# Formal reasoning about the security of AWS (Invited paper)

Byron Cook

Amazon Web Services &
University College London

**Abstract.** We report on the development and use of formal verification tools within Amazon Web Services (AWS) to increase the security assurance of its cloud infrastructure and to help customers secure themselves. We also discuss some remaining challenges that could inspire future research in the community.

## Introduction

Amazon Web Services (AWS) is a provider of *cloud services*, meaning on-demand access to IT resources via the Internet. AWS adoption is widespread, with over a million active customers in 190 countries, and $5.1 billion in revenue during the last quarter of 2017. Adoption is also rapidly growing, with revenue regularly increasing between 40–45% year-over-year.

The challenge for AWS in the coming years will be to accelerate the development of its functionality while simultaneously increasing the level of security offered to customers. In 2011, AWS released over 80 significant services and features. In 2012, the number was nearly 160; in 2013, 280; in 2014, 516; in 2015, 722; in 2016, 1,017. Last year the number was 1,430. At the same time, AWS is increasingly being used for a broad range of security-critical computational workloads.

Formal automated reasoning is one of the investments that AWS is making in order to facilitate continued simultaneous growth in both functionality and security. The goal of this paper is to convey information to the formal verification research community about this industrial application of the community's results. Toward that goal we describe work within AWS that uses formal verification to raise the level of security assurance of its products. We also discuss the use of formal reasoning tools by externally-facing products that help customers secure themselves. We close with a discussion about areas where we see that future research could contribute further impact.

*Related work.* In this work we discuss efforts to make formal verification applicable to use-cases related to cloud security at AWS. For information on previous work within AWS to show functional correctness of some key distributed algorithms, see [43]. Other providers of cloud services also use formal verification to establish security properties, *e.g.* [23, 34].

Our overall strategy on the application of formal verification has been heavily influenced by the success of previous applied formal verification teams in industrial settings that worked as closely with domain experts as possible, *e.g.* work at Intel [33, 50], NASA [31, 42], Rockwell Collins [25], the Static Driver Verifier project [20], Facebook [45], and the success of Prover AB in the domain of railway switching [11].

External tools that we use include Boogie [1], Coq [4], CBMC [2], CVC4 [5], Dafny [6], HOL-light [8], Infer [9], OpenJML [10], SAW [13], SMACK [14], Soufflé [37], TLA+ [15], VCC [16], and Z3 [17]. We have also collaborated with many organizations and individuals, *e.g.* Galois, Trail of Bits, the University of Sydney, and the University of Waterloo. Finally, many PhD student interns have applied their prototype tools to our problems during their internships.

## Security of the cloud

Amazon and AWS aim to innovate quickly while simultaneously improving on security. An original tenet from the founding of the AWS security team is to never be the organization that says *"no"*, but instead to be the organization that answers difficult security challenges with *"here's how"*. Toward this goal, the AWS security team works closely with product service teams to quickly identify and mitigate potential security concerns as early as possible while simultaneously not slowing the development teams down with bureaucracy. The security team also works with service teams early to facilitate the certification of compliance with industry standards.

The AWS security team performs formal security reviews of all features/services, *e.g.* 1,430 services/features in 2017, a 41% year-over-year increase from 2016. Mitigations to security risks that are developed during these security reviews are documented as a part of the security review process. Another important activity within AWS is ensuring that the cloud infrastructure *stays* secure after launch, especially as the system is modified incrementally by developers.

**Where formal reasoning fits in.** The application security review process used within AWS increasingly involves the use of deductive theorem proving and/or symbolic model checking to establish important temporal properties of the software. For example, in 2017 alone the security team used deductive theorem provers or model checking tools to reason about cryptographic protocols/systems (*e.g.* [24]), hypervisors, boot-loaders/BIOS/firmware (*e.g.* [27]), garbage collectors, and network designs. Overall, formal verification engagements within the AWS security team increased 76% year-over-year in 2017, and found 45% more pre-launch security findings year-over-year in 2017.

To support our needs we have modified a number of open-source projects and contributed those changes back. For example, changes to CBMC [2] facilitate its application to C-based systems at the bottom of the compute stack used in AWS data centers [27]. Changes to SAW [13] add support for the Java programming language. Contributions to SMACK [14] implement automata-theoretic

constructions that facilitate automatic proofs that s2n [12] correctly implements the *code balancing* mitigation for side-channel timing attacks. Source-code contributions to OpenJML [10] add support for Java 8 features needed to prove the correctness of code implementing a secure streaming protocol used throughout AWS.

In many cases we use formal verification tools *continuously* to ensure that security is implemented as designed, *e.g.* [24]. In this scenario, whenever changes and updates to the service/feature are developed, the verification tool is re-executed automatically prior to the deployment of the new version.

The security operations team also uses automated formal reasoning tools in its effort to identify security vulnerabilities found in internal systems and determine their potential impact on demand. For example, an SMT-based semantic-level policy reasoning tool is used to find misconfigured resource policies.

In general we have found that the internal use of formal reasoning tools provides good value for the investment made. Formal reasoning provides higher levels of assurance than testing for the properties established, as it provides clear information about what has and has not been secured. Furthermore, formal verification of systems can begin long before code is written, as we can prove the correctness of the high-level algorithms and protocols, and use under-constrained symbolic models for unwritten code or hardware that has not been fabricated yet.

## Securing customers *in* the cloud

AWS offers a set of cloud-based services designed to help customers be secure *in* the cloud. Some examples include AWS Config, which provides customers with information about the configurations of their AWS resources; Amazon Inspector, which provides automated security assessments of customer-authored AWS-based applications; Amazon GuardDuty, which monitors AWS accounts looking for unusual account usage on behalf of customers; Amazon Macie, which helps customers discover and classify sensitive data at risk of being leaked; and AWS Trusted Advisor, which automatically makes optimization and security recommendations to customers.

In addition to automatic cloud-based security services, AWS provides people to help customers: *Solutions Architects* from different disciplines work with customers to ensure that they are making the best use of available AWS services; *Technical Account Managers* are assigned to customers and work with them when security or operational events arise; the *Professional Services* team can be hired by customers to work on bespoke cloud-based solutions.

**Where formal reasoning fits in.** Automated formal reasoning tools today provide functionality to customers through the AWS services Config, Inspector, GuardDuty, Macie, Trusted Advisor, and the storage service S3. As an example, customers using the S3 web-based console receiving alerts—via SMT-based

reasoning—when their S3 bucket policies are possibly misconfigured. AWS Macie uses the same engine to find possible data exfiltration routes. Another application is the use of high-performance datalog constraint solvers (*e.g.* [37]) to reason about questions of reachability in complex virtual networks built using AWS EC2 networking primitives. The theorem proving service behind this functionality regularly receives 10s of millions of calls daily.

In addition to the automated services that use formal techniques, some members of the AWS Solutions Architects, Technical Account Managers and Professional Services teams are applying and/or deploying formal verification directly with customers. In particular, in certain security-sensitive sectors (*e.g.* financial services), the Professional Services organization are working directly with customers to deploy formal reasoning into their AWS environments.

The customer reaction to features based on formal reasoning tools has been overwhelmingly positive, both anecdotally as well as quantitatively. Calls by AWS services to the automated reasoning tools increased by four orders of magnitude in 2017. With the formal verification tools providing the semantic foundation, customers can make stronger universal statements about their policies and networks and be confident that their assumptions are not violated.

## Challenges

At AWS we have successfully applied existing or bespoke formal verification tools to both raise the level of security assurance *of* the cloud as well as help customers protect themselves *in* the cloud. We now know that formal verification provides value to applications in cloud security. There are, however, many problems yet to be solved and many applications of formal verification techniques yet to be discovered and/or applied. In the future we are hoping to solve the problems we face in partnership with the formal verification research community. In this section we outline some of those challenges. Note that in many cases existing teams in the research community will already be working on topics related to these problems, too many to cite comprehensively. Our comments are intended to encourage and inspire more work in this space.

**Reasoning about risk and feasibility.** A security engineer spends the majority of their time informally reasoning about risk. The same is true for any corporate Chief Information Security Officer (CISO). We (the formal verification community) potentially have a lot to contribute in this space by developing systems that help reason more formally about the consequences of combinations of events and their relationships to bugs found in systems. Furthermore, our community has a lot to offer by bridging between our concept of a counterexample and the security community's notion of a *proof of concept* (PoC), which is a constructive realization of a security finding in order to demonstrate its feasibility. Often security engineers will develop partial PoCs, meaning that they combine reasoning about risk and the finding of constructive witnesses in order to increase their confidence in the importance of a finding. There are valuable

results yet to be discovered by our community at the intersection of reasoning about and synthesis of threat models, environment models, risk/probabilities, counterexamples, and PoCs. A few examples of current work on this topic include [18, 28, 30, 44, 48].

**Fixes not findings.** Industrial users of formal verification technology need to make systems more secure, not merely find security vulnerabilities. This is true both for securing the cloud, as well as helping customers be secure in the cloud. If there are security findings, the primary objective is to find them *and* fix them quickly. In practice a lot of work is ahead for an organization once a security finding has been identified. As a community, anything we can do to reduce the friction for users trying to triage and fix vulnerabilities, the better. Tools that report false findings are quickly ignored by developers, thus as a community we should focus on improving the fidelity of our tools. Counterexamples can be downplayed by optimistic developers: any assistance in helping users understand the bugs found and/or their consequences is helpful. Security vulnerabilities that require fixes that are hard to build or hard to deploy are an especially important challenge: our community has a lot to offer here via the development of more powerful synthesis/repair methods (*e.g.* [22, 32, 39]) that take into account threat models, environment models, probabilities, counterexamples.

**Auditable proof artifacts for compliance.** Proof is actually two activities: *searching* for a candidate proof, and *checking* the candidate proof's validity. The searching is the art form, often involving a combination of heuristics that attempt to work around the undecidable. The checking of a proof is (in principle) the boring yet rigorous part, usually decidable, often linear in the size of the proof. Proof artifacts that can be re-checked have value, especially in applications related to compliance certification, *e.g.* DO-333 [26], CENENLEC EN 50128 SIL 4 [11], EAL7 MILS [51]. Non-trivial parts of the various compliance and conformance standards can be checked via mechanical proof, *e.g.* parts of PCI and FIPS 140. Found proofs of compliance controls that can be shared and checked/re-checked have the possibility to reduce the cost of compliance certification, as well as reduce the time-to-market for organizations who require certification before using systems.

**Tracking casual or unrealistic assumptions.** Practical formal verification efforts often make unrealistic assumptions that are later forgotten. As an example, most tools assume that the systems we are analyzing are immune to *single-event upsets*, *e.g.* ionizing particles striking the microprocessor or semiconductor memory. We sometimes assume compilers and runtime garbage collectors are correct. In some cases (*e.g.* [20]) the environment models used by formal verification tools do not capture all possible real-world scenarios. As formal verification tools become more powerful and useful we will increasingly need to reason about what has been proved and what has not been proved, in order to avoid misunderstandings that could lead to security vulnerabilities. In applications of security this reasoning about assumptions made will need to interact

with the treatment of risk and how risk is modified by various mitigations, *e.g.* some mitigations for single-event upsets make the events so unlikely they they are not a viable security risk, but still not impossible. This topic has been the focus of some attention over the years, *e.g.* CLINC stack [41], CompCert [3], and DeepSpec [7]. We believe that this will become an increasingly important problem in the future.

**Distributed formal verification in the cloud.** Formal verification tools do not take enough advantage of modern data centers via distributing coordinated processes. Some examples of work in the right direction include [21, 35, 36, 38, 40, 47]. Especially in the area of program verification and analysis, our community still focuses on procedures that work on single computers, or perhaps *portfolio* solvers that try different problem encodings or solvers in parallel. Today large formal verification problems are often decomposed manually, and then solved in parallel. There has not been much research in methods for automatically introducing and managing the reasoning about the decompositions automatically in cloud-based distributed systems. This is in part perhaps due to the rules at various annual competitions such as SV-COMP, SMT-COMP, and CASC. We encourage the participants and organizers of competitions to move to cloud-based competitions where solvers have the freedom to use cloud-scale distributed computing to solve formal verification problems. Tool developers could build AMIs or CloudFormation templates that allow cloud distribution. Perhaps future contestants might even make Internet endpoints available with APIs supporting SMTLIB or TPTP such that the competition is simply a series of remote API calls to each competitor's implementation. In this case competitors that embrace the full power of the cloud will have an advantage, and we will see dramatic improvements in the computational power of our formal verification tools.

**Continuous formal verification.** As discussed previously, we have found that it is important to focus on *continuous verification*: it is not enough to simply prove the correctness of a protocol or system once, what we need is to *continuously* prove the desired property during the lifetime of the system [24]. This matches reports from elsewhere in industry where formal verification is being applied, *e.g.* [45]. An interesting consequence of our focus on continuous formal verification is that the time and effort spent finding an initial proof before a system is deployed is not as expensive as the time spent maintaining the proof later, as the up-front human cost of the pre-launch proof is amortized over the lifetime of the system. It would be especially interesting to see approaches developed that synthesize new proofs of modified code based on existing proofs of unmodified code.

**The known problems are still problems.** Many of the problems that we face in AWS are well known to the formal verification community. For example, we need better tools for formal reasoning about languages such as Ruby, Python, and Javascript, *e.g.* [29, 49]. Proofs about security-oriented properties of many large open source systems remain an open problem, *e.g.* Angular, Linux,

OpenJDK, React, NGINX, Xen. Many formal verification tools are hard to use. Many tools are brittle prototypes only developed for the purposes of publication. Better understanding of ISAs and memory models (*e.g.* [19, 46]) are also key to prove the correctness of code operating on low-level devices. Practical and scalable methods for proving the correctness of distributed and/or concurrent systems remains an open problem. Improvements to the performance and scalability of formal verification tools are needed to prove the correctness of larger modules without manual decomposition. Abstraction refinement continues to be a problem, as false bugs are expensive to triage in an industrial setting. Buggy (and thus unsound) proof-based tools lose trust in formal verification with the users who are trying to deploy them.

## Conclusion

In this paper we have discussed how formal verification contributes to the ability of AWS to quickly develop and deploy new features while simultaneously increasing the security of the AWS cloud infrastructure. We also discussed how formal verification techniques contribute to customer-facing AWS services. In this paper we have outlined some challenges we face. We actively seek solutions to these problems and are happy to collaborate with partners in this pursuit. We look forward to more partnerships, more tools, more collaboration, and more sharing of information as we try to bring affordable, efficient and secure computation to all.

## References


1. Boogie program prover. https://github.com/boogie-org/boogie.
2. CBMC model checker. https://github.com/diffblue/cbmc.
3. CompCert project. http://compcert.inria.fr.
4. Coq theorem prover. https://github.com/coq/coq.
5. CVC4 decision procedure. http://cvc4.cs.stanford.edu.
6. Dafny theorem prover. https://github.com/Microsoft/dafny.
7. DeepSpec project. https://deepspec.org.
8. HOL-light theorem prover. https://github.com/jrh13/hol-light.
9. Infer program analysis. https://github.com/facebook/infer.
10. OpenJML program prover. https://github.com/OpenJML.
11. Prover Certifier. https://www.prover.com/software-solutions-rail-control/prover-certifier.
12. s2n TLS/SSL implementation. https://github.com/awslabs/s2n.
13. SAW program prover. https://github.com/GaloisInc/saw-script.
14. SMACK software verifier. http://smackers.github.io/.
15. TLA+ theorem prover. https://github.com/tlaplus.
16. VCC program prover. https://vcc.codeplex.com.
17. Z3 decision procedure. https://github.com/Z3Prover/z3.
18. A. Aldini, J.-M. Seigneur, C. B. Lafuente, X. Titi, and J. Guislain. Design and validation of a trust-based opportunity-enabled risk management system. *Information and Computer Security*, 25(2), 2017.

19. J. Alglave and P. Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In *POPL*, 2017.
20. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85, 2006.
21. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers, 2016.
22. R. Bloem, B. Cook, and A. Griesmayer. Repair of Boolean programs with an application to C. In *CAV*, 2006.
23. S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer. Verifying cloud services: present and future. *ACM SIGOPS Operating Systems Review*, 47(2), July 2013.
24. A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, S. Magill, C. MacCarthaigh, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook. Continuous formal verification of Amazon s2n. In *CAV*, 2018.
25. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *NFM*, 2012.
26. D. Cofer and S. Miller. DO-333 certification case studies. In *NSF*, 2014.
27. B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model checking boot code from AWS data centers. In *CAV*, 2018.
28. T. F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, PP(99), 2017.
29. M. Eilers and P. Müller. Nagini: A static verifier for Python. In *CAV*, 2018.
30. V. Ganesh, S. Banescu, and M. Ochoa. The meaning of attack-resistant programs. In *International Workshop on Progamming Languages and Security*, 2015.
31. A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *NFM*, 2013.
32. S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4, 2017.
33. J. Harrison. Formal verification of IA-64 division algorithms. In *TPHOLs*, 2000.
34. C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *SOSP*, 2015.
35. M. Heule, O. Kullmann, and V. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *SAT*, 2016.
36. G. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the Swarm tool. In *SPIN*, 2008.
37. H. Jordan, B. Scholz, and P. Subotic. Towards proof synthesis by neural machine translation. In *CAV*, 2016.
38. R. Kumar, T. Ball, J. Lichtenberg, N. Deisinger, A. Upreti, and C. Bansal. CloudSDV: Enabling Static Driver Verifier using Microsoft Azure. In *IFM*, 2016.
39. C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Jornal*, 21, 2013.
40. N. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *VMCAI*, 2011.
41. J. S. Moore. Machines reasoning about machines: 2015. In *ATVA*, 2005.
42. A. Narkawicz and C. uñoz. Formal verification of conflict detection algorithms for arbitrary trajectories. In *Reliable Computing*, 2012.
43. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4), 2004.

44. M. Ochoa, S. Banescu, C. Disenfeld, G. Barthe, and V. Ganesh. Reasoning about probabilistic defense mechanisms against remote attacks. In *IEEE European Symposium on Security and Privacy*, 2017.
45. P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *LICS*, 2018.
46. A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. End-to-end verification of ARM processors with ISA-Formal. In *CAV*, 2016.
47. K. Rozier and M. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *FM*, 2011.
48. J. Rushby. Software verification and system assurance. In *IEEE International Conference on Software Engineering and Formal Methods*, pages 3–10, 2009.
49. J. F. Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. JaVerT: JavaScript verification toolchain. In *POPL*, 2017.
50. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.
51. M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. 2010.