Editors: Mohamed Kaâniche, mohamed.kaaniche@laas.fr | Aad van Moorsel, aad.vanmoorsel@ncl.ac.uk

The Tests-versus-Proofs Conundrum

George Candea | EPFL

In recent years, advances in formal-proof systems and constraint solvers have enabled us to dream of a day when all the software we write can be proven correct. Software practitioners now must decide whether to persist in using testing to improve confidence in their code or to invest effort in learning and adopting more formal approaches. How are they to choose? Well, ... it depends.

Proofs or Tests?

Proofs are powerful. They're mathematical constructs that guarantee that all possible executions of a program satisfy (under certain assumptions) a particular desired property, such as freedom from crashes or undivertible control flow. Mathematics never lies. In an era when attackers can use a mere SMS to hack mobile phones, and malware can bring down critical infrastructure systems, we yearn for the certainty that proofs offer.

Yet proofs are as weak as their assumptions, and the easiest way to attack a "proven" system is by violating these assumptions. For example, the simplest way to attack a Java-based system might be to exploit Java runtime vulnerabilities; once the attacker owns the runtime, any proofs that assume the runtime's integrity are moot. Even compilers can be tricky. For example, we might prove on the basis of source code that a C program that checks for pointer wrap-around using `if (buf+len<buf)` is memory safe. However, `gcc -O2` will optimize away this check because pointer overflow is undefined behavior in the C specification and thus produces an unsafe executable.[1] Can we assume device firmware running on our network card is trustworthy? Can we assume our CPU is correct and secure?

System tests and penetration tests are the time-honored approach to increasing confidence in a computer system. Tests exercise actual binaries running in the actual environment on the actual hardware, thus reducing the number of assumptions. We can then measure code coverage: the higher the number, the better our managers sleep at night. Testing is also a good match for agile development practices, which dominate today. In contrast, formal approaches are prone to the ossifying waterfall model.

Yet tests are surprisingly incomplete, which is particularly problematic in the security domain. It's impractical to use classic testing to exercise a nontrivial fraction of the program paths in any realistic software. A beefy test suite might exercise as much as 90 percent of the statements in a piece of code but cover not even 1 percent of the (exponentially many) possible paths. Serious bugs often hide like needles in a haystack, and incomplete testing lets them persist through many code revisions.

Neither a formal approach nor testing is a clear winner in the war on bugs. So what are we to do? One answer to this conundrum is tools and techniques that let developers combine tests and proofs.

Bridging Tests and Proofs with Symbolic Execution

We can view a program as the encoding of a decision tree. For example, in Figure 1, each `if` instruction on the left corresponds to a decision in the tree on the right. To test this program, we give it some concrete input value—say, 1,200 RPMs—which makes the program follow a specific path (in green in Figure 1) through the tree. We then check the outcome. If we wanted to prove this program's correctness, we'd have to test it for all possible inputs, which implies 2^{32} tests (assuming 32-bit integers)—a tall order.

Symbolic execution (SE) can make this process considerably more efficient. Introduced in the 1970s, SE posits that, instead of using concrete inputs to execute a program, we can use symbolic inputs that subsume a range of possible concrete values. That is, we use as input λ, which initially represents all possible integers.

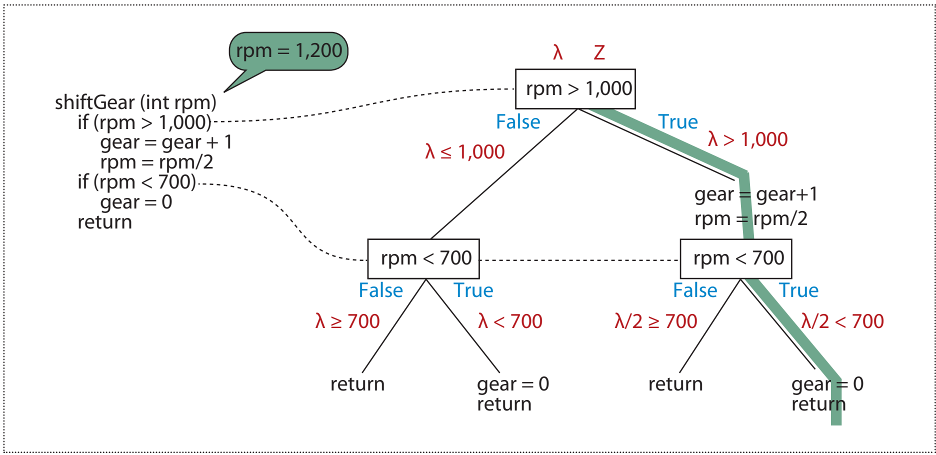

Figure 1. A program is a decision tree. Each instruction on the left corresponds to a decision in the tree on the right. Green indicates a program path; red indicates path constraints.

SE entails simulating a program's behavior using the symbolic input and collecting the constraints (in red in Figure 1) that the branch conditions impose. SE therefore automatically unfurls the program into the decision tree that it encodes, without missing any paths (that is, with no false negatives), unlike classic testing.

The beauty of SE is that we can use it for automated test generation. If the SE engine detects a violation of a desired property on a path, it can hand the conjunction of that path's constraints to a constraint solver to obtain a concrete value that takes the program down that path. For example, say that, before returning, the program accesses `sensors[rpm]`, where `sensors` is a zero-based array of 1,000 elements. Along the left-most path in the tree, `sensors[rpm]` could be out of bounds. Passing $\lambda \in \mathbb{Z} \wedge \lambda \le 1{,}000 \wedge \lambda \ge 700 \wedge \lambda > 999$ to a constraint solver yields the concrete value $\lambda = 1{,}000$, which, when used as the input, causes the out-of-bounds access.

SE automatically produces test cases that demonstrate unequivocally the bugs it finds, and is thus free of false positives.

To verify our example program's correctness, SE needs four symbolic values, one for each path: l_1 = [`MIN_INT`, 700), l_2 = [700, 1,000], l_3 = (1,000, 1,400), and λ_4 = [1,400, `MAX_INT`]. This covers the entire input space much faster than 2^{32} tests.

So, SE bridges tests to proofs by verifying an entire path at a time for all possible inputs that lead down that path. When the input isn't a single integer but entire files or multiple network packets, SE's benefit over classic testing is even greater.

SE is powerful; Microsoft uses a variant of it to save millions of dollars in potential security vulnerabilities.[2] At EPFL, we've been working on making SE practical while building on valuable prior research, such as the KLEE SE tool.[3] Other teams worldwide have been making significant contributions in this area, and I can't even attempt to do them justice in this short article. I'll just briefly describe our efforts at chipping away at three key challenges that arise when SE meets the real world.

Three Real-World Scalability Challenges

When trying to use SE for real software problems, a practitioner is likely to hit three main barriers: real software has tons of code, it's tightly connected to its execution environment, and deploying SE faces social challenges.

Real Programs ⇒ Path Explosion

The number of possible paths through a body of code is roughly exponential in code size, because each `if` might split execution into two paths. Loops exacerbate this problem, commonly called *path explosion*. In our Cloud9 system (http://cloud9.epfl.ch), we attack this problem in both depth (through algorithmic changes) and breadth (through parallelization on large compute clusters).

By merging nodes (symbolic program states) in the decision tree, we can turn it into a directed acyclic graph and reduce the number of paths exponentially. However, to preserve SE's precision, we must also merge the corresponding constraints. For instance, merging the two left-most leaves in our example tree would generate a single path with the disjunctive constraint $(\texttt{rpm} < 700 \wedge \lambda < 700 \vee \texttt{rpm} \not< 700 \wedge \lambda \ge 700) \wedge \lambda \le 1{,}000 \wedge \lambda \in \mathbb{Z})$. Unfortunately, such constraints can be much harder to solve than the two simpler constituent constraints. So, although this approach explores fewer paths, exploring some paths could take longer, with the net effect that SE takes longer than if we hadn't merged the states.

The key is to recognize when such a merge would be beneficial, and merge only then. Cloud9 employs static analysis to estimate each symbolic variable's impact on solver queries that follow a potential merge point. It merges only when doing so will likely be advantageous. Furthermore, Cloud9 merges states dynamically, during SE, in a way that interacts favorably with automated test generation. Using these techniques, Cloud9 explored up to 11 orders of magnitude more paths in the same time budget when compared to previously published results on the same software.[4] The best part is that the benefit of this state merging grows exponentially with the input size.

Cloud9 also parallelizes SE on public-cloud infrastructures, letting us "throw hardware at the problem." By employing a complete model of the Posix environment and full support for multithreaded software, Cloud9 found bugs that classic testing missed when used for systems such as Memcached, the Apache HTTP Server, and the Python interpreter.[5]

Nevertheless, for most real-sized software, exploring every path is still infeasible. So, instead of using SE for proofs, modern SE engines use it for better testing. Instead of exploring all possible paths, these engines employ heuristics to explore the paths likely to contain bugs. This trades completeness for reasonable runtime.

Real Execution Environments ⇒ Polyglot Tools Needed

Accurate analysis of program behavior typically requires understanding how that program interacts with its environment. Even small programs allocate memory, read and write files, send and receive network packets, and so on. Unfortunately, a real execution environment consists of many libraries, an OS kernel, and device drivers, written by many parties in many languages, often with no source code available.

One way to deal with this Babelesque situation is to abstract the environment and use a model to encapsulate all the assumptions that SE makes about the environment's behavior. Most SE engines, including Cloud9, take this approach. By employing models, the engine trusts the environment to behave according to its assumptions. Unfortunately, models are almost never fully accurate, they tend to further lose accuracy as the modeled environment evolves, and writing them is labor intensive (up to multiple person-years[6]). Real environments, such as the Java virtual machine or Windows OS, have bugs that make them vulnerable to attackers, so trusting them blindly can be risky.

Another approach is to symbolically execute the lowest common denominator: machine code. The S2E engine (http://s2e.epfl.ch) does this and can therefore symbolically execute programs together with arbitrary execution environments. S2E presents the developer with a virtual machine (currently x86 or ARM) in which a developer can install an entire software stack, in its executable binary form (for example, a full Windows system). S2E executes the software symbolically "in vivo," as the software operates in its live, real environment.

In order to scale, S2E employs *selective SE*, in which execution seamlessly weaves between symbolic and concrete mode. This automatically avoids exploring paths that aren't relevant to the currently explored path. For example, if a program calls `malloc()`, which is part of the environment, S2E will return $\lambda \in$ {valid pointer, `NULL`} instead of symbolically executing `malloc()`. This symbolic return value is sufficient to capture the memory allocator's behavior.

A real environment, though, includes hardware devices. So that software doesn't have to make assumptions about such devices, S2E provides *symbolic hardware*. A nice side effect is that testing can occur even when the devices aren't present—this is how we automatically tested dozens of Windows device drivers and found many bugs. We also developed on S2E a multipath performance profiler that found performance bugs in widely used software, a reverse-engineering tool for proprietary software, and a testing tool for distributed systems. Other groups worldwide are using S2E for a range of other analyses, including security-oriented ones.

Real Users ⇒ Social Challenges

Even if all the technical challenges of SE were resolved, getting developers to use such tools is difficult—this is the "social scalability" challenge. Understanding SE results is difficult because we can't just attach a debugger to the running program and visualize thousands of simultaneously executing paths. State merging, as I described earlier, makes this even more difficult. There's still much left to do to integrate SE into a developer's toolbox.

At EPFL, we're working on making SE-based automated testing accessible to everyone everywhere through the CodeTickler Web service (www.codetickler.org). We aim to let users upload an executable to the service, click on Test, and check it for undesired behaviors (memory errors, data races, resource leaks, and so on). Currently, the service can test Windows device drivers. It produces detailed, executable traces for every path that leads to a failure. These traces can reproduce the bugs in a debugger, one path at a time, enabling users to understand the bugs and fix them quickly. It is also possible for end users to check untrusted drivers before installing them.

To address other related social challenges, we developed techniques for employing users' executions to verify software (as in RaceMob[7]), preserving user privacy in such crowdsourced systems,[8] and automating debugging with techniques such as execution synthesis.[9]

When faced with the tests-versus-proofs conundrum, the most promising path for the modern developer is to combine formal methods and traditional testing practice. By devising tools that creatively combine tests with proofs, we can improve today's software, despite its increasing complexity. If at the same time we make these tools fully automatic, we can also improve developers' productivity, thus surmounting the inherent social challenge of changing how software is written. ■

References

1. X. Wang et al., "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior," *Proc. 24th ACM Symp. Operating Systems Principles* (SOSP 13), 2013, pp. 260–275.
2. E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," *Proc. 2013 Int'l Conf. Software Eng.* (ICSE 13), 2013, pp. 122–131.
3. C. Cadar, D. Dunbar, and D.R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. 8th Usenix Conf. Operating System Design and Implementation* (OSDI 08), 2008, pp. 209–224.
4. V. Kuznetsov et al., "Efficient State Merging in Symbolic Execution," *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 12), 2012, pp. 193–204.
5. S. Bucur et al., "Parallel Symbolic Execution for Automated Real-World Software Testing," *Proc. 6th Conf. Computer Systems* (EuroSys 11), 2011, pp. 183–198.
6. T. Ball et al., "Thorough Static Analysis of Device Drivers," *Proc. 1st ACM/SIGOPS EuroSys European Conf. Computer Systems* (EuroSys 06), 2006, pp. 73–85.
7. B. Kasikci, C. Zamfir, and G. Candea, "RaceMob: Crowdsourced Data Race Detection," *Proc. 24th ACM Symp. Operating Systems Principles* (SOSP 13), 2013, pp. 406–422.
8. S. Andrica and G. Candea, "Mitigating Anonymity Concerns in Self-Testing and Self-Debugging Programs," *Proc. Int'l Conf. Autonomic Computing*, 2013, pp. 259–264.
9. C. Zamfir and G. Candea, "Execution Synthesis: A Technique for Automated Software Debugging," *Proc. 5th European Conf. Computer Systems* (EuroSys 10), 2010, pp. 321–334.

George Candea leads the Dependable Systems Lab and is an associate professor of computer science in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne (EPFL). Contact him at george.candea@epfl.ch.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

IEEE computer society

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.
MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.
COMPUTER SOCIETY WEBSITE: www.computer.org

Next Board Meeting: 4–7 February 2014, Long Beach, Calif., USA

EXECUTIVE COMMITTEE
President: Dejan S. Milojicic
President-Elect: Thomas M. Conte; **Past President:** David Alan Grier; **Secretary:** David S. Ebert; **Treasurer:** Charlene ("Chuck") J. Walrad; **VP, Educational Activities:** Phillip Laplante; **VP, Member & Geographic Activities:** Elizabeth L. Burd; **VP, Publications:** Jean-Luc Gaudiot; **VP, Professional Activities:** Donald F. Shafer; **VP, Standards Activities:** James W. Moore; **VP, Technical & Conference Activities:** Cecilia Metra; **2014 IEEE Director & Delegate Division VIII:** Roger U. Fujii; **2014 IEEE Director & Delegate Division V:** Susan K. (Kathy) Land; **2014 IEEE Director-Elect & Delegate Division VIII:** John W. Walz

BOARD OF GOVERNORS
Term Expiring 2014: Jose Ignacio Castillo Velazquez, David. S. Ebert, Hakan Erdogmus, Gargi Keeni, Fabrizio Lombardi, Hironori Kasahara, Arnold N. Pears
Term Expiring 2015: Ann DeMarle, Cecilia Metra, Nita Patel, Diomidis Spinellis, Phillip Laplante, Jean-Luc Gaudiot, Stefano Zanero
Term Expriring 2016: David A. Bader, Pierre Bourque, Dennis Frailey, Jill I. Gostin, Atsuhiro Goto, Rob Reilly, Christina M. Schober

EXECUTIVE STAFF
Executive Director: Angela R. Burgess; **Associate Executive Director & Director, Governance:** Anne Marie Kelly; **Director, Finance & Accounting:** John Miller; **Director, Information Technology & Services:** Ray Kahn; **Director, Membership Development:** Eric Berkowitz; **Director, Products & Services:** Evan Butterfield; **Director, Sales & Marketing:** Chris Jensen

COMPUTER SOCIETY OFFICES
Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928
Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614
Email: hq.ofc@computer.org
Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720
Phone: +1 714 821 8380 • **Email:** help@computer.org

MEMBERSHIP & PUBLICATION ORDERS
Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • **Email:** help@computer.org
Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan • Phone: +81 3 3408 3118 • **Fax:** +81 3 3408 3553 • **Email:** tokyo.ofc@computer.org

IEEE BOARD OF DIRECTORS
President: J. Roberto de Marca; **President-Elect:** Howard E. Michel; **Past President**: Peter W. Staecker; **Secretary:** Marko Delimar; **Treasurer:** John T. Barr; **Director & President, IEEE-USA:** Gary L. Blank; **Director & President, Standards Association:** Karen Bartleson; **Director & VP, Educational Activities:** Saurabh Sinha; **Director & VP, Membership and Geographic Activities:** Ralph M. Ford; **Director & VP, Publication Services and Products:** Gianluca Setti; **Director & VP, Technical Activities:** Jacek M. Zurada; **Director & Delegate Division V:** Susan K. (Kathy) Land; **Director & Delegate Division VIII:** Roger U. Fujii

revised 17 Dec. 2013

IEEE