Contents

1 Background and Related Works **1**

1.1 Overview of Software Verification 1

1.1.1 Software Verification Techniques 2

1.1.1.1 Static Analysis 2

1.1.1.2 Human-directed Analysis 3

1.1.1.3 Dynamic 4

1.1.2 The Use of Software Verification 4

1.1.2.1 Hunting bugs 5

1.1.2.2 Proving correctness 5

1.1.2.3 Other applications 6

1.2 Software Model Checking 6

1.3 Bounded Model Checking 7

1.4 C Bounded Model Checker (CBMC) 10

1.4.1 Ready for verifying a program 10

1.4.2 Assumption & Assertion 12

1.4.3 Non-Determinism 12

1.4.4 Automatic Verification 12

1.5 Overview of s2n 12

1.6 Other approaches using BMC 13

Bibliography **17**

List of Figures

1.1 The Triangle Model of Software Verification adapted from the paper by M. Brain and D. Kroening [6]. 2
1.2 The Model Checking Structure adapted from E. Clarke [11] 7
1.3 The idea behind Bound Model Checking adapted from H. Rocha et al. [38] 8
1.4 The LOF caption 11
1.5 The LOF caption 11

Chapter 1

Background and Related Works

Clarification The terms of soundness and completeness of a software verification technique often have various meanings in different papers. Sometimes soundness and completeness refer to an under-approximation and an over-approximation to a program behaviour respectively while sometimes they are defined in the opposite way. In order to be more clear, we will avoid using them throughout the report instead of using "no missing bugs" and "no false alarms" of a detection.

1.1 Overview of Software Verification

Software verification is an important step in the software development life cycle, which is easily confused with software validation. It is questioning about "*are we building the right product?*", while software verification cares about "*are we building the product right?*" [30]. In other word, we concern about **whether a system meet its specification**, including safety properties, liveness properties, concurrency properties, as well as functional and non-functional properties. This is the key to ensuring quality and reliability of the software, especially for those *critical software*, such as safety-critical, security-critical, financial-critical, etc. By analysing the system behaviour and detecting errors in order to minimise or prevent system failure.

There are numerous verification techniques, based on formal or informal methods, available into to achieve the ultimate goal of software verification. These techniques can be divided into three categories according to their approach: *Static Analysis*, *Dynamic Analysis*, and *Human-directed Analysis*. Although each of them has their own focuses, strengths and weaknesses, their verification achievements can generally be concluded in three main properties: being automatic, not missing bugs, and excluding false alarms by M. Brain and D. Kroening [6]. These can also be visualised in a triangle model, which has been illustrated in Figure 1.1 below. Since there always

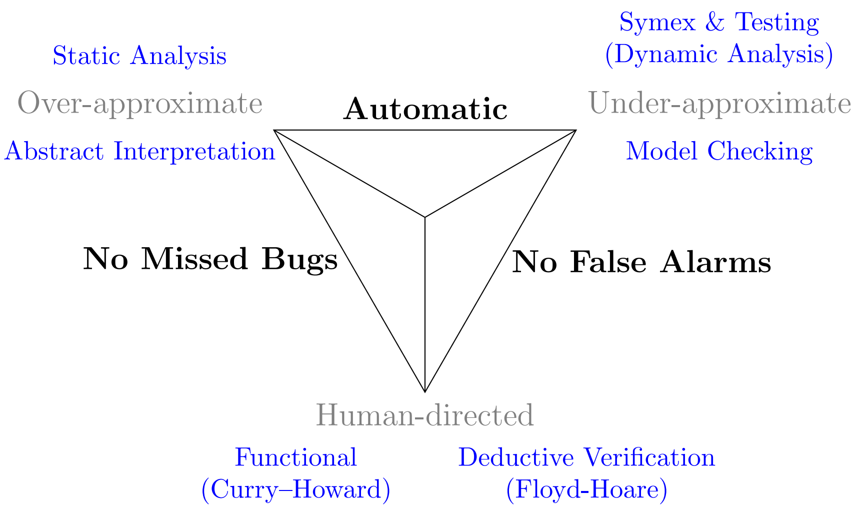

Figure 1.1: The Triangle Model of Software Verification adapted from the paper by M. Brain and D. Kroening [6].

is a trade-off between the three proprieties, it becomes a weakness of such techniques. They can be considered as sitting at a corner of the triangle and climbing to the top of a pyramid while they are trying to tackle their own weakness. We will discuss each of them with some highlighted techniques as examples and the use of software verification in the following subsections.

1.1.1 Software Verification Techniques

1.1.1.1 Static Analysis

Static analysis generally refers to a family of techniques determining the run-time properties of a program without execution [17]. With an approximation of the fixpoint collecting semantics, it can indicate run-time errors automatically at compilation time without any code instrumentation or user interaction [16,45]. However, the capability of static analysers can be various depended on techniques implemented. Sometimes predefined facts or patterns are often used for efficiently showing the absence of simple errors, which can be found in compiler optimisation, such as Clang[1], GCC[2]. Besides that, there are two well-known static analysis techniques, *Abstract Interpretation* and *Model Checking*, that are widely researched and capable to analyse large size software systems with minimal manual effort.

[1]A C language family frontend for LLVM: http://clang.llvm.org/
[2]The GNU Compiler Collection: https://gcc.gnu.org/

2

Model Checking a formal technique to examine each reachable states whether a correctness property holds without execution and generate a counterexample once a violation exists, which will be discussed in detailed in Section 1.2.

Abstract Interpretation is a theory of semantics approximation to predict the program behaviour by using an abstract domain instead of the actual domain of the execution [17]. It guarantees a full coverage of the analysis by over-approximating the possible execution paths, therefore no bugs will be missed [15]. However, the accuracy of the analysis relies on the approximation, which becomes a limitation to static analysis. Due to undecidability of static analysis, it is not possible to come up with an approximation that produces no false positive and no false negative detections, which means no false alarms [17]. Moreover, it is difficult to generate counterexamples as the precision loss during the analysis which is a significant trade-off for efficiency.

Despite of the weakness above, the performance and the scalability of abstract interpretation are beneficial to the industry as some abstract interpretation-based analysers, such as **ASTRÉE**, have been successfully applied on the verification of flight and rocket control, safety-critical, software. Moreover, a study conducted by P. Cousot et al. [16] further uses ASTRÉE as an example to claims that such a difficulty of abstract interpretation has been overcome by domain-specific static analysers.

1.1.1.2 Human-directed Analysis

Human-directed analysis, as its name suggested, refers to techniques relied human assistance to determine the software correctness by using mathematical proof. **Deductive Verification** is an example of such technique to verify the correctness of a program by expressing them into a set of mathematical statements, such that the validity can then be proven by deduction [18, 22]. In contrast to static analysis and model checking, it can be the most precise way to model the program behaviour without abstracting any data structures and instructions as expressed in the triangle model in Figure 1.1. However, the price of such precision is relatively high as it requires huge amount of human efforts to construct the inductive arguments as well as computing resources to achieve the verification. This technique is intended to trade off automation for precision by heavily relying on human interaction during the theorem proving process. Hence, it is capable to verify the functional correctness, but, hardly to be scaled up for large size system [15].

1.1.1.3 Dynamic

Dynamic analysis refers to techniques that test and evaluate a program by running it. It can derives the correctness properties that hold for a certain number of executions, but not for all executions, which is in contrast to static analysis mentioned above [3]. Although it is not able to prove a particular property holds, violations of properties can still be detected as well as provide some useful information about the program behaviour. This technique is easier and cheaper to carry out comparing with other formal verification techniques as it generally requires less knowledge than formal method does. Therefore, various testing techniques exists and commonly be used in the software industry.

Concolic testing is a hybrid software verification technique joining symbolic static analysis and concrete dynamic analysis, which is able to generate test cases automatically and to examine execution paths exhaustively. M. Kim et al. [27] conducted an experiment on concolic testing to analyse the multi-sector read operation for a flash memory, and summarised its advantages and weaknesses compared to model checking techniques. Concolic testing algorithm in general consists of five steps: 1) Instrumentation, 2) Concrete execution, 3) Symbolic execution, 4) Deciding the next execution path, and 5) Choosing the next input values. As the symbolic execution preforms along the concrete execution path, it can prevent any false alarms be created. Once the path formulas are not able to be solved by a constraint solver, some symbolic constraints will be simplified by replacing some of the symbolic values with concrete value. This may leads to an incomplete coverage. The experimental result showed it had a better applicability and lower memory usage. However, the study also reflected that concolic testing is a time-consuming method as time will be wasted on generating invalid test cases in a complex environment model. Concolic testing can be a good choice for finding bugs, but not for demonstrating program correctness. Comparatively, model checking generally provides high accuracy and a better performance of the verification than concolic testing, which is essential for verifying critical software.

1.1.2 The Use of Software Verification

Software verification are commonly used for two main purposes: 1) hunting bugs, and 2) proving the program functional correctness. Below are several case studies making good use of the model checking, which is the technique we focused on in this project:

1.1.2.1 Hunting bugs

M. Voelter et al. [44] took several well-know bugs as examples to show how the modular extension of the C programming language can improve software security. One of the examples is the Heartbleed bug in OpenSSL TLS mentioned in Section **??**. By constructing a simple harness and a message with a nondeterministic data buffer, CBMC is able to identify the failure in a short period. This is a great demonstration of the importance of formal verification in the software development.

Another remarkable case study presented by Y. Kim and M. Kim [28] has successfully detected four hidden bugs in an embedded software, lazybox ls, using CBMC. This study highlighted the pivot of loops analysis, determining the minimum iterations to exit a loop, to avoid false negative detections in practice. As an accurate unwinding loop bound can minimise the chances of missing bugs, state explosion, and execution timeout. The experiment also showed the effectiveness of using such model checking technique and its limitation on the loop analysis.

There is a case study conducted by J. Yang et al. [48] demonstrated a systematical way to model check three widely-used and heavily-tested file systems. Two classes of checks are suggested: 1) generic checks for those general properties should hold at any point, and 2) consistency checks for the functional properties that the file system specified. Concerning the complexity of such file systems, it is impractical to be fully tested in the traditional way as there could be an exponential number of test cases for checking the recovery mechanism. Model checking make use of several state-reducing techniques, which makes it capable to explore such vast state spaces efficiently. As a result, 32 serious bugs are found during the experiment.

1.1.2.2 Proving correctness

An experience report presented by R. Metta [34] described their experience in verifying the correctness of a 200-line implementation of Cyclic Redundancy Check. The study indicated that sometime the specification and verification process can hardly be automated and required intense human efforts even for such a small code bases. It is hard to scale up if such a program containing huge loops. The authors further suggested a bottom-up approach to verify the correctness of each functions independently through CBMC first, and then to verify the correctness of the loop through an induction manually using CBMC.

A. Jääskeläi et al. [23] verified the implementation of the 2003 voting scheme in a systematic capability 3 level shutdown system through model checking. The experi-

5

ment is conducted in two stages: 1) refine and translate the requirements into pseudo code directly for a preliminary verification, and then move on to 2) verify the actual implementation of the voting scheme. The study concluded in several suggestions for using model checking techniques. Beginning with some simple test cases can help to understand how to give precise and effective assertions before jumping into those complicated properties. Fault seeding techniques is recommended for inexperienced users in order to check the correctness of assertions by introducing violation to the corresponding properties.

Another study conducted by B. Beckert et al. [4] verified the functional and relational properties of voting rules, and demonstrated the effectiveness of using bounded model checking and deductive verification. Moreover, it also showed that the verification effort can be greatly reduced by using symmetry properties.

F. Werner and D. Farago [47] investigated CBMC to the domain of wireless sensor networks (WSNs) and proved that it is capable to perform automatic verification on a system about 21000 LoC. It showed that the scalability can be greatly improved by the proposed abstractions and simplification heuristics, with the –slice-formula option enabled. The study further concluded that CBMC with the abstractions and heuristics is powerful enough to verify large scale programs.

1.1.2.3 Other applications

There are several studies showing that model checking is applied to various usages, such as embedded software verification [7, 14], cross-platform verification [46], sequentialisation-based verification [19], fault localisation [20], equivalence checking [31], cyber-physical system co-verification [50], and test-vector generation [2].

1.2 Software Model Checking

Model Checking is an algorithmic technique, based on graph theory, automata theory and logic, [43] to establish the correctness of the model of a state system regarding to the given properties, which is formalised from the specification [11, 12]. A model checking problem can be defined as whether all reachable states s of a given model M satisfies the given property φ, which is expressed by a temporal logic formula, such that $M, s \models \varphi$. This concept can be fully adapted to software verification considering that M and φ refer to a model of a program and its specification respectively. While a program consists of a set of *states* describing its memory state at a particular time frame and a set of *transitions* describing how the program evolves from one state to another. Model checking will first compute all the reachable states, then by traversing exhaustively, the model algorithm is able to tell whether the property p holds at all states if there is finite number of states. Once a violation is found on any execution path, an *counterexample* can be generated accordingly. Otherwise, *witnesses* are present to confirm that the given property φ is true. These concepts are all simplified into the Figure 1.2 below.

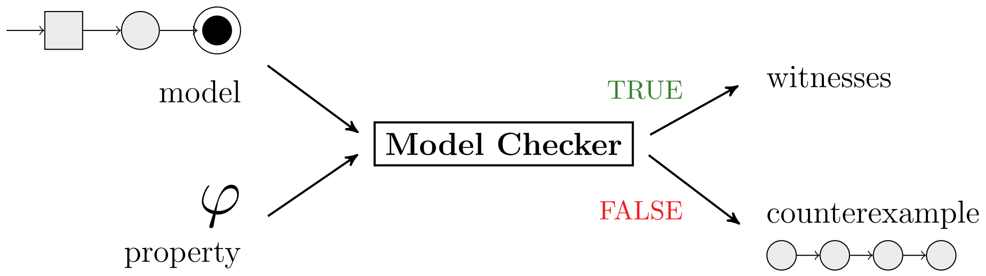

Figure 1.2: The Model Checking Structure adapted from E. Clarke [11]

Model checking is capable to verify both *safety* and *liveness* properties of a software in sightly different manner [17]. Safety properties means nothing bad happens in the program execution, such as arithmetic overflow, buffer overflow, or NULL dereference, which can be verified by proving the unreachability of these states. Liveness properties means something good eventually happens. This can be proved by showing that a program fulfilled its functional specifications or it simply guaranteed termination at the end. A program can be claimed as verified while all the given properties are satisfied by the given model. This technique is precise as all reachable states have been examined, however, the number of states can grows exponentially along with the complexity of the program [5, 8]. The state explosion problem becomes a major weakness of such techniques, while analysing a medium sized code bases [49]. In the history of model checking, several techniques have been applied in order to increase the capacity of the exploration states, for examples, some efficient graph traversal techniques, the symbolic representation, Binary Decision Diagrams (BDDs), and so on. However, the size of BDDs also can grow exponentially, which becomes a bottleneck of verification as the efficiency and capability are restricted, and it is still not enough to carry out a full verification to meet the industrial needs.

1.3 Bounded Model Checking

Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) was introduced by A. Biere et al. in 1999 [5], which is a complementary technique to model checking with an upper bound of the number for exploration state. Although it does not overcome the entire complexity weakness of model checking, it is more capable to verify different circumstances comparing with using the BDD-based techniques. While BMC only cover the execution paths within the user-defined bound, the properties beyond the bound will not be verified. Instead of proving "*all reachable states satisfies the property*, is true directly, the formula can also be checked in its negation by showing that "*there exists a reachable state violates the property*", is false [8]. Hence, the concept of BMC is to check the negation of given property φ up to a given depth k, in order to find a witness for showing such a violation exists [38], shown in the Figure 1.3 below.

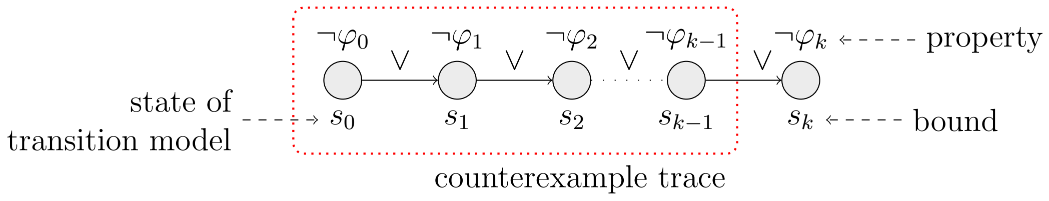

Figure 1.3: The idea behind Bound Model Checking adapted from H. Rocha et al. [38]

In addition, by unrolling the transition operations for k times and then combining those properties to form a propositional formula, such a boolean formula can be reduced as a SAT problem and hence it can be solved by a SAT solver [5, 39]. The formula is satisfiable as long as the property is violated by a trace of length k. The advantage of replacing BDDs with SAT is that the exponential growth of space in BDDs can be avoid, and also smart depth first search can be applied to reduce the memory consumption when using breadth first search in BDDs.

Sometimes there is a chance of missing bugs in the verification, due to the given bound does not cover all the possible execution paths, hence choosing an appropriate bound is fundamental to minimise such a chance. In practise, it is suggested to begin with a smaller length k, and then continue with a larger one if there is no error is found. Nevertheless, the verification result is promising while a counterexample can locate exactly where the violation is and all bugs within the bound are guaranteed to be found. Even when the property is not allow for proving the correctness, BMC is still useful and reliable for finding counterexamples. Because of such improvements and this impressive feature, BMC based on SAT has been successfully used in the formal verification of industrial practises. Moreover, V. D'Silva et al. [17] further concluded that BMC is the best technique to find shallow bugs comping with static analysis and model checking, and is able to provide a complete counterexample trace once a bug is found.

A practical experiment conducted by J. Toman et al. [41] combines exhaustive test generation and bounded model checking technique in order to detect memory safety errors. A bounded verifier, **CRUST**[3], was developed, which relies on **CBMC** for performing automated analysis. It will first generate a set of testing drivers for the relevant APIs and then convert the driver code from Rust to C for bounded model checking. With the help of exhaustive test generation, large input spaces can be explored efficiently and multiple test drivers can be generated for covering the input spaces as many as possible. By evaluating data structures from Rust standard library, CRUST successfully detect some underlying memory safety bugs from it.

In contrast to static analysis, model checking is more precise in proving more complicated properties since an under-approximated model is used in the verification. This is supported by the experimental result conducted by K. Vorobyov and P. Krishnan [45]. The experiment compares the verification result of static program analysis with the one of model checking by applying **Parfait**[4] and CBMC respectively on some benchmarked code bases. By configuring CBMC to focus on particular types of error, it showed a high accuracy rate (97% overall true positive detections, and found about 19% more bugs), whereas only 77% with Parfait. Moreover, both of them reported a 0% false positive detections, which is common for model checking and a specific design to minimise it for static program analysis. However, model checking is much more expensive with respect to computation time and memory usage, as it reached the memory limit in the experiment. The study further concluded that: 1) an insufficient unwinding was one of the reasons for false negative detections of CBMC, and 2) scalability can be a problem for model checking on a large size code base, such as an operating system.

A counterexample-guided abstraction refinement (CEGAR) based technique, which was presented by A. Yeolekar and D. Unadkat [49], utilises dynamic analysis to overcome the scalability limitation of model checking. Dynamic inference is used to guess invariants and refine the abstraction from spurious counterexamples, thus a better precision of the abstraction and an accelerated loop refinement are achieved in their experiment.

[3]CRUST github repository: https://github.com/maidsafe/crust

[4]Parfait github repository: https://github.com/performancecopilot/parfait

1.4 C Bounded Model Checker (CBMC)

CBMC [5], despite its name, is a bound model checking tool for the formal verification of ANSI-C programs using SAT solvers developed by D. Kroening et al. [10]. It aims at reasoning about the safety properties of low level ANSI-C implementation as many safety-critical software are written in such low level languages.

1.4.1 Ready for verifying a program

In order to analysis a given C/C++ code, CBMC will first reduce the model checking problem to a validity of bit vector equation problem by translating the program statement into an static single assignment (SSA) form [9,10]. The translation consists of three main procedures described as follows:

Program translation Suppose the ANSI-C program has been preprocessed already. The program will be translated into a control-flow graph only consisting of `if`, `while`, `goto` statements and simple assignments without any side effects, and then be analysis by using standard compiler technique.

Loop unwinding In order to model the computation up to a given depth, loops are necessary to be unwound into a fixed number of iterations. This can be done by repeating the loop body with certain amounts of copies. While each copy is guarded by an `if` statement with the same condition as the loop, it is prepared for the case that less iterations are required for the loop. In addition, an assertion with the negated condition can be added after the last copy, which can ensure that the loop does not require more iterations by using `--unwinding-assertions` option in CBMC. This is essential for showing that whether the unwinding bound is large enough to model the program behaviour. Note that, after the program translation process above, loops can be constructed by using `while` statements, recursive function calls, and `goto` statements. We use `while` as an example below:

[5]The CBMC Homepage: http://www.cprover.org/cbmc/

10

```
                                        if(cond) {
                                            instr; /** 1st copy **/
                                            if(cond) {
                                                instr; /** 2nd copy **/
                                                    ...
while(cond) {                                   if(cond) {
    instr;          →                               instr; /** kth copy **/
}
                                                    /** unwinding assertion **/
                                                    assert(!cond)];
                                                }
                                            }
                                        }
```

Figure 1.4: Loop unwinding for k times with unwinding assertion adapted from CPROVER tutorials [7]

Variable Renaming After the previous operations, the program, containing `if` statements, assignments, assertions, `goto` instructions and labels only, is ready to be transformed into SSA form by using pointer analysis. The transformation is demostrated by the simple example below:

```
                    x1=x0+y0;
x=x+y;              if(x1!=1)
if(x!=1)                x2=2;
    x=2;            else
else          →         x3=x1+1;       →
    x++;
                    x4=(x1!=1)?x2:x3;
assert(x<=3);       assert(x4<=3);
```

$$
\begin{aligned}
C &:= x_1{=}x_0{+}y_0 \wedge \\
&\quad x_2{=}2 \wedge \\
&\quad x_3{=}x_1{+}1 \wedge \\
&\quad x_4{=}(x_1{!=}1)?x_2{:}x_3 \\
P &:= x_4 \leq 3
\end{aligned}
$$

Figure 1.5: SSA form transformation adapted from [9, 10]

The procedure above renames all variables with states to ensure that each variable is fresh and will only be assigned once. As a result, the program can be viewed as a set of constraints and two bit-vector equations, constraints C and properties P, can be produced accordingly. Hence, the property can now be verified by converting $C \wedge \neg P$ into Conjunctive Normal Form (CNF) through a SAT solver. Once the equation is *satisfiable* which means a violation of the given property is found, otherwise, the property holds when the equation is *unsatisfiable*.

[7]The CPROVER manual: http://www.cprover.org/cprover-manual/

1.4.2 Assumption & Assertion

As mentioned above, CBMC considers the verification conditions as a pair of bit-vector equations, constraints C and properties P, which is specified by using *assumption* (`__CPROVER_assume()`) and *assertion* (`__CPROVER_assert()`) statement respectively. The `__CPROVER_assert()` statement takes a Boolean condition, which is an ANSI-C logic expression, and a string description as arguments. CBMC will check the given condition holds for all executions. The description is useful for locating the violated assertion. The `__CPROVER_assuem()` statement will also takes a Boolean expression, which is used to restrict the program traces exploration. The program trace examination will be aborted while the given assumption is false.

1.4.3 Non-Determinism

Non-deterministic choice functions is supported in CBMC by declaring `nondet_` as the prefix of their names and it is usefully for modelling the user inputs. The range of the generated values depends on the return type of the function and also the restriction given by the corresponding assumptions.

1.4.4 Automatic Verification

Concerning to the verification on safety properties, CBMC takes care of a widely range of program behaviour regarding to such properties, including dynamic allocations using `malloc()` and `free()`, buffer overflow, pointer safety, arithmetic overflow, user-defined assertions, etc. Moreover, by enabling CBMC with different options, the verification condition generator of CBMC will automatically generate the safety conditions respectively, which allows the above verifications become fully automated. The options of supported build-in safety check are as follows: `--divided-by-zero-check`, `--bounds-check`, `--pointer-check`, `--memory-leak-check`, and `--signed-overflow-check` as well as `--unsigned-overflow-check`. Because of the advantage of being automated and the verification capability, CBMC has been widely used in a variety of applications, such as error explanation, verifying embedded programs, finding security bugs in windows binaries, etc.

1.5 Overview of s2n

s2n [1, 40] is an open-source implementation of the TLS/SSL protocols that was released in late June 2015 by the Amazon Security Labs. It is designed with the concern of security, reviewability, ease of use and efficiency. It can support various version of the TLS/SSL protocols, such as SSLv3, TLS1.0-1.2, with different cipher suites. The implementation of s2n is only around 6,000 lines of C99 code as it relies on OpenSSL or other its forks for handling the low-level cryptographic computation of the TLS protocol. Comparing with the implementation of OpenSSL which needs about 70,000 lines of code to implement the protocol. This makes s2n easier to be reviewed, indeed, Amazon also announced that three external security evaluations and penetration tests had been done on s2n at that time.

1.6 Other approaches using BMC

The use of bounded model checking techniques have been presented and widely researched over the last couple of decades. In this section, we discuss some experiences of applying model checking on software security audit and assurance. It is important to learn from the successful cases from others in order to observe how the model checking is being used effectively and what kind of contribution to did it make to the software development process and the overall quality.

Abstract testing is a software testing approach proposed by F. Merz et al. [33], which focuses on the relation between requirements and the corresponding test cases in order to replace the traditional test cases by an abstract one. Each abstract test cases is derived form the requirements and formulated into assumption and assertion statements on the source code level, which is often a one to one relation. Assumption and assertion statements are the constraints encoding the preconditions and postconditions respectively. Setting up these constraints for the testing environment with respect to non-deterministic values, rather using concrete values. Hence, an abstract test case is possible to represent a large number of concrete test cases, which can remarkably simplify the test cases generation and maintenance process. The experimental study showed that abstract testing is more capable and efficient to discover underlying bugs than traditional software testing. Moreover, the performance of CBMC is promising with less than 20s runtime on average for an abstract test case, which is suitable for agile-like development process. Once there is a change of requirement, it is easily and directly visible to the corresponding abstract test cases.

Besides the incomplete coverage of conventional testing stated above, M. Kim et al. [24–26] also mentioned that once a violation is detected, it still require a large amount of human effort to locate where the violation occurred by replaying the scenario and tracing the execution step-by-step. Hence, model checking techniques is well-suited to overcoming these weaknesses of conventional testing method through exhaustive analyses. This study applied SAT-based bounded model checking technique to verify the functional correctness and increase the reliability of the device driver software for Samsung's OneNAND^{TM} flash memory. In order to extract the code level properties, the study also suggested a top-down approach to identify the properties from several design documents, such as software requirement specifications, architecture design specifications, and detailed design specifications. It demonstrated a practical way on how the functional correctness properties can be extracted from the real world environment. By replacing the conventional testing with constraint-based exhaustive testing, which is the same as abstract testing above, CBMC achieved a great success in the experiment that discovered some bugs including incomplete exceptions handling and logical bugs. This promising result shows that model checking techniques are mature enough to be applied on verifying an industrial level software.

Property-based code slicing Despite the fact that model checking is a powerful technique, it often requires more knowledge and cost than testing. In order to reduce the cost while maintain comprehensiveness, M. Park et al. [35] proposed an approach that consists of three strategies: 1) property-based environment generation, 2) property-based abstraction, and 3) collaborative verification using model checking and testing. The study applied it to verify the Trampoline operating system. The result showed that the approach is able to reduce the verification cost by scaling down the target code, and to simplify the analysis process by localising the verification activity.

False positive elimination Software verification based on abstract interpretation is scalable for verifying industrial level code bases, but imprecise. Many spurious errors are generated as the abstract interpretation over-approximated the execution traces than the program can actually perform. Manual investigation is needed to review each error, such manual efforts are costly to the development process. To overcome this problem, H. Post et al. [37] and P. Darke et al. [29] integrate model checking to reduce the number of false positive generated by the static analysis tools. The experimental result shows that 69% of warnings have been successfully removed by CBMC.

Another study by T. Muske et al [42] points out the above approach could involve numerous verifications, and further proposes an approach consisting of three techniques to achieve a faster false positives elimination. By avoiding redundant equivalent assertions, results in a 60% accelerated false positives elimination.

Aspect-oriented programming technique M. Ludwich and A. Frohlich [32] introduced an approach to formally verify both *functional correctness* and *safety properties* of embedded operating system components. Such components are supposed to be shifted between different domains, therefore, the implementation of formal verification must also be domain-independent. The study proposed a corresponding strategy that consists of three main steps: 1) creating contracts for each components that to be verified , 2) implementing such components with the corresponding contracts accordingly, and 3) performing software bounded model checking to verify whether the implementation of components respect to their contracts. The benefits of keeping the specification and implementation close to each other is that any violations in the implementation can be detected at the early stage of development. However, this may introduce an extra run-time overhead to its instances. Hence, those contracts should be partially eliminated from the implementations beyond the verification stage. The experiment made used of aspect-oriented programming technique to achieve this purpose, and further facilitate a modular verification by isolation different components. Note that components isolation is not suitable for verifying a monolithic design. More importantly, the experiment also demonstrated that the functional correctness and safety properties can be verified by the contracts captured from requirements and those properties generated by the model checker respectively. This approach is a kind of white box testing and is more suitable to carry out at the early stage or during the development.

Falsification-Driven Verification A. Groce et al. [21] stated the idea of a "verification successful" in model checking or even theorem proving can be a sign of insufficient verification properties. In order to gain more knowledge of the meaning behind "successful" results, the study proposed a falsification-driven methodology that adapts mutation testing to show the weakness of current specification. Mutate both the harness and the program with a small syntactic change assuming a good test suit should be able to detect a bug introduced by such a change. Hence, the mutation kill rate can act as a measurement unit of the accuracy rate and correctness of a harness. As a better harness should able to detect more errors from the implementation. This approach is useful for ensuring the quality of harnesses, thus a more reliable verification. However, it requires many manual efforts to verify and examine the mutants while some of them can be semantically equivalent. It would be suitable for developers who are not familiar with formal verification but intend to verify a system.

Configuration lifting is a specification analysis technique presented by H. Post and C. Sinz [36], which transforms all variants into a meta-program and facilitates the following specification analysis in three different domains: 1) inconsistencies within a feature model, 2) inconsistencies between the feature model and their implementations, and also 3) the coverage of run-time errors in all product variants. The study demonstrated the technique on verifying Linux kernel and device driver, which successfully found two bugs in the kernel configuration system. The experimental result showed that this technique is able to apply on a large size system included more than 4600 features and model checking on the meta-program provides a significant speed-up compared with traditional enumeration based analysis.

Hybrid approach L. Cordeiro et al. [13] propose a semi-formal verification approach combined dynamic and static verification to stress and cover the state spaces of embedded software exhaustively, in order to improve the coverage and reduces the verification time. This approach allows developers to reason both the functional and temporal properties quantitatively to guarantee the timeliness and correctness of the design. The experimental result shows that model checkers have limitation to specify more complex temporal properties and state space explosion problem.

Bibliography

[1] The s2n user manual. `https://github.com/awslabs/s2n/blob/master/docs/USAGE-GUIDE.md`.

[2] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. *Automatic Test Generation for Coverage Analysis Using CBMC*, pages 287–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[3] Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[4] Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich. Automated verification for functional and relational properties of voting rules. In *Sixth International Workshop on Computational Social Choice (COMSOC 2016)*, Toulouse, France, June 2016.

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[6] M. Brain and D. Kroening. The software verification pyramid. Submitted to the Royal Society Philosophical Transactions A.

[7] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Software and Systems*, 84(10):1693–1707, 2011.

[8] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[9] Edmund Clarke and Daniel Kroening. Hardware verification using ansi-c programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM.

[10] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[11] Edmund M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[12] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[13] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 396–403, May 2009.

[14] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 137–148, Washington, DC, USA, 2009. IEEE Computer Society.

[15] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.

[17] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[18] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[19] B. Fischer, O. Inverso, and G. Parlato. Cseq: A concurrency pre-processor for sequential c verification tools. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 710–713, Nov 2013.

[20] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electron. Notes Theor. Comput. Sci.*, 174(4):95–111, May 2007.

[21] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney. How verified is my code? falsification-driven verification (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748, Nov 2015.

[22] Reiner Hähnle. Introduction deductive software verification.

[23] Antti Jääskeläinen, Mika Katara, Shmuel Katz, and Heikki Virtanen. Verification of safety-critical systems: A case study report on using modern model checking tools. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 44–56, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] M. Kim, Y. Kim, and H. Kim. Unit testing of flash memory device driver through a sat-based model checker. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 198–207, Sept 2008.

[25] M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering*, 37(2):146–160, March 2011.

[26] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver — an experience report. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 144–159, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Moonzoo Kim and Yunho Kim. Formal methods: Foundations and applications. chapter Concolic Testing of the Multi-sector Read Operation for Flash Memory File System, pages 251–265. Springer-Verlag, Berlin, Heidelberg, 2009.

[28] Y. Kim and M. Kim. Sat-based bounded software model checking for embedded software: A case study. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 55–62, Dec 2014.

[29] Shrawan Kumar, Bharti Chimdyalwar, and Ulka Shrotri. Precise range analysis on large industry code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 675–678, New York, NY, USA, 2013. ACM.

[30] David Kung and Hong Zhu. Software verification and validation. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[31] Dong-Ah Lee, Junbeom Yoo, and Jang-Soo Lee. *Equivalence Checking between Function Block Diagrams and C Programs Using HW-CBMC*, pages 397–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[32] Mateus Krepsky Ludwich and Antônio Augusto Fröhlich. On the formal verification of component-based embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 47(1):28–34, January 2013.

[33] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. Bridging the gap between test cases and requirements by abstract testing. *Innov. Syst. Softw. Eng.*, 11(4):233–242, December 2015.

[34] Ravindra Metta. Verifying code and its optimizations: An experience report. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 578–583, Washington, DC, USA, 2011. IEEE Computer Society.

[35] Mingyu Park, Taejoon Byun, and Yunja Choi. Property-based code slicing for efficient verification of OSEK/VDX operating systems. In *Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2012, Kyoto, Japan, November 12, 2012.*, pages 69–84, 2012.

[36] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 347–350, Sept 2008.

[37] H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 188–197, Sept 2008.

[38] H. Rocha, H. Ismail, L. Cordeiro, and R. Barreto. Model checking embedded c software using k-induction and invariants. In *2015 Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 90–95, Nov 2015.

[39] M. Sagiv. Lecture notes in automatic software verification: Bounded model checking, April 2015.

[40] S. Schmidt. Introducing s2n, a new open source tls implementation. `https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-NewOpen-Source-TLS-Implementation`.

[41] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for rust (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 75–80, Nov 2015.

[42] Mayur Khanzode Kumar Madhukar Tukaram Muske, Advaita Datar. Efficient elimination of false positives using bounded model checking. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–to. IARIA, 2013, 2013.

[43] Moshe Y. Vardi. *Model Checking for Database Theoreticians*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[44] Markus Voelter, Zaur Molotnikov, and Bernd Kolb. Towards improving software security using language engineering and mbeddr c. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 55–62, New York, NY, USA, 2015. ACM.

[45] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. *Proceedings of SSV*, 2010.

[46] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. *Cross-platform verification framework for embedded systems*, pages 137–148. Lecture Notes in Computer Science. Springer, 9 2007.

[47] Frank Werner and David Faragó. Correctness of sensor network applications by software bounded model checking. In *FMICS*, 2010.

[48] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, November 2006.

[49] Anand Yeolekar and Divyesh Unadkat. *Assertion Checking Using Dynamic Inference*, pages 199–213. Springer International Publishing, Cham, 2013.

[50] Y. Zhang, F. Xie, Y. Dong, X. Zhou, and C. Ma. Cyber/physical co-verification for developing reliable cyber-physical systems. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 539–548, July 2013.