A Security Analyser for Finding Vulnerabilities in C Programs

The Efficient SMT-Based Context-Bounded Model Checker Tool

Author:
Alecsandru-Catalin Balan

Supervisor:
Lucas Cordeiro

Final year project report submitted for the degree of BSc. (Hons) Computer Science

University of Manchester
School of Computer Science
May 2020

Abstract

As modern embed software become more and more complex, the need for automatic verification is now greater than ever. Numerous programs contain many unidentified vulnerabilities, which could be exploited and could lead to faulty behaviors, crashes, information leakage and even data theft. In order to avoid such scenarios, automatic verification can be used for detecting hidden bugs and vulnerabilities. This project evaluates one tool used for automatic verification, the Efficient SMT-based Bounded Model Checker, an open source software which consists of different security check methods like: pointer dereference, double free, memory leaks and many others. This report presents the tool's architecture, components, as well as the background theory it is based on. The effectiveness and capabilities are demonstrated by rigorous testing and experiments show the tool's performance compared to other existing tools. The last section presents the achievements and reflections of this project, as well as possible future work, with this project acting as a base.

Acknowledgements

My greatest thanks are expressed towards my coordinator, Lucas Cordeiro. His support and constructive feedback helped me to overcome all the challenges encountered during the completion of this project.

In addition, I want to acknowledge my parents and my sister for providing the wise counsel and the moral support needed during these challenging years and special thanks to my friends for the given recommendations and advice this year.

Contents

1 Introduction **7**
- 1.1 Motivation and Related Work . . . 7
- 1.2 Aim and Objectives . . . 8
- 1.3 Report Structure . . . 8
- 1.4 Impact of COVID-19 . . . 9

2 Context **9**
- 2.1 Boolean Satisfiability and Satisfiability Modulo Theories . . . 10
  - 2.1.1 Boolean Satisfiability (SAT) . . . 10
  - 2.1.2 Satisfiability Modulo Theories (SMT) . . . 10
- 2.2 Bounded Model Checker . . . 12
- 2.3 Bounded Model Checker based on SMT . . . 13
- 2.4 C programs vulnerabilities . . . 13
- 2.5 Related Work . . . 14

3 Design **14**
- 3.1 Architecture . . . 14
- 3.2 Handling Input . . . 16
  - 3.2.1 Scalar Data Types . . . 16
  - 3.2.2 Fixed Point Arithmetic . . . 16
  - 3.2.3 Arrays . . . 17
  - 3.2.4 Structures and Unions . . . 17
  - 3.2.5 Pointers . . . 17
  - 3.2.6 Dynamic Memory Allocation . . . 18
  - 3.2.7 Floating Point . . . 19
- 3.3 Illustrative examples . . . 20

4 Testing and Results **23**
- 4.1 Regression testing . . . 23
- 4.2 Concurrency testing . . . 23
- 4.3 Evaluation . . . 24

5 Reflection and Conclusion **26**
- 5.1 Reflection . . . 26
- 5.2 Conclusion and Future Work . . . 27

List of Figures

1 ESBMC tool's architecture. 15
2 C program converted into GOTO program. 20

Listings

1 Pointer to array example 20
2 SSA form of the C program 21
3 ESBMC output after running the fixed code from Listing 1 . . . 22
4 C program with global variable and faulty behaviour 24

List of Tables

1 Data representing conversions and run time of ESBMC 25

1 Introduction

In this section, we introduce the main concepts of Bounded Model Checking (BMC), Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) as well as the motivation behind choosing this project. An overview as well as the aim and objectives for this project are also provided. This chapter also highlights the impact of COVID-19 and the changes that were necessary for completing the project.

1.1 Motivation and Related Work

Over the past 3 decades, there has been a great interest for automatic verification techniques for software systems as well as hardware systems where there is a finite number of states. One of the earliest techniques for solving Boolean functions is using a binary decision diagram (BDD) where, given a truth table and its decision tree, a BDD can be created from the decision tree by maximally reducing it using the reduction rules. The main advantage of BDD is that it is unique for every given function and variable order. Because systems and programs are very complex, their system state space grows exponentially, thus creating the 'state explosion problem' [1]. One solution for solving this problem is Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT), method introduced as complementary to BDD. BMC will check for negations of given properties at a given depth of the Finite State Machine (FSM). Given a transition system T, a property ϕ and a depth (bound) k, BMC will unfold the system k times and encode it into a verification condition (VC) ψ which will be satisfiable if and only if a counterexample of ϕ is found at depth k or less. Due to the growth of software complexity every year, Satisfiability Modulo Theories (SMT) solvers are used as back-ends for solving the VC generated by the BMC. Older work on SMT-based BMC [2] for software did not encode constructs such as pointers, unions and bit-level operation, thus minimizing its usefulness for verifying programs written in ANSI-C, but nowadays such tools fully supports verification of ANSI-C programs, as well as C/C++ applications. Moreover, this project particularly focuses on embed software written in C language, which utilise dynamic memory allocation, one of the most common cause of unwanted vulnerabilities such as: buffer overflow and string manipulation, with the consequence of information leakage and memory overwriting. Nowadays, software becomes more and more complex, which leads to a greater risk of creating vulnerabilities, thus the need of automatic verification for finding possible threats is higher.

1.2 Aim and Objectives

This paper makes use of previous work on the Efficient SMT-Based Bounded Model Checker (ESBMC) [3][4], with the aim of demonstrating the efficiency and effectiveness of formal verification techniques to prove the absence of memory safety and undefined behavior issues in C programs. With various approaches which could be pursued for completing the project, a list of objectives must be considered, so the general goal is correlated with the following specific ones:

- Describing the background problem and the context. This is done by rigorous research over the existing C vulnerabilities, in order to gain a thorough understanding of common vulnerabilities in this environment;
- Providing enough level of detail regarding the background theories used and the current capabilities of the ESBMC tool. Again, this is achieved as a result of laborious research and documentation;
- Extending ESBMC in order to further enhance its ability of proving correctness of the C programs. In order to achieve this, suitable extension must be identified, as well as documentation over the development of the ESBMC must be made, as ESBMC is an open source project;
- Testing and evaluating the ESBMC after the implementations and then comparing with other existing SMT-based BMC tools. This is achieved by setting up the environment and dependencies of tools, as well as writing suitable tests.

1.3 Report Structure

The chapters of this report cover:

- **Chapter 2: Context** discuss the background theory needed to understand this project and to evaluate this report. Includes details of Boolean Satisfiability and Satisfiability Modulo Theories used by the tool and it also provides the theory behind bonded model checking. Furthermore, an outline of the related work is given at the end of the chapter.
- **Chapter 3: Design** explains the architecture of the ESBMC tool and describes the encodings used to convert the constraints and properties of C programs into the background theories discussed in Chapter 2. It also provides an illustrative examples of a C programs being converted as it goes through the ESBMC pipeline, using relevant figures and listings.
- **Chapter 4: Testing and Results** provides the methods to measure both the quality and performance of the ESBMC when verifying C programs for faulty behaviors and memory safety using relevant tables and data collected from using the tool.

- **Chapter 5: Reflexion and Conclusion** highlights the achievements and challenges of this project, contains the student reflection for the project and ends with a discussion for further work.

1.4 Impact of COVID-19

Unfortunately, the rapid evolution of COVID-19, from an epidemic to a pandemic in just a few months, has greatly influenced the last part of the project and changes were necessary in order to ensure the project's completion. One of the greatest impacts the pandemic had over this project was the environment change caused by leaving Manchester. The home desktop from Manchester was properly set up to conduct rigorous experimentation and testing of the ESBMC as it had all the dependencies installed. Because leaving was necessary, testing and experimenting was conducted over an older version of the environment, which did not allow for comparative evaluation with other tools, as installing the required dependencies and tools would have taken a considerably amount of time. Thus, the approach for presenting relevant comparison data was changed and comparative results from other papers were used in order to prove the efficiency of the tool. Other changes were regarding the project management aspect, because COVID-19 had quite the impact on the project planning, thus objectives and aims dates needed to be changed accordingly with the deadlines' change and so the project planning is not very accurate. Despite the unfortunate changes caused by the pandemic, the student considers the project as being completed, details covering the conclusion and reflection being covered in the last chapter.

2 Context

C Bounded Model Checker (CBMC) [5] is a BMC used to check C/C++ programs for memory safety, exceptions and user-specified assertions. While it comes with a built-in solver based on MiniSat to solve bit-vector formulas, it also supports external SMT solvers. Building on the front-end of CBMC, ESBMC generates the VCs for a program, converts them based on different background theories and uses external SMT libraries to decide satisfiability of first-order formulae. This section describes the above-mentioned background theories and how BMC makes use of them in order to find vulnerabilities in a C environment.

2.1 Boolean Satisfiability and Satisfiability Modulo Theories

2.1.1 Boolean Satisfiability (SAT)

Given a Boolean formula, we can say that the formula is satisfiable if and only if, for all the interpretations of formula's variables, there exists at least one such that the function evaluates to TRUE. On the other hand, if for all possibilities of variables interpretation, the formula is FALSE, then we say that it is unsatisfiable.

2.1.2 Satisfiability Modulo Theories (SMT)

By replacing Boolean variables with predicates from background theories, SMT generalize the SAT problem. Given a theory τ and a quantifier-free formula ϕ, we say that ϕ is τ-satisfiable if and only if there exists a structure that satisfies both the formula and the sentences of τ, or equivalently, if $\tau \cup \{\phi\}$ is satisfiable [6]. Let us assume $\Gamma \cup \{\phi\}$, a set of formulae in the same language as τ. We say that ϕ is a τ-consequence of Γ, or $\Gamma \models \tau\phi$, if and only if every model of $\tau \cup \Gamma$ is a model of ϕ. Showing that $\Gamma \models \tau\phi$ holds it is clearly a problem that can be reduced to the problem of checking the τ-satisfiability of $\Gamma \cup \{\neg\phi\}$ [3]. The SMT-LIB [7] aims to create a common standard for the specification of background theories, but many SMT solvers (e.g. Boolector, CVC3, and Z3) provide additional functions apart from those specified in the SMT-LIB. Bellowed are described fragments of SMT solvers used by ESBMC [3] for theory of linear, non-linear, and bit-vector arithmetic:

$F ::= F\, con\, F | \neg F | A;$

$con ::= \wedge | \vee | \oplus | \rightarrow | \Rightarrow;$

$A ::= T\, rel\, T | Id | true | false\ ;$

$rel ::= < | \leq | \geq | > | = | \neq\ ;$

$T ::=\ T\, op\, T | \sim T | ite(F, T, T) | Const | Id |$

$\quad Extract(T, i, j) | SignExt(T, k) | ZeroExt(T, k);$

$op ::= + | - | * | / | rem | << | >> | \& |\ |\ | \oplus | @;$

with the following notations:

- F: a Boolean-valued expression with atoms A;
- T: terms built over integers, reals, and bit-vectors;
- con: logical connectives such as conjunction ($\wedge$), disjunction ($\vee$), exclusive-or ($\oplus$), implication ($\rightarrow$) and equivalence ($\Rightarrow$);
- rel: relational operators $<, \leq, \geq, >, =, \neq$;

- bit-level operators:and (&), or (—), exclusive-or ($\oplus$), complement ($\sim$), right shift (>>), and left shift (<<), concatenation;
- linear and nonlinear arithmetic operators: +, -, $*$, /, remainder (rem);
- Extract (T, i, j): bit vector extraction from bits i to j, resulting in a new bit vector of size i - j + 1;
- SignExt (T, k): bit vector extension from size v to the signed equivalent bit-vector of size v + k;
- ZeroExt (T, k): bit vector extension with zeros from size v to the unsigned equivalent bit-vector of size v + k;
- ite(f, t1, t2): conditional expression with input f (Boolean formula) and output t1 or t2 (arguments).

In order to cope with the array theories, SMT solvers use McCarthy axioms [8]. Two functions are used, the select function and the store function. Select(a, i) simply returns the value of a at the position i, while store(a, i, v) returns the same array as a where index i holds the value v. Formally, these functions are described by the following axioms [9]:

- $i = j \Rightarrow \ \mathit{select}(\mathit{store}(a, i, v), j) = v$;
- $i \neq j \Rightarrow \ \mathit{select}(\mathit{store}(a, i, v), j) = \mathit{select}(a, j)$;

Moreover, the theory of equality allows reasoning of array equality. For example, the following Boolean expression $a = b \wedge i = j$ can be written as *select(a, i) = select(b, j).* Explicitly, two more axioms can be deduced from the array theory [9]:

- $a = b \Leftarrow \ \forall i, \ \mathit{select}(a, i) = \mathit{select}(b, i)$;
- $a \neq b \Rightarrow \ \exists i, \ \mathit{select}(a, i) \neq \mathit{select}(b, i)$;

By using tuple theory, modelling of union and structures becomes possible. Similar as in the array theory, tuples provide the same two functions: *select* and *store*, but they are used on the tuple elements. The function $\mathit{select}(t, f)$ will return the constant value hold by the field f of the tuple t and $\mathit{store}(t, f, v)$ returns the same tuple as tuple t, except that the value of the field f is v.

By using decision procedure [10] SMT efficiently handles terms in the given background theory, in comparison to SAT solvers, which need to replace all higher-level operators by bit-level circuit equivalents, resulting in a loss of word-level information used to formulate problems, thus scaling very poorly.

2.2 Bounded Model Checker

Model Checking was first proposed by Clarke and Emerson in [11] with the first algorithms enumerating all the reachable states of a system and then checking the correctness for a given property. This model was heavily restricted by the number of states a system could have, so it did not scale very well with industrial systems and was more appropriate for small designs. Model checking has three fundamental features as described in [12]:

- It is automatic, so there is no need for user interaction in order to prove given properties;
- Every system checked is assumed to be finite, for example, communication protocols or digital sequential circuits;
- It makes use of temporal logic to describe the properties of the system.

Thus, Clarke et al. [2001] stated that 'model checking is an algorithmic technique for checking temporal properties of finite systems.'

Because model checking was used as complementary to BDDs, this method was limited by the amount of memory necessary for manipulating and storing the BDDs, so it could only check systems with around a hundred latches, as these would be converted into formulae with a very few numbers of variables. Because of the huge success of solving Boolean formulas using SAT, there was a great motivation for studying the technique called Bounded Model Checking (BMC), first proposed in [13]. In its early stages, it did not fully solve the problem of systems complexities, but results have shown that it could handle instances with a huge number of variables (around hundreds of thousands) and much more clauses (even millions). Instead of the naïve approach of enumerating states and checking the given specification, BMC focuses on generating a counterexample of the specified property, with its execution length bounded by a variable k. Some systems could have a pre-known upper bound, otherwise known as 'Completeness Threshold of the design' described by Cimatti et al. in [13] where, if no bug is found until this bound is reached, then the verification is successful. Although BMC aim was the same as BDD-based model checking, there were two noticeable characteristics which made the method unique:

- It required an input the user had to provide, the bound, which would represent the number of cycles the method had to explore, thus if the bound is not high enough, implies that the method is incomplete;
- It opted for SAT techniques over traditional BDDs. Many experiments showed that SAT solver outperforms BDD, moreover, many problems considered hard for BDDs were easily solved by SAT.

2.3 Bounded Model Checker based on SMT

In order to analyze a program, BMC will transform it into a state transition system created from the control-flow graph (CFG)[14] which is discussed in the next chapter. As a formal definition, a transition system M, written as M = (S, T, S_0), is an abstract machine consisting of the set of states S, the set of initial states S_0 and the transition relation T, describing possible moves from one state to another. Assuming a transition system M, a given property ϕ and the bound specified by the user k, BMC will unfold the system k-times creating the VC ψ, where ψ is a quantifier-free formula, which will be checked by an SMT solver for satisfiability. The following logical formula represents the above-mentioned model checking problem [3]:

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_j + 1) \wedge \neg\phi(s_i)$$

with the following notations:

- ϕ = the safety property for checking;
- I = the set of the initial states;
- $\gamma(s_j, s_j + 1)$ = the transition from state j to state j+1;

The above formula is satisfiable if and only if there is a reachable state at time I, with I < k where ϕ is violated. If the formula is unsatisfiable, that means there is no error state that can be reached within the bound k. In case of satisfiability, the BMC must generate a counterexample for the property ϕ, which is represented by a sequence of states s_0, $s_1 \ldots$, s_k, where s_k is reachable within the bound k.

2.4 C programs vulnerabilities

As a formal definition, a vulnerability in a program is a property which violates the CIA Triad (Confidentiality, Integrity and Availability) allowing sensitive data to be leaked or even malicious code to be written as part of the program. Most common vulnerabilities regarding the C/C++ language are related to buffer overflow, such as: type overflow, array overflow, string overflow.

- Type overflow: happens when the value e of a variable v defined as type t exceeds the maximal value of t at runtime;
- Array overflow: happens when the variable a[i] is used when i¿size_a, where a is an array of size_a.

These vulnerabilities among many others are specified in detail in [15].

2.5 Related Work

In the early days of automatic verification, SAT-based BMC [16] was first proposed as a complementary technique to BDD, but this raised new challenges, especially in handling recursive function calls and in handling complex data structures. The next step, as solutions to the above problems were proposed in [5][17] and later implemented in the CBMC tool. The approach was to first build a propositional formula to model those program traces (within an upper bound) which violates given properties and second to use the SAT solvers to prove the satisfiability of the resulted formulae, or to generate a counterexample if the formulae is unsatisfiable. Following the solutions implemented in CBMC, a generalization was proposed in [18] which would have programs to be encoded into quantifier free formula, instead of propositional formula, which later would be checked for satisfiability using SMT solvers. This approach leads to more dense formulae when handling programs with complex data structures (e.g. arrays), because the encoding techniques of CBMC depends on both the size of data structures present in the program as well as the size of bit-vector representation of basic data types, whereas the size of quantifier free formula is independent. Experimental results obtained in [18] showed that using SMT is more efficient, not only because some formulae generated by SAT-CBMC where of such a size that makes solving them impractical, but also because SMT-CBMC scales better as the size of the data structures increases.

3 Design

This chapter covers the design of the Efficient SMT-based Bounded Model Checker (ESBMC), as well as its architecture, components and features. A description of ESBMC encoding of C programs into SMT-solvers background theories, introduced in the previous chapters, is also provided in this section alongside with illustrative examples.

3.1 Architecture

When a C program is provided to ESBMC, by default, it will check for user-defined properties, but also for pointer safety and alignment, division by zero and array bounds violations. As an option, it can check for deadlocks, memory leaks, overflows, and data-races. Figure 1 shows ESBMC architecture [4].

- **Front-end:** In its early days, ESBMC built on top of the CBMC model checker. This included the scanning of C/C++ source and computing its Parse tree, creating the intermediate representation of the program (IRep) and modeling the IRep into GOTO program using the control-flow graph generator. Now, ESBMC uses clang [19], a compiler suite for C/C++/ObjectiveC/ObjectiveC++ widely used in industry [20], as it provides a number of advantages for both the developers and the users: (i) the CBMC front-end needed maintenance as the ESBMC was developing;

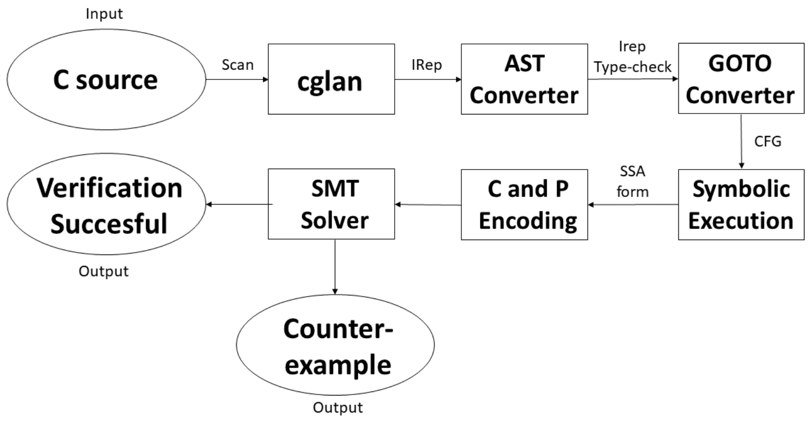


Figure 1: ESBMC tool's architecture.

clang removed this need of maintenance, thus helping developers significantly, (ii) it allowed ESBMC to provide both compilation error messages and meaningful warnings when the program was parsed, (iii) the analysis of the input code is easier with clang, as this allows for simplification of the program (e.g. evaluate static assertion, calculate sizeof expressions).

- **Control-flow Graph (CFG) generator:** The CFG generator is responsible for the creation of GOTO programs. These are created from the Abstract syntax tree (AST) of the input programs, and are a simplified representation of the source code, because it only consists of conditional and unconditional branches, assertions, assignments and assumptions. Basically, it removes all loops statements (for, while, do-while) and the switch statements. At this stage, additional checks are made such as out of bound access, division by zero and, optional, variable types overflow (e.g. integer overflow, floating-point overflow). Section 3.3 provides an ilustrative example of a simple code piece being converted into a GOTO program.

- **Symbolic execution:** The GOTO symex of ESBMC performs a symbolic execution of the GOTO programs: firstly, it will use the bound k provided by the user to unfold the program, secondly, the unfolded program will be converted into the static single assignments (SSA) and lastly, the safety properties are derived from the SSA and passed to the SMT solver, in order to be checked. At this step, dynamic memory allocation is checked by inserting the pointer safety checks. It is mandatory for the program to be unrolled first, so that the maximum set of dynamically allocated structures is known in advance by the pointer analysis.

- **SMT back-end:** At the time of writing, ESBMC supports five SMT back-ends solvers [4]: Boolector (default), Z3, MathSAT, CVC4 and Yices. Apart from Boolector, which does not support encoding for real arithmetic and linear integer, all solvers support encoding of arrays, tuple, bit vectors, fixed-point arithmetic and floating-point arithmetic into quantifier free formulas, which makes the back end very flexible and highly configurable. The SSA are converted into two sets of quantifier free formulas, one represents the constraints, written as C and the other set represents the properties, written as P. So the SMT-solver checks the satisfiability of $C \wedge \neg P$, more precisely, C is $I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_j + 1)$ and not P is $\bigvee_{i=0}^{k} \neg\phi(s_i)$. Depending on the result, we can say that the property holds (up to bound k) if C $\models$T P is unsatisfiable, otherwise, in case of satisfiability, a violation was found and ESBMC proceeds to generate a counterexample, using the set of SSAs which lead to the violation of the property.

3.2 Handling Input

For the SMT solver to determine the satisfiability problem for the properties of a given program, it first requires the constraints and properties of the C/C++ program to be converted into the background theories described in the previous chapter. In this section, the encoding for: scalar data types, fixed-point arithmetic, arrays, structures and unions, pointers, dynamic memory allocation and floating-point are briefly described, as they are detailed in [3].

3.2.1 Scalar Data Types

ESBMC comes with two approaches when modelling unsigned and signed data types. The first approach is to use the default integers which are provided by SMT-LIB theories. However, the second approach transforms the data types into bit-vectors with different bit widths. Thus, int, char, long int, long long int are treated as signedbv (while the unsigned version for these are considered unsignedbv). Relational and arithmetic operators, described in Section 2.1, are encoded depending on their operands' encoding. Conversions of integers into fixed point types is supported and performed using the functions described in Section 2.1, precisely Extract, SignExt and ZeroExt. Bool is also converted into signedbv and unsignedbv by using the ite function and the reverse conversion is possible, by comparing singedbv and unsignedbv to constants representing zero.

3.2.2 Fixed Point Arithmetic

Non integral numbers are used in many application domains such as telecommunication and discrete control, so ESBMC uses two approaches for encoding those numbers: binary, for bit-vector arithmetic and decimal for rational arithmetic.

Every rational number has an integral number I and a fractional number F, with m being the number of bits for representing I and n the number of bits used by F. Thus, ESBMC represents the number as the pair $\langle I.F \rangle$ which can be interpreted as $I + F/2^n$. As an example, 0.25 is represented as $\langle 0000.01 \rangle$ and 3.125 is $\langle 0011.001 \rangle$. Fixed-point arithmetic encoding depends on the approached used and is encoded as either bit-vector arithmetic or rational arithmetic by rounding. For bit vector arithmetic, operands must have the same bit widths for both the integral and fractional part. In order to achieve same bit width, the shorter bit sequence is extended with 0's from the left, if there are missing bits before the radix point, or from the right otherwise. For rational arithmetic by rounding, the fractional part F is divided by 2^n, the result being rounded to a given number of decimal places, the integer part is extracted as it is and everything is converted into a rational number in base 10. As an example, if $m = 2, n = 8, 4$ decimal places, the number 2.7 ($\langle 10.10110011 \rangle$) is converted to: $I = 2,\ F = 179/2^8$ and then rounded to 26992/10000. With this approach, speed and accuracy are traded off, but multiple SMT background theories can be explored, this is detailed in [3].

3.2.3 Arrays

Because of the functions store and select presented in Section 2.1 of the array theory, arrays are encoded with ease. For example, the assignment *value = array[i]* can be mapped using the *select* function and is encoded as *value = select(a, i)* and *array' = array WITH [i:=v]* is encoded as *array' = store(array, i, v)*. The only problem is the array out of bounds which can cause a program to crash, because in a program, the arrays are bounded, but in array theory, they are not. But ESBMC easily checks for this violation by simply generating VCs provable if and only if the indexing is within the bound, which ESBMC keeps a track of.

3.2.4 Structures and Unions

As for the arrays, encoding of structures and unions is done using the store and select functions from tuples theory. The expression *store(t, f, v)* returns a tuple t, where the value of field f is v. Unions are encoded in a similar matter, the only difference being an additional field l, which holds a number indicating the last field used for writing.

3.2.5 Pointers

For pointers encoding, ESBMC creates a tuple p with two fields:

- *p.o*: representing the object being pointed at by the pointer. This can be dynamically updated using the *store* function from tuple theory, depending on the changes of the object.

- $p.i$: representing the offset of that object. This depends on the object type, as pointers could point to an array (in which case $p.i$ is the index), a scalar (in this case, $p.i$ is fixed to 0) or a structure (here, $p.i$ represents the field of the structure).

Formally, regarding pointer property like: SAME_OBJECT, LOWER_BOUND, UPPER_BOUND and INVALID_POINTER, different literals are used with the following constraints:

- l_same_object $\Leftrightarrow\ p_a.o = p_b.o$;
- l_lower_bound $\Leftrightarrow\ \neg(p_a.i < b_l) \vee \neg(p_a = p_b)$;
- l_upper_bound $\Leftrightarrow\ \neg(p_a.i \geq b_u) \vee \neg(p_a = p_b)$;
- l_invalid_pointer $\Leftrightarrow\ (p.o \neq \nu) \wedge (p.i \neq \eta)$;

Where b_l is the lower bound of object b, b_u is the upper bound of object b, ν represents an invalid object and η is the encoding of the NULL pointer. A more detailed encoding of pointers is described by Cordeiro et al [3].

3.2.6 Dynamic Memory Allocation

Using dynamic memory allocations in embed software is a very discussed subject and although most disagree with this method, there are many programs that use dynamic memory allocations and ESBMC can deal with the use of functions malloc and free. By using the SMT solver theory on arrays and modelling the memory into an array of bites, basic operations of write and read on this array are converted to the logic level. Thus, three properties are checked by the ESBMC: (i) any call of malloc, free, or other dereferencing operations have as argument a **dynamic object**, (ii) free and dereferencing operations are called only on **valid objects**, (iii) the memory allocated using malloc is **deallocated** before the end of the execution. Thus, as for the pointer properties, three literals are used by ESBMC in order to check these properties:

- l_is_dynamic_object: checks whether the object is dynamic and between the memory bounds:

$$l\_is\_dynamic\_object \Leftrightarrow (\bigvee_{n=0}^{k-1} d_o.p_j = n) \wedge (0 \leq 1 < n)$$

- l_valid_object: checks whether the object is alive, by using one additional bit, which is true when malloc is called for the object (denoting is still alive) and false when free is used on the object (denoting is no longer alive):

$$l\_valid\_object \Leftrightarrow (l\_is\_dynamic\_object \Rightarrow d_o.\nu)$$

- l_deallocated_object: checks whether all the objects have been deallocated, at the end of the unfolded program:

$$l\_deallocated\_object \Leftrightarrow (l\_is\_dynamic\_object \Rightarrow \neq d_o.\nu)$$

Here, do represents a dynamic object, which is identified using the pointer p (which is bounded by k, so $0 \leq p < k$) and k is the number of dynamic objects. The dynamic object does also consist of m, representing the memory array with size n bytes and ν is the additional bit used for validating d_o.

3.2.7 Floating Point

In addition to the SMT standards, the SMT floating point theory was first proposed by Rummer and Wahl [21] in 2010 and deals with floating point arithmetic, comparison and arithmetic operators, NaNs, positive and negative infinities and zeroes, covering almost all the operations performed by the program and the encoding is that described above. The only limitations are casts on Boolean type and equality operators, which will be briefly described, as they are detailed in [22].

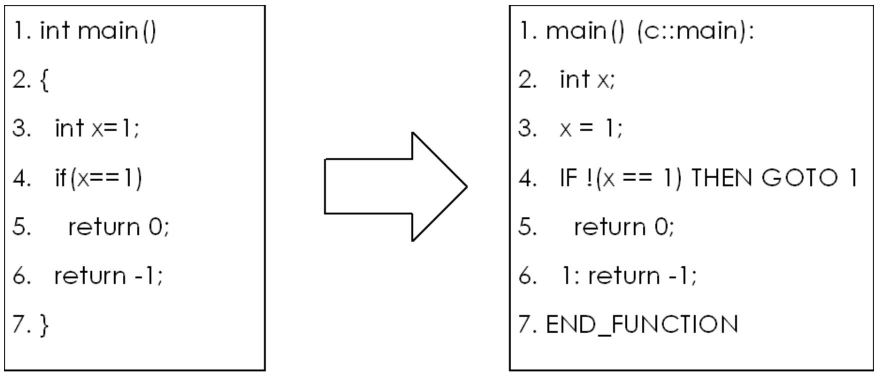
In order to deal with the with the encoding, ESBMC comes with two approaches: (i) Using the SMT theory of floating points (fully available in Z3, but partially in CVC4), (ii) using bit vectors, allowing support for all integrated solvers of floating point arithmetic. SMT-LIB provides two non-standard function for converting floating point into and from bit vectors: *fp_as_ieebv* and *fp_from_ieebv*.

- In order to deal with Boolean casting, the *ite* function is used and returns either 1.0 if the Boolean evaluates to *true*, or 0.0 otherwise. On the other hand, conditional assignments are used to encode floating points into Boolean: *true* if the floating point is anything but 0.0, $false$ if it is 0.0.

- Assignments and equality of bit vectors can be encoded with the equality operator (==), however, this dose not follow the SMT standard, which instead defines the **fp.eq** operator which is capable of handling special symbols, e.g. $NaNs$. The operator ($fp.eqxy$) returns $true$, if x is negative zero and y is positive zero, or if x is positive zero and y is negative zero. In case of an argument being a NaN, the $fp.eq$ will return $false$. Thus, ESBMC uses the $fp.eq$ operator to encode the equality of floating points, while the equality operator is only used for assignments.

3.3 Illustrative examples

This section provides illustrative examples of pieces of code being transformed into quantifier free formulas and encodings of different elements to further enhance the understanding of ESBMC capabilities. Figure 2 shows the conversion of a C program to a GOTO program, described in the ESBMC architecture.



```
int main()
{
  int x=1;
  if(x==1)
    return 0;
  return -1;
}
```

```
main() (c::main):
  int x;
  x = 1;
  IF !(x == 1) THEN GOTO 1
    return 0;
  1: return -1;
END_FUNCTION
```

Figure 2: C program converted into GOTO program.

Consider the C program from Listing 1, where the array a is initialised in a for *loop* and the pointer p points to a (line 8). Obviously, this program contains an array out of bounds, as i is initialised with 3 in line 3, line 10 violating the bounds property. ESBMC is able to spot the violated property by checking that $i_0 < 3$. Assuming that the bug is fixed and the C program initialises i with 2. ESBMC unfolds the program and transforms it into the SSA form as shown in Listing 2.

```
1  int main()
2  {
3    int a[3], x, i = 3, *p;
4    for(int j = 0; j < 3; j++)
5    {
6      a[j] = j + 1;
7    }
8    p = a;
9    if(x == 0)
10     a[i] = 0;
11   else
12     a[i-1] = 0;
13   assert(*(p+2) == 0);
14 }
```

Listing 1: Pointer to array example

This consist of assignments, where every variable is unique (e.g. the j integer from the for loop is replaced with j_0 , ... j_3). As described in Section 3.2.3, the WITH symbolic notation of the SSA is replaced with *store* function from the SMT theories.

```
i@1!0&0#1 == 2
j@1!0&0#1 == 0
a@1!0&0#1 == (a@1!0&0#0 WITH [0:=1])
j@1!0&0#2 == 1
Unwinding loop 1 iteration 1 file test2.c line 4
a@1!0&0#2 == (a@1!0&0#1 WITH [1:=2])
j@1!0&0#3 == 2
Unwinding loop 1 iteration 2 file test2.c line 4
a@1!0&0#3 == (a@1!0&0#2 WITH [2:=3])
j@1!0&0#4 == 3
Unwinding loop 1 iteration 3 file test2.c line 4
p@1!0&0#1 == &a@1!0[0]
guard@0!0&0#1 == (x@1!0&0#0 == 0)
a@1!0&0#4 == (a@1!0&0#3 WITH [2:=0])
a@1!0&0#5 == a@1!0&0#3
a@1!0&0#6 == (a@1!0&0#5 WITH [1:=0])
a@1!0&0#7 == (guard@0!0&0#1 ? a@1!0&0#4 : a@1!0&0#6)
\guard_exec@0!0 => a@1!0&0#7[2] == 0
```

Listing 2: SSA form of the C program

Using the SSA form, ESBMC builds the constraints C and the properties P described bellow, using the background theories.

$$C := \begin{bmatrix} i_0 = 3 \wedge j_0 = 0 \\ \wedge a_1 = store(a_0, j_0, j_0 + 1) \wedge j_1 = j_0 + 1 \\ \wedge a_2 = store(a_1, j_1, j_1 + 1) \wedge j_2 = j_1 + 1 \\ \wedge a_3 = store(a_2, j_2, j_2 + 1) \wedge j_3 = j_2 + 1 \\ \wedge p_1 = store(p_0, 0, a) \\ \wedge p_2 = store(p_1, 1, 0) \wedge g_1 = (x_1 = 0) \\ \wedge a_4 = store(a_3, j_2, 0) \\ \wedge a_5 = a_3 \\ \wedge a_6 = store(a_5, j_1, 0) \\ \wedge a_7 = ite(g_1, a_4, a_6) \\ \wedge p_3 = store(p_2, 1, select(p_2, 1) + 2) \end{bmatrix}$$

$$P := \begin{bmatrix} j_0 \geq 0 \wedge j_3 < 3 \\ \wedge i_0 \geq 0 \wedge i_0 < 3 \\ \wedge i_0 - 1 \geq 0 \wedge i_0 - 1 < 3 \\ \wedge select(p_3, select(p_3, 1)) = a \\ \wedge select(select(p_4, 0), select(p_4, 1)) = 0) \end{bmatrix}$$

The assertion made in line 13 is checked by first adding 2 to the value of p_i. This is done in the last *store* operation from C. The *select* functions from P check if the pointer and the array point to the same memory location. The verification fails because the SAME_OBJECT property is violated and a counterexample is generated, as shown in Listing 2.

```
VERIFICATION FAILED
Building error trace

Counterexample:

State 1 file test2.c line 6 function main thread 0
main
----------------------------------------------------
  a[0] = 1 (00000000000000000000000000000001)

State 2 file test2.c line 6 function main thread 0
main
----------------------------------------------------
  a[1] = 2 (00000000000000000000000000000010)

State 3 file test2.c line 6 function main thread 0
main
----------------------------------------------------
  a[2] = 3 (00000000000000000000000000000011)

State 4 file test2.c line 12 function main thread 0
main
----------------------------------------------------
  a[1] = 0 (00000000000000000000000000000000)

State 5 file test2.c line 13 function main thread 0
main
----------------------------------------------------
Violated property:
  file test2.c line 13 function main
  assertion
  (_Bool)(*(p + 2) == 0)
```

Listing 3: ESBMC output after running the fixed code from Listing 1

4 Testing and Results

This chapters presents how ESBMC was tested in order to prove the absence of memory safety and undefined behavior issues in C programs. Moreover, evaluation on ESBMC speed of generating GOTO programs as well as generating VCs is presented in order to demonstrate the efficiency and effectiveness of formal verification techniques.

4.1 Regression testing

Regression testing was mandatory to ensure that previous developed software performs as expected during the implementation of new components. ESBMC allows for regression testing, as this contains a python binding, which provides access to ESBMC internal API's and data structures using python language. Running the testing tool provided by ESBMC would return enough information in order to determine new bugs resulted during implementation and help debugging, because it dynamically generates unit tests and identifies the specific parts of the ESBMC that failed. In order to mitigate bugs as much as possible and to ensure that previous state of the software could be reached, Git was used for storing the project, because of its capabilities of controlling systems and tracking changes during development. Thus, control of the implementation as well as a better evaluation were ensured, which helped achieving this project aim.

4.2 Concurrency testing

Because C language allows running threads with shared memory, this could lead to a numerous of bugs and/or undefined behavior of programs. Most reliable for proving ESBMC capability of detecting such behaviors are concurrency tests, which validates different aspects of concurrency: mutexes, atomicity, race conditions, locks and others. For example, the code from Listing 4 [23] provides a simple lock created using C language, where LOCK is a global variable in line 1 and two functions, *lock* a Bool type in line 4 and *unlock* a void type in line 13, use it in order to lock and unlock the variable (change its value to 1 and 0 respectively). This program is considered to have an unbounded loop, the *while* from line 22 has no run-time bound. The option $--unwindx$ allows ESBMC to unroll the program a number of x times. The verification is successful if the unwinding assertion are off, which means that the first run of the program does executes as expected, however, if we set a bound to the program, ESBMC can spot the faulty behavior and generate a counter example for any $K \geq 2$, meaning that starting from the second iteration, the program does not execute correctly. It is important to notice that, in this case, the program correctness is not proved, however, ESBMC is able to find bugs.

```c
_Bool  LOCK = 0;
_Bool  nondet_bool();

_Bool  lock () {
  if(nondet_bool()){
    assert (!LOCK);
    LOCK=1;
    return 1;
  }
  return 0;
}

void unlock (){
  assert (LOCK) ;
  LOCK=0;
}

int main (){
  int t;
  unsigned is_locked = 0;

  while(t > 0){
    if(lock()) is_locked = 1;
    if (is_locked != 0) unlock();
    is_locked = 0;
    t--;
  }
}
```

Listing 4: C program with global variable and faulty behaviour

4.3 Evaluation

As there are a lot of components that need to be computed in order to validate a program, the speed as well as the effectiveness must be mandatory properties of the ESBMC tool. The data presented in Table 1 is collected from running ESBMC on a few of the concurrency tests which shows how effective conversions are made. The back end SMT solver used for validating these programs is Boolector v.3.2.0 [24]

Boolector v3.2.0							
Program	L	C	B	GOTO time	Symex time	Runtime decision	Verification Output
assume1	15	2(1)	1	0.074	0.0	0.001	(T)Failed
atomic section 1	16	2(1)	1	0.083	0.0	0.001	(T)Successful
atomic section 5	34	1(1)	1	0.072	0.001	0.0	(T)Successful
cond spawn 1	37	14(2)	1	0.108	0.02	0.001	(T)Failed
global pointer	33	26(2)	1	0.078	0.002	0.005	(F)Successful
loop	12	1(1)	5	0.069	0.0	0.0	(T)Failed
malloc	28	14(6)	1	0.097	0.001	0.004	(T)Failed
recursion	28	1(1)	10	0.064	0.0	0.0	(T)Failed
simple lock	28	35(34)	17	0.076	0.003	0.035	(T)Failed

Table 1: Data representing conversions and run time of ESBMC

Here, *Program* is the name of the concurrency test from the artefact, L represents the number of lines of the program, C is the number of constraints created by ESBMC x before the simplification and (y) the remaining constraints after simplification and B represents the unwinding bound. The GOTO time represents the time it took to create the GOTO programs from the control flow graph(CFG) discussed in Chapter 3, while the Symex time represents the symbolic execution of the GOTO programs. We can clearly see that the tool performs extraordinary well when generating GOTO programs and even better while executing them. This is also backed up by the choice of the SMT-solver, Boolector in our case, which has been proved in [3] by rigurous evaluation to perform better than other available SMT-solvers. Lastly, the verification output represents if a bug was found in one of the test, where (T) means that ESBMC did manage to make correct assertions of the test, and (F) means that the ESBMC fails to verify the program, due to known bugs.

A detailed comparison of different SMT solver being used as back ends, as well as a direct comparison between ESBMC and others BMC is described by Cordeiro et al. in [3]. The experiments showed that both Boolector and Z3 performs extraordinary well, compared to CVC3, due to the fact that they use memory more efficient and so memory overflow is absent for both Boolector and Z3, while also being more efficient in solving the background theories, where time is significantly less. Compared to SAT-version of CBMC, ESBMC noticeably outperforms CBMC in both time and memory, as being more precised and scalling significantly better for programs involving non-linear arithmetic, bit vectors and structures manipulation.

5 Reflection and Conclusion

The project's objective was to prove the absence of faulty behaviors and memory safety in C programs. This was motivated by the need of automatic verification, especially for enterprise grade software, where vulnerabilities can easily go undetected, until they are exploited. This was achieved by testing and evaluating the performance of ESBMC tool.

5.1 Reflection

The student had prior knowledge regarding Boolean Satisfiability and Satisfiability Modulo Theories, which greatly helped in understanding back end processes. However, this project greatly enhanced the student understanding of Bounded Model Checking, especially based on SMT. The student had previous experience with C/C++ programing languages and this project helped in further enhancing programing skills. Several challenges were encountered while working on this project. ESBMC is an open source software in continuous development, so the student had to ensure that the latest version was used for testing. One major challenge was the upgrade of building the ESBMC tool. In the beginning this was done by using a configure script, but after two months since the beginning, a rebase was done so that the tool would build using cmake [25], but a building guide was available only after three months from the update, so the student had to work with an older version for the time being. Another obstacle was setting up the environment for the tool. ESBMC makes use of several dependencies, some of which are very difficult to install due to lack of documentation. One major constant obstacle was the lack of proper equipment for this project, explicitly the student did not use a laptop for creating the environment in the beginning, but a home desktop was used instead. This led to inconsistent progress, as consultation with the coordinator was online. Finally, this is the first long term individual project of the student. Thus, the student learnt relevant lessons regarding project management, the most important being to take clear notes, which can be used throughout the project, as well as to create a project journal, which would clearly show the progression of the project. Regarding the above-mentioned challenges, as well as the ability to address them, the student considers this project a success. Through rigorous testing, the tool's efficiency has been proved and the proposed objectives have been completed. The student presented a theoretical understanding of the main concepts covered by this project, but had no practical experience, so laborious research and documentation was conducted throughout the whole duration of the project, ranging from understanding ESBMC to experimental evaluation of different existing automatic verification methods implemented by other BMC tools. If the project was to be completed again, several changings would be made regarding different approaches taken and decisions made. First, a laptop would be the appropriate choice of equipment for developing, as this allows feedback with progress being presented face to face. Secondly, the student would create issues within the tool's GitHub, allowing indications and solutions directly from the developers, which would have speed up the progress significantly. Lastly, the student would make use of available software such as Trello or OneNote, which would considerably help with the project management aspect.

5.2 Conclusion and Future Work

Within a limited time, the student managed to demonstrate the efficiency and capability of automatic verification of an open source tool, thus proving high understanding of the presented concepts. During this time, the student improved his knowledge significantly for a broad range of domains, while also elevating interest for this specific field, which is heavily researched.

The results presented could act as a base for further improvement of this project, which consist in verification of large embed software written in C used in security domain. Numerous encryption applications such as OpenSSL, OpenSSH, PuTTY are written in C and potentially contain numerous bugs which could be identify using ESBMC.

Improvement could be made on the front-end of the ESBMC as currently this is hard to maintain due to its huge size. One proposed solution was writing a front-end based on C++ clang, which would also allow automatic verification of C++ programs which use the standard template library (STL).

One addition to this tool would be the creation of a GUI, which would greatly help users getting started with ESBMC, as currently there is no documentation on that part and new users must be self-taught.

References

[1] Clarke E.M., Klieber W., Nováček M., Zuliani P. , *Model Checking and the State Explosion Problem.* [*In: Meyer B., Nordio M. (eds) Tools for Practical Software Verification*] 2012.

[2] A. Armando, J. Mantovani, and L. Platania *Bounded model checking of software using SMT solvers instead of SAT solvers* [*In: SPIN, LNCS 3925, pp. 146–162.*] 2006.

[3] Cordeiro, L., Fischer, B., and Marques-Silva, J., *SMT-Based Bounded Model Checking for Embedded ANSI-C Software* [*In: IEEE Transactions on Software Engineering (TSE), v. 38, pp. 957-974, IEEE*] 2012.

[4] Gadelha, M., Monteiro, F. R., Morse, J., Cordeiro, L., Fischer, B., Nicole, D., *ESBMC 5.0: An Industrial-Strength C Model Checker* [*In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 888-891*] 2018.

[5] E. Clarke, D. Kroening, and F. Lerda, *A tool for checking ANSI-C programs* [*In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2988, pp. 168–176*] 2004.

[6] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification.* [*In: Springer*] 2007.

[7] SMT-LIB, The Satisfiability Modulo Theories Library, `http://combination.cs.uiowa.edu/smtlib,` 2009

[8] J. Mccarthy, *Towards a mathematical science of computation* [*In: IFIP Congress. North-Holland, pp. 21–28*] 1962.

[9] R. Brummayer and A. Biere, *Boolector: An efficient SMT solver for bit-vectors and arrays* [*In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 5505, pp. 174–177*] 2009.

[10] L. M. de Moura and N. Bjørner, *Satisfiability modulo theories: An appetizer* [*In: Brazilian Symposium on Formal Methods (SBMF), LNCS 5902, pp. 23–36*] 2009.

[11] E. M. Clarke and A. Emerson, *Synthesis of synchronization skeletons for branching time temporal logic* [*In: Logic of Programs: Workshop, Yorktown Heights, volume 131 of Lecture Notes in Computer Science, pages 52–71. Springer-Verlag*] 1981.

[12] Clarke, E., Biere, A., Raimi, R. et al. *Bounded Model Checking Using Satisfiability Solving* [*Formal Methods in System Design 19, 7–34*] 2001.

[13] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs* [*In Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS. Springer-Verlag*] 1999.

[14] S. S. Muchnick, *Advanced compiler design and implementation* [*In: Morgan Kaufmann Publishers Inc.*] 1997.

[15] Technical report of the joint FCP Russian-French grant Nr. 02.514.12.4002, Step 4.

[16] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y., *Symbolic model checking without BDDs* [*Cleaveland, R., ed.: Proceedings of TACAS99. Volume 1579 of Lecture Notes in Computer Science., Springer193–207*] 1999.

[17] Kroening, D., Clarke, E., Yorav, K, *Behavioral consistency of C and Verilog programs using bounded model checking* [*In: Proceedings of DAC03, ACM Press 368–371*] 2003.

[18] A. Armando, J. Mantovani, and L. Platania, *Bounded model checking of software using SMT solvers instead of SAT solvers* [*In: Int. J. Softw. Tools Technol. Transf., vol. 11, no. 1, pp. 69–83*] 2009.

[19] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries* [*In: Packt Publishing*] 2014.

[20] C. Metz, "Why Apple's swift language will instantly remake computer programming."
`http://www.wired.com/2014/07/apple-swift/` 2014

[21] *Rümmer*, P., Wahl, T., *An SMT-lib theory of binary floating-point arithmetic* [*In: SMT Workshop*] 2010.

[22] Gadelha, M. Y. R., Cordeiro, L. C., Nicole, D., *Encoding floating-points using the SMT theory in ESBMC: An empirical evaluation over the SV-COMP benchmarks* [*In: 20th Brazilian Symposium on Formal Methods (SBMF), LNCS 10623, pp. 91-106*] 2017.

[23] The CProver User Manual [online]
`Available from: https://www.cprover.org/cbmc/doc/manual.pdf`

[24] Boolector SMT solver [online]
`Available from: https://boolector.github.io/`

[25] Cmake, designed to build, test and package software. [online]
`Available from: https://cmake.org//`