Research Statement:
Software Correctness at Scale through Testing and Verification

Leonidas Lampropoulos
University of Maryland, University of Pennsylvania

15-slide summary of this statement

Software correctness is becoming an increasingly important concern as our society grows more and more reliant on computer systems. From a discrepancy in units of measurement causing the Mars Climate Orbiter crash 20 years ago to a missing bounds check causing Heartbleed, even the simplest of software errors have devastating financial or security implications. How can we find errors in real-world applications? How can we guarantee that such errors don't exist? To even begin to answer these questions we need a *specification*: a description of what it means for a piece of code to be correct, stated either explicitly (e.g. my private data are never leaked to third parties) or implicitly (e.g. this code will not terminate with an uncaught exception). My goal is to develop efficient ways to write, debug, and reason about software and specifications at scale.

In my research so far, I have focused on two techniques and their interplay: **property-based testing** and **mechanized formal verification**. In property-based testing, a developer specifies expected system behavior as a collection of *properties*—executable boolean predicates over structured test data—and a tool synthesizes inputs to repeatedly test these properties. Testing can quickly uncover errors in complex software [1; 2], but it cannot guarantee their absence. Formal verification, on the other hand, provides strong correctness guarantees that a system conforms to a desired specification. However, despite recent success stories [3; 4], the verification process is still expensive in both time and effort. Together, testing and verification are a match made in heaven. The properties to be tested are supplied "for free" in the form of the specification to be verified. In turn, testing can provide confidence in auxiliary lemmas and uncover useful program invariants, before a full proof is attempted.

Armed with this realization, I integrated both techniques in a single framework. I brought the benefits of testing to the Coq proof assistant, a widely-used interactive theorem prover, by developing QuickChick [ITP '15], a modern property-based random testing framework. QuickChick has since been successfully used in a variety of projects, from designing and implementing a fully-verified web server (DeepWeb), to a framework for reasoning about LLVM programs (Vellvm). QuickChick is the subject of the fourth volume of the popular **Software Foundations** series of online textbooks [SFv4], being taught both as a standalone summer school module (DeepSpec Summer School '17 & '18), and as part of a broader course on Program Analysis and Understanding (UMD, CMSC 631). Since then I have mainly focused on techniques for efficient random generation of constrained test data [POPL '17; Haskell '17; POPL '18; ICFP '18; OOPSLA '19].

In the immediate future, I plan to work on the principles and practice of property-based testing, and to use the resulting advances to facilitate large-scale verification efforts. My focus will be on two real-world security applications: enforcing programmable security policies (called *micropolicies*) for the RISC-V ISA [5], and extending the full C programming language to enforce spatial

memory safety (*Checked C*). I have already worked on testing and formalizing models of these systems [ICFP '13; JFP '16; POST '19]; During the course of this work, it became abundantly clear that automatic testing techniques were (1) desperately needed, as a lot of verification effort was wasted on futile proof attempts with a faulty design (which testing could have revealed early on), and (2) inadequately advanced to deal with the intricate invariants involved without significant user effort. To scale these developments to real-world analogues, I plan to develop efficient semi- or fully- automatic techniques targeting all aspects of testing, from generating data to minimizing counterexamples. In particular, I'm currently working on *smarter* techniques, that leverage coverage or other runtime information to produce a better distribution of test inputs, and on more *principled* techniques, exploring the combinatorial nature of algebraic data types to maximize the impact of each input.

Longer term, I want to bring the advantages of property testing and verification to the mainstream. The largest barrier to entry is *writing the specifications* themselves: no matter how efficient the testing or verification infrastructure is, it still requires specifications that abstract away from traditional unit tests to capture the deeper meaning of a piece of code. Developing ways to automatically derive specifications, as well as teaching how to come up them in the first place, are both essential steps towards that goal.

It is equally important to transfer techniques from the world of proof assistants, where testing and verification can be integrated tightly, to a setting that is more commonly used in practice. To that end, I plan to address the issue of program correctness in a dynamically typed language: *Python*. Python offers an interesting trade-off from a testing perspective because it is notoriously slow. Therefore, traditional random testing techniques that rely on executing a program on concrete inputs many times are less attractive. Symbolic techniques on the other hand execute an instrumented program, gather information about the paths taken, and use external tools like SMT solvers to construct inputs that are guaranteed to explore new paths. In most settings, the overhead of such techniques means they can run order of magnitude fewer tests; in a interpreted context where execution is slower, this overhead is much more palatable and presents an interesting opportunity for future work.

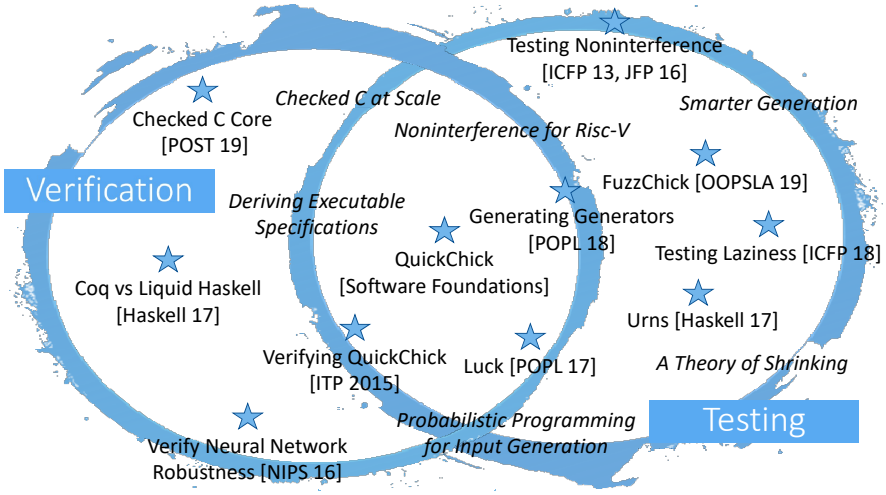

Figure 1: My prior and future research on testing and verification. Both prior and planned future research (in *italics*) will be discussed in detail in the rest of the statement.

Property-Based Random Testing in Coq

Property-based random testing (PBRT) tools operate on properties: executable specifications over structured data [6]. Given such a property, a PBRT tool generates random inputs and executes the property repeatedly; should a counterexample be found, it is minimized through a process called *shrinking* before it is reported to the user. All three components—generation, executable specification, and shrinking—are necessary for PBRT to succeed. Automating each one comes with its own set of challenges.

Generation. In a PBRT tool, the generation of inputs is determined by *generators*: programs that describe a distribution of test data. Much of the time, these generators can be inferred automatically based on existing type information. However, most properties of interest involve *sparse preconditions*, imposing restrictive constraints on their inputs. For such properties, generating inputs using only type information, and then filtering out any that fail to satisfy a precondition, is a recipe for extremely inefficient testing. For example, if we wanted to test the correctness of compiler optimizations—that is, that a program compiled with and without optimization yields the same output—then we would need to generate programs for which compilation succeeds. If we attempt to generate such programs by generating random ones (based on, e.g., a grammar) and the filtering out those that fail to compile, we would waste the vast majority of testing time generating inputs that will just be discarded. In such cases, for random testing to be effective, users have to provide custom generators for random data that satisfy a given precondition by construction. Writing such a generator by hand, however, can be extremely time consuming and error prone. A better approach is to *derive* such generators using the precondition as a guide.

Following this approach, I developed Luck [POPL '17], a domain-specific functional language with a rigorous formal foundation. In Luck, users write a precondition as a standard executable predicate P. Instead of generating completely random data and testing whether P holds, Luck walks over the definition of P and instantiates each unknown variable x at the first point where it meets a constraint involving x. Using lightweight annotations, Luck users can decorate P to control both the distribution of such instantiations, and to further defer them until more constraints are encountered, using an ad-hoc constraint solver to take them all into account. I then further generalized and adapted this approach for QuickChick [POPL '18]. I developed a derivation procedure that, given a predicate in the form of an inductively defined Coq relation, synthesizes generators for data that satisfy this relation. These generators come with certificates of correctness: they are sound and complete with respect to the predicate they were derived from. Moreover, the generators are regular programs: they can be read, modified, and, more importantly, composed together by users that want to further infuse domain-specific knowledge. This derivation procedure is now part of the standard QuickChick release, being used to quickly gain confidence in complex developments.

A pervasive theme throughout both of these projects is the notion of *backtracking*. When test data can be produced by multiple different generation strategies, they need to be composed together into a single strategy that chooses one of them at random. Often, however, generation strategies can fail; the only solution is to backtrack and pick a different one. More importantly, for the user to control the distribution of generated test data, different strategies need to be combined using user-specified weights. To tackle this issue I developed the Urn [Haskell '17], an efficient data structure for sampling from updateable discrete distributions, with an optimized implementation available online. While the Urn is an efficient implementation of backtracking, there remains the question of exposing high-level *programming abstractions* that facilitate writing and reasoning about backtracking generators. I'm currently working on such a set of abstractions by extending QuickChick's underlying generation monad with backtracking support.

This year, I also pursued a different approach to enhance the efficiency of automatic generation using coverage information which gave rise to Coverage Guided, Property Based Testing [OOPSLA '19]. Coverage-guided fuzz testing, exemplified by the popular AFL fuzzer [2], can be viewed as a very restrictive form of property-based random testing, where the inputs are always streams of binary data and the property is always "the system under test does not crash unexpectedly". Fuzzing has found a lot of success by mutating binary data coming into programs in this restricted setting. But it is far more advantageous to mutate test data at the algebraic datatype level, exploiting the additional structure and type information available. I implemented this technique on top of QuickChick, demonstrating orders of magnitude better bug-finding efficiency compared to existing automatic type-based approaches, but also orders of magnitude worse than expertly handwritten random generators. This technique shows promise, but there is also much room for improvement.

Executable Specifications. Another component that is necessary for property-based random testing is the property itself. In Coq, theorems are usually specified using *inductive relations*. Inductive relations can describe more behaviors than traditional functional computations in proof assistants, from potentially nonterminating computations to nondeterministic or concurrent semantics. To make use of property-based testing in such a setting, users currently need to provide an executable variant of their specifications, ideally with a proof linking the two variants together. My goal is to reuse the backtracking framework discussed above to write and reason about decidability procedures for inductive relations.

Shrinking. A final aspect of property-based testing that needs to be automated deals with counterexample *minimization*. Just like generating data that satisfy preconditions directly can be exponentially more efficient than generating arbitrary data and filtering them, the same is true for minimizing reported counterexamples that need to satisfy the same invariants. A related problem is that of counterexample *deduplication*; when testing a piece of software for a non-trivial amount of time, usually many different bugs are discovered. Being able to characterize which counterexamples correspond to the same underlying problem in the code helps both reduce the effort on the end user (that otherwise has to sort through numerous instances of the same issue), and improve the efficiency of testing (by avoiding input data that exhibit identical erroneous behavior). Existing techniques in the coverage-based fuzzing world, like comparing stack trace hashes or coverage profiles, fall way short of discriminating instances of the same error correctly [7]. By integrating testing and theorem proving in a single framework, there is a unique opportunity to study this problem as we have ground truth: we can establish the correctness of a program via verification and observe the outcome of different techniques in a controlled setting with artificially injected errors.

Specifications at Scale

Noninterference for RISC-V. Towards the start of my PhD, I was part of a project to design security monitors with the help of random testing [ICFP '13]. These monitors enforce *noninterference*: intuitively, secret data cannot affect public channels. I later scaled up this infrastructure to more complex and realistic monitors [JFP '16], with much more advanced security features. I'm currently working towards enforcing security policies for RISC-V, working with Draper Labs [8] and Dover Microsystems [9] as industry partners. Using FORVIS as a starting point (a reference implementation of RISC-V in Haskell that is a candidate to be *the* specification of the RISC-V ISA) [10], I have developed a co-processor that serves as a monitor for programmable security

policies. The policy I'm most interested in enforcing is a stronger variant of noninterference, where the set of data that is considered secret can change throughout the execution. For this application, the primary focus is to increase confidence in the design through testing, with formal verification as a secondary goal.

Memory Safety for C. In the beginning of my postdoc, I jumped on a chance to formalize a proof of memory safety for Checked C [POST '19]. Checked C is a backwards-compatible extension of C in the form of a type system that prevents *spatial safety violations* (e.g. buffer overflows). My contribution to this project was a formalization of the type system for a core calculus—a subset of C extended to mix safe and unsafe pointers, and a proof that checked code cannot be blamed for memory safety violations. The next step in this project is to extend the development to the real C language by extending CompCert, an optimizing verified C compiler that precisely captures legal C behaviors. The translation between Checked C and C introduces runtime checks in appropriate places and ensures that for well-typed Checked C programs spatial safety is never violated. The scale of this verification effort is much larger than the one for the core calculus and would greatly benefit from a testing infrastructure that will nip futile proof attempts in the bud by uncovering flaws early. As CompCert is verified using Coq, QuickChick is the natural choice for developing this testing infrastructure. It is a perfect application to stretch its testing capabilities to the limit and to perform research on automating various aspects of testing that will have immediate impact in practice.

Non-Functional Specifications. Most of the work I have done so far has focused on writing, debugging and reasoning about specifications of *functional correctness*: input-output behaviors. A particularly interesting and important problem is to apply these same techniques to domains that are traditionally thought of as not amenable to testing or verification. For example, I have already participated in a project to verify robustness properties of neural networks using formal methods [NIPS '16] and to check the strictness behavior of Haskell programs [ICFP '18].While both of these efforts dealt with non-functional specifications in an ad-hoc basis, exploring more principled approaches for such undertakings is an exciting avenue of future work.

Improving Correctness for Python. My prior work is mostly centered around QuickChick, exploring the synergy of testing and verification in a proof assistant. Most practical software, however, is not written in a proof assistant . In fact, most of the time applications are not even written in a single framework: for instance, it is extremely common to see Python code interfacing with native C/C++ code for critical components. At the same time, the dynamic features of Python make it error prone, with undetected type mismatches leading to surprising runtime errors, and with boundary mismatches in the foreign function interface between Python and C/C++ leading to wrong results or memory corruption. Worse, Python's inefficiency also severely limits the effectiveness of automatic testing approaches like fuzz testing, greatly reducing the number of tests that can be executed in a reasonable time. I aim to extend the automatic testing capabilities of the Python ecosystem [11; 12], using smarter symbolic or statistical generation techniques that don't rely on program execution.

Machine Learning and Testing. The techniques I developed for QuickChick bring together ideas from traditionally disparate fields, bridging together functional logic programming with constraint solving and coverage-guided fuzzing with property-based testing. Still, there is a lot of

potential for cross-fertilization of ideas between even more fields of Computer Science. For example, the Luck generation approach resembles on-the-fly model checking in the formal methods world [13], and Markov Chain Monte Carlo sampling methods in probabilistic programming [14]. I'm interested in mixing together such ideas from the formal methods, probabilistic programming, and machine learning communities to improve both the state-of-the-art in random testing as well as these fields themselves. For example, I aim to use probabilistic inference to reason about random generators (after all, a random generator *is* a probabilistic program), and to use reinforcement learning techniques to tune the distributions of the semantic mutations of coverage-guided fuzzing, which can significantly affect testing performance.

Specifications for Non-Experts. Finally, I want to make property-based random testing and verification more accessible to non-experts. The trend towards increased automation in recent years is a significant first step, but still does not address one main issue: the need to write specifications in the first place. Random testing techniques can be used to uncover patterns of program behavior and propose such properties. At the same time, it is important to teach the art of writing specifications and showcase the benefits of test-driven methodologies to foster mainstream adoption. To that end, I was the principal author of the fourth volume in the popular Software Foundations series titled **"QuickChick: Property-Based Testing for Coq"** [SFv4], exploring interactions between specifications, testing, and verification in the context of a proof assistant.

Conclusion

I envision a future where every piece of software is accompanied by a formal specification of its behavior, where testing provides confidence throughout the design process and verification provides stronger guarantees for critical components. I am actively working towards that future by making it easier to reap the benefits of specification-driven development, automating and improving all aspects of property-based testing, and facilitating large-scale design and verification of security applications.

Publications

[OOPSLA '19] **Lampropoulos, L.**, Hicks, M., Pierce, B. C., "Coverage Guided, Property Based Testing". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. Oct. 2019.

[POST '19] Ruef, A., **Lampropoulos, L.**, Sweet, I., Tarditi, D., Hicks, M., "Achieving Safety Incrementally with Checked C". In: *Proceedings of the Symposium on Principles of Security and Trust (POST)*. Apr. 2019.

[ICFP '18] Foner, K., Zhang, H., **Lampropoulos, L.,** "Keep your laziness in check". In: *Proceedings of the ACM International Conference on Functional Programming (ICFP)* (2018).

[POPL '18] **Lampropoulos, L.**, Paraskevopoulou, Z., Pierce, B. C., "Generating Good Generators for Inductive Relations". In: *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)* (2018).

[SFv4] **Lampropoulos, L.**, Pierce, B. C., *QuickChick: Property-Based Testing In Coq*. Software Foundations series, volume 4. Electronic textbook, Aug. 2018.

[POPL '17] **Lampropoulos, L.**, Gallois-Wong, D., Hritcu, C., Hughes, J., Pierce, B. C., Xia, L., "Beginner's Luck: a language for property-based generators". In: *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. 2017.

[Haskell '17] **Lampropoulos, L.**, Spector-Zabusky, A., Foner, K., "Ode on a Random Urn (Functional Pearl)". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 2017.

[Haskell '17b] Vazou, N., **Lampropoulos, L.**, Polakow, J., "A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 2017.

[NIPS '16] Bastani, O., Ioannou, Y., **Lampropoulos, L.**, Vytiniotis, D., Nori, A. V., Criminisi, A., "Measuring Neural Net Robustness with Constraints". In: *Advances in Neural Information Processing Systems (NIPS)*. 2016.

[JFP '16] Hriţcu, C., **Lampropoulos, L.**, Spector-Zabusky, A., Azevedo de Amorim, A., Dénès, M., Hughes, J., Pierce, B. C., Vytiniotis, D., "Testing Noninterference, Quickly". In: *Journal of Functional Programming (JFP)* (2016).

[ITP '15] Paraskevopoulou, Z., Hriţcu, C., Dénès, M., **Lampropoulos, L.**, Pierce, B. C., "Foundational Property-Based Testing". In: *6th International Conference on Interactive Theorem Proving (ITP)*. 2015.

[ICFP '13] Hriţcu, C., Hughes, J., Pierce, B. C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., **Lampropoulos, L.,** "Testing Noninterference, Quickly". In: *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. 2013.

[WWV 12] **Lampropoulos, L.**, Sagonas, K., "Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services". In: *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems (WWV)*. 2012.

Other References

[1] Yang, X., Chen, Y., Eide, E., Regehr, J., "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2011.

[2] *American Fuzzing Lop (AFL)*. `http://lcamtuf.coredump.cx/afl/`. 2019.

[3] *Everest VERified End-to-end Secure Transport*. `https://project-everest.github.io/`.

[4] Kastner, D., Wunsche, U., Barrho, J., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S., "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler". In: *ERTS 2018: Embedded Real Time Software and Systems*. 2018.

[5] *The Free and Open RISC ISA*. `https://riscv.org/`.

[6] Claessen, K., Hughes, J., "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2000, pp. 268–279. URL: `http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf`.

[7] Klees, G. T., Ruef, A., Cooper, B., Wei, S., Hicks, M., "Evaluating Fuzz Testing". In: 2018. URL: `https://arxiv.org/abs/1808.09700`.

[8] *Draper Labs*. `https://www.draper.com/`.

[9] *Dover Microsystems*. `https://www.dovermicrosystems.com/`.

[10] *Forvis: A Formal RISC-V ISA Specification*. `https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec`. 2019.

[11] *Hypothesis: Test faster, fix more*. `https://hypothesis.works/`.

[12] Wilk, J. *American Fuzzy Lop fork server and instrumentation for pure-Python code*. `https://github.com/jwilk/python-afl`.

[13] Bhat, G., Cleaveland, R., Grumberg, O., "Efficient on-the-fly model checking for CTL". In: *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*. 1995.

[14] Gordon, A. D., Henzinger, T. A., Nori, A. V., Rajamani, S. K., "Probabilistic programming". In: *Proceedings of the on Future of Software Engineering (FOSE)*. 2014.