Chapter 1

Background

1.1 Software Model Checking

Model Checking is an algorithmic technique to formal prove the correctness properties of the model of a state system [9, 10]. In other words, a model checking problem is defined as whether the states s of a given model M satisfies a property P specified by a temporal logic formula, such that $M, s \models P$. The concept can be fully adapted to software verification considering that M consists of *states* and *transitions*, and P refer to a program and its specification respectively. By traversing all the reachable states exhaustively, the model checking algorithm is able to tell whether the program meet its specification or not if there is finite number of states. Once a violation is found on any execution path, an *counterexample* can be generated accordingly. Model checking is capable to verify both *safety* and *liveness* properties of a software in different manner. 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 function fulfilled its responsibilities or the program will terminate at the end. This technique is precise by examining the states exhaustively, however, state explosion becomes a real problem. Although the combination of symbolic model checking with Binary Decision Diagrams (BDDs) slightly improved the state space, BDDs can still grow exponentially, which becomes the bottleneck of verification as the efficiency and capability are restricted.

1.2 Bounded Model Checking

In order to overcome the weakness of model checking and able to make use of it, Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) was proposed by A. Biere et al. [6]. The basic idea of BMC is check the negation of a given property with in a given bounded length k [32]. Considering we have a model M of a state machine, a property P, and a constant bound k states, a BMC problem becomes whether a counterexample with at most k state transitions exists satisfies a property P for a given state machine M. This problem can be formulated as a SAT problem and hence be solved by a SAT solver [31]. Hence, even if there is infinite number of states, BMC can still be applied by bounding the number of exploring states. Concerning the coverage of the verification, the bounded can lead to an unsound verification as bugs may exist beyond the bound. Nevertheless, all bugs within the bound are guaranteed to be identified.

1.3 C Bounded Model Checker (CBMC)

CBMC [1] [8] is a bounded model checker developed by D. Kroening et al. at Carnegie Mellon University. It utilizes a SAT solver in order to verify the software. It first translates a program into a static single assignment form without loops and function calls by bounding each loop with a number of executions and unwinding the loop to the bounded. Then, all variables can be renamed with states, ensuring 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 be solved by using a SAT solver. Moreover, CBMC can also support the verification on dynamic allocations using malloc and free, buffer overflow, pointer safety, and also user-defined assertions. It has been widely used in a variety of applications, such as error explanation, verifying embedded programs, finding security bugs in windows binaries, etc.

1.4 Overview of s2n

s2n [2] is a lightweight and open-source implementation of the TLS/SSL protocols developed by the Amazon Security Labs. TLS/SSL protocols are cryptographic protocols providing the communication security over the network. s2n is designed to be simple, small, fast, and also with security as a priority. It is about 6,000 line of code ignoring tests, blank lines, and comments. It supports various version of the TLS/SSL protocols, such as SSLv3, TLS1.0-1.2, and also support different cipher suites.

1.5 Related Work

The use of model checking technique for automated formal verification of software has been presented and widely researched over the last couples of decades. In this section, we discuss some practical applications and experiences of applying model checking techniques on 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 effect it has contributed to the development process and the overall quality.

1.5.1 Formal software verification techniques

There are numerous formal verification techniques available for software verification. Each of them is designed for a particular purpose. We discuss those techniques which are highly researched or presented recently. A survey that conducted by V. D'Silva et al. [13] gave a detailed overview of three main automatic formal verification techniques, and brought out their strengths and weaknesses for applying to practical problems. The three techniques are: 1) Static Analysis, 2) Model Checking, and 3) Bounded Model Checking. We discuss each of them in detail below.

Static analysis generally refers to a family of techniques for determining the run-time properties of a program without executing it. Static analysis is able to indicate run-time errors automatically at compilation time without any code instrumentation or user interaction. Therefore, these techniques are commonly used in compiler optimisation. In term of software verification, the soundness of an analysis relies on the approximation, which becomes a limitation to static analysis. As the undecidability of static analysis problems, it is not possible to come up with an approximation that produces no false positive and no false negative detections. It uses some simple predefined fact for showing the absence of simple errors, such as no assertion violation, no arithmetic overflow, and no exceed of array bounds, efficiently. However, it is difficult to generate counterexamples, due to the precision loss during the analysis which is a significant trade-off for efficiency.

In contrast to static analysis, model checking is more precise as it can prove more complicated properties and provides counterexamples once a bug is found. The above statements are supported by the experimental result conducted by K. Vorobyov and P. Krishnan [35]. The experiment compares the verification result of static program analysis with the one of model checking by applying Parfait and CBMC respectively

on some benchmarked code base. 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) scalability can be a problem for model checking a large size code base, such as an operating system, and 2) an insufficient unwinding was one of the reasons for false negative detections of CBMC.

Model Checking is a technique for verifying the correctness of a finite-state system as mentioned in Section 1.1. It computes the run-time states of the software without actually running the program. Each reachable state is examined whether a correctness property holds. If such an execution path violating any of the properties is found, a counterexample is formed. This process is guaranteed to terminate only if there are finite states. This technique is frequently used on verifying the safety properties of device drivers and systems code. However, it is easy to run into a state space explosion problem while analysing a medium sized code bases [39]. Even with the help of SAT or SMT solver and some abstraction techniques, model checking is hard to scale up to the size level that static analysis does.

Bounded Model Checking is a complementary technique to model checking with an upper bound of the number for explored state as mentioned in Section 1.2. It unrolls the transition operations for k times and combines the properties to form a propositional formula, which is solvable by a SAT solver. The formula is satisfiable as long as the property is violated by a trace of length k. Such technique is sound for the execution paths up to the specified length k, while a bug can exist beyond the bound. Although it may result in an inconclusive outcome, BMC is still a great successful technique for finding bugs, as well as proving the liveness and safety properties by applying in different manner. V. D'Silva et al. [13] further concluded that bounded model checking 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. However, the completeness is not guaranteed if the program contains deep loops.

Dynamic analysis is a technique to analyse the run-time proprieties by executing a program, which is in contrast to static analysis. It is not able to prove a particular property holds, but nevertheless it is useful for detecting violations of properties [4] and able to scale to large-size code bases. A. Yeolekar and D. Unadkat [39] presented a counterexample-guided abstraction refinement (CEGAR) based technique by utilising 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.

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. [21] 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, no false alarms will be produced. 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. Model checking generally provides high accuracy and a better performance of the verification than concolic testing, which is essential for verifying critical software.

1.5.2 Software Verification approaches through MC

Abstract testing is a software testing approach proposed by F. Merz et al. [26], 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. [18–20] 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. [28] 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. [30] and P. Darke et al. [23] 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 [33] 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 [25] 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. [16] 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 [29], 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.

Other approaches L. Cordeiro et al. [11] 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.

1.5.3 Practical software verification

Software verification can generally be categorised into two main purposes: 1) hunting bugs, and 2) proving the program correctness. Below are several case studies which are making good use of the model checking are discussed below:

1.5.3.1 Hunting bugs

M. Voelter et al. [34] 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 [22] 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. [38] 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.

1.5.3.2 Proving correctness

An experience report presented by R. Metta [27] 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. [17] verified the implementation of the 2003 voting scheme in a systematic capability 3 level shutdown system through model checking. The experiment 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. [5] 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 [37] 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.

1.5.4 Model Checking tools

CRUST is a bounded verifier developed by J. Toman et al. [32] combines exhaustive test generation and bounded model checking technique in order to detect memory safety errors. It will first generates a set of testing driver for those relevant APIs, converts the driver code from Rust to C, and then performs bounded model checking. With the help of exhaustive test generation, it is able to efficiently explore large input spaces and generated multiple test drivers for covering the input spaces as many as possible. CBMC is used as a back-end service for automated analysis. By evaluating data structures from Rust standard library, CRUST is capable to detect some memory safety bugs from it.

1.5.5 Other applications

There are several studies showing that model checking is applied to various usages, such as embedded software verification [7, 12], cross-platform verification [36], sequentialisation-based verification [14], fault localisation [15], equivalence checking [24], cyber-physical system co-verification [40], and test-vector generation [3].

Bibliography

[1] The cprover user manual. `http://www.cprover.org/cbmc/doc/manual.pdf`.

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

[3] 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.

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

[5] 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.

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

[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, 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

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

[38] 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.

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

[40] 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.