Formal Methods at Scale

Mike Dodds[1], John Launchbury[1], Stephen Magill[2]
[1]Galois, [2]MuseDev

Application 1: Formal verification of Amazon Web Services (AWS) cryptographic code.

- Galois has formally verified cryptographic algorithms that are deployed as part of the s2n TLS library, in use in AWS's production systems. Through this, the verified code is used to secure traffic for hundreds of millions of end users.
- Proofs are integrated into continuous integration (CI) and rerun on every change to the code. This means that incorrect code cannot enter production.

Results / insights:

- Galois proofs require careful initial design, but can often be rerun automatically after code changes. This is necessary for adoption in production, as engineers typically do not have the expertise to repair a proof.
- Cryptographic algorithm implementations are a good target for formal methods tools, as they are typically (1) clearly specified through RFCs, (2) bounded in size, (3) have clear interfaces, (4) rarely change once developed, (5) failures result in severe security consequences. Other applications which have proved attractive targets include state machine code and serialize/deserialize code.
- Proofs of this kind are limited in the scale of software they can be applied to, by several factors:
    1. Speed of SMT solvers: requires improvements in cloud-based solver infrastructure.
    2. Number of formal methods experts to design proof: requires more useable tools that can be applied by domain experts.
    3. Ability of engineers to maintain / repair proofs: requires techniques that can auto-repair the majority of trivial fixes, and auto-triage the remainder of deeper issues.
- All formal proofs have background assumptions that limit the scope of the guarantees they provide. For example, Galois proofs assume the compiler is well-behaved, and that functions are called as expected. There is considerable risk to user confidence if these assumptions are mis-communicated, and a bug is found in 'verified correct' software.

Application 2: Muse[1] (https://muse.dev), a cloud platform providing formal methods tools "as a service".

- Muse supports multiple off-the-shelf tools, including Infer, ErrorProne, Pyre, and others. Muse supports tools with a 'push-button' interaction model, that can be used on large scale code.
- Much of the challenge in using formal methods tools lies in simply running tools. Muse provides integrations that support common build systems, languages, and code hosting platforms.

Results / insights:

- Cloud-hosted services can solve a number of roadblocks to adoption of formal methods tools.
- Integration with CI / CD tooling can be provided in a manner that is tool-agnostic. As an example, Muse integrates with GitHub to perform analysis of every code change and report errors as pull request comments, and is able to provide this transparently for all analysis tools.
- Building on a common platform allows multiple tools to be supported with a single configuration file. This vastly simplifies setup and enhances usability—for example, this allows Muse to support both Infer and ErrorProne, tools with very different underlying analysis requirements.
- Tools can be managed centrally, relieving users from the maintenance burden of keeping several tools up-to-date, integrated into the execution environment, and properly configured.

---

[1] *As of June 2021, Muse is now Sonatype Lift and can be found at https://lift.sonatype.com/*

- Cloud-hosting formal methods tools allows data to be collected and aggregated, providing feedback to drive tool improvement. This enables the sort of data-driven approach to tool development that has benefited formal methods teams at Facebook and Google.
- Cloud hosting allows tool errors to be detected and logged without interrupting the development workflow. This lets developers use early-stage tools so that these tools can mature and the tool authors can learn from developer usage.