Report – Formal Methods for Security

Leads: Adam Chlipala (MIT), Deian Stefan (UCSD)
Scribes: Eunsuk Kang (CMU), Andrew Tolmach (Portland State)

Agreeing on the Topic

Of course every breakout needs to begin with some healthy debate about what the topic text actually means. One suggestion was that formal methods are all the things you do with tool support when you have some kind of formal logical semantics for a programming language. We agreed we want to include all of interactive (e.g., Coq) and automated (e.g., Z3 SMT solver) proof, both verification (proving implementation against spec) and synthesis (turning specs into implementations), and methods both static and dynamic and proof-based versus testing-based. Even type systems (at least when they get as fancy as Rust's) should be in-scope.

In further discussions about what we should drill down on, suggestions were somewhat bimodal. One cluster had to do with helping the broad cybersecurity community understand and adopt formal methods. Another cluster had to do with helping the existing formal-methods experts find the highest-impact topics within cybersecurity to tackle next.

Education and Adoption

An example was given of an aerospace company interested in offering a formal-modeling course to its engineers. Do we have community consensus on what knowledge base such a course needs to convey? Is there a role for NSF-facilitated engagement with companies to learn what they are looking for in educational experiences?

We also discussed instilling the fundamentals in early undergraduate curriculum, like discrete math, where formal notions of correctness and perhaps even program proof could fit well. An example was given in the context of a discrete-math class, where students were asked to formulate the N-queens problem as SMT constraints, at which point they could see the benefit of having reusable logic solvers around, which also built their appreciation for specifications and formal logic.

What can we learn from the incentive structure exploited in organizations like Amazon Web Services, to drive formal methods adoption? Techniques like fuzzing may be relatively easy to get started with, compared to e.g. program proof. Is there a gradual adoption curve that can port across companies?

Whatever consensus can be built around these questions may be worth writing down, perhaps in the style of a systematization-of-knowledge paper, Oakland-style. It would explain the different kinds of formal methods, their pros and cons, and how to get started using each style

effectively. Or that message could be spread through tutorials at security conferences, summer schools, etc. And yet another idea is matchmaking between formal-methods experts and other potential adopters, whether at conferences or online, at least for short intro conversations.

It is important to motivate the importance of the area. We should make a case that formal methods are a principled way to build secure and trustworthy systems, eliminating whole classes of bugs. Further, formal thinking is an opportunity to harden a design, exerting pushback on complexity and encouraging good modularity. Formalism also provides a good way to "measure" security, thanks to the need to be explicit about threat models and edge cases.

Application Areas

Projects Already Far Along

The best-known examples of formal methods in the software industry are in cryptography and distributed systems. For instance, Amazon Web Services and Microsoft each have active and visible projects in both. These examples make for great motivation to get more people excited about formal methods.

In the open-source world, it's noteworthy that all major browsers now draw parts of their TLS implementations from code built with formal-methods tools.

Another domain getting increasing traction, though often as a kind of "under-the-hood" element not as obvious to the broader developer audience, is compilers. Actually, the applications in cryptography and distributed systems often involve formal-methods-centric compilers, and there is other ongoing work, e.g. in JavaScript implementations, that may be poised to make the same kind of splash.

There are also many applications in blockchain and smart contracts, where many experts have been pulled into companies in that sector.

Challenges (i.e., good ideas for SaTC proposals)

Industrial control systems and cyberphysical systems more generally were raised as especially compelling applications, where the payoff to security analysis is high. Another (overlapping) good application focus is autonomous and ML-based systems. NSF's new DASS program is relevant.

Some of the most practical formal-methods techniques depend on programming languages that make certain kinds of security bugs impossible, e.g. Rust and memory management. How can we push that frontier further?

Many successful projects build on formal semantics or models. One example that came up was models of Android permissions. The question, then, is how to gain confidence in accuracy of these models. Formal proof against open-source implementations is an option with intuitive appeal but also high cost. What other methods could researchers develop? Are there opportunities for curating centralized repositories of vetted semantics and models?

Compartmentalization mechanisms, from browser sandboxes to enclave systems, were raised as a good target for formal methods. Can we characterize isolation guarantees formally and prove that they are respected? How can these methods be part of the story for gradually adding more use of formal methods in legacy projects?

From an attack perspective, it was suggested that it could pay off to analyze formal proofs for evidence of which parts of system designs are most fragile, that is, where the proof would fail in the face of seemingly small design changes. Could those aspects of systems be most fruitful to attack?

Cost of evolving formal developments over time, alongside changing implementations, remains a central concern and never-ending source of challenges in basic research. Techniques should support reuse of old proof effort, through incremental verification, e.g. caching lemma proofs to replay. Modular and compositional techniques are especially critical here.

Usability of formal-methods tools is another such dimension. For instance, SMT solvers' error messages are often inscrutable. Is there a place here for HCI-style research to improve the user experience?

How should the community split its effort between low-level generic bugs like memory errors and high-level semantic bugs like implementing a textbook algorithm incorrectly? Potential security issues at the hardware-software boundary may be especially worth studying, at either granularity.

Which properties are best enforced with runtime monitoring, e.g. with a browser killing a tab that does something inappropriate? The performance overhead pushes against adoption, though implementation is usually simpler.

How could engineering practices change to prepare for effective use of formal methods, even if formal methods are not yet being applied to a project?