A survey on static analysis and model checking

Iván García-Ferreira, Carlos Laorden, Igor Santos, Pablo Garcia Bringas

Deustotech Computing, University of Deusto,
Avenida de las Universidades 24, 48007, Bilbao, Spain
{ivan.garcia.ferreira,claorden,
isantos,pablo.garcia.bringas}@deusto.es

Abstract. The error detection in software is a problem that causes the loss of large amount of money in updates and patches. Many programmers spend their time correcting code instead of programming new features for their applications. This makes early detection of software errors become essential. Both in the fields of static analysis and model checking, great advances are being made to find errors in the software before the products are released. Although model checking techniques are more dedicated to find malware, it can be adapted for errors in the software. In this article we will discuss the techniques used today for the search of patterns and vulnerabilities within the software to know what are the possible solutions to this issue. We examine the problem from the point of view of their algorithms and their effectiveness in finding bugs. Although there are similar surveys, none of them addresses the comparison of best static analysis algorithms against the best mathematical logic languages for model checking, two fields that are becoming very important in the search for errors in software.

Keywords: Static binary code analysis, mathematical logic, model checking, bugs

1 Introduction

The era of computer bugs began with the first computers in the late 1940's. More specifically, in 1947 was when the first computer bug arose [42]. Since then, there have been many computer bugs causing the loss of thousands of dollars. In some cases a computer bug costs millions of dollars, as we can observe in the arianne 5 case [31], or, in the worst cases, lives, as we can observe in the Therac-25 case [30] in which seven people died because they received radiation 100 times higher than normal. More recently, in 2012, the company "Knight Capital" lost more than 400 million in the stock market because the program used to buy and sell shares bought everything in the market for 45 minutes.

These kind of problems in computers show that a correct programming is required. For this reason, many programmers analyse the source code manually, but they lose a lot of time with this task, and the human error is always present. For that purpose, programmers build tools to analyse source code and even the binary file generated by the compiler.

This survey is divided in four sections. In section 2 we analyse the algorithms used today for the static analysis of code. In section 3 we show which are the temporal logic languages most used in model checking. In section 4, we show previous works in static analysis and model checking. And, finally, section 5 offers the conclusions and future work.

2 Static analysis

Although, nowadays, the use of dynamic analysis among programmers is arising, the high computational requirements involved in this task make this technique unproductive. Because of this, static analysis is the approach most frequently used to perform code analysis.

Static analysis is not only used for the detection of computer bugs in the source code, but is largely used for pattern searching that might lead to discover if a binary file is malware or goodware. We can see an example of this research in some of Christodorescu et al. works [6, 7] .

For this reason, almost from the beginning of computer history, many mathematicians and computer theorists have developed some algorithms for searching patterns in software.

In the next sections we will see some of the best algorithms to search patterns.

2.1 Hoare logic

One of the first steps in mitigating errors was developed by C.A.R. Hoare [23] who created the Hoare Logic with the contributions of Robert Floyd.

The most important element in the Hoare logic is the Hoare Triple. The Hoare Triple has the following form: "{Q} S {R}". Where Q and R are the precondition and the postcondition that a computer program has to met for the code S to work correctly. In short, you could say that if it is true that the program starts in a state Q and ends in a state R, the program will work correctly. Hoare logic is able to avoid many bugs to make programming safer thanks to a correct specification.

$$
\begin{array}{l}
\{Q \equiv 1 \leq n \wedge n \leq 1000\} \\
fun \;\; maximum(a : vect; n : integer) dev(x : integer) \\
\{R \equiv (\forall \alpha \in \{1..n\}.x \geq a\,[\alpha])\}
\end{array}
$$

Fig. 1. Hoare logic example.

The pseudocode showed in Fig. 1, is a function to compute the maximum of a vector of integers. With Q, we define what values are allowed in variable "n", in this case, "n" can not be less than 1 or greater than 1000. In the second line,

we can see the specification of the function with its input and output variables and the types of these variables. In the last line, we see the postcondition R, in this example, it returns the highest value of the entire array.

Edsger W. Dijkstra [15] invented an extension of Hoare logic, Predicate transformer semantics, where he established the concept of the weakest precondition:

$$wp(\mathcal{S}, \mathcal{R}) \tag{1}$$

Where S is a script to perform and R one postcondition that must be met.

2.2 Shape analysis

In 1967 Reynolds [38] created *Shape analysis* to inspect the heap pointers and to know how these pointers access to the memory. This analysis is performed using a graphic called *shape graphs* or *alias graphs*, where it is shown how pointers are related to the memory locations that are being targeted.

One problem with this approach is that it is not sensitive to the flow of the program, which may not detect many vulnerabilities. This makes the *Shape analysis* become inaccurate, and can often generate errors showing program behaviours not corresponding to the actual ones.

The advantage of this technique is the clarity in the representation of results in its tools. With the graphical results you can know how the relationship between pointers and memory addresses is.

2.3 Abstract interpretation

Abstract interpretation is an approach to the semantic structure of a program. It was created by Cousot [13] as a method for static program analysis. Its main applications are to decide optimisations and transformations, and error detection.

With *abstract interpretation* an application can be tested into a domain of abstract values. With this, we obtain an abstract version of the application with abstract data. Also, with this point of view is not necessary to run the application, getting a broader view of program behaviour.

One advantage of this technique is that it is possible to know if the results of the program could be obtained in a finite time. Another advantage, is that the results describe the behaviour of some program executions. But *abstract interpretation* also has a disadvantage, the information obtained in the analysis is always an approximation of what really occurs in a program execution.

2.4 Value-Set analysis

Balakrishnan and Reps introduced the first way to recognise the values in memory using the *Value-Set Analysis* [3]. VSA is an abstract way of analysing an executable to know what values will take each variable in a program.

Summarising, we could say that VSA is a line by line analyser of all the information that exists in the registers and variables in memory (on the stack, which are called a-locks). An example of VSA at a given point of a function could be:

$$esp \rightarrow (\perp, -44), var_1 \rightarrow (0, \perp), var_2 \rightarrow (0, \perp), eax \rightarrow (\perp, 4[1, \infty])$$

Fig. 2. Value-Set analysis example

In the example of Fig. 2, VSA checks the value of the main registers and variables declared in the program. In this example, the ESP register is pointing to a value which is located in the position -44, regarding the value of the stack. The variable "*var*_1" is the first global variable, the same happens with the variable "*var*_2" which, in this case, is the second global variable. The EAX register contains the same value that the global variable has, which ranges from 1 to infinity, because it is not specified which is the highest value that the variable can have.

To make the job easier VSA uses IDAPro [25] to find known addresses, stack offsets, information on the limits of procedures and calls to system libraries (using the FLIRT technology [21]). With all this, it generates a small database which is used for subsequent analysis.

Balakrishnan et al. also published another work in which they doubted that compilers were performing their job properly [4]. Balakrishnan et al. performed tests which concluded that although a programmer had made the code perfectly, the compiler could introduce mistakes in optimisation time and cause failures in the program.

3 Model Checking

Model checking is a technique that, given a formal property checks whether that specific property has been met in the state model, where the state model is the description of the states and events of a system using a diagram or table. The property is a formula that represents some behaviour within the system described by the state model.

Model checking was introduced by Emerson et al. [17], Clarke et al. [8], [9], and Queille and Sifakis [37]. But, if we have to talk about the first steps of model checking in software verification, the pioneer was Holzmann [24].

For model verification, these systems need a system that shows the properties of states and transitions. For this purpose, the researchers use temporal logic, which was introduced by Pnueli [35] in 1977.

Although there are many applications today for model checking, the main tools in this field are based on LTL or CTL to make their calculations. One of the most used tools is SPIN [41], which use LTL to perform formal verification.

Another widely used tool is NuSMV [34], which in this case is based on CTL, but also partly based on LTL.

3.1 CTL

The creation of CTL was made by Emerson and Clarke [8], although the complete axiomatisation was made by Ben-ari et al. [5] and Emerson and Halpern [18].

The basic operations in CTL are the same that in normal mathematical logic, and we can see them in Fig. 3.

p	q	~p	p ∨ q	p ∧ q	p ↔ q	p → q
V	V	F	V	V	V	V
V	F	F	V	F	F	F
F	V	V	V	F	F	V
F	F	V	F	F	V	V

Fig. 3. Basic operations in CTL.

One of the benefits of CTL is the possibility to knwo whether it is feasible to satisfy a certain condition over the time (see Fig. 4). This fact is possible because there are several types of operations that do not exist in standard propositional logic. CTL introduces Universal ($\forall$) and existential ($\exists$) operators, to determine if a condition met in one or more paths in the graph.

In addition to these operators, CTL introduces new operators ($\Diamond\psi$, $\bigcirc\psi$, $\Box\psi$, $\psi\ \mathcal{U}\phi$) to perform operations with paths. The operator $\Diamond\psi$ is used to indicate if a path is going to satisfy a particular property at a given time. The operator $\bigcirc\psi$ checks whether a property is going to satisfy in the second position of the graph. The operator $\Box\psi$ checks if a path satisfies a certain property in all states of the graph. And finally the operation $\psi\ \mathcal{U}\phi$, which checks if there is a path that start with ψ and in any moment contain a path that met ϕ.

In CTL these operators can not be separated from the existential and universal quantifiers, in order to fulfil that condition operations must always be defined as you have seen in Fig. 4.

3.2 LTL

LTL was created by Manna and Pnueli [32]. It is a temporal logic that reasons about only one timeline, and uses the same operators as temporal logic.

As LTL has a single execution path, unlike CTL, so the universal and existential operators are not needed. The main feature that has LTL is that it can chain temporal formulas, unlike in CTL, where the temporal operators should always go with the universal or existential operator.

An example of an LTL formula that can not be done in CTL would be: $\Diamond\Box$p. On the contrary, in LTL it would be imposible to define the following CTL formula: $\forall\Diamond\forall\Box$p.

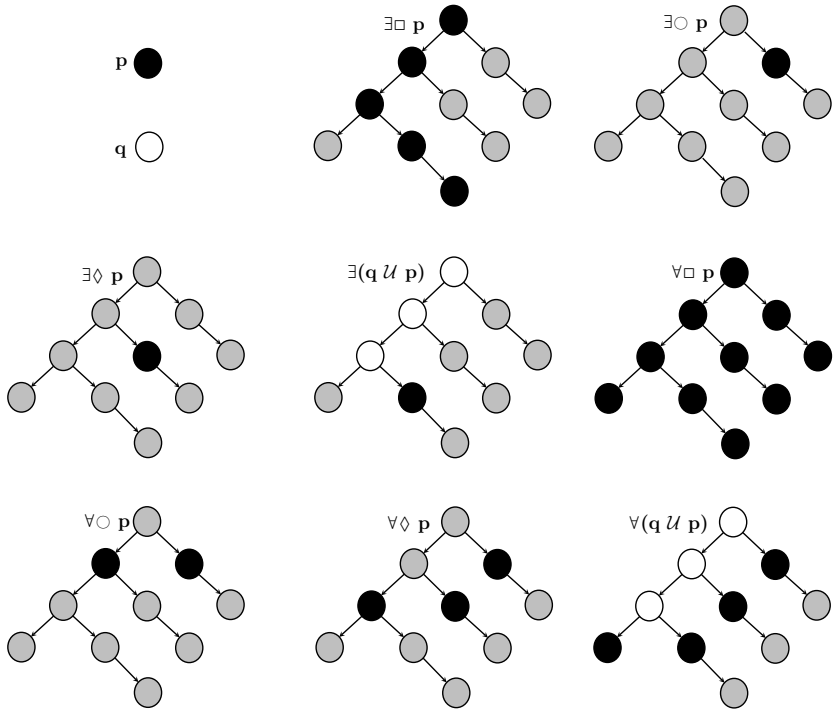

Fig. 4. Graphic representation of some CTL formulas, where the black state satisfies p, the white state satisfies q and the gray circle is any state.

3.3 CTL*

CTL* (pronuncied CTL star) was created by Emerson and Halpern [19]. This logic is more expressive than the temporal logics seen so far, and for this reason there are formulas that can be performed in CTL* which are not conceivable in other temporal logics. Thus it can be said that CTL* comes directly from LTL and CTL, because it contains the linear operators of LTL and the path operators of CTL.

Although CTL* is more complete than the other temporal logics, it has not been practically used for model checking. And this is even more surprising because the computational complexity of CTL* is exactly the same as LTL, PSPACE. One possible reason for this is that the complexity of CTL* formulas can be very high, and may be difficult to verify their correctness.

3.4 CTPL

CTPL is a temporal logic created by Kinder et al. [27] to search malware in binary files. The syntax used by CTPL is similar to that used by CTL, but for Kinder et al. states are assembly instructions. Besides, CTPL uses the predicate `#loc(L)` to control the order in which the arguments are passed to the stack.

EF (mov eax, 937 ∧ AF(push eax))

Fig. 5. Example of CTPL formula.

In Fig. 5 we can see an example of the CTPL syntax. This example searches the instruction `mov eax 937` and later the instruction `push eax` is encountered, regardless of the execution path of the binary.

In this approach, the first thing that Kinder did was unpack the binaries with different tools (UnFSG, Petite Enlarger, etc.). Packing the binary is a very common technique among malware programmers to prevent the code from being detected as malware. Once unpacked, the binary assembly code was ready to be extracted using a dedicated tool, IDA Pro [25].

In 2012, Song and Touili [39, 40] developed SCTPL, an extension of CTPL. SCTPL allows building predicates with the stack. With this feature it is possible to increase the detection of malware. Song translated binary code into a pushdown system that mimicked the program's behaviour.

3.5 μ-calculus

μ-calculus can be seen as an extension of CTL. It was created by Kozen [29] in 1983. The μ-calculus uses the same operations that are used in CTL. One difference is that, instead of using existential and universal operators, μ-calculus uses $\langle\rangle$ for the existential operator operations, and [] for the universal operator operations.

The main difference between CTL and μ-calculus is the use of fixpoints. The μ-calculus has least fixpoint (μ) and a greatest fixpoint (ν), which makes it possible to give an external fixpoint characterisation of correctness properties.

3.6 State explosion problem

One of the biggest problems in model checking is the large amount of data that must be managed. Some problems can arise for this reason, but the most relevant is the time needed to compute all data. This problem is called "state explosion problem", and some researchers are fighting against this problem for many years [11, 10].

Some techniques tried to abstract from the data to make it more workable. Cousot and Cousot [14] tried this approach and then verified that abstraction. Alur et al. [1] analysed the data, then used the information obtained in the previous step to abstract from the data, and finally verified the abstractions.

4 Related work

Studies on the static analysis and model checking have previously been presented by D'Silva et al. [16], but temporal logics are not mentioned in the article.

D'Silva's study is about the verification of source code, but as we know the compiler can introduce errors in the compilation process.

Kunur [28] performed a study of temporal logic in 2010. In this study we can observe the different temporal logics that have appeared in recent years, but many of these have not been used in model checking.

Engler and Musuvathi [20] performed a study in which they checked the effectiveness of static analysis against model checking. This study concluded that although the static analysis detected more errors, the model checking detects errors that a static analysis can not detect, but it took much longer to prepare the tests.

These studies demonstrated that static analysis techniques are very powerful in finding errors in the source code (*Hoare logic*, *Shape analysis*, *Abstract interpretation*) or x86 assembler (*Value-Set Analysis*). They get good results quickly for general and known errors, for example well documented buffer overflows(e.g. strcpy). If we are talking about finding poorly documented errors, model checking is the best option. It is also able to find errors in both, the source code (CTL, LTL, CTL *), and x86 assembly code (CTPL). The problem with model checking is that you can only know if the source code contains the error or not, often making it very difficult correction.

All algorithms showed in this article have tools that demonstrate its effectiveness, for example Frama-C [22] (abstract interpretation), Java+ITP [26] (Hoare logic), Predator [36] (Shape analysis), CodeSurfer [12] (Value-Set analysis), NuSMV [34] (CTL), SPIN [41] (LTL), ARC [2] (CTL*) or mCRL2 [33] (μ-calculus). The only algorithm that has no tool with which we can test their effectiveness, is CTPL. This algorithm was developed only for scientific purposes and there is not a tool or source code available.

5 Conclusions

As we have seen, techniques to find bugs in software are evolving to become more accurate. Both in the field of static analysis and model checking, the researchers are getting more information from an executable without the source code.

Each year there are new techniques to improve the proposed algorithms in different ways. In many cases the analysis try to be as accurate as possible and others try to do the algorithm as fast as possible. Although it seems that the trend is to improve the algorithms, it is possible that new techniques will make obsolete current algorithms.

Static analysis and model checking have limitations. In static analysis the time required to generate results can consume a lot of CPU time, so the time needed to make an analysis will be bigger than in other techniques. In model checking, the analyses are faster but in a lot of cases a Boolean result can be insufficient. With Model Checking it is possible to know if a binary file is malware or contains an error, but it is impossible to know, where is the bug in the code, or where is the malicious code in malware.

For future work, we are planning to improve the model checking techniques using set theory to get more accurate results instead of the boolean output used in of traditional model checking. This approach can be considered a hybrid system between the techniques of model checking and static analysis, because it will have the accuracy that the formulas of model checking have and the ease of bug correction that provide static analysis techniques.

References

1. Alur, R., Alfaro,L. de, Henzinger, T.A., Mang, F.Y.C. Automating Modular Verification. In *Proceedings of the Tenth International Conference on Concurrency Theory*, Volume 1664 LNCS, pp. 82—97, Springer-Verlag, 1999.
2. ARC. http://altarica.labri.fr/wp/?page_id=32 (Last accessed: 20 Feb. 2014)
3. Balakrishnan, G., Reps, T., Analyzing memory accesses in x86 executables. In *Compiler Construction*, pp. 2732-2733, Springer, 2004.
4. Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T., WYSINWYX: What You See Is Not What You eXecute. In *Proc. IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2005.
5. Ben-Ari, M., Pnueli, A., Manna, Z. The temporal logic of branching time. In *Acta informatica*, 20(3), pp. 207-226, 1983.
6. Christodorescu, M., Jha, S., Seshia, S. A., Song, D., Bryant, R. E., Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy*, pp. 32-46, 2005.
7. Christodorescu, M., Jha, S. Static analysis of executables to detect malicious patterns. In *Wisconsin Univ-Madison dept of Computer Sciences*, 2006.
8. Clarke, E. M., Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Springer Berlin Heidelberg*, pp. 52-71, 1982.
9. Clarke, E. M.; Emerson, E. A.; Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems*, 8 (2): 244, 1986.
10. Clarke, E. M., Grumberg, O.. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pp. 294-303, ACM, 1987, December.
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H. Progress on the state explosion problem in model checking. In *Informatics*, pp. 176-194, Springer Berlin Heidelberg, 2001, January.
12. CodeSurfer. http://www.grammatech.com/research/technologies/codesurfer (Last accessed: 20 Feb. 2014)
13. Cousot, P., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN*, pp. 238-252, 1977.
14. Cousot, P., Cousot, R. Refining Model Checking by Abstract Interpretation. In *Automated Software Engineering Journal*, 6(1), pp. 69-95, 1999.
15. Dijkstra,Edsger W., Guarded commands, nondeterminacy and formal derivation of program. In *Communications of the ACM*, pp. 453-457, 1975.
16. D'Silva, V., Kroening, D., Weissenbacher, G. A survey of automated techniques for formal software verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), pp. 1165-1178, 2008.

17. Emerson, E.A.; Clarke, Edmund M. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, 1980.
18. Emerson, E.A., y Halpern, J.Y. Decisions procedures and expressiveness in the temporal logic of branching time. In *Handbook of theorical Computer science*, vlo B: Formal models and Semantics. Elsevier, 1985.
19. Emerson, E. A., Halpern, J. Y. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. In *Journal of the ACM (JACM)*, 33(1), pp. 151-178, 1986.
20. Engler, D., Musuvathi, M. Static analysis versus software model checking for bug finding. In *Verification, Model Checking, and Abstract Interpretation*, pp. 191-210, Springer Berlin Heidelberg. ISO 690, 2004, January.
21. F.L.I.R.T., https://www.hex-rays.com/products/ida/tech/flirt/index.shtml (Last accessed: 20 Feb. 2014)
22. Frama-C. http://frama-c.com/ (Last accessed: 20 Feb. 2014)
23. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. In *Commun. ACM 12*, 1969.
24. Holzman, G.J. Design and validation of computer protocols. Prentice-Hall, 1990.
25. IDA Pro. https://www.hex-rays.com/products/ida/. (Last accessed: 20 Feb. 2014)
26. Java+ITP. http://maude.cs.uiuc.edu/tools/javaitp/ (Last accessed: 20 Feb. 2014)
27. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H. Detecting malicious code by model checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 174-187, Springer Berlin Heidelberg, 2005.
28. Konur, S. A survey on temporal logics. arXiv preprint, 2010.
29. Kozen, D. Result on the Propositional µ-calculus. In *Journal of Theoretical Computer Science*, 27, pp. 333-354, 1983.
30. Leveson, Nancy. An Investigation of the Therac-25 Accidents, In *IEEE Computer*, 26, pp. 18-41, 1993.
31. Lions, J.L. ARIANE 5, Flight 501 Failure. http://www.di.unito.it/ damiani/ariane5rep.html , 1993. (Last accessed: 20 Feb. 2014)
32. Manna, Z. Pnueli, A. The Temporal Logic of Reactive and Concurrent Systems. In *Springer-Verlag*, 1991.
33. mCRL2. http://www.mcrl2.org/ (Last accessed: 20 Feb. 2014)
34. NuSMV. http://nusmv.fbk.eu/. (Last accessed: 20 Feb. 2014)
35. Pnueli, A. The temporal logic of programs. In *Foundations of Computer Science 18th*, 1977.
36. Predator. http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/ (Last accessed: 20 Feb. 2014)
37. Queille, J. P.; Sifakis, J. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, 1982.
38. Reynolds, J. Automatic computation of data set definitions, In *Science*, 1967.
39. Song, F., Touili, T. Efficient malware detection using model-checking. In *FM 2012: Formal Methods*, pp. 418-433, Springer Berlin Heidelberg, 2012.
40. Song, F., Touili, T. PoMMaDe: pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 607-610, ACM, 2013, August.
41. SPIN. http://spinroot.com/spin/whatispin.html. (Last accessed: 20 Feb. 2014)
42. The First "Computer Bug". http://www.history.navy.mil/photos/images/h96000/h96566kc.htm. (Last accessed: 20 Feb. 2014)