Scaling up Bounded-Model-Checking for Internet of Things devices

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

Feb 2022

Student Name: Yihan Lu

Student Number: 10463149

Supervisor: Lucas Cordeiro

Course: MSc Advanced Computer Science

Department of Computer Science

Table of Contents

Word Count: 14281

Abstract

This dissertation explores the applications of bounded model checking to Internet of Things devices. Bounded model checking is a software verification technique that can be used to find bugs or alternatively prove they do not exist in software. This is useful when making secure software. Bounded model checking can be used to detect common bugs and vulnerabilities such as array out-of-bounds errors, NULL pointer dereferences, and double-free.

Internet of things (IoT) devices are becoming increasing popular and becoming more involved in our everyday lives. Some examples include of IoT, smart home devices, wearables such as smart watches, and smart city devices such as smart streetlamps. In many cases, internet of things devices could be collecting sensitive data and are likely to be part of a wider network. For both reasons, internet of things devices can be a target for hackers. Therefore. It is important to able to develop software for them that will robust and not have bugs that hackers can exploit. To do ensure this, bounded model checking can be used.

The main focus of this dissertation will be on extending a popular bounded model checking tool, known as ESBMC. The goal is to develop an extension that will make it more suitable for verifying internet of things applications. This extension will use the concept of domain partition to attempt to make the problem it needs to solve easier. This extension will be evaluated by using a popular set of tests for bounded model checking tools. Finally, we will conclude this dissertation by using this tool to verify some functions from a software designed for an internet of things device.

Declaration

I declare that no portion of the work referred to in the 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.

Intellectual Property Statement

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) 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 dissertation, 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 entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, 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 dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy, in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations.

1 Introduction

The use of Internet of things (IoT) devices is a massive tendency in-home use and industry use. For example, they are applied in the areas of smart homes, health, traffic monitoring, agriculture etc. While this technique brings lots of benefits, potential risks of security vulnerabilities cause people's concern. The security weakness could exist in physical devices, connectivity protocols, and the software on IoT platforms, caused by malicious invasions, platform bugs, faulty devices, etc. IoT systems' risks bring inconveniences to people's lives, for instance, turn off the heat in cold winter or turn on the light when the family is sleeping. More severe consequences include leaking private information and leading car accidents. Since safety violations are inevitable in software programs, verification techniques are adopted by people to check the correctness of software.

This dissertation explores and extends the bounded model checking (BMC) method, which is a method of verification technique. BMC checks a given property in a system by unrolling the system for k times, translating it into a verification condition and testing its satisfiability [1].

The tool we will use is ESBMC [2]. It is a mature bounded model checking tool to find subtle bugs in multi-threaded C programs, which can detect array bounds violations, NULL- pointer dereferences, arithmetic overflows, or deadlocks. It allows users to add assertions in the code to find violations and it is not mandatory to add them. In addition, ESBMC implements a k-induction algorithm [3] and use satisfiability modulo theory (SMT) solvers to check the verification conditions of safety properties automatically. Model-checking is suitable to apply to IoT systems. On the one hand, there could be many states when an IoT system is running, so building model can cover all the states to find a counterexample of a safety property effectively and precisely. On the other hand, many IoT programs use C programs, which is suitable for ESBMC to check.

1.1 Problem Description

First, there are many components in an IoT system; the compromise of one component would influence other components. Second, missed interactions between components could be led by legitimate device failures and induced communication failures.

Besides, despite model checking is an effective way to verify IoT systems, the state space explosion problem can reduce its efficiency. This is because the state-space grows exponentially when the problem size increases, making it hard to find all possible violations. Here we introduce a main problem: How could BMC methods scale up to large IoT systems using domain partitioning?

1.2 Objectives

The state space explosion problem influences the performance of model checking tools. This thesis scale-up BMC by simplifying the constraint problem passed to the BMC's solver to find vulnerabilities in distributed IoT software to make full use of computational resources. This thesis is focused on addressing the following 4 objectives:

1. Develop an approach to select variables to be partitioned for a given domain
2. Choose a good partitioning strategy to precisely represent different cases in the model.
3. Evaluate the proposed approach on a set of open-source applications related to the IoT domain.
4. Evaluate the effectiveness and accuracy of ESBMC with the extension by using the SV-benchmarks GitLab repository.

2 Background and Theory

2.1 Internet of Things

The Internet of things (IoT) describes physical objects (i.e., things) that are connected and can share useful data over a network – In many cases, this is the internet. The devices are

usually embedded with sensors so that they can collect and share information. Examples of IoT devices include smart home devices such as lighting and home security, which can be controlled over the internet, typically from a mobile device. One of the main goals of Internet of Things devices is to improve the efficiency of everyday life.

The beginning of the Internet of things is often attributed to a Coca-Cola vending machine modified at Carnegie Mellon University in 1982 [4]. The modified vending machine could report its inventory and whether the drinks were cold or not over a computer network. The phrase “Internet of Things” was only later established in 1985 by Peter T. Lewis in a speech to the Congressional Black Caucus Foundation 15th Annual Legislative Weekend in Washington, D.C.

Since then, the internet of things has continued to grow, supported by the growth of the internet and the increasing usage of electronic devices. It has recently established itself as an important component in the everyday life of many people through applications such as smart homes.

Smart homes are designed to automate the usage of household devices such thermostats, lights, security systems, and appliances. This requires special IoT versions of these devices that can be connected over the internet so they can be controlled from a mobile device or a virtual assistant such as Alexa. In addition to being controlled wirelessly, these devices can save on electricity costs by automatically switching off when nobody is at home. There are many providers of smart home appliances such as Apple’s HomeKit, Samsung’s SmartThings, and Amazon’s Alexa smart home range. It should be noted that devices from different ranges may not be compatible, and the users may be locked into a single eco-system. Other consumer Internet of Things devices include wearable technology such as smart watches and optical head-mounted display such as Google Glass.

Outside of consumer applications there are many other uses of Internet of Things devices. Some other popular applications are medical, transportation, and the military. Applications to medical can include wearables that can measure heartrate and blood pressure and

specialized devices that can measure implants, such as pacemakers [5]. These devices allow for the health of a patient to monitored remotely, and automatically. This can help save millions on medical costs and catch potentially otherwise unnoticed health issues. The internet of things can be applied to transportation in many ways. A current popular application is to toll collection systems where drivers can pay using a transponder, or using their license plate information, instead of using cash [5]. This has the overall benefit of reducing congestions. A second application is to smart parking, where parking lots can keep track of the number of free spaces and their locations. They can also collect the payment in a similar fashion to the smart toll booths. A final application is to military purposes. For this application biometrics and environmental sensors can be used to keep track of threats on the battlefield and to also keep track of resources and allied soldiers.

There are many ambitious future applications of the internet of things, many being completely reasonable. One such example is self-driving cars which has been making steady progress over recent years. A more ambitious application is to creating smart-cities. This is where a city uses numerous types of electronic devices to collect data which can then be used to improve operations across the city. For example, data could be collected on transportation usage, which can be used to allocate resources to a certain route. Similarly, data could be collected on waste, which can inform the garbage collection services to clean a specific location [6]. There have been many cost-benefit analysis' done on smart city technologies, which give support to many of these technologies. Many tech companies have recently invested into developing smart city technology such as Google, Microsoft, Alibaba and Tencent.

Due to the amount and type of data that can be sent between internet of things devices, naturally, security concerns arise. These security concerns are like ones in existing computer networks such as webservers. One problem could be running old software which can be exploited. Furthermore, the wireless communication needs to be sufficiently encrypted. Distributed Denial of Service (DDOS) attacks could be used bring down IoT networks. Most likely, issues could arise from users using weak passwords. All devices need to keep up to

date and secured, as only a single weak device can lead to the whole network being compromised. As internet of things devices become more popular, there will be more data that can be stolen and as we further integrate these devices into our lives, this data will become more sensitive. In particular, applications to home security, healthcare and military need to be stringent about security.

It is important that developers take into consideration these potential vulnerabilities when creating IoT devices. Communication should be strongly encrypted, and both the hardware and software should be bug and exploit free. This can be verified using many of the already existing software and hardware verification tools.

2.2 Software Verification & Validation

Software verification and validation (often shortened to V & V) is a stage in software development with the aim checking that the software matches the requirements set out beforehand and the software is bug free and cannot be exploited. The verification and validation stages where summarised by Boehm as follows:

- Verification: Are we building the product right?
- Validation: Are we building the right product?

The validation stage is simply concerned with checking that the software satisfies the end users requirements. The verification step ensures the software itself is secure. This thesis is concerned with the verification stage, so we will now explore it in more depth.

Software verification is different for every program. The level of verification needed will depend on its size, complexity, purpose, and features. For small programs running at the user level, basic testing is likely sufficient. For programs running at the kernel, much stricter verification should be performed. This could include more advanced techniques such as fuzzing and model checking. It is important to not allow bugs and vulnerabilities to be present in kernel level programs because it allow give hackers to gain root access to the device. Multi-threaded programs are an easy source of bugs and vulnerabilities. This is

because multiple threads can access the same blocks of memory. This can cause problems such as race-conditions and dead-locks, which are particular to multi-threaded programs, but also can also lead to bugs such as NULL pointer dereferences.

There are two main classes that software analysis tools fall into:

- **Dynamic analysis**: This is type of software analysis that can be done to software by running it. Some examples of dynamic analysis are unit testing and performance analysis. Dynamic analysis is often fast to perform but will not provide the full picture because only certain paths of execution are used. Dynamic analysis usually does not require access to the source code.
- **Static analysis:** This is type software analysis that is done without executing the software, instead the analysis is performed on the source-code. Sometimes this is referred to as source-code analysis. Some examples include bounded model checking, code optimisation, enforcing code style. Many static analysis techniques are built-in to compilers such as code optimisation. To perform static analysis, the source code usually needs to be converted into some intermediate representation that is easier to use. In many cases this could be a control flow graph or an abstract syntax tree. Static analysis can be expensive to perform because they may need to consider possible paths of execution.

Two important concepts in software verification when it comes to detecting bugs and vulnerabilities are *soundness* and *completeness*:

- A software verification technique is *sound* if it can correctly conclude that a program has no vulnerabilities, i.e., there are not false positives.
- A software verification technique is *complete* if any vulnerability it finds is an actual vulnerability, i.e., there are not false negatives.

Ideally, we would like our software verification techniques to be both sound and complete, however this is hard to achieve. Soundness would require all possible paths of execution and all possible inputs to be considered. This suited to static analysis techniques. In contrast,

completeness is more suited to dynamic analysis. This is because a dynamic technique will provide concrete example of when a bug or vulnerability occurs. This cannot always be done with static analysis because the code is not executed. Software verification techniques from both categories should be used together to get to get a balance of soundness and completeness.

One of the most basic forms of software verification which falls under the dynamic analysis is unit testing. Unit testing is used to check if the output of a parts of a program, such as a function, matches the expected output for given parameters. This is done by writing tests in the same language as the rest of the source code. These tests can also be written to test edge cases that could cause issues, for example, division by zero. Most programming languages have libraries that make writing tests easy such as JUnit for java. It can be difficult to ensure a program is bug-free using only units tests, since each test usually only corresponds to one path of execution and possible sets of inputs, but unit testing is good to ensure that the program is doing what it is mean to do.

A more advanced technique for dynamic analysis is fuzzing. Fuzzing is the process of giving semi-random data to a program that may cause unexpected behaviour, memory leaks or crashes. Fuzzing can identity common vulnerabilities in programs without requiring much setup. Because of this fuzzing is a popular technique to find zero-day exploits in software. More specifically, there are two types of fuzzing: white-box fuzzing, this is where the source-code is available; and black-box fuzzing, where the source code is not available. To highlight the power of fuzzing, In 2019, Google discovered more than 20,000 vulnerabilities in Chrome using fuzzing. There are many open-source and freely available fuzzers online such as AFL++.

Model checking is static analysis technique that can prove that a program satisfies certain properties or will provide a counter-example. For example, a model checking tool can be used to prove a program will not dereference a NULL pointer. This can be expensive to perform as the tool will need to consider all paths of execution and all possible inputs. In many cases, it is only feasible to verify such properties up to a certain depth, for example, by

assuming a for loop is only executed a finite number of times. However, this may mean that we will lose out on soundness, if we cannot unwind the loops fully. In the next section, we explore bounded model checking in much more detail, which forms the basis of this thesis.

To help prevent bugs and vulnerabilities appearing in the first place, code can be written to match coding standards. These standards encourage good programming habits and can help prevent many bugs such as NULL pointer dereferencing by requiring NULL checks. In addition, if code conforms to a standard, it is easier for other developers to understand and make changes to the existing code. For example, there is a C programming standard known as SEI CERT C Coding Standard which provides rules to help C programs be secure and reliable. C compilers such as clang can check if code meets these standards by analysing the source-code in a form of static analysis. Most other popular programming languages have programming standards such as SEI CERT C++ for C++ and SEI CERT Oracle Coding Standard for Java.

Another import consideration in software security is cryptography. Strong forms of encryption should be used when devices are sending sensitive data to each other, for example, over the internet. This can prevent other users eavesdropping on the connection. Additionally, sensitive data that is stored on a device should be encrypted. The encryption algorithms should be modern and vulnerability free. Programmers should use up-to-date existing libraries which implement modern encryption algorithms as they are extensively tested and more likely to be bug and vulnerability free.

2.3 Bounded Model Checking

Model checking is one approach to verifying software. This is done by checking whether certain properties of a program hold for all its different possible states and paths of execution. A challenging problem in model checking is the explosion of the state space. As programs become more complex, by using more variables and control flow structures, the number of possible states blows up exponentially. This makes it increasingly difficult to

check whether the desired properties hold for all states. A common approach to address the state explosion problem is to use bounded model checking.

Bounded model checking reduces the state space by unwinding the loops and recursions a finite number of times. On this reduced state space, the desired properties are verified. This technique is efficient and well established. Although, bounded model checking can only show that these properties do not hold in the full state space by providing counter examples, but it cannot prove that these properties hold in the full space, unless the program is fully unwound. Therefore, this technique is not sound. But however, this technique is complete.

We can formalise the problem of model checking as follows. First, we formalise all the possible states the system can reach by a *Kripke* structure. A Kripke-structure is a 4-tuple

$$(S, S_0, R \subseteq S \times S, V)$$

Where S is the set of all possible states, S_0 is the set of all initial states, R is the set of all possible state transitions, and $V: S \to 2^{Prop}$ is a function that tells us which propositions are satisfied in a particular state. The Kripke structure encodes all possible paths of execution a program starting from an initial state S_0. The set of state transitions R corresponds to all branching statements in a program such as if statements and for loops.

The model checking problem is then deciding whether a set of logical formula $\phi_1, \dots, \phi_n$ are satisfied in the Kripke structure. These logical formulae are often expressed in a form temporal logic, so they can include references to the current and future state. If one of these logical formulas are satisfied, this would correspond to finding a bug in the program. We explore the problem of satisfiability of logical formulae in the subsequent chapters. In the context of bounded model checking, the maximum length of any path in the Kripke structure is bounded. This is because all the loops and recursions are unrolled a finite number of times.

Given a property ϕ we want to verify holds r our program and a bound k, bounded model checking will create a verification condition ψ which is related to ϕ in the following way:

ψ is satisfiable ⟺ there exists a counter example to ϕ in less than k unwindings

Because of this definition ψ will only need to refer to states that are possible after k unwindings, this ensures it is finite. If the verification condition ψ is satisfiable, then a counter example of the property ϕ occurring in program has been found within k unwindings. In the case that the system can be completely unwound in k steps, the verification condition ψ is equivalent to the original security property ϕ. And therefore, if ψ is unsatisfiable the program can declare free of property ϕ. This soundness property is not guaranteed to hold if the program is not fully unwound, as a counter example could exist for a higher number of unwindings.

There are many approaches that can be used to perform bounded model checking. An older approach used *binary decision diagrams* (BDD's) [7]. In this approach, a graph representing the possible states and transitions is never actually constructed, instead the set of states and transitions are encoded as binary decision diagrams. This approach was popular before the rise in popularity of SAT and SMT solvers. A more modern approach, which is similar to the approach implemented by ESBMC, is to convert to program and verification conditions into logical formulae and check the satisfiability using a SAT or SMT solver. The first stage of this process is to compute the state space of the program. This can be modelled by creating a graph $G = (V, E)$, where the vertices V are the states, and the edges E represent the state transitions and therefore the possible paths of execution. This is known as the control flow graph. A control flow graph can be created by using symbolically executing the code. This structure is analogous the Kripke structure. Once we have a control flow graph, a breadth-first search algorithm can be used to traverse the graph and symbolically execute it and convert the program and assertations it into a set of logical formulae which can be verified by a SAT solver or and SMT solver. Further details on this process are explored in the following sections section. A third approach known as Counterexample-guided Abstraction Refinement [8] attempts to find bugs in a program by considering an upper approximation of the original program. If a property holds in this upper approximation, it must also hold in the original program. However, if a counter example (a bug) is found, it

may not exist in the original program and may be a result of the upper approximation. In this case, the process is repeated using a more refined upper approximation.

Many programs make use of multiple threads which have access to a shared block of memory. This can lead to a multitude of bugs if two threads are competing to read and write from the same memory address at once. Bounded model checking can be used to verify that this won't occur in a multi-threaded program (up to a certain depth). One approach is for the model checker to consider all the possible *interleaving's*, which is the order that the CPU can execute the instructions between the different threads. To make checking this efficient, we can place a bound on the number of context switches, i.e., how many times the CPU changes between threads. Like the previous situations, this reduces the state space and allows for checking the satisfiability of these properties feasible.

Some common properties that can be checked with a bounded model checker include, NULL pointer dereferencing, array out-of-bounds violations and arithmetic overflows. Most popular BMC tools will support checking these properties out the box. Additionally, the user can write annotations in the source for further properties to be checked. Pretty much anything that can be encoded as a logical statement can be verified by the model checker.

A common application of model checking is to verify computer hardware designs. This is important because if there are bugs that are found after production, it will likely be impossible to repair the hardware without completely replacing it. This is very costly. Additionally, since hardware always has a finite number of states an input, it is theoretically possible to verify the hardware completely.

2.4 Boolean Satisfiability (SAT Solvers)

The Boolean satisfiability problem asks whether a given Boolean expression can be satisfied. This means can we replace the variables in the expression with TRUE or FALSE such that the expression is satisfied. This is commonly referred to as the SAT problem. A program that can solve this problem is known as a SAT solver.

Formally, a Boolean expression is a mathematical formula where the variables can take two possible values, TRUE or FALSE, and these variables can be connected by Boolean operators, such as ∧ (and),∨ (or) and ¬ (not). These evaluate how you would expect from any popular programming language. Some examples of Boolean expression are:

- $x \wedge (y \vee z)$
- $(x \Rightarrow y) \wedge z$
- $(x \vee y) \Leftrightarrow (x \wedge y)$

Here ⇒ is the implies connective and ⇔ is the if-and-only-if connective, this is true if the two side have the same truth value. In these formulae, x, y, z play the role of free variables. Free variables can be substituted for the truth values, TRUE or FALSE. A Boolean expression is said to be satisfiable if the free variables can be replaced with truth values such that the expression evaluates to TRUE. For example, $x \wedge (y \vee z)$ is satisfiable, since we can substitute all x, y, z with TRUE and the Boolean expression becomes TRUE ∧ (TRUE ∨ TRUE) which evaluates to TRUE.

An import property in Boolean algebra is that all Boolean expressions can be converted to an equivalent expression which uses only the ∧ (and),∨ (or) and ¬ (not) connectives. For example, $x \Rightarrow y$ is equivalent to $\neg x \vee y$. Every Boolean expression can be written in *conjunctive normal form (CNF).* This is where the expressions are formed from disjunctions from smaller formulas only using ∨ and ¬. Many algorithms for SAT solving require Boolean expression input to be written in conjunctive normal form. This is useful because the satisfiability of each clause can be checked independently. The following expression is an example of a Boolean expression in conjunctive normal form

- $(\neg x) \wedge (y \vee z) \wedge (\neg \mathrm{z} \vee \mathrm{x})$.

Any Boolean expression can be converted to conjunctive normal form by first, rewriting connectives such as ⇔ with equivalent expression in terms of ∨, ∧ and ¬, and then systematically applying double negation elimination, De Morgan's Laws, and the distributive law. This approach can be problematic as the number of terms can blow-up exponentially. A

more refined approach is to use Tseitin's transformation which produces a Boolean formula in conjunctive normal form that can only be linearly larger than the input [9]. Although, the Boolean expression produced by this algorithm is not necessarily equivalent to the original formula, as it may have more variables, but it is equisatisfiable. This means that the transformed expression is satisfiable if, and only if, the original input expression is satisfiable, which is sufficient for the purposes of SAT solving and model checking.

There does not exist an algorithm that can solve every instance of the SAT problem efficiently as it is NP-COMPLETE which follows from the Cook-Levin Theorem [10]. However, there exists many heuristics and fast implementations that can solve the SAT problem for certain expressions with thousands of variables in a reasonable amount of time. This is usually sufficient for model checking given that the state space is small enough.

A popular algorithm, which forms the basis of many SAT solvers, is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [11]. This is a backtracking-based algorithm that can check the satisfiability of Boolean expressions in conjunctive normal form. Although the DPLL algorithm maybe able solve some instances quickly, it has a worse case performance of $O(2^n)$. This worse case can occur when the Boolean expression is unsatisfiable, and all possible states have to be considered. The other possible outcome is that the expression is satisfiable, and the algorithm will return an assignment of each variable that gave the satisfiability. To improve the performance of this algorithm in practice parallelization and heuristics can be used.

There exist many popular implementations of SAT solver algorithms such as MiniSat[1] (which uses the DPLL algorithm) and CP-SAT[2] that are open-source and freely available online. MiniSat takes as input a file that expresses a Boolean formula in conjunctive normal form in the DIMACS-CNF format. An example of a Boolean formula written in the DIMACS-CNF format is:

[1] http://minisat.se/MiniSat.html
[2] https://developers.google.com/optimization/cp/cp_solver

```
 1  2 -3
-1  4  3
 4 -2  1
```

Which translates to the Boolean expression:

$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_3) \wedge (x_4 \vee \neg x_2 \vee x_1)$.

In the DIMACS-CNF format the numbers correspond to variables and the minus sign "-" represents the negation of a variable. The numbers on the same line are disjuncted and then the lines are conjucted, producing an expression in conjunctive normal form.

Although SAT solvers can only use Boolean variables, most things in computer science can be encoded as Boolean variables. For example, a 32-bit integer can be represented by 32 Boolean variables and its mathematical operators such as addition and subtraction can be encoded using only $\vee$, $\wedge$, $\neg$, this is identical to what is done in digital electronics. This means that we can use SAT solvers to check satisfiability of more complicated expressions such as those expressing properties of a program. Therefore, SAT solvers can be used for software verification.

The applications of SAT solvers are mostly restricted to constraint restriction problems such as bounded model checking, integer-programming, scheduling, and electrical circuit design. More complicated problems, often opt to use Satisfiability Modulo Theories (SMT) solvers, which build on similar ideas to sat solvers, but can use more complicated theories which for example can include floating-point numbers, integers, and arrays.

2.5 Satisfiability Modulo Theories (SMT Solvers)

Like the Boolean Satisfiability problem, satisfiability modulo theories is problem of determining whether a given mathematical formula is satisfiable. Unlike the Boolean satisfiability problem, this formula no longer needs to a propositional logic formula, but can be a formula in first-order logic. A first-order formula is an extension of a propositional logic formula that can include quantifiers such as for all and exists ($\forall, \exists$). This can introduce a huge amount of complexity, making determining if a formula is satisfiable very expensive.

In general, deciding if an arbitrary first-order logic formula is satisfiable is undecidable. Instead, SMT solvers implement certain first order logic theories that are usually decidable and are of practical interest, usually this involves not allowing quantifiers. Before discussing SMT solvers, we briefly review some important ideas from first order logic.

A formula in first order logic is a well-formed formula which can quantify over variables using ∀, ∃. The symbols available in first order logic are ∀, ∃ for quantification, ∨,∧, →, ¬, as logical connectives, ≡ for equality and parenthesis (,) to group terms together. It is common to see abbreviations being assigned

First order theories can introduce new functions and predicate symbols. For example, the first-order theory of arithmetic will include a `+` symbol and ZFC would include the ∈ symbol to denote membership. Some examples of first order formulas are:

- $\forall x \, \exists z \; x \; + \; y \; \equiv z$, is a first-order formula in the theory of arithmetic. Here z is a free variable.
- $\forall x \forall \mathrm{y} \exists \mathrm{z} (\mathrm{x} \in \mathrm{z} \wedge \mathrm{y} \in \mathrm{z})$, is a first-order formula in ZFC.
- $select(store(arr, i, val), i) = val$, is a first-order formula in the theory of arrays.

To assign meaning to a first-order formula we can interpret it using a model. A model assigns meaning to each function and predicate symbol and substitutes each free-variable for a value. Many theories have standard models, which give typical definitions for each symbol. For example, in the standard model of arithmetic, the variables take values which are integers and the `+` symbol is interpreted as the standard addition operation.

Formally, A model $\mathcal{M}$ of a first-order formula consists of

- A nonempty set D that is the domain variables can take values in
- An interpretation of each function symbol f that appears in the formula,
- An interpretation of each predicate symbol P that appears in the formula,
- A variable assignment $s: FV(\phi) \rightarrow D$ of each free variable to an element of D.

If there exists a model $\mathcal{M}$ such that a first order formula ϕ is true under the interpretation of $\mathcal{M}$, we say that $\mathcal{M}$ is a model for ϕ, or that $\mathcal{M}$ satisfies ϕ. This is denoted by $\mathcal{M} \vDash \phi$. This is more general but analogous to the definition of satisfiability in the bounded model checking section.

To give an example consider the formula $\phi(x, y, z) \coloneqq f(x, y) \equiv z$, where '+' is a binary function. A model for this is

- $D = \mathrm{N}$.
- f is addition of natural numbers.
- The variable assignment $s(x) = 3, s(y) = 2, s(z) = 5$.

Under this model the formula becomes $3 + 2 \equiv 5$, which is true. Therefore ϕ is satisfiable.

SAT solvers support many useful theories for both programming and mathematics, some of these theories are:

- linear arithmetic
- bit vectors
- arrays

These theories are helpful when encoding programs as first order logic formulae and can be used to guarantee that certain properties of a program hold. When SAT solvers find a model for a first order formula, the functions and predicates that appear in the formula have a standard meaning that is assumed and finding the variable substitution is the important part.

As an example, we will explore the theory of arrays. This is implemented in many SMT solvers by default as it is very useful for software verification. As expected, the objects in the theory of arrays are arrays, which are defined to satisfy the axioms we will see in a moment. Firstly, the theory of arrays has two key functions:

- $store(arr, i, val)$
- $select(arr, i)$

These functions allow us to formalise storing and retrieving values in an array. In addition, the theory of array includes some fundamental axioms about these two functions. These are

- $select(store(arr, i, val), i) = val$
- $select(store(arr, i, val), j) = select(arr, j)$ for any $i \neq j$
- $\big(\forall i : select(A, i) \equiv select(B, i)\big) \rightarrow A = B$

These axioms essentially define the theory of arrays and are necessary to prove properties of arrays.

There are many ways that SMT solvers can determine the satisfiability of a first order formula. The most simple and original approach is to convert the formula to a Boolean formula. This can then be directly fed into an existing sat solver. This of course grossly expands the number of variables, for example a 32-bit integer, would be replace with 32 single-bit variables. Additionally, this approach could miss out on any simplifications that can be performed which are specific to the implemented first order theories. A more modern approach is to use a similar algorithm to DPLL algorithm specific for first-order formulas. This is known as DPLL(T) [12]. This works very similarly to DPLL, where it is a backtracking-based algorithm but instead it can reason about other first order theories in addition to Boolean logic. Popular SMT solvers such as Z3[3] and MathSat[4] use the DPLL(T) algorithm and achieve state-of-the-art speeds. Z3 is an open-source SMT solver developed by Microsoft, MathSat is propriety and is a joint project of Fondazione Bruno Kessler and DISI-University of Trento.

Many SMT solvers support the SMT-LIB format, which is a Lisp-like language which the mathematical formulae can be easily expressed in. The choice to use lisp is due to its popularity, its simplicity, and its ease to parse. Additionally, its declarative and functional

[3] https://github.com/Z3Prover/z3
[4] https://mathsat.fbk.eu/

nature suits expressing mathematical formulae. The following snippet of code is an example of an SMT-LIB program that expresses the Boolean formula $\bar{a} \vee \bar{b} \Leftrightarrow \overline{a \wedge b}$.

```
(declare-fun a () Bool)
(declare-fun b () Bool)
(assert (not (= (not (and a b)) (or (not a)(not b)))))
(check-sat)
```

We can simply provide this program as input into Z3 and it will check if the logical formula is satisfiable. Z3 supports more complicated theories than Boolean logic. We can check the satisfiability of a formula involving integers, such as $a \times b = 100$. This is expressed below as a program in Z3.

```
(declare-const a Int)
(declare-const b Int)
(assert (= (* a b) 100))
(check-sat)
```

This of course is satisfiable and has many models. One such model is one that assigns the variables $a = 10, b = 10$.

2.6 Encoding a program as a set of logical formula

To further understand the relationship between code and logical formulae, in this section, we give an example of how a program and its desired properties can be encoded as logical formulas which can then be checked by an SMT solver such as Z3. This will highlight a typical approach a bound model checking problem could implement. The goal will be to construct a conjunction of logical formulas C which will represent the instructions and a conjunction of logical formulas P that represent the desired properties of the code. For example, P may include properties that ensure there is no array out of bounds violations or null pointer references. Consider the following small C program that assigns a value to an array to an element in the array.

```
int main() {
 int arr[10], v, x;
 if (x < 10 && x >= 0)
   arr[x]=v;
}
```

Because this program is simple, we can see that this program will not produce an array out-of-bounds violation, but we will see how we can do this in automatically. Firstly arr, v, x are free variables since they are not assigned to any values. We begin by encoding the expression in the if statement.

$$g_1 \coloneqq x < 10 \wedge x \geq 0$$

This is simply just a logical formula in the theory of linear arithmetic. Next, we encode the assignment of the array if g_1 is satisfied:

$$arr_1 \coloneqq store(arr_0, x, 10)$$

Here, $store$ is the function from the theory of arrays, which is supported by many SMT's. Next, we combine this with an if statement:

$$C \coloneqq [g_1 \coloneqq x < 10 \wedge x \geq 0 \wedge arr_1 \coloneqq store(arr_0, x, 10) \wedge arr_2 \coloneqq ite(g_1, arr_1, arr_0)]$$

This logical formula represents the program. Notice that each time we modified the array we assigned it to a new variable which are distinguished by different subscripts. This means that we have performed single static assignment.

Next we will construct the logical formulas P which represent bugs that we do not want to occur in the program C. Particular to this example, we might want to ensure an array out-of-bounds violation does not occur. To do this we just need to check that the expressions used to index the array are within the bounds of the array. In the above program, the time the array is indexed is by the variable x, and this only happens if the if statements are executed. Therefore, our verification condition is

$$P \coloneqq g_1 \rightarrow x \leq 10 \wedge x \geq 0$$

Most bounded model checkers have support to automatically created such properties for array out of bounds violations. To check if this property is violated for the program, we want to check if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models C \wedge \neg P$. This would provide us a counterexample for int terms of the free variables arr, v, x which causes the property P to be violated.

In practice the code and verification conditions, may not be converted to a logical formula written in mathematical notation like above, but may be written in the lisp-like SMT-LIB2 language. This process would be essentially equivalent and is necessary to use existing SMT solvers.

In a more complicated example, the first two steps will typically involve unwinding loops and explicitly converting all of the variables in the program to a single static assignment form (meaning each variable can only be assigned a single value) and then converting the SSA to a logical expression. These steps are often necessary to encode the program as a logical formula. In more complicated programs, things such as floating-point numbers will need to be carefully converted into a logical representation. Many SMT solvers support fixed point theories, but not all support a complete floating-point theory. Sometimes this is addressed by encoding floating points numbers a bit-vector representation.

Overall, this is similar to the approach ESBMC takes which will be further detailed in the relevant chapter.

2.7 Bounded Model Checking and K-induction

Proof by induction is an important proof technique that can be used to prove a proposition $P(n)$ holds for all natural numbers n. This is useful for bounded model checking because we can use induction on the number of unwindings of the loops to prove a property holds for a program. Although there are other forms of induction that are more effective in bounded model checking such as K-induction. Like proof by induction, k-induction is a proof technique that can be used to prove that a statement holds for every natural number. We will define k-induction formally shortly after reviewing some basic properties of induction. Firstly, proof-by-induction can be stated as following axiom

$$\phi(0) \wedge (\forall n(\phi(n) \rightarrow \phi(n+1))) \rightarrow \ \forall n\, \phi(n)$$

Which holds for any logical formula ϕ. A stronger form of this statement (which is in fact equivalent) is commonly used which is know as strong induction, it can be stated as follows:

$$\Big(\forall \mathrm{n}\Big(\big(\forall \mathrm{m} \leq \mathrm{n}\phi(m)\big) \rightarrow \phi(n+1)\Big)\Big) \rightarrow \forall n \phi(n)$$

In many cases this statement can be easier to use since it assumes that $\phi(m)$ holds for all $m \leq n$ rather than for a single value of n. The final form of induction we consider is known as k-induction. This is stated as follows: for any k we have that

$$\big(\forall i < (k-1)\phi(i)\big) \wedge \Big(\forall n\Big(\big(\forall i \leq (k-1)\,\phi(n+i)\big) \rightarrow \phi(n+k)\Big)\Big) \rightarrow \forall n\,\phi(n+k)$$

This is similar to strong induction but some of the terms from the inductive assumption have been moved into the base case. K-induction is suitable to be used with bounded model checking because we can use the number of times the loops are unwound as the k in the k induction. This is particularly useful if the property we are trying to prove is dependent on a loop. Next will we discuss how k-induction can be used by a model checker. This is a summary of the ideas presented in [13].

To implement k-induction in a bounded model checker it is we consider three propositions. Firstly, let $\phi(\mathrm{s})$ denote a verification condition about the program and let $\psi(\mathrm{s})$ denote a formula representing if a loop is fully unwound, where s is the current state of the program. Additionally, let $I(s)$ be a proposition that denotes the properties valid in an initial state and let $T(s, s')$ represent a transition from state s to s'. Essentially I and T repesent the code from the program. We define the base case as:

$$B(k) \coloneqq I(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg\phi(s_k)$$

This proposition is standard bounded model checking. If it is satisfiable, then we have generated an example for the verification condition $\phi(s)$ which corresponds to sum kind of bug. If the base case holds, we can continue trying to prove that property ϕ holds. To do this, we can first try to prove that the loop is fully unwound. If this is the case, the property ϕ must hold. This is the same as standard bounded model checking. To do this we use the forward condition:

$$F(k) \coloneqq I(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg\psi(s_k)$$

In the case that the loops are not fully unwound, but the base case holds, we can try to use an inductive step. Here we making the assumption that the verification condition ϕ hold for all states s_i with $i \leq k + n$ we want to show that it must hold for s_{k+n+1}. This is equivalent to showing the following statement, known as the inductive step, is unsatisfiable.

$$S(k) \coloneqq \exists n \in N\colon \phi(\mathrm{s_n}) \wedge \ldots \wedge \phi(\mathrm{s_{n+k}}) \wedge T(s_n, s_{n+1}) \wedge \ldots \wedge T(s_{n+k}, s_{n+k+1}) \wedge \neg\phi(s_{n+k+1})$$

What follows from this is the K-induction rule which can be used prove properties hold or to produce counter-examples. Given a number k to initially unwind the loops, and a verification condition ϕ the k-induction rule is then defined:

$$kind(\phi, k) \coloneqq \begin{cases} a\ counter\ example\ is\ found\ to\ \phi & if\ B(k)\ is\ satisfiable \\ property\ \phi\ holds & if\ B(K)\ \wedge F(k)\ \ is\ unsatisfiable \\ property\ \phi\ holds & if\ B(K)\ \wedge S(k)\ \ is\ unsatisfiable \\ kind(\phi, k\ +\ 1) & otherwise \end{cases}$$

In summary, the k-induction rule is able to disprove a property ϕ by generating a counter example for the base case. K-induction can prove a property holds in two different ways, the first is equivalent to standard bounded-model checking, where we prove the assertation holds after k unwindings and the loop are fully wound. The second is by using the base case and the inductive step. This will be used in the case that we cannot prove the loop is fully unwound (the forward condition). If a counterexample or a proof of correctness cannot be constructed, k-induction will then try again for k+1 unwindings.The k-induction rule is used by many bounded model checkers such as ESBMC as it is known to be more efficient that other induction techniques.

In practice, it can be hard check the inductive case holds because it may be expensive to compute what the values of the variables will take after $n + k\ +\ 1$ iterations. In this case, an upper approximation of $S(k)$ can be created where the variables that are changed in the loops are assigned non-deterministic values. If this upper approximationis unsatisfiable, then so is the S(k) and therefore $\phi(s)$ must hold for all s. This process of assigning the variables non-deterministic values is known as *havocing*. To get a closer upper

approximation, an invariant can be inserted into the inductive step. This can be much easier for an SMT solver can check. This is explored in more depth in [13].

For a mathematical perspective on k-induction, including proofs of its correctness consult [14]. For reference to how it was first introduced in the context of bounded model checking see [15].

2.8 Bounded Model Checking Tools

In this thesis we will use a modern and Industrial strength bounded model checking tool known as ESBMC [16]. ESBMC can verify properties in single-threaded and multi-threaded ANSI C programs. It has built in features to check for many common bugs such as array-out-of-bounds errors, NULL pointer dereferencing, and arithmetic overflow. In addition, ESBMC allows users to define their own assertions in the code to be checked. ESBMC has multiple features which can bound the state space, it can perform standard bounded model checking or use a k-induction rule, both limit the number of iterations considered in unbounded loops. To verify the assertations, ESBMC converts the program and assertations into a set of quantifier-free first order formulae which can be checked by an SMT solver. ESBMC supports a range of popular SMT solvers out-of-the box such as Z3, CVC4 and uses Boolector by default. ESBMC uses the C compiler clang to parse and generate an abstract syntax tree of the inputted C code. It is also used to inform the user if there are problems with compiling their program before verifying it. ESBMC has been used to verify applications such as control software in Unmanned Aerial Vehicles [17]. ESBMC is fast as proven by its performance on the SV benchmarks. ESBMC is a fork of CBMC, the C Bounded Model Checker.

ESBMC checks for many bugs by default, such as such as array-out-of-bounds errors, NULL pointer dereferencing, but a user can annotate a C program to instruct ESBMC to check if a specific property will hold for a program. In particular, the user can assign variables non-deterministic values which will cause ESBMC to consider all possible values for them. The non-deterministic variables will be encoded as free-variables in the logical formula sent to

the SMT solver. Additionally, special assume and assert functions can be used in the code to assume and assert properties of variables. The following is an example of a C program that has been annotated to be checked by ESBMC.

```
int main() {
  unsigned int x = _nondet_uint();
  unsigned int y = _nondet_uint();
  __ESBMC_assume(y < 10);

  int sum = 0;

  for (int i = 0; i < y; i++) {
    sum += x;
  }
  assert(x * y == sum);
}
```

The programs assigns x and y non-deterministic values so we can verify the following code over a large range of values. We can also make assumptions about variables appearing in the code or other logical properties using the __ESBMC_assume. Finally, the program asserts that two values are equal. ESBMC will attempt to prove that this property holds or try to find a counter example. In the next section we look at more examples of how ESBMC can be used to verify software.

Figure 1 shows the steps taken by ESBMC to verify ANSI C code. We break down these stages in detail throughout the rest of this section.

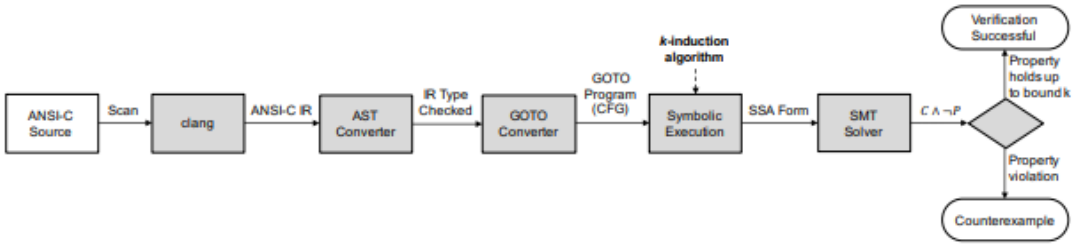

Figure 1: Steps Taken in ESBMC 5.0 to verify ANSI-C code. This image is taken from [16]

In the first stages, the source code is given to clang which tokenises and then parses the source code and generates an abstract syntax tree of the program. The abstract syntax tree is a type of intermediate representation of the source code, which is easier to perform static analysis on. Clang performs type-checking on the AST as would be necessary when

compiling a C program. Furthermore, clang can use the AST to simplify to the program, for example, by replacing sizeof expressions and constants.

The next step is to convert the AST into another intermediate representation which is a *control-flow-graph.* A node in the CFG represents a (non-) deterministic assignment or a conditional statement, and an edge in the CFG represents a jump in the program's control flow. In ESBMC this control flow graph is represented by a GOTO program. This is a simplified version of the original source-code where all branching statements, such as, for and while are replaced with if-statements and goto-statements. In this step, ESBMC can also automatically insert assertations for properties such as division by zero and array out-of-bounds errors.

In the next step, the GOTO program is symbolically executed. The symbolic execution engine unrolls the program and generates the SSA (static single assignment) form of the program. In this form, each variable will only be assigned once. This is necessary to be able to convert the program and assertations into logical statements that can be checked, so the possible values of the variable can be distinguished over time.

Finally, the SSA form of the code is then converted into a conjunction of quantifier-free logical statements C and the assertations in the code are also converted into a conjunction of quantifier-free logical statements P, known as the verification conditions. To conclude, we want to check that if $C \wedge \neg P$ is satisfiable, in other words, check if there exist a model $\mathcal{M}$ such that $\mathcal{M} \vDash C \wedge \neg P$. This can be verified by giving the statements into an SMT Solver such as Z3. If it is satisfiable a bug has been found.

When converting the program to the logical formulae, ESBMC can encode most variables and operations using the built-in theories in the chosen SMT solver. One exception is to floats and doubles (floating-point numbers). If ESBMC is using Z3 as a backend, it can use its floating-point theory, which is based on the IEEE standard. Otherwise ESBMC will fall back to encoding floating points as bit vectors, which can be much more inefficient to work with. A third option is for ESBMC to encode floating point variables using the theory of fixed-point

numbers, however this will be an over approximation of floating points, so it may find spurious bugs that are not present when using a floating-point representation. One benefit of using the fixed-point theory is efficiency. In addition, it may be useful when the main properties that want to be checked does not concern floating point arithmetic.

By default, ESBMC aims to be sound. Meaning that it will only certify a program as bug-free if it is in fact bug-free. This means that ESBMC will insert assertations after unwinding a loop to ensure the loop has been fully unwound. Soundness in some cases may be unnecessary, particularly in the case where a user is only trying to find bugs and errors. Different modes and settings can turn this off. ESBMC has a couple of main modes it can operate in. These are:

- Standard BMC (used by default),
- Falsification (--falsification flag),
- Incremental BMC (--incremental-bmc flag),
- K-induction (--k-induction flag).

These will be explored in detail along with some examples in the next section.

A big selling point for ESBMC is that is also has support for checking multi-threaded C programs which use posix threads. ESBMC implements two approaches to do this. The first approach is to explore all possible interleaving's of the threads. Naturally, this can become incredibly expensive to compute, so ESBMC limits the number of interleaving's it can explore by placing a bounded on the number of context switches - this is the number of times the CPU can switch can between threads. The second approach is to use scheduling. ESBMC has support to generate verification conditions that will check for issues that can occur in multi-threaded programs such as deadlocks.

In addition to supporting C, ESBMC has support for both C++ [18] and solidity smart contracts [19]. These languages are both natural targets for bounded model checking. C++, much like C, can have bugs easily introduced into the source code from poor programming or bad memory management. Additionally, C++ is a very popular language. Solidity is a

programming language that can be used to write smart contracts for the Ethereum virtual machine. These smart contracts can be used for large cryptocurrency transactions. Because of this, bugs in Solidarity code can be devastating, making this language a natural target for bounded model checking.

In addition to ESBMC, the exists many bounded checking tools for detecting bugs and verifying software. Another popular bounded model checking tool for C is FRAMA – C. This takes a different approach to bounded model checking which is based on the weakest-precondition calculus, but still uses SMT solvers such as Z3 to check verification conditions. In particular interest to this thesis, there are current applications of FRAMA-C to internet of things software [20] [21] [22] [23]. Some bounded model checkers that support languages other than C and C++ are JBMC for java and Gomela for Go. Currently there is more interest in verifying C and C++ programs than other languages because of their popularity and their applications to embedded systems, which can be running critical software. One example of using model checking software to critical software is when Airbus used a predecessor of FRAMA-C know as CAVEAT to verify the C code that would later run on their A380's [24].

ESBMC continues to prove to be one of the best performing bounded model checking tools as proven by recent competition results[5]. In addition, ESBMC is currently one of the best tools that uses k-induction. ESBMC is written in C++ and it is open-source where its source code can be found on github[6]. It also has a Python API that can be used to access it internal data structures.

2.9 Verifying Software with ESBMC

In this section we explore how we can perform common software verification tasks using ESBMC. We will explore the some of the different options and dissect how they work internally.

[5] http://www.esbmc.org/
[6] https://github.com/esbmc/esbmc

The standard BMC mode, which is operated by default, simply unwinds the loops a given number of times and then checks the satisfiability of the generated verification conditions from the program. By default, ESBMC will add assertations after the loop unwinding, to assert that the loop has been fully unwound. This is to ensure completeness. This can be problematic and prevent ESBMC from checking the rest of the program if the goal is to only find bugs. Instead, these assertations can be turned off using the --no-unwinding-assertions flag. An alternative approach is to use falsification mode.

The falsification mode iterates through all possible number of unwindings with the goal of finding a counterexample to the verification conditions. This mode does not care about proving the verification conditions are unsatisfiable and showing the program is correct. In particular, the falsification mode does not check if a loop was fully unrolled by inserting assertations unlike the default, incremental-bmc and k-induction mode.

The incremental BMC mode performs bounded model checking by iterating through the possible number of loop unwindings. Once all loops are unwound, two stages are performed to check the program. The first stage involves checking that all of the assertations hold. If this is not the case, a counterexample has been found to the verification conditions and it stops here. Otherwise, stage two, involves checking if the loop is fully unwound by inserting an assertation after the loop. If the loop is fully unwound, the verification conditions are unsatisfiable and therefore the program is correct. If the loops are not fully unwound, the process is repeated for a higher number of unwindings. ESBMC has settings to change the how much the unwinding should be incremented after each attempt and the initial value it should start at. In the case that memory is completely used up or a maximum number of unwinding's is reached, the program quits without deciding if the program is bug-free or not.

K-induction is the leading feature of ESBMC. It is an efficient method that can prove a program is bug free using the k-induction rule (a variant of proof by induction). In many cases it requires less iterations to prove the correctness of a program over incremental-BMC. Similar to incremental BMC, k-induction increments through the number of

unwindings until either it finds a counterexample to the verification conditions, or it is able to prove the verification conditions are unsatisfiable. To prove this, the k-induction rule is used, which was discussed in a previous section. In the case that memory is completely used up or a maximum number of unwinding's is reached, the program quits without deciding if the program is bug-free or not.

ESBMC provides us with some special functions we can use to verify properties of programs. Firstly, we can assign non-deterministic values to variables. This can be done using the $nondet$ functions. For example we can create a non-deterministic integer and a floating point number as follows.

```
int x = nondet_int();
float y = nondet_float();
```

When ESBMC encounters a non-deterministic variable in a program, it will translate it to a free-variable in the final logical formula. This means all possible values of it will be considered. We can restrict these free variables using the __ESBMC_assume function. For example, the following code restricts the free-variables to take a value between 0 and 10.

```
int x = nondet_int();
__ESBMC_assume(x =< 10 && x >=0);
```

__ESBMC_assume can takes any expression that has a Boolean value as a parameter. This Boolean expression will be directly inserted into the final logical formula generated from this program. The final important function that can be used by ESBMC is $assert$. This is infact a built-in function to C, but ESBMC can verify the asserted properties for all possible states and paths of execution, whereas the assertions would usually be checked at run time for determined values.

With knowledge of this different modes and the different ESBMC functions, we will consider some example applications that would be useful in practice.

First of all consider the following simple program. This program contains a function sum that is expected to add the number 2 together n times. This expected behaviour is encoded in an assertion in the main function.

```
int sum(n) {
  int c = 0;
  for(int i=1; i<=n; i++) {
    if (i<1000)
    c += 2;
  }
  return c;
}
int main() {
  int n= nondet_uint();
  int s = sum(n)
  assert(s == n*2 || sn == 0);
}
```

Suppose that we are only concerned in finding a bug in this program. To do this we can search for a counter example using any of ESMBC modes. However, it may be more effective to use the falsification mode, because then assertions won't be inserted to check if a loop has been unwound fully. Trying to prove this program is bug-free using ESBMC will be difficult and may be impossible, because the loop can go on arbitrarily long, so it won't be possible to unwind it completely, and also because the condition we want to check depends on the number of iterations.

Another interesting problem we can use ESBMC to check is the termination of a loop. Consider the following program:

```
int main()
{
  int x=_nondet_int();
  int *p = &x;

  while(x<100) {
   (*p)++;
  }
  assert(0);

  return 0;
}
```

The termination of this loop depends on x becoming greater than 100. But nowhere in the code is x directlyincremented. However, x is indirectly incremented through the pointer p.

We can use ESBMC to try and check if this program will terminate by inserting the $assert(0)$ after the loop. Of course, this method cannot not work in general because of the halting problem. To be able to prove this, ESBMC needs to be able to reason about pointers, which it can, as a theory of pointers is built into most popular SMT solvers.

For further detail on the usage of ESBMC consult the online documentation or use the help flag on the command line.

3 Original Work

3.1 Software Security in IoT devices.

Software security in internet of things devices is an important issue since these devices can be collecting sensitive data and running important operations such as monitoring a pacemaker. If bug is present in these devices or a hacker can exploit the device, the consequences can be devastating. Internet of things devices are connected wirelessly; this makes internet of things devices a target to hackers because they don't physically need to device to hack it. Furthermore, many internet of things hacks will not target the device itself but use the device as an entry point to a wider network.

Since internet of things devices are small and usually use cheap hardware, their computing power is limited. This means that it may be impossible to run sophisticated software such as a firewall to help prevent unwanted traffic and other kinds of intrusion detection software. Therefore, there is more of a reliance on the software running on the device to be secure. Furthermore, because many Internet of things devices are some form of embedded system it is likely that it will be running C code. Because of this, common problems such as NULL pointer deference's, array out of bounds errors and double-free could exist in the source code. Therefore, we can leverage existing bounded model checking tools to find bugs and exploits that may be present in the software such as ESBMC.

In addition to typical programming vulnerabilities, IoT devices are often run indefinitely, meaning that the source code will contain loops which will be iterated a very large number

of times, and this could lead to large variables, overflows and memory leaks. ESBMC has support deal with these cases, where it can unwind loops, check for the possibility of overflows and memory leaks if the correct flags are set.

There have been many recent examples of where internet of things devices has been attacked by hackers. One example is when a Bluetooth exploit was found in a Tesla Model X which allows a hacker to unlock the door without the original FOB key. In 2018, a ransomware virus took over many UK's National Health Services computers and other electronic devices. This prevented many of devices being used unless a bitcoin payment was made to the hackers which would automatically decrypt the systems. The total cost of dealing with this hack to the NHS was calculated to be £92 million. If hospitals become more reliant on internet of things devices attacks such as this could be much more devastating. Additionally, there have been many cases of hackers gaining unauthorised access to surveillance networks. Similar attacks could be carried out on home surveillance networks, which may be more intrusive. As highlighted by these attacks, when addressing the security concerns of an internet of things device, it is particularly important to verify that the modules responsible for wireless communication are exploit and bug free.

Contiki-ng[7] is a modern operating system designed to be run on lower power devices with small amounts of memory such as internet of things devices. Contiki-ng is a next-generation version of an older version just Contiki. Contiki has seen applications in smart city devices such as streetlamps and alarms. Both versions of Contiki are open-source and released under the BSD-3-clause license. The use of the Contiki operating system is that it implements many network communication stacks that can be used out of the box. Since many devices will be running using this network stack it is important to verify that it does not have any active exploits. We will check whether this is the case using our extension of ESBMC. There have been recent applications of FRAMA-C to verify some of the important

[7] https://www.contiki-ng.org/

modules of Contiki. This includes the memory allocation module [20], the AES-CCM encryption module [23].

Proving that such software is bug-free remains a challenge this is because the code will include many control structures such as loops that need to be unwound and there will be lots of data received by the device whose contents will not be known in advance. Such things will be modelled as non-deterministic variables. Furthermore, the program will be run over long periods of time, so it may be necessary to unwind the loops of a program a huge number of times. Because of this the verification conditions generated by ESBMC will likely be large and contain many free-variables and this can be difficult for an SMT solver to address this. To tackle this problem, we propose an approach to perform domain splitting on the free variables which appear in the verification conditions.

3.2 Domain Partitioning

To scale-up Bounded Model Checking so it can be used on IoT software more effectively we propose an approach based on domain partitioning. Give a verification condition $\phi(\mathrm{x})$ where x is a free variable, we construct "smaller" verification conditions $\phi_1(x), \ldots, \phi_n(x)$ such that

$$\vDash \phi(x) \iff \phi_1(x) \vee \ldots \vee \phi_n(x)$$

for all x. Each "smaller" verification condition will be formed by making an additional assumption $\psi_i(x)$ that restricts the domain of x. Therefore, for each $1 \leq i \leq n$, we define

$$\phi_i(x) \coloneqq \psi_i(x) \vee \phi(x)$$

We must require that the additional assumptions satisfy $\vDash \psi_1(x) \vee \ldots \vee \psi_n(x)$ for the intersection of the smaller equations to be equisatisfiable to the original verification condition.

Consider an example where we have a verification condition $\phi(x, y, z)$ where x is an integer. We can partition the domain of x into three by consider the following assumptions

$$\psi_1(x) \coloneqq x \leq -1000$$

$$\psi_2(x) \coloneqq -1000 < x \leq 0$$

$$\psi_3(x) \coloneqq 0 < x$$

It is clear that one of these three equations must be satisfiable, i.e. $\vDash \psi_1(x) \vee \psi_2(x) \vee \psi_3(x)$. Next, we construct the three smaller verification conditions:

$$\phi_1(x) \coloneqq (x \leq -1000) \wedge \phi(x, y, z)$$

$$\phi_2(x) \coloneqq (-1000 < x \leq 0) \wedge \phi(x, y, z)$$

$$\phi_3(x) \coloneqq (0 < x) \wedge \phi(x, y, z)$$

Since the disjunction of these three conditions are equisatisfiable to the original verification condition.

Domain splitting can benefit model checking in many ways. First of all, it can speed up the verification process as the smaller equations can be easier to check than the original. Additionally, less computational resources such as random-access memory will be required, since each equation will have a smaller search space. Multi-threading can also be used to check the equations in parallel to speed up the verification process.

There are two potentially important factors that should be considered when deciding how to partition the domain of a verification condition.

1. Which variable(s) should we partition the domain of?
2. How should we partition the domain of said variable? – How many times should we partition.

The strategy to partition the domain of a verification condition needs to be chosen carefully. If we choose poorly, it is possible that the smaller verification conditions will be just as hard to verify as the original verification condition.

To address the first point, one approach is to count the number of times each variable appears in the formula. Domain partitioning can then be applied to the most popular variables. For the second point, a natural and simple approach is to divide the domain into

disjoint intervals of equal size. I have implemented these approaches in my extension to ESBMC. More advanced approaches could instead weight the occurrences of variables differently for appearing in different types of sub-formulas. Additionally, the domain of variables do not need to be partitioned by using only constants. Instead, two variables can have their domains partition relatively to each other. For examples, we could consider the cases when $x < y$ and $x \geq y$. In the context of bounded model checking, these domain partition heuristics should keep in mind the formulas that are likely to occur in the logical representation of a program.

Although choosing good heuristics for domain partitioning could lead to much better performance for bounded model checking, however, there is always a natural limitation to how well the approach can scale since SAT and SMT solving is NP-HARD.

3.3 Implementation

The domain partitioning approach mentioned in the previous section has been implemented as an extension to ESBMC. In summary, this does two things:

1. The variable with the most applications is selected.
2. The domain of this variable is partitioned by a number of times specified by the user (by default this is 2).

This extension has been created by extending the existing C++ source code of ESBMC. The extension supports two additional command line options:

- --domain-partition-smt – This will perform domain splitting on the most popular variable.
- --domain-partition-depth <depth> – This flag specifies the number of intervals the domain should be partitioned into. By default, this is two.

This domain partitioning extension only modifies the bounded model checking component of the program and so can work seamlessly with k-induction and incremental-bmc. The extension also supports both floating-point and fixed-point bit-vector representations of

floating points numbers. It is expected that most of the existing features of ESBMC can be used with the extension while using the domain partitioning feature. Although, it has not been tested whether this can extension works with the C++ and Solidarity features of ESBMC. The extension has been tested on many different programs from the SV-Benchmarks so we can be sure of the correctness of the implementation. The extension has also been successfully applied to verifying C modules from Contiki-mg, an operating system for internet of things devices written in C. These last two points will be explored in the subsequent sections.

The domain partitioning features are implemented after the single static assignment form of the code has been generated. This is useful for multiple reasons; firstly, symbolic execution has been performed and all loops have been unrolled therefore it is possible to count the number of occurrences of each variable in the final verification condition. Secondly, since the next step is to convert everything into logical formulas, the extra conditions can be constructed here and easily sent to the SMT solver. Finally, we need to create multiple SMT each corresponding to the original verification condition but with a restricted domain, this can be done using the SSA form.

The actual code modifications and extensions are included in two main sources files

1. src/esbmc/bmc.cpp
2. src/esbmc/bmc_domain_split.cpp

The first source file is included in the original ESBMC source code. This file is responsible for performing symbolic execution, generating the SSA and then converting this to the relevant SMT format and checking its satisfiability. The second file is a new file included in my extension that helps domain partitioning. This file define a class called BMC_Domain_Split. In summary, this class can be used to find the most used variable in the SSA code and then can generate the logical expressions which are used to partition the domain of this variable a given number of times. We will break down the key components of this class now.

Firstly, the BMC_Domain_Split class has the following constructor:

```
BMC_Domain_Split(
  const messaget &msg,
  symex_target_equationt::SSA_stepst &steps)
  : msg(msg), steps(steps)
{
}
```

This class has two parameters, the first being a *messaget* object which is used for logging output. The second is an object representing the single static assignment of the program. Both of these objects are defined in the base ESBMC project. These parameters are stored in the class as private fields.

The class defines six methods all facilitate of domain partitioning. These functions following declarations.

```
void bmc_get_free_variables();
void bmc_free_var_occurances();
std::pair<expr2tc, int> bmc_most_used() const;
std::vector<expr2tc>
bmc_split_domain_exprs(const expr2tc &var, unsigned int depth) const;
void print_free_vars() const;
```

In addition, this class has two public fields:

```
std::vector<int> free_var_counts;
std::vector<expr2tc> free_vars;
```

These fields will hold the result of various function calls. The `free_vars` vector will contain all the expressions that represent free variables. The `expr2tc` class is defined in the original ESMBC code and is the internal representation of an expression from the C code. The `free_var_counts` variable will count the number of times each free variable is used in the code. Further information can be found in the source code where these functions are annotated using oxygen style documentation

Although, the purpose of these functions should be clear from the names, we explain two of the important ones in more details. Firstly, the `bmc_get_free_variables` function retrieves the free variables from the SSA code. These are the variables that are assigned to non-deterministic values. The result of this function is stored in the `free_vars` field. This is done by iterating through the SSA steps and checking if there is an assignment to a non-

deterministic variable. Next function of interest is `bmc_split_domain_exprs`. This function takes two parameters, the first being the variable that will have its domain partitioned and the second is number of pieces the domain should be split into. This function returns a vector of expressions, that can be inserted into the generated verification conditions to split the domain.

We will now see how this class is used in bmc.cpp file to facilitate domain partition. The modified bmc.cpp file now uses the BMC_Domain_Split class to find the most used variable and creates the domain partitioning expressions exactly after generating SSA. It then creates copies of the original verification conditions each with a restricted domain. All these restricted verification conditions will then be checked by an SMT solver.

In the `run_thread` method in bmc.cpp, I have added the following snippet of code.

```
if(options.get_bool_option("domain-partition-smt"))
{
  int depth = std::stoi(options.get_option("domain-partition-depth"));

  BMC_Domain_Split bmcds(msg, eq->SSA_steps);

  bmcds.bmc_get_free_variables();
  bmcds.bmc_free_var_occurances();

  std::pair<expr2tc, int> v = bmcds.bmc_most_used();

  expr2tc var = v.first;

  //generate the exprs to perform domain splitting on
  //dsc is an member of the bmc class
  dsc = bmcds.bmc_split_domain_exprs(var, depth);
}
```

This code is placed after the symbolic execution has occurred and the single static assignment steps have been generated. The code begins by checking if the `--domain-partition-smt` flag has been set. If this is the case, we proceed with generating the domain splitting formulae. This begins by finding the depth the user wants to partition the domain, if this is unspecified it defaults to 2. Next the `BMC_Domain_split` class is initialised using the already generated SSA code. Afterwards, we use the object to find the free variables and count the number of occurrences of each of them. Next, the most

variable is retrieved and used with the `bmc_split_domain_exprs` function to generate expression that will later be used to split the domain. The output of this function is stored in a newly added field `dsc` in the `BMC` class.

The next stage is to generate verification conditions from the SSA code and the domain splitting assumptions. This is done in the new `run_decesion_procedure_dp` function, which is a variant of the original `run_decesion_procedure` function. The `run_decesion_procedure_dp` function differs from the original since it will create multiple equations to verify, which is one per domain splitting expression that are stored in the `dsc` field. Each equation of these equations will have a restricted domain and will sent to an SMT solver and will checked if its satisfiable.

The program then uses the following decision procedure to decide correctness: The program will be correct if all the expression are unsatisfiable. The program will produce a counter example if one of the verification conditions is satisfiable.

In the extension, there is also some additional code that has not been used in the domain partitioning feature. This can be found in the src/goto-programs/goto_domain_split.cpp files. The purpose of this code was also to find the non-deterministic variables and partition the domain by inserting additional assumptions. However, this is done at the goto-program level. The difficulty with inserting the assumptions here is that multiple copies of the goto-program would need to be created and each with a unique domain partition assumption be inserted. Then each goto-program would need to be symbolically executed. After implementing this module, it was realised that it would be more efficient to do this after the SSA generation.

The extension to ESBMC can be built by following the same instructions on the ESBMC GitHub repository.

3.4 Evaluating the Extension

In this section we will verify that our extension is implemented correctly by testing it on some the test programs form the SV-benchmarks GitLab repository. We will also compare the speed of the original ESBMC and ESBMC with the domain partitioning extension, known as ESBMC-DP.

Firstly we compare the correctness of the programs on 25 different tests from the loops[8] folder. The results are as follows:

	ESBMC	**ESBMC-DP**
Correct True	16	16
Correct False	7	7
Incorrect True	0	0
Incorrect False	0	0
Total Correct	25	25
Total False	0	0

For both versions, the k-induction rule was used. ESBMC-DP was set to divide the domain 8 times. Unsurprisingly, for these tasks, both versions achieve the same perfect result. This result suggests to us that the implementation of ESBMC-DP is correct.

We now will compare the speed of ESBMC-DP and ESBMC on a variety of problems from SV-Benchmarks. For ESBMC-DP I have set its domain partition depth to 4.

[8] https://github.com/sosy-lab/sv-benchmarks/tree/master/c/loops

	ESBMC	ESBMC-DP
sum01-1.c	0.51s	0.62s
sum01-2.c	1.07s	1.09s
matrix-1.c	0.10s	0.07s
sum-array-2.c	1.02s	1.05s

From these results, we can observe that ESBMC with domain partitioning may not improve the time it takes to verify a program over the original ESBMC. It should be noted that ESBMC-DP could be parallelised, where the multiple verification conditions can be checked in parallel. This could lead to performance gains overall. Furthermore, these results are not telling of their general performance, because these problems appear to be easy to solve.

A more refined approach to domain partition could lead to performance gains. For example, the domain splitting conditions could somehow consider the relationships between different variables.

3.5 Verifying IoT Applications

In this section, we will apply our extension to verify a some module from the Contiki-ng[9] operating system.

The module we will verify is `aes.c`. This is responsible for implementing the Advanced Encryption Standard (AES) algorithm, which is a modern form of encryption that is commonly used for communication. It is important that this module is verified to be secure as it will be responsible for any kind of encryption that is used by Contiki-ng. This module has been previously verified using FRAMA - C [23]. The `aes.c` module implements three functions:

```
static uint8_t galois_mul2(uint8_t value);
static void set_key(const uint8_t *key);
```

[9] https://github.com/contiki-ng/contiki-ng/tree/develop/os

```
static void encrypt(uint8_t *state);
```

The `galois_mul2` function computes the product between a number value and 2 in the Galois Field $GF(2^8)$. The `set_key` functions creates an 11x16 array based on this key, which will be used to encrypt data. The `encrypt` function encrypts the state parameter using the generated array from the `set_key` function. We can verify that each function one is bug-free using ESBMC and its built-in features.

The first function is incredibly simple with only two lines of source code. This a small mathematical procedure, which is unlikely to causes any problems if it is implemented correctly.

```
static uint8_t galois_mul2(uint8_t value) {
  uint8_t xor_val = (value >> 7) * 0x1b;
  return ((value << 1) ^ xor_val);
}
```

We can work out if this mathematical function is implemented correctly by comparing it to another of the same function. For example, consider another well-known implementation of this function.

```
uint8_t gmul(uint8_t a, uint8_t b) {
  uint8_t p = 0; /* the product of the multiplication */
  while (a && b) {
    if (b & 1) /* if b is odd, then add the corresponding a to p (final
product
                  = sum of all a's corresponding to odd b's) */
      p ^= a;  /* since we're in GF(2^m), addition is an XOR */

    if (a & 0x80) /* GF modulo: if a >= 128, then it will overflow when
shifted
                     left, so reduce */
      a = (a << 1) ^ 0x11b; /* XOR with the primitive polynomial x^8 + x^4
+ x^3
                               + x + 1 (0b1_0001_1011) - you can change it
but
                               it must be irreducible */
    else
      a <<= 1; /* equivalent to a*2 */
    b >>= 1;   /* equivalent to b // 2 */
  }
  return p;
}
```

Using ESMBC we can compare these two functions and check that they are equal. This is the contents of the following function.

```
void gmul_equiv() {
  uint8_t x = nondet_char();
  uint8_t y = galois_mul2(x);
  uint8_t z = gmul(2, x);

  assert(y == z);
}
```

We can send this function to ESBMC for verification. (Using the `--function` flag, we can choose which function we want ESBMC to check). ESBMC proves that these two functions are equal. Because of this result we can be confident in the implementation of `galois_mul2`.

We can verify the functions `set_key` and `encrypt` are bug-free using similar approaches. This is because they both make use of the same arrays and iterate over them with for loops. Below we provide a snippet from the `set_key` function.

```
  for (i = 1; i <= 10; i++) {
   round_keys[i][0] =
       sbox[round_keys[i - 1][13]] ^ round_keys[i - 1][0] ^ rcon;
   round_keys[i][1] = sbox[round_keys[i - 1][14]] ^ round_keys[i - 1][1];
   round_keys[i][2] = sbox[round_keys[i - 1][15]] ^ round_keys[i - 1][2];
   round_keys[i][3] = sbox[round_keys[i - 1][12]] ^ round_keys[i - 1][3];
   for (j = 4; j < AES_128_BLOCK_SIZE; j++) {
     round_keys[i][j] = round_keys[i - 1][j] ^ round_keys[i][j - 4];
   }
   rcon = galois_mul2(rcon);
  }
```

Since ESBMC checks for array-out-of bounds errors by default, all we need to do is pass these functions to ESBMC. ESBMC verifies that neither function will produce an out-of-bounds error.

Here, we have shown how we verified key properties of the `aes` module in the Contiki-ng operation system. This s surprisingly different to the approach required when using FRAMA-C, as the program needs to manually annotated to check for out-of-bounds errors, where as

ESBMC can insert these automatically. It should also be noted, that as expected from the results in the previous section, the naïve domain partitioning approach I implemented did not give a significant performance gain when verifying this module.

4 Conclusion and Summary

This dissertation explored both bounded model checking and its applications to internet of things devise. It began reviewing the historic, current, and future uses of internet of things devices. Afterwards it gave an in-depth guide through all the prerequisites needed to under how a bounded model checking tool operates. It then went on to discuss the internals of bounded model checking and, how ESBMC works internally.

The main challenge of this dissertation was trying to understand existing code for ESBMC and the building and extension on it. This took a considerable amount of time, but it really consolidated the theoretical knowledge explored in the earlier sections.

After developing this extension, an explanation of how it works was given in chapter 4 and it was also evaluated on popular tests for bounded model checking tools. It was finally then used to verify a module from a program designed for an internet of things device.

The four goals set out in the introduction were addressed in the following ways:

1. Imply an approach to select variables to be partitioned for a given domain – The implemented approach selected the most applied variable from the code.
2. Choose a good partitioning strategy to precisely represent different cases in the model. The implemented partitioning strategy was simple but can be easily extended now some basic source code is in place.
3. Evaluate the proposed approach over a large set of open-source applications related to the IoT domain. – The extension was used to verify a module from the Contiki-ng operating system for Internet of things devices.
4. Evaluate the effectiveness and accuracy of ESBMC with the extension. – This was done by testing the extension on multiple examples from the SV-benchmarks GitHub repository.

To summarise, In this dissertation I have implemented a working and usable extension to facilitate domain partitioning in ESBMC. Although the performance metrics of the extension

do not improve significantly over those of the original ESBMC, the new code developed for the extension serves as foundation for further exploration into using domain partitioning to improve bounded model checking.

In the future, the implemented technique of domain splitting can be extended in many directions. Some ideas include performing domain partitioning on more than one variable at once, using more complicated constraints to divide up the domain of the variables, and using domain partitioning in conjunction with parallelism.

5 Bibliography

[1] L. Cordeiro, B. Fischer and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering,* vol. 38, no. 4, pp. 957-974, 2011.

[2] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *International Conference on Computer Aided Verification*, 2018.

[3] M. Y. R. Gadelha, H. I. Ismail and L. C. Cordeiro, "Handling loops in bounded model checking of C programs via k-induction," *International Journal on Software Tools for Technology Transfer,* vol. 19, p. 97–114, 2017.

[4] J. Teicher, *The little-known story of the first IoT device,* 2018.

[5] D. Bandyopadhyay and J. Sen, "Internet of things: Applications and challenges in technology and standardization," *Wireless personal communications,* vol. 58, p. 49–69, 2011.

[6] J.-y. Wang, Y. Cao, G.-p. Yu and M.-z. Yuan, "Research on application of IOT in domestic waste treatment and disposal," in *Proceeding of the 11th World Congress on Intelligent Control and Automation*, 2014.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L.-J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and computation,* vol. 98, p. 142–170, 1992.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*, 2000.

[9] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of reasoning*, Springer, 1983, p. 466–483.

[10] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[11] M. Davis, G. Logemann and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM,* vol. 5, p. 394–397, 1962.

[12] R. Nieuwenhuis, A. Oliveras and C. Tinelli, "Abstract DPLL and abstract DPLL modulo theories," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2005.

[13] M. R. Gadelha, F. Monteiro, L. Cordeiro and D. Nicole, "ESBMC v6. 0: verifying C programs using k-induction and invariant inference," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

[14] T. Wahl, "The k-induction principle," *Northeastern University, College of Computer and Information Science,* p. 1–2, 2013.

[15] L. d. Moura, H. Rueß and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *International Conference on Computer Aided Verification*, 2003.

[16] M. R. Gadelha, F. R. Monteiro, J. Morse, L. Cordeiro, B. Fischer and D. A. Nicole, "ESBMC 5.0: an industrial-strength C model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[17] L. Chaves, I. V. Bessa, H. Ismail, A. B. dos Santos Frutuoso, L. Cordeiro and E. B. de Lima Filho, "DSVerifier-aided verification applied to attitude control software in unmanned aerial vehicles," *IEEE Transactions on Reliability,* vol. 67, p. 1420–1441, 2018.

[18] F. R. Monteiro, M. R. Gadelha and L. C. Cordeiro, "Model checking C++ programs," *Software Testing, Verification and Reliability,* vol. 32, p. e1793, 2022.

[19] K. Song, N. Matulevicius, L. C. Cordeiro and others, "ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts," *arXiv preprint arXiv:2111.13117,* 2021.

[20] F. Mangano, S. Duquennoy and N. Kosmatov, "Formal verification of a memory allocation module of Contiki with Frama-C: a case study," in *International Conference on Risks and Security of Internet and Systems*, 2016.

[21] A. Blanchard, N. Kosmatov and F. Loulergue, "A Lesson on Verification of IoT Software with Frama-C," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018.

[22] A. Blanchard, N. Kosmatov and F. Loulergue, "Ghosts for lists: a critical module of Contiki verified in Frama-C," in *NASA Formal Methods Symposium*, 2018.

[23] A. Peyrard, N. Kosmatov, S. Duquennoy and S. Raza, "Towards formal verification of Contiki: Analysis of the AES–CCM* modules with Frama-C," in *RED-IOT 2018-Workshop on Recent advances in secure management of data and resources in the IoT*, 2018.

[24] S. Duprat, D. Favre-Félix and J. Souyris, "Formal verification workbench for Airbus avionics software," in *Conference ERTS'06*, 2006.