Toasters, Seat Belts, and Inferring Program Properties

David Evans

University of Virginia, Department of Computer Science
Charlottesville, Virginia
evans@cs.virginia.edu

Abstract. Today's software does not come with meaningful guarantees. This position paper explores why this is the case, suggests societal and technical impediments to more dependable software, and considers what realistic, meaningful guarantees for software would be like and how to achieve them.

If you want a guarantee, buy a toaster.
Clint Eastwood (*The Rookie*, 1990)

1 Introduction

Software today doesn't come with guarantees. Should it? What kinds of guarantees should they be?

I wouldn't want to argue with "Dirty Harry", but toasters don't really come with guarantees either, certainly not in the sense of a mathematical proof that they will satisfy a set of precisely defined requirements. What a toaster does come with is: (1) a reasonable expectation that a semi-intelligent user will be able to get the toaster to transform a typical slice of bread into toast; (2) a warranty that the manufacturer promises to replace the toaster if it is defective, and (3) in the United States to a large degree, and to varying degrees in other countries, the assurance that if the defectively designed or manufactured toaster causes your house to burn down, you will be able to sue the toaster manufacturer for damages far in excess of the cost of the toaster.

Software is a long way from satisfying any of those properties: (1) purchasers of software do not expect it to work correctly; instead of returning misbehaving software, users are conditioned blame themselves; (2) software usually comes with an offer to replace defective disks, by no warranty on correct behavior; and (3) software vendors, to date, have managed to be immune from liability lawsuits even in cases where negligent implementations produce serious losses. We do, however, have many of the essential technologies in place to provide meaningful guarantees regarding software systems. Research tools have been developed to check properties of large programs [5, 10, 12, 14, 20, 22], and dozens of companies are now offering analysis tools and services (e.g., Coverity, Fortify, Ounce Labs, PolySpace, Reflective).

The rest of this paper discusses four major impediments remaining before routine software comes with effective guarantees: a lack of mechanisms for providing the necessary incentives to encourage software vendors to invest resources and delay products to improve dependability; inadequate ways to identify properties worth checking; insufficient theoretical understanding of how to interpret the outcome of checking, especially unsound analyses, as meaningful guarantees; and deficiently educated developers unable to effectively use and insist on the use of appropriate program verification tools and techniques.

> *The first principle was security... A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not to—they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.*
>
> Tony Hoare, describing Elliott Brothers' Algol 60 implementation
> *The Emperor's Old Clothes*, 1980 Turing Award Speech

2 Incentivizing Verification

Twenty-five years after Hoare's speech, computing is still not a "respectable branch of engineering": software developers and language designers continue to release code where memory references are unchecked and no one has yet been sent to jail or even fined for doing it. The technologies for preventing this particular type of error have been available for many decades, yet vendors still ship software without using them. If jail sentences had been established in 1980 for the CEO of any company that sells a product containing a buffer overflow vulnerability, I suspect there would have been no programs with buffer overflow vulnerabilities sold in 1981, and certainly not in 2005. Alas, I know of no jurisdiction that has made programming in C++ or designing a language without bounds checking a criminal offense [8]. This is an incentive problem, not a technology problem.

An automobile company could not sell a car that suffers from a problem like unchecked array references, without losing billions of dollars in lawsuits. As a result, technologies that improve safety are quickly deployed throughout the industry. Some parallels can be drawn between safety belts in cars and bounds checking in software. Safety belts were introduced in the 1950s because of biomechanical research suggesting their effectiveness; they were, however, rarely used by car occupants until mandatory belt wearing laws were passed [16]. As with bounds checking, a very

effective technology was available but largely unused for many decades. However, unlike the case with software, legal mechanisms in the form of both regulation and liability, placed pressure on vendors to incorporate the best known technology in their products. In 1986, General Motors became the first US auto manufacturer to decide to install lap/shoulder belts in the rear seats of cars, instead of lap-only belts. GM began installing lap/shoulder belts in selected 1987 model cars. The other auto companies followed within a few years, and it later became a government standard. GM faced a $200M lawsuit (which was settled under seal) claiming that GM was negligent in not making the change sooner since its internal research indicated that lap-only belts were less effective than previous government estimates [16, 17].

Software in embedded systems is subject to potential lawsuits if the containing device fails with disastrous results. As a result, the development and validation practices for such software is quite different from that typically used for software-only systems. With a few notable exceptions, critical software in embedded systems today is remarkably reliable compared to software in software-only systems.

Applying product liability to software is not without risks, however. If companies that do not deploy the best known technology can be liable for negligence, this provides a strong disincentive to developing new technologies and stifles creativity and innovation. Software liability also raises serious issues for open source developers and academic researchers who wish to develop and distribute software without fear of lawsuits or needing approval from lawyers.

An alternative is to use market forces. This has proven difficult so far with software, primarily because of the difficulty in measuring software quality, especially security. A research community is emerging that considers economic approaches to improving security [3, 4] as well as measuring it [23]. Good ideas for software security metrics, however, remain elusive. One promising direction is work on measuring relative attack surfaces [19].

There are no easy answers here, but it seems many of the challenges we face in improving software dependability and security are not so much in developing tools and techniques to analyze programs, but in making using those tools cost effective in a business sense. This involves the technical challenges of decreasing the costs of using them and increasing the value they provide, but also the large contextual challenge of making the costs of improving software quality economically justifiable by increasing the cost disadvantages associated with low qualify software. The trends are in the right direction as evidenced most clearly by Microsoft's trustworthy computing initiative [18] and increasing willingness to sacrifice functionality and delay product releases to enhance security over that past few years [21].

It is easier to write an incorrect program than understand a correct one.
Alan Perlis

3 Identifying Properties

We can group properties into three categories:

1. Generic language semantics properties.
2. Documented application properties.
3. Unknown (but necessary) application properties.

The first category comprises those properties that should always be true of all programs, such as all memory references are in bounds and the program never leaks memory. Since these properties are universal, they should be identified once by the programming language designers and there is no need to identify them for a particular application. Most program analysis tools available today are focused on checking or detecting violations of this type of property. Few viable excuses remain for releasing software today that suffers from these kinds of flaws.

In most cases, documented application-specific properties do not exist. There is no precise description of required application properties, and even the developers don't know what those properties are. Efforts to improve education for software developers (discussed in Section 5) may increase the likelihood of there being documented properties, but progress here will be slow and limited. Except for the most safety-critical (and thus expensive) software, it is unreasonable to expect required application properties to be clearly documented in the near future. Even when developers are willing to spend the effort required to formally document these properties, they often do not know what properties are necessary for correctness or would be useful to document.

Our efforts should focus, then, primarily on the third category – unknown and undocumented, but necessary, application properties. Over the past few years, several research groups have developed techniques for inferring those properties. Daikon infers data invariants on programs by analyzing execution traces on a test suite [13]. Other researchers have developed techniques for inferring specifications of programs from their dynamic behavior [2, 9, 32] and static analysis of their program texts [1, 31, 32].

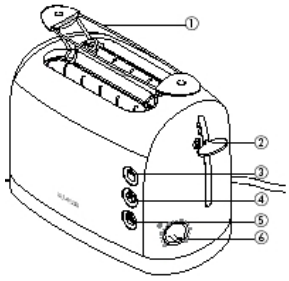
My research group's work in this area is motivated by the observation that many properties in the third group are true during many or all test executions, but when they are violated during real executions they produce serious consequences. We have developed a tool, Terracotta [30], that takes a program and a test suite and produces a set of inferred properties. We have focused primarily on inferring simple temporal properties that constrain the order and occurrence of events in the program such as calls to a particular method (such as all calls to the `lock` method must be followed by

calls to the corresponding unlock method) or combinations of temporal and data properties (such as, object O is never modified between events A and B). The goal is not to produce a specification of the program for human use, but rather to infer properties that are useful for other purposes. We have used inferred properties to identify undesirable behaviors [27]; unexpected differences between similar programs or different versions of a program [28]; and as input to a model checker [29]. When a counterexample to an inferred property is found, it may reveal a bug in the program or a deficiency in the testing approach. By inferring properties this way and using them with automatic checking and comparison tools, we are able to discover essential properties about a program that developers would not think to document.

4 Towards Software Guarantees

Let's return to the rather degenerate toaster example and the toaster guarantee is shown in Figure 1. The guarantee does not claim that the toaster will always behave according to a particular specification. Rather, it states that if the toaster "goes wrong" it will be replaced, provided the user does not misuse, "neglect", modify, or damage the toaster. Despite its limitations, this guarantee has some value to the purchaser, and it would be a major advance if software came with a similar guarantee.

The technical challenge is to determine the equivalent of "goes wrong" for a complex software system. Automatic property inference and checking is a step towards this goal. Instead of attempting to formally specify the exact behavior for software, by using property inference techniques to infer properties that are true of the "normal" behavior of the software, and checking (or ensuring at run-time) that they are always true we can establish claims about the scope of executions covered by the testing strategy. If the inferred properties capture enough of the behavior of the software, then we can claim that executions that satisfy those properties are "okay", and executions that do not satisfy them have "gone wrong". In such circumstances,



safety

- Burnt food can catch fire, so:
  - never leave your toaster on unattended;
  - keep your toaster away from anything (eg curtains) that could catch fire;
  - set the browning control lower for thin or dry bread;
  - set the browning control no higher than ☹ when using the warming rack; and
  - never warm food with a topping or filling (eg pizza): if it drips into the toaster, it could catch fire.
- To avoid electric shocks, never:
  - let the toaster, cord or plug get wet: or

guarantee

If your toaster goes wrong within one year from the date you bought it, we will repair or replace it free of charge provided:

- you have not misused, neglected or damaged it;
- it has not been modified (unless by Kenwood);
- it is not second-hand;
- it has not been used commercially;
- you have not fitted a plug incorrectly; and
- you supply your receipt to show when you bought it.

This guarantee does not affect your statutory rights.

Figure 1. Kenwood TT360/TT390 Toaster Instructions (excerpted).

measures can be taken to put the software right again to return to the "normal" behavior. Rinard and his colleagues' work on acceptability-oriented computing [11, 24, 25] and Swift et al.'s work on hiding device driver failures from executing programs [26] illustrate the possibility of executing programs in ways that programming errors are automatically recovered from. When unsound analysis techniques are used, we cannot expect to make full correctness guarantees; instead, we should strive to find ways to formalize guarantees more like the toaster guarantee of nothing "goes wrong", and to develop tools and techniques that allow us to make such guarantees.

The use of COBOL cripples the mind;
its teaching should, therefore, be regarded as a criminal offence.
Edsger W.Dijkstra, *How do we tell the truths that might hurt?* (EWD 498), June 1975

5 Education

The single most important factor in determining the quality of software is the knowledge, experience and attitudes of people who design and implement it. People choose the programming languages, compilers, analysis tools and testing and validation approaches to use. Hence, it is unlikely that software quality will improve dramatically without also changing the ways we educate programmers. Although increasing automation can make analysis tools accessible to less sophisticated developers, it will be up to developers to decide to use those tools and to correctly interpret their results.

Computer science curricula have traditionally followed industry, not led it. With rare exceptions, the choices of programming languages and tools used in most introductory software engineering courses follow a few years behind the current needs of industry rather than envisioning the future and focusing on producing graduates with conceptual understanding and the ability to lead industry forward. To improve the state of software engineering, academia needs to take the lead in teaching students in introductory software engineering courses the theories, tools and techniques that will be important for verified programming. Instead of focusing on the technical details of complex programming languages that are popular in industry, introductory software engineering courses should be teaching students to think about preconditions, postconditions, data invariants, and temporal properties and to understand what program analysis tools and testing techniques can allow them to state about their programs. At the University of Virginia, we are developing a curriculum towards these goals [6, 7] (which draws heavily from the MIT curriculum), and have experimented with introducing static analysis tools in our introductory software engineering course [7]. Although the state of the art in available tools presents some challenges, and it is difficult for students in introductory courses to formally document complex invariants, we are optimistic that incorporating automatic property inference tools into the process can help [15] and that this approach can provide

students with the necessary background to develop more secure and dependable software.

6 Summary

The research and industrial communities have made tremendous progress in program analysis and verification tools over the past several years, and these tools have now reached the point where they can be usefully applied to large, complex programs. In order for their use to become prevalent, however, the appropriate incentive structure must be in place. Technical challenges remain in determining useful properties to check that go beyond generic language properties, and in better understanding the claims that can be made as a result of unsound analyses. Promising directions for research towards these goals include automatic property inference and automatic detection of and recovery from errors. Full program verification against a precise specification will remain expensive and rare, but perhaps advances in technology and changes in incentive structure will make meaningfully guaranteed software commonplace.

References

1. Rajeev Alur, Pavol Černý, P. Madhusudan and Wonghong Nam. Synthesis of Interface Specifications for Java classes. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 2005.
2. Glenn Ammons, Rastislav Bodik and James R. Larus. Mining Specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. January 2002
3. Ross Anderson. *Economics and Security Resource Page*. http://www.cl.cam.ac.uk/users/rja14/econsec.html
4. L. Jean Camp and Stephen Lewis, editors. *Economics of Information Security*. Kluwer Academic Publishers. September 2004.
5. Hao Chen, Drew Dean, and David Wagner. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium* (NDSS). February 2004.
6. *CS150: Computer Science from Ada and Euclid to Quantum Computing and the World Wide Web*. University of Virginia Course. http://www.cs.virginia.edu/cs150
7. *CS201j: Engineering Software*. University of Virginia. http://www.cs.virginia.edu/cs201j
8. *CS655: Graduate Programming Languages*. University of Virginia Course. Spring 2000. http://www.cs.virginia.edu/evans/cs655-S00/mocktrial/
9. J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, pp. 297-319, Vol. 53, No. 3, April 2004.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification In Polynomial Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2002.
11. Brian Demsky and Martin Rinard. Data Structure Repair Using Goal-Directed Reasoning. *Proceedings of the 2005 International Conference on Software Engineering*. May 2005.

12. Dawson Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific Programmer-Written Compiler Extensions. *Symposium on Operating Systems Design and Implementation*. October 2000.
13. Michael D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*. February 2001.
14. David Evans. Static Detection of Dynamic Memory Errors. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.
15. David Evans and Michael Peck. *Simulating Critical Software Development*. University of Virginia Computer Science Technical Report, UVA-CS-TR2004-04. February 2004.
16. Leonard Evans. *Traffic Safety*. Science Serving Society Press, 2004. http://scienceservingsociety.com/traffic-safety.htm
17. Leonard Evans. Personal communication, April 2005.
18. Bill Gates. *Trustworthy Computing Initiative* (memo to all Microsoft employees). 15 January, 2002.
19. M. Howard, J. Pincus, and J.M. Wing. Measuring Relative Attack Surfaces. Proceedings of *Workshop on Advanced Developments in Software and Systems Security*. Taipei, December 2003.
20. David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. *USENIX Security Symposium*, August 2001.
21. Microsoft Corporation. *Trustworthy Computing.* http://www.microsoft.com/twc
22. M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.
23. Andy Ozment. Bug Auctions: Vulnerability Markets Reconsidered. In *Workshop on Economics and Information Security*. May 2004.
24. Martin Rinard. Acceptability-Oriented Computing. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion* (OOPSLA '03 Companion) Onwards! Session. California October 2003.
25. Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-Oblivious Computing. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. December 2004.
26. Michael Swift, Muthukaruppan Annalamai, Brian Bershad and Henry Levy. Recovering Device Drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. December 2004.
27. Jinlin Yang and David Evans. Dynamically Inferring Temporal Properties. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2004.
28. Jinlin Yang and David Evans. Automatically Inferring Temporal Properties for Program Evolution. *15th IEEE International Symposium on Software Reliability Engineering*, November 2004.
29. Jinlin Yang and David Evans. Automatically Discovering Temporal Properties for Program Verification. In submission, January 2005.
30. Jinlin Yang. *Terracotta*. http://www.cs.virginia.edu/terracotta
31. Westley Weimer and George Necula. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems* (TACAS 05), April 2005.
32. John Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *International Symposium on Software Testing and Analysis*, July 2002.