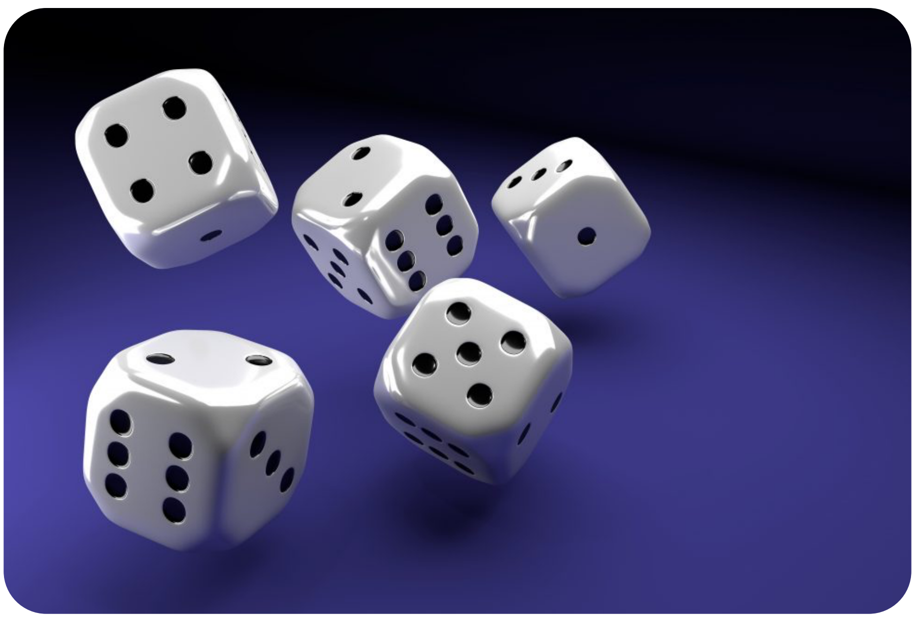
# PL Perspectives

Perspectives on computing and technology from and for those with an interest in programming languages.

# Verifying Randomized Algorithms: Why and How?

by Justin Hsu on Oct 20, 2020 | Tags: formal verification, probabilistic programs, randomized algorithms



After decades of research, randomized algorithms are playing a key role in many areas of computer science. Probabilistic procedures like stochastic gradient descent are at the heart of recent advances in machine learning. Recent notions of data privacy leverage statistical noise to hide sensitive individual

CONTRIBUTE

**Editors**:Todd Millstein and Adrian Sampson

Contribute to the Blog

RECENT POSTS

Circuit Scaling for Analog Computing

Bad Reasons to Reject Good Papers, and vice versa

A New ASPLOS Conference Submission Process

Undefined Behavior deserves a better reputation

When Compiler Optimization Meets Binary Code Difference

information. Cryptographic protocols crucially rely on unpredictable, random choices to achieve their desired security guarantees. Approximate computing, previously covered on this blog, involves running programs on hardware with randomized faults in order to achieve better energy efficiency.

While exciting applications are growing rapidly, our ability to assure the correctness of randomized algorithms has not kept pace. Like all sufficiently-complex programs, implementations of randomized algorithms have bugs. In groundbreaking recent work, Joshi, Fernando, and Misailovic uncovered serious errors in *5 out of 15* implementations of randomized hashing and sketching algorithms. At the same time, these programs are particularly difficult to assess with standard software engineering techniques, like testing—the correct behavior often describes a probability distribution of outputs, and errors can't be uncovered by a single faulty execution. The bugs discovered by Joshi, et al. came from open-source projects with a high level of visibility, but the errors were not caught by test suites (in at least one case, due to errors in the testing framework itself). Moreover, the correctness of sophisticated randomized algorithms rely on highly technical mathematical proofs. Though errors in the research literature are rare (but not unheard of), such flaws can render algorithms incorrect before they are even implemented.

What makes randomized algorithms hard to get right, and what can we do about it? This post considers research that aims to answer these questions.

## Small programs, big proofs

At a high level, randomized programs are standard programs

ARCHIVES



TAGS

that can draw random samples. For instance, they may use a special command to generate a fair coin flip, or sample a uniform number from 1–6. More sophisticated algorithms may use samples from domain-specific distributions, like the Gaussian distribution.

This small addition to the programming language leads to a significant increase in complexity, posing interesting challenges for PL and verification. (The seemingly minor step of adding an "observe" command for conditioning—used by probabilistic programming languages to describe machine learning models—leads to further large jump in complexity.) While most existing verification techniques are geared towards large programs, which form most of today's software, randomized algorithms are typically *small programs*—it is hard to imagine a randomized algorithm that is even a hundred lines of high-level code. Instead, the complexity lies in *big proofs*: arguing correctness for an algorithm that takes just a few lines to write down often involves applying theorems from probability theory, and can sometimes be a research contribution.

Even though they are a challenging target, randomized algorithms offer an opportunity to develop and broaden the reach of PL techniques. For instance, this line of research fits with a broader recent trend establishing *quantitative* program properties, where correctness is not binary (e.g., techniques for analyzing a program's resource usage or running time). Understanding how to verify randomized programs may also shed light on PL for emerging models of computation. For instance, probabilistic behavior is a fundamental aspect of quantum programs.

AI safety
architecture
awards
carbon footprint
carbon offset
climate change
compilers
computer security
concurrency
conferences
covid-19
diversity
formal reasoning
formal verification
functional programming
fuzzing
gradual typing
hardware
history
industrial adoption
language design
machine learning

The idea of verifying randomized algorithms is not new, but the recent surge in compelling applications has led to a new wave of research in this area. While many research directions are still in their early stages, we can identify a few common themes across recent work.

## Lesson 1: Pick your abstractions carefully

*Abstraction* is a fundamental concept in all verification techniques: an abstraction—of a program, state, or behavior—discards irrelevant details while retaining enough information to prove a property of interest. In the probabilistic setting, program behaviors transform probability distributions over states. These distributions are often too complex to describe exactly, even for simple programs. We would like to abstract away unnecessary information, but unlike abstractions for standard, non-probabilistic programs, abstractions for probabilistic programs typically need to keep quantitative information about probabilities.

The best case for verification is when the target property P is *compositional*: P holds on a program c if P holds on sub-programs of c. In this case, the property P itself can serve as a natural abstraction. As has been discussed several times on this blog, compositional reasoning can enormously simplify all kinds of verification tasks, allowing simpler proofs, more automated techniques, and better scalability. Though probabilistic programs are usually quite small, techniques for verifying probabilistic programs can still benefit from compositional reasoning. However, many properties from the algorithms literature are not compositional. It often requires a change of perspective to reformulate non-compositional proofs in a compositional fashion; in some cases, we can carefully generalize the target property, akin to strengthening the induction hypothesis.

A good illustration of this idea can be seen in verification techniques for *differential privacy*, a strong definition of data privacy for randomized queries. Proposed about 15 years ago, differential privacy has rapidly emerged as the current gold standard of data privacy, with numerous applications across computer science and increasing adoption in industry and government. Crucially, differential privacy is a compositional property: the privacy of a program follows from the privacy of its sub-components. For this reason, despite being a probabilistic property, differential privacy has been a fruitful and surprisingly tractable target for many verification techniques.

Application of verification of Probabilistic programing on DP

However, methods based on privacy composition cannot verify algorithms where privacy requires more sophisticated arguments. Perhaps the most well-known example is the *Sparse Vector Technique* (SVT), a subroutine that is widely used across the privacy literature. The textbook privacy proof of SVT is notoriously subtle, and does not use compositional reasoning. Accordingly, SVT appeared to be a challenging example to verify, even with the aid of an interactive theorem prover. By slightly generalizing the differential privacy property, however, we were able to give a compositional proof for SVT, eventually leading to fully automated verification for this interesting algorithm.

## Lesson 2: Look to standard PL techniques, but with a twist

virtual conferences

CATEGORIES

Select Catego

RSS SUBSCRIBE

Subscribe via E-Mail

ACM SIGPLAN

The ACM Special Interest Group on Programming Languages (SIGPLAN) explores programming language concepts and tools, focusing on design, implementation, practice, and theory. Its members are programming

The PL and verification community has developed a broad array of abstractions for standard, non-probabilistic programs. Given the drastically different challenges between verifying randomized programs and verifying standard programs, we might worry: Do any ideas from existing techniques carry over? Do we need to scrap the well-developed verification methods we know and love, and develop radically new approaches from scratch? Fortunately, and somewhat surprisingly, the answer appears to be no. A wide variety of classical methods from PL have been adapted to the probabilistic setting, opening up new applications and revealing unexpected connections. For some examples:

- Linear types, originally designed for reasoning about resource usage, can be used to prove differential privacy by reasoning about a program's sensitivity.
- Refinement types, originally designed for specifying functional correctness, can be used to verify security of cryptographic protocols and incentive properties of randomized auctions.
- Potential types, designed for bounding space and time usage through amortized analysis, can be used to bound expected resource usage.
- Information-flow control, originally designed to separate secret and public data, can be used to prevent probabilistic dependencies and analyze oblivious data structures.
- Predicate transformers, developed by Dijkstra to reason about imperative programs, can be generalized to expectation transformers and used to bound average values of expressions and expected runtimes in probabilistic programs.

language developers, educators, implementers, researchers, theoreticians, and users.

## JOIN US

Keep up-to-date with the latest technical developments, network with colleagues outside your workplace and get cutting-edge information, focused resources and unparalleled forums for discussions.

Join here: https://www.acm.org/special-interest-groups/sigs/sigplan

- Relational Hoare logics, designed for verifying compiler optimizations, can be used to verify security of cryptographic protocols, differential privacy, and convergence of Markov chains.
- Separation logics, designed to analyze heap-manipulating and concurrent programs, can use separation to model probabilistic independence.
- Methods from automated verification, like abstract interpretation and trace abstraction, can be adapted to bound expected values and prove accuracy guarantees.

While the probabilistic versions differ in some key aspects, they retain the spirit of their standard counterparts. Taken together, these technology transfers are a great sign: they give further evidence that many classical PL methods can generalize beyond their intended domains.

## Lesson 3: Learn from what human experts do

Designing proper abstractions is a difficult part of verifying randomized algorithms. However, we don't need to do this work on our own—we can look to the rich variety of proofs in the algorithms literature for inspiration. Algorithm designers have developed a powerful toolbox to analyze probabilistic programs, including *basic properties* that are useful as stepping-stones, and *proof techniques*, patterns for establishing certain kinds of properties. While these techniques may appear mysterious at first sight, there are often neat compositional abstractions lurking behind the scenes—understanding these ideas can lead to new verification methods. For example, the union bound technique has led to new techniques for verifying accuracy, and the coupling proof technique has led to new methods for showing probabilistic relational properties.

## Much more to be done!

Across computer science, many research areas tackling today's cutting-edge problems (e.g., privacy and fairness, AI robustness, quantum computing) leverage probabilistic methods. To ensure that solutions to these and other problems are working as intended, we need to develop better verification techniques for randomized programs. While there has been much progress in this area, there is still a large gap between the algorithms that PL methods can easily verify and the algorithms that humans routinely analyze—many randomized algorithms taught in undergraduate classes are still not easy to handle with formal tools.

There is a great opportunity to close this gap, making verification of randomized programs easier and more practical. While we have a growing toolbox of formal techniques for particular properties and proof techniques, we are still lacking a unified system that can combine different methods to verify randomized algorithms. There are many natural directions for further research, including improved automated techniques —most existing techniques are designed to reason about a program encoding a single distribution, rather than an algorithm generating a family of output distributions—and synthesis for probabilistic programs, linking up with an exciting recent trend in PL research. More broadly, verification work in this area can help build bridges between PL and other fields—there are many opportunities to examine intricate proofs, learn clever proof techniques, and understand ingenious algorithms from all areas of computer science.

***Acknowledgments:*** *Mike Hicks greatly improved this post with numerous edits and suggestions.*

***Bio:*** *Justin Hsu is an Assistant Professor of Computer Sciences at the University of Wisconsin–Madison. His research develops techniques for verifying randomized algorithms (surprise!).*

## About the Site

This site is maintained by volunteers working in many programs of ACM SIGPLAN. We thank you for visiting! If you have questions about the site, please send a note to our information director.