fuzzer architecture. White rectangles represent components reused from ESBMC; grey rectangles represent new created components

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
        size_t Size, string filepath)
{
    // use libFuzzer as seed generator
    Goto_program = generator(Data, Size);
    do_bmc(Goto_program);
    return 0;
}
```

Figure 3.5: Programming Logic of Goto-Fuzzer

statement is considerably simpler. In addition, contextual properties are initialised automatically during the conversion from symbol table to goto program, which will be a tough task if building Goto program directly. To ensure that the symbol table based generator works indentically as the direct goto program generator, We will prove that the types of generated statement will cover that of goto program instruction during the Goto converson. A simplified version of the programming implementation is shown in Figure 3.5.

Chapter 4

Ethics and professional considerations

This project does not require the participation of any volunteers, nor will user data be collected in any form, the involvement of humans and data can therefore be excluded. Considering that ESBMC is under the Apache License 2.0, our modifications and development should follow the same open-source rules. In addition, given that ESBMC will be used in commercial and other production environments, the fuzzer needs to be tested before it is released, and the project documentation needs to be supplemented with relevant notes.

Chapter 5

Risk consideration

5.1 Design Stage

To design an efficient structure-awared fuzzer, which is the combination of generation and mutation fuzzer, will be the first challenge. This will have a direct impact on the final outcome. For example, test cases generated from a mutation algorithm that lacks logic may have the following problems: (1) the test cases are not useful, which is often reflected in the simplicity of the test cases (2) the test cases are virtually identical to others from the aspect of code coverage, which will result in test objects being limited to only some branches.

5.2 Implement Stage

The difficulty of the implement of Goto-fuzzer can be reflected in several aspects. Coupling and component independence needed to be firstly considered at the top-level design stage in order to avoid fuzzer having side effects on other functions in ESBMC. In addition, although the data structures and libraries for the intermediate representation were already provided, there was still a lot of coding to be done. Furthermore, the complex internal cross-referencing within ESBMC and the use of ”magic” functions put the developer's C++ reading skills to the test.

5.3 Debug and Test Stage

The debuging and testing of Goto-fuzzer will be challenge, as the interdependencies of the libraries make debugging Goto-fuzzer as time consuming as debugging the entire ESBMC, requiring recompilation and relinking of the entire ESBMC project.

Chapter 6

Project evaluation

The project can be evaluated from the following aspects: First, the primary purpose of a fuzzer is to use it to find hidden bugs in the program, thus the ability of error detection will be tested. 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 [12]. What's more, the improvement of code coverage will be measured. Tools for statistical code coverage are provided in ESBMC and can be output visually. In the worst-case scenario, where our Goto fuzzer fails to find any bug, then the code coverage will be the key evaluation of the effectiveness. Besides, despite the libFuzzer as the mutator being a pre-requisite for the project, we could also propose other mutation-based fuzzer to compare and evaluate their relative performance. This optional task will be done if possible.

Chapter 7

Planning

As the stage division in the section above, the general arrangement of this project will be divided into four parts, as shown in the Gantt Chart in Figure 7.1. First, the design stage will take 2 weeks and will be devoted mainly to summarizing the syntax grammar for the generator and to working out how to apply mutation to this fuzzer. Second, the implementation stage will last a month, and this stage will complete all. As an optional arrangement, we consider building an additional interface to output the goto program for debugging purposes. This task will be cancelled due to insufficient time. Third, the debug and testing stage will take three weeks, during which we focus on collecting data to verify the effectiveness of our Goto fuzzer. Finally, the writing stage will begin with the design stage simultaneously and will continue until August 8th.

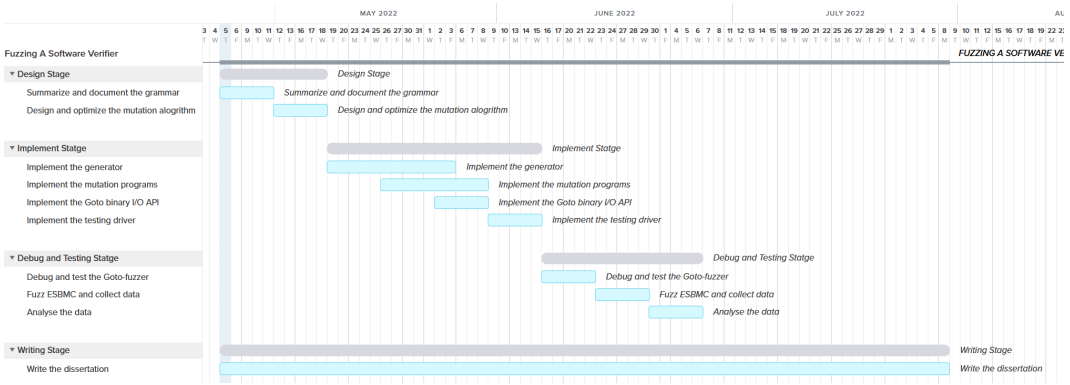

Figure 7.1: Gantt Chart

Bibliography

[1] Robert C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 1st edition, 2008.

[2] Esbmc. http://www.ESBMC.org/.

[3] libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.htm.

[4] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38, 07 2009.

[5] Paul J. Deitel and Harvey M. Deitel. C. *How to Program; with an Introduction to C++*. Pearsonl, 2016.

[6] Alfred V. Aho and Alfred V. Aho. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2009.

[7] cprover. http://cprover.diffblue.com/.

[8] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, may 2018.

[9] Csmith. https://embed.cs.utah.edu/csmith/.

[10] Hoang Lam Nguyen and Lars Grunske. Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing. 2022.

[11] Structure-aware fuzzing with libfuzzer. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[12] 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.