On Narrowing the Gap between Verification and Systematic Testing

Maria Christakis

Abstract: Our work on narrowing the gap between verification and systematic testing has two directions: (1) complementing verification with systematic testing, and (2) pushing systematic testing toward reaching verification. In the first direction, we explore how to effectively combine static analysis with systematic testing, so as to guide test generation toward properties that have not been previously checked by a static analyzer in a sound way. This combination significantly reduces the test effort while checking more unverified properties. In the second direction, we push systematic testing toward checking as many executions as possible of a real and complex image parser, so as to prove the absence of a certain class of errors. This verification attempt required no static analysis or source code annotations; our purely dynamic techniques targeted the verification of the parser implementation, including complicated assembly patterns that most static analyses cannot handle.

ACM CCS: Software and its engineering → Software creation and management → Software verification and validation → Formal software verification
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Keywords: Verification, static analysis, systematic testing, dynamic test generation.

1 Introduction

Software systems are ubiquitous in modern life. Their robustness and reliability is, therefore, vital in our society and constitutes a primary goal of the computer-science community. Formal verification and automated systematic testing are two fundamental research areas of computer science, aiming to ensure software correctness and identify code issues as early as possible.

Verification has been studied for approximately five decades and is increasingly applied in industrial software development to detect errors. Verification techniques allow developers to prove that a program satisfies a set of desired properties, for instance, the absence of runtime errors. So far, verification tools have been so effective in detecting errors in real-world programs that they are increasingly and routinely used in many software development organizations. In fact, there is a wide variety of such tools, targeting mainstream programming languages and ranging from relatively simple heuristic tools [2], over abstract interpreters [20] and software model checkers [3, 5], to verifiers based on automatic theorem proving [23, 4].

Over the last ten years, there has been revived interest in systematic testing, and in particular, in testing techniques that rely on symbolic execution [29], introduced more than three decades ago [25, 9, 10]. Recent significant advances in constraint satisfiability and the scalability of simultaneous concrete and symbolic executions have brought systematic dynamic test generation [26, 7] to the spotlight, especially due to its ability to achieve high code coverage and detect errors deep in large and complex programs. As a result, dynamic test generation is having a major impact on many research areas of computer science, for instance, on software engineering, security, computer systems, debugging and repair, networks, and education.

Sound software verification over-approximates the set of possible program executions, as shown in Fig. 1, to prove the absence of errors in a program. Due to this over-approximation, sound verification typically generates spurious warnings about executions that are not erroneous, or even possible, in a program, thus, making it difficult to identify any real errors in the code. Systematic testing, on the other hand, typically under-approximates the set of possible program executions with the purpose of proving the existence of errors in the program [28]. As a result, systematic testing typi-

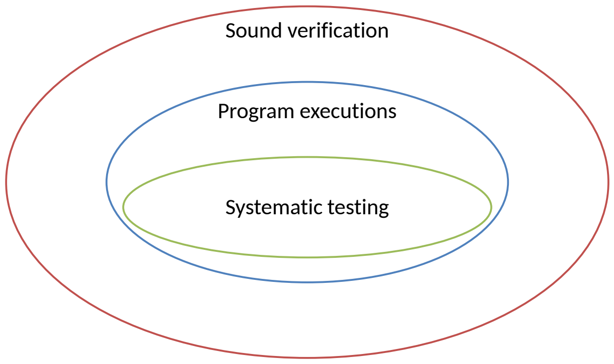

Figure 1: Sound verification over-approximates the set of all possible program executions, whereas systematic testing typically under-approximates this set.

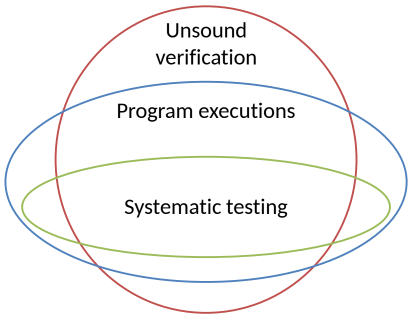

Figure 2: Unsound verification typically neither over-approximates nor under-approximates the set of all possible program executions.

cally misses errors in the code. Our work on narrowing this gap between software verification and systematic testing [11] focuses on two directions: (1) complementing verification with systematic testing, and (2) pushing systematic testing toward verification.

In the first direction, we complement verification (or static analysis) with systematic testing, to maximize software quality while reducing the test effort. In particular, we precisely define the correctness guarantees that verifiers provide, such that they can be effectively compensated for by dynamic test generation. At the same time, we enhance systematic testing techniques with better oracles, and enable these techniques to consider factors that affect the outcome of such oracles but were previously ignored. This research direction enables the detection of more software errors, earlier in the development process, and with fewer resources.

In the second direction, we explore how far systematic testing can be pushed toward reaching verification of real applications. Specifically, we assess to what extent the idea of reaching verification with systematic testing is realistic, in the scope of a particular application domain, namely, that of binary image parsers. This research direction sheds light to the potential of dynamic test generation in ensuring software correctness.

2 Complementing Verification with Systematic Testing

Modern software projects use a variety of *unsound* static program analysis techniques to detect errors, most of which do not check all possible executions of a program, as shown in Fig. 2. For instance, such techniques often fail to verify certain program properties (due to the complexity of these properties), or they verify some program paths under unsound assumptions (such as the absence of arithmetic overflow), which simplify the analysis but might not hold for all executions.

Making such assumptions is customary in static analysis to improve the precision (i.e., number of spurious warnings), performance, and automation of the analysis, and because some program features simply elude static checking [30]. That is, most practical static analyses sacrifice soundness in favor of other important qualities.

Despite these compromises, static analyzers find real errors in real code. However, as a result of these compromises, it is not clear what guarantees a static analysis actually provides about program correctness. This means that users who are not familiar with an analyzer's implicit compromises do not know how to interpret the absence of warnings about their code. It is also not clear how to use systematic testing to check exactly those properties that are not soundly verified by a static analysis. Consequently, software engineers need to test their programs as if no static analysis were applied, which is inefficient and requires large test suites.

Until our work, various approaches had combined verification and testing [21, 22, 24], but mainly to determine whether a warning emitted by a verifier is a false positive. However, these approaches do not take into account that unsound static analyses might generate false negatives (that is, they might miss errors), and therefore, do not address compromises of verifiers. In other words, testing aims to target only program executions for which a verification warning has been emitted, thus ignoring executions that have not been previously checked by a static analyzer due to its unsoundness. This is shown by the shaded area in Fig. 3 representing the executions that were not statically proven correct and are targeted by testing.

To address this problem, we developed a technique for combining verification and systematic testing, which guides the latter not only toward program executions for which a verification warning has been emitted, but also toward executions that unsound verification has missed. The program executions that systematic testing aims to cover with our technique are depicted by the shaded areas in Fig. 4.

In particular, we proposed a tool architecture, presented in Fig. 5, that (1) combines multiple, complementary static analyzers that check different properties and make different unsound assumptions, and (2) complements

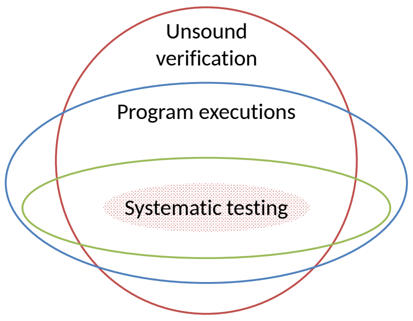

Figure 3: In existing work, systematic testing targets only those program executions for which a static verification warning has been emitted (shaded area), thus ignoring the executions that unsound verification has missed.

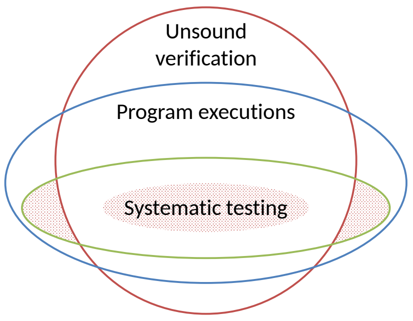

Figure 4: In our work, systematic testing targets those program executions for which a verification warning has been emitted as well as those that unsound verification has missed (shaded areas).

static analysis with systematic test generation to cover those properties that have not been checked statically.

The first stage of the architecture allows the user to run an arbitrary number (possibly zero) of static analyzers. An analyzer reads the program, which might also contain results of prior static analysis attempts, and tries to verify any properties that have not already been proven by upstream tools. Each analyzer also records its verification results in the program, which serves as input to the next downstream tool. The intermediate versions of the program precisely track which properties have been fully verified and which remain to be validated.

In the second stage, we apply dynamic test generation to automatically generate test cases from the program and the results of static analysis. In particular, the properties that remain to be checked as well as the assumptions made by static analyzers occur, in the form of runtime checks, in an instrumented version of the program. The resulting instrumented program can then be fed to one or more test generation tools. Our instrumentation causes the symbolic execution of these tools to generate the constraints and test data that exercise exactly the properties that have not been statically verified (in a sound way), thus reducing the size of the generated test suites.

A key originality of this tool architecture is that it makes explicit which properties have been checked statically and under which assumptions. Therefore, the correctness guarantees provided by static analyzers are documented precisely, and can guide dynamic test generation toward those properties that are not verified yet, leading to smaller and more effective test suites. These test suites will consist of a series of successful test cases that will boost the user's confidence about the correctness of their programs or concrete counterexamples that reproduce an error. Moreover, by automatically generating tests from the explicit verification results of static analyzers, our technique makes the degree of static analysis configurable; it may range from zero to complete. This allows developers to stop the static verification cycle at any time, which is important in practice, where the effort that a developer can devote to static analysis is limited.

By developing this architecture, we investigated the following scientific topics:

How to design an annotation language that supports both verification and systematic testing [16]. The main virtues of our annotations are that they are simple and easy to support by a wide range of static and dynamic tools, expressive, and well suited for test generation.

What the compromises of mainstream verifiers are and how to make these compromises explicit [18]. We used our annotations to encode typical compromises made by deductive verifiers. We also encoded most soundness compromises in a widely used, commercial static analyzer. We measured the impact of its unsound assumptions on several open-source projects, which constituted the first systematic effort to document and evaluate the sources of unsoundness in an analyzer. These results can guide users of static analyzers in using them fruitfully, for instance, in deciding how to complement static analysis with testing, and assist designers of static analyzers in finding good trade-offs for their tools.

How to combine verification and systematic testing to maximize code quality and minimize the test effort. We presented a technique for effectively reducing redundancies with static analysis when complementing its verification results by test generation [19]. Our main contribution is a code instrumentation that causes test generation to abort tests that lead to verified executions, prune parts of the search space, and prioritize tests that lead to unverified executions. To increase the usability of our technique, we also extended the IDE of a known verifier to seamlessly integrate test generation, among other approaches, for diagnosing verification errors [15]. We investigated how to present the results of these approaches in the IDE without overwhelming the user with too much information.

How to generate tests for program properties that are difficult to verify and lie beyond the capabilities of systematic testing [12, 17]. We proposed approaches for efficiently generating test oracles (in the form of runtime checks) for rich properties that are difficult to statically verify, and test inputs for thoroughly evaluating these oracles.

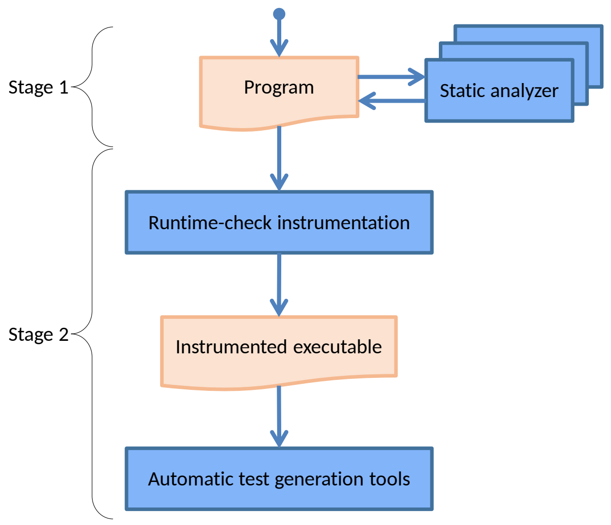
Stage 1
Stage 2
Program
Static analyzer
Runtime-check instrumentation
Instrumented executable
Automatic test generation tools

Figure 5: Tool architecture for complementing static verification with systematic testing.

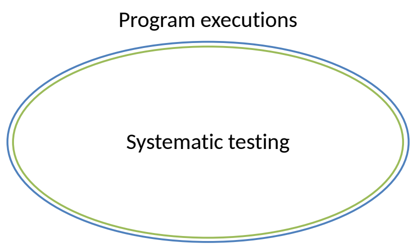

Figure 6: Assessing whether it is realistic to push systematic testing to cover all possible program executions.

3 Pushing Systematic Testing Toward Verification

Systematic dynamic test generation has been implemented in many popular tools over the last decade [8, 31, 27, 33, 6, 32, 1]. Although effective in detecting program errors, such testing tools have never been pushed toward proving that a large and complex application is free of certain classes of errors. In this research direction, we assess to what extent reaching verification with systematic testing is feasible in practice in the scope of binary image parsers, that is, we assess whether Fig. 6 is realistic. Specifically, we used and enhanced systematic dynamic test generation to get closer to proving memory safety of the ANI Windows image parser.

The ANI parser is responsible for processing structured graphics files in order to display “ANImated” cursors and icons. Such animated icons are ubiquitous in practice (like the spinning ring or hourglass on Windows), and their domain of use ranges from web pages, instant messaging, and e-mails, to presentations and videos. The choice of this parser was motivated by the fact that in 2007 a critical out-of-band security patch was released for code in this parser costing Microsoft and its users millions of dollars. The ANI parser is included in all distributions of Windows, that is, used on more than a billion PCs, and has been tested for years. Given the ubiquity of animated icons, our goal was to determine whether the ANI parser is now free of security-critical errors.

In the context of this parser, we showed how systematic dynamic test generation can be applied, extended, and automated toward program verification. This was the first application of dynamic test generation to verify as many executions as possible of a real, complex, security-critical program, and more importantly, the first attempt to prove that an operating-system (Windows or other) image parser is free of security-critical errors. We are also not aware of any past attempts at verification without using any static analysis. All the techniques and tools used in this work were exclusively dynamic, thus seeking verification of the execution of the parser, including complicated assembly code patterns that most static analysis tools cannot handle. During this verification process, we found several security vulnerabilities in the parser, which have been fixed in the latest Windows versions. Overall, when excluding the code parts that were actually memory unsafe, we were able to prove that the ANI parser is memory safe, that is, free of any buffer-overflow security vulnerabilities, modulo the soundness of our tools and a few additional assumptions that we had to make [14].

These results required a high level of automation in our tools and verification process although a few key steps were performed manually. To achieve further automation, we then devised a new compositional test strategy for automatically and dynamically decomposing large programs such that code coverage and error detection are increased in significantly less testing time in comparison to the state-of-art test strategy used in production at Microsoft [13].

4 Conclusion

My vision for the future is to enable developers to rely on and benefit from a wide range of tools and techniques that improve their development workflow as well as the quality of their software. This cannot be achieved by simply providing developers with yet another tool. I plan to continue leveraging both novel and existing techniques, such that they complement each other symbiotically, to streamline the software development process and alleviate its bottlenecks, such as bug finding, manually augmenting test suites, or code reviewing. I consider our work until now a first milestone in this journey.

Acknowledgments. I am grateful to Peter Müller, Valentin Wüstholz, and Patrice Godefroid; this work would not have been possible without them. I would also like to thank my other collaborators, Patrick Emmisberger and Rustan Leino.

Literature

[1] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *TSE*, 36:474–494, 2010.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.

[3] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[4] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *STTT*, 9:505–525, 2007.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.

[7] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.

[9] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE*, pages 1066–1071. ACM, 2011.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *CACM*, 56:82–90, 2013.

[11] M. Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, Switzerland, 2015.

[12] M. Christakis, P. Emmisberger, and P. Müller. Dynamic test generation with static fields and initializers. In *RV*, volume 8734 of *LNCS*, pages 269–284. Springer, 2014.

[13] M. Christakis and P. Godefroid. IC-Cut: A compositional search strategy for dynamic test generation. In *SPIN*, volume 9232 of *LNCS*, pages 300–318. Springer, 2015.

[14] M. Christakis and P. Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *VMCAI*, volume 8931 of *LNCS*, pages 373–392. Springer, 2015.

[15] M. Christakis, K. R. M. Leino, P. Müller, and V. Wüstholz. Integrated environment for diagnosing verification errors. In *TACAS*, volume 9636 of *LNCS*, pages 424–441. Springer, 2016.

[16] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[17] M. Christakis, P. Müller, and V. Wüstholz. Synthesizing parameterized unit tests to detect object invariant violations. In *SEFM*, volume 8702 of *LNCS*, pages 65–80. Springer, 2014.

[18] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 336–354. Springer, 2015.

[19] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155. ACM, 2016.

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[21] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[22] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17:1–37, 2008.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[24] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994. ACM, 2011.

[25] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25:30–37, 2008.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[27] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.

[28] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56. ACM, 2010.

[29] J. C. King. Symbolic execution and program testing. *CACM*, 19:385–394, 1976.

[30] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58:44–46, 2015.

[31] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.

[32] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.

[33] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

Dr. Maria Christakis is currently a lecturer (assistant professor) in the School of Computing at the University of Kent, England. She was previously a post-doctoral researcher at Microsoft Research Redmond, USA. She received her Ph.D. from the Department of Computer Science of ETH Zurich, Switzerland in the summer of 2015, and feels very fortunate to have been advised by Peter Müller. Maria was awarded with the ETH medal for an outstanding doctoral thesis. She completed her Bachelor's and Master's degrees at the Department of Electrical and Computer Engineering of the National Technical University of Athens, Greece.
Maria's goal is to develop theoretical foundations and practical tools for building more reliable and usable software and increasing developer productivity. She is mostly interested in software engineering, programming languages, and formal methods. Maria particularly likes investigating topics in automatic test generation, software verification, program analysis, and empirical software engineering. Her tools and techniques explore novel ways in writing, specifying, verifying, testing, and debugging programs in order to make them more robust while at the same time improving the user experience.

Address: School of Computing, University of Kent, UK
E-Mail: M.Christakis@kent.ac.uk