



# The Application of Formal Methods to Real-World Cryptographic Algorithms, Protocols, and Systems

**Nicky Mouha and Asmaa Hailane**, National Institute of Standards and Technology

*Computer hosts a virtual roundtable with four experts in formal methods to discuss recent developments in the area of cryptography.*

Digital Object Identifier 10.1109/MC.2020.3033613  
Date of current version: 14 January 2021

**A**s you read the title of this article, at least one of the following two questions probably occurred to you. First, can one really use the terms “formal methods” and “real world” so close to each other? And second, why such a narrow domain of application for formal methods, namely, cryptography? Doesn’t that restrict the scope of discussion too much?

We are not going to answer the first question for you here. We will let our panelists convince you that the terms do belong together. Actually, we are not really going to answer the second question either, at least not directly. However, we do want to suggest that by choosing a specific application of formal methods, we enable our experts to communicate with greater detail and include more concrete examples than would be possible using general terms. Besides, we note that cryptographic algorithms, protocols, and systems are increasingly considered essential (and security-critical) infrastructure for our virtual world, rather than merely specialized applications. Our “narrow” domain is really quite large.

## ROUNDTABLE PANELISTS

**Karthikeyan Bhargavan** is a research director at Inria, Paris, France, where he leads the Prosecco project (Programming Securely With Cryptography). Contact him at [karthikeyan.bhargavan@inria.fr](mailto:karthikeyan.bhargavan@inria.fr).

**Adam Chlipala** is an associate professor of computer science at the Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, where he leads the Programming Languages and Verification Group. Contact him at [adamc@csail.mit.edu](mailto:adamc@csail.mit.edu).

**Jonathan Protzenko** is a senior researcher in the Research in Software Engineering (RISE) group, Microsoft Research, Redmond, Washington, USA. Contact him at [protz@microsoft.com](mailto:protz@microsoft.com).

**Bow-Yaw Wang** is a research fellow and professor at the Institute of Information Science, Academia Sinica, Taipei, Taiwan. Contact him at [bywang@iis.sinica.edu.tw](mailto:bywang@iis.sinica.edu.tw).

## EMERGING TOOLS

Everest (<https://project-everest.github.io/>) is a project that aims to build and deploy a verified version of Transport Layer Security (TLS). There have been several serious attacks on TLS, ranging from the protocol's design to its most commonly used ciphers and most modes of operation.

The High-Assurance Cryptographic Library (HACL\*) (<https://hacl-star.github.io/>), a verified library of modern cryptographic primitives written in F\*, includes modern cryptographic algorithms that are used in Networking and Cryptography Library (NaCl) and popular protocols, such as Signal and TLS.

Fiat Cryptography (<https://github.com/mit-plv/fiat-crypto>), based on a verified compilation scheme, aims to generate verified field arithmetic code for several curves, including the first verified high-performance implementation of P-256, the most widely used elliptic curve in TLS.

Cryptoline (<https://github.com/fmlab-iis/cryptoline>) is a tool and a language for the verification of low-level implementations of mathematical constructs. It has been used to verify implementations in OpenSSL, BoringSSL, and mbed TLS.

You may have a final question as well. What exactly is a virtual roundtable? This question we will answer. It is relatively straightforward: we ask a series of questions about an important technical topic to a group of expert panelists via email (see “Roundtable Panelists” for more information about the panel). This is a simple format, but there are two important differences between this and an in-person panel. One is that no expert knows who the others are.

The second is that each panelist must answer the questions without seeing the others' responses. And now it is time for us to step out of the way so you can see what our panelists have to say. We hope you enjoy their insightful perspectives.

**COMPUTER:** Numerous techniques have been referred to as *formal methods*. How would you define formal methods, either in general or more specifically when applied to cryptography?

**KARTHIKEYAN BHARGAVAN:** In general, *formal methods* refers to the application of logical reasoning techniques to understand, model, and verify computer systems. For cryptography, some of us like to use a more focused term—computer-aided cryptography—which describes “formal, machine-checkable approaches to the design, analysis, and implementation of cryptography.”<sup>1</sup> To me, writing a detailed formal specification of a cryptographic algorithm or protocol is an application of a formal

method and so is the use of (semi)automated tools to find attacks and build proofs for cryptographic mechanisms.

**ADAM CHLIPALA:** Let me give a general definition and then touch briefly on what's specific to cryptography. I would say formal methods are fundamentally about using formal logic to characterize behavioral similarities between different pieces of code. If one piece of code is very simple and if you know a more complicated piece of code behaves similarly, then you increase your confidence in the second one. Some pieces of code are so simple and "obviously" correct that we decide to call them *specifications*, though the defining criteria tend to be nebulous. What's important is that we do not rely on humans to write or check arguments for similarity. Instead, we should use algorithms at least to check the arguments (for example, written out in ASCII source code) and ideally find the arguments in the first place. At a minimum, algorithms should make the construction of arguments (proofs) less labor intensive. With the right tools and choice of specifications, it becomes possible to trust a complex system without needing to run it or read its implementation.

Formal methods and cryptography are a great match, and there tend to be a few important kinds of behavioral similarity that folks want to prove. The simplest is functional correctness, where we focus on a system that produces correct answers. A trickier one is the proof of traditional security properties, such as, "An attacker who doesn't know the private key has little hope of figuring out the contents of these encrypted messages." A last but also very important category is the proof that side channels can't be used to break higher-level properties, for example, the execution time doesn't leak bits of a key.

**JONATHAN PROTZENKO:** *Formal methods* is an umbrella term that has been historically hard to define and, in practice, doesn't evoke much for people outside our field of expertise. I generally try to use more specific terms, such as model

checking, abstract interpretation, or program proof. But as long as the technique enables seeing code as a mathematical object that can be symbolically manipulated, on which you can prove theorems, I consider it to fall under the formal methods umbrella.

**BOW-YAW WANG:** In my view, formal methods generally involve techniques that manually or automatically apply logical or mathematical reasoning to achieve clearly stated goals. When applied to cryptography, formal methods can refer to the construction of cryptographic programs or proofs for mathematically specified security properties.

**COMPUTER:** Where do you see the application of formal methods to cryptographic algorithms, protocols, and systems within five to 10 years?

**BHARGAVAN:** In the past few years, we have started to see a transformation in the attitude toward formal methods for cryptography in both industry and academia. Part of this change can be traced to the Transport Layer Security (TLS) 1.3 standardization process, which involved a multiyear collaboration between the Internet Engineering Task Force (IETF), all major browser and operating system vendors, and several research groups that analyzed the protocol in detail before it was published. Considering the complexity of the protocol, many of these analyses relied on mechanized provers, that is, formal methods. In the coming years, I see this process being replicated, and it will become understood that a new cryptographic protocol cannot be standardized without formal machine-checked proofs. Another new direction has been the incorporation of formally verified implementations into mainstream cryptographic libraries. Going forward, I see more and more of the core cryptographic algorithms in browsers and operating systems being replaced by verified implementations.

**CHLIPALA:** First, I think it helps to emphasize how much progress has been made already (see "Emerging Tools"). The Project Everest<sup>2</sup> team kicked off the recent wave of open source adoption with the verified High-Assurance Cryptographic Library (HACL\*)<sup>3</sup> and its use in Firefox. A verified crypto-primitive compiler that I've been involved with, Fiat Cryptography,<sup>4</sup> has now been adopted for (some aspects of) finite-field arithmetic in both Chrome and Firefox, the WireGuard virtual private network (VPN) in the Linux kernel, and the primary libraries for Facebook's Libra and other blockchain systems.

There are two main dimensions where I expect to see substantial progress in the five-to-10-year time frame. First, at least my own work with Fiat Cryptography hasn't involved the proof of higher-level security properties, such as resistance to forged signatures, and I expect the community will make good progress on scaling that kind of proof. Perhaps more importantly, I expect to see a much more satisfying integration of formal results into full verified systems. The instances of adoption I just mentioned involve copying and pasting formally validated code within much larger systems (typically not subjected to formal methods). The boundaries among applications, libraries, compilers, and hardware are major opportunities for bugs that invalidate guarantees. I expect to see good proofs of concept with formal guarantees that stretch from Verilog hardware designs to whiteboard-level pseudocode for cryptographic protocols, covering both functional correctness and lack-of-information leaks through timing.

**PROTZENKO:** We are witnessing a very exciting time wherein many teams are adopting different approaches toward proving cryptographic algorithms. This has fostered a friendly competition, and many of those teams (Fiat Cryptography,<sup>4</sup> Jasmine,<sup>5</sup> Cryptoline,<sup>6</sup> HACL\*, and others) have done wonderful work that significantly advanced the state of the art in the span of just a few years. The challenge is now to rise beyond

cryptographic primitives and tackle layers that are further and further up the software stack.

In a sense, a cryptographic primitive is a pure function: data in, data out—this was a great “warm-up” for the software verification community. But a large chunk of more complex critical code sits immediately above the primitives, dealing with state machines, buffering, incremental application programming interfaces (APIs) that enable passing data across several API calls, and so on, which complicates the problem statement. And then, above these high-level crypto APIs, there are protocols, which typically orchestrate many primitives, have several intertwined state machines, and deal with many more pieces of state and much more complex invariants and data structures than just a single algorithm.

I expect that, within a few years, the state of the art in program proofs will have advanced enough that verifying primitives will be considered mundane and a strong requirement for any new proposed algorithm; that high-level crypto APIs will also be fully verified; and that the latest advances will provide efficient, fully verified implementations of complete protocols, including all the primitives, state machines, data structures, and bookkeeping. By then, and on the somewhat longer horizon, I expect that we will see large software subsystems, such as a hypervisor, an Internet of Things device, or an entire operating system subcomponent, fully verified, of which the protocol will be only a small chunk.

**WANG:** Formal methods have been used to verify cryptographic algorithms, protocols, and systems at smaller scales. With such (limited) success, the cryptography community has noted the advantages of applying formal methods. Better formal methods for cryptography will surely be developed in the near future. I believe theorists and practitioners in cryptography will adopt formal methods more broadly. More specifically, formal methods can help theorists carry out mathematical proofs for security.

Protocol designers can benefit from a rigorous analysis of corner cases. Formal methods can also verify cryptographic programs at a large scale. Of course, it will not be possible without interdisciplinary collaboration. I also expect more communication between formal method and cryptography communities.

**COMPUTER:** How would you compare the assurance provided by formal methods versus those from other techniques, such as static analysis, dynamic analysis (for example, fuzzing), or known-answer tests? Are there types of bugs that can be found using formal methods but not other techniques?

**BHARGAVAN:** When analyzing a cryptographic system, the first goal is always to verify that the input-output behavior is functionally correct. The harder goal is to prove that the system preserves its security invariants even in the presence of a hostile adversary who can use malformed inputs and employ side-channel attacks. Classic software analysis and testing tools can be very effective in finding bugs in cryptographic systems and are widely used in industry. However, they can miss the kind of low-probability functional correctness bugs that often appear in cryptographic code, such as an integer overflow bug that appears in only one out of, say,  $2^{64}$  inputs. Furthermore, these techniques have little hope of finding side-channel leaks or protocol flaws that depend on cryptographic weaknesses.

Formal methods can close this gap by providing comprehensive guarantees for a cryptographic mechanism against some well-defined set of attackers, under some assumptions about the underlying cryptographic algorithm and about the application that uses the mechanism. Of course, the guarantees hold only in this model, and any attack that exploits an attack vector that was not covered in the model may still succeed. So, I see formal methods as yet another tool in the analyst’s arsenal: they eliminate an entire class of attacks,

enabling us to focus on others that were not covered by the model.

**CHLIPALA:** Different groupings of those techniques deserve different answers. Let me start with dynamic analysis and known-answer tests lumped together as approaches that rely on careful execution of the code by running it under many different inputs. The trouble here is that true exhaustive testing of a system, under all possible inputs, is infeasible. The lifetime of the universe might not be enough to test all inputs in some cases! That places the burden on developers to devise a theory of what the important corner cases are, to be sure to exercise them all sufficiently. However, in a security setting, you always worry that your adversary did a better job than you did at intuiting the tricky corner cases. He potentially just needs to find one to break all your guarantees. In contrast, formal verification enables the certification of correct behavior in all scenarios. Many techniques don’t even require more analysis runtime as the scenario space grows since they rely on symbolic proofs, not state-space exploration.

Static analysis is another important class that we generally consider as proving shallow properties (for example, no null pointer dereferences) in a relatively fast, automatic way, which is appealing for large legacy code bases. Actually, the boundary between static analysis and formal methods is sort of like the one between “artificial intelligence” (AI) and other tasks—we call tasks “AI” when they seem hard to us today! So, static analysis typically falls short of establishing functional correctness, and folks are liable to call it *formal methods*, instead, if it is used for functional correctness. In my experience, compared to most computing professionals, cryptographers are relatively quick to agree that it is important to validate that every bit an implementation outputs is correct, if we’re talking about crypto libraries.

**PROTZENKO:** There is a whole spectrum of techniques, ranging from

simple unit tests, moving on to fuzzing and static analysis, and then culminating with full program proofs. Naturally, they all provide different kinds of guarantees, and some of the more subtle bugs that would not be found by, say, fuzzing (because the problem space is too large or finding the bug requires deep mathematical examination that cannot be automated) will be found by program proof. But more often than not, these techniques are complementary: for instance, it is crucial to fuzz and known-answer-test your specifications if you want to have strong trust in your proof that the code meets the specification. Similarly, a quick round of fuzzing can be a great way to make sure your tentative optimization looks solid before attempting to prove it. And perhaps more pragmatically, a tool such as American Fuzzy Lop (AFL),<sup>7</sup> which does not require a substantial time investment, can be a great way to get your management to believe in formal methods.

**WANG:** Two types of assurances can be made by various bug-finding techniques. One is that no bug has been found so far; the other is that any bug is logically impossible. Engineering techniques explore known corner cases to find bugs. If the exploration is not exhaustive, such techniques provide the first type of assurance. Formal methods, on the other hand, try to find proofs for the absence of bugs. If such a proof is found, it is logically impossible to have bugs under the assumptions made by such techniques. Some formal methods even produce witnesses (that is, bugs) when they fail to find proofs.

Consider field arithmetic in cryptographic programs. A field multiplication has hundreds of bits as inputs. Since it is computationally infeasible to exhaustively explore the input space, engineering techniques offer only the first type of assurance by exploring corner cases. I can give at least two accounts where cryptographic programmers missed a carry flag in their code (one intentionally, the other unintentionally). Both buggy

programs successfully passed random and known-answer tests. Using formal methods, inputs witnessing the missed carry and hence yielding incorrect answers were found. These witnessing inputs become a known-answer test for the program. I want to point out that static analysis is a formal method, in my opinion. It tries to construct proofs for the absence of bugs and hence provides the second type of assurance as well.

**COMPUTER:** For cryptographic applications, there are large existing code bases such as open source crypto libraries. Are there effective and efficient ways that formal methods can be applied here?

**BHARGAVAN:** The past few years have seen a number of successful projects applying formal methods to cryptographic code. Code from the Fiat Cryptography<sup>4</sup> project has been integrated into BoringSSL (used in Google Chrome). Code used in OpenSSL has been verified using Verified Software Toolchain,<sup>8</sup> Vale,<sup>9</sup> and CryptoLine.<sup>6</sup> Code from my own project, HACL\*,<sup>3</sup> is deployed in Network Security Services (NSS) (used in Mozilla Firefox) as well as the Linux kernel, WireGuard VPN, Microsoft MsQuic, and the Tezos blockchain. These projects take a variety of approaches; some verify C code, others verify assembly code, and still others generate low-level C or assembly code from verified cryptographic code in domain-specific high-level languages. These techniques are used to prove memory safety, functional correctness, and resistance against some kinds of timing side channels.

**CHLIPALA:** I'd say there are two main ways. One is to apply formal method tools that work on source code in widely deployed languages, such as C. For a good example that meets a high standard of functional correctness and a higher-level security proof, see Beringer et al.<sup>10</sup> The other main strategy is to use formal tools to generate cryptographic code in the first place,

ideally outputting C or assembly code for easy integration with legacy code bases. All the adoptions I highlighted earlier went this route.

**PROTZENKO:** The adoption of formal methods is happening right before our very eyes: BoringSSL and NSS both have replaced large chunks of their code with formally verified variants. New libraries (for example, Linux's Zinc) make it an explicit goal to use as many verified implementations as possible. My hope is that this creates friendly peer pressure and that more legacy libraries are nudged into adopting formally verified implementations.

**WANG:** I think automated or automatic formal methods are more effective and efficient for such libraries. I personally would recommend model checking or static analysis, among others. These two techniques are perhaps the well-established formal methods that have the least human intervention. They have also been used in the hardware industry for decades and in the software industry more recently. Academic and commercial tools are also available. Of course, these techniques are not immediately applicable to crypto libraries at the moment. Using these techniques, successful case studies of selected crypto libraries have been reported. They are the most promising to be applied to open source crypto libraries at large scale, in my opinion.

**COMPUTER:** Can formal methods be used to synthesize implementations of cryptographic algorithms that have security properties other than provable correctness?

**BHARGAVAN:** For low-level algorithms, such as the Advanced Encryption Standard (AES) and Secure Hash Algorithm 3 (SHA-3), the main security goal (beyond function correctness) is side-channel resistance. There are various formal techniques for proving the absence of secret-independent code (for example, see Barthe et al.<sup>11</sup>), which eliminates various kinds of remote



timing attacks. More recently, formal techniques have also been proposed to find and prevent microarchitectural attacks on cryptographic code.

Beyond low-level algorithms, there are many other security properties of interest. One may want to prove that a composite construction [for example, authenticated encryption with associated data (AEAD) and the Rivest–Shamir–Adleman probabilistic signature scheme (RSA-PSS)] provides strong security guarantees, given some assumptions about the underlying algorithms. For example, most of the postquantum key encapsulation mechanism submissions to the National Institute of Standards and Technology competition include a proof of indistinguishability under chosen plaintext attack (IND-CCA) security, and these proofs have been buggy in the past and could benefit from mechanized provers. Going even further, formal methods can be used to synthesize verified implementations of cryptographic protocols, such as TLS,<sup>12</sup> providing strong authentication and secrecy guarantees against powerful network attackers.

**CHLIPALA:** Absolutely! I think freedom from information leaks through timing is a good example (and, no doubt, we will increasingly see work extending results to other potential side channels, such as electromagnetic emissions). Our Fiat Cryptography<sup>4</sup> tool generates code in a restricted language that is constant time by construction, and we hope to explore the extension of such guarantees to a richer output language. The HACL\*<sup>3</sup> team has applied a type system to establish such properties.

**PROTZENKO:** There are many properties of interest beyond functional correctness, notably side-channel resistance. One of the main challenges for the next generation of tools will be to evolve our models and techniques to be able to deal with the latest results in microarchitectural and side-channel attacks, notably Spectre and Meltdown.

**WANG:** It is not entirely clear what “security properties other than provable correctness” means. I will simply interpret the statement as referring to security properties that cannot be proved. By definition, formal methods entail proofs associated with goals. Any security property that can be ensured by formal methods needs to be provable. Subsequently, formal methods cannot synthesize implementations with security properties other than provable correctness. Let me elaborate my points a bit. When formal methods claim that a synthesized program has a security property, there must be an explicit or implicit proof for the claim, by definition. Subsequently, any claimed security property is provable and, in fact, proved. For unprovable security properties, there is neither mathematical nor logical reasoning to prove or disprove such properties. Formal methods just cannot claim whether such security properties hold on synthesized programs.

**COMPUTER:** To what extent do we need to sacrifice the speed of cryptographic algorithms to obtain provable properties of the implementations? How much do we need to sacrifice in terms of portability?

**BHARGAVAN:** Perhaps surprisingly, one does not really need to sacrifice speed. Projects such as Vale<sup>9</sup> and Jamin<sup>5</sup> have been used to build and verify assembly code for cryptographic algorithms that are faster than unverified crypto. CryptoLine<sup>6</sup> verifies manually optimized assembly code from OpenSSL. Fiat Cryptography<sup>4</sup> and HACL\*<sup>3</sup> generate portable C code that is faster than unverified C implementations, and HACL\* can even get very close to assembly speeds.<sup>13</sup> In general, one can choose to forego portability and verify assembly code or sacrifice some performance and verify portable C code. More recently, EverCrypt<sup>14</sup> shows how to mix and match verified assembly from Vale with verified C from HACL\*, hence obtaining portable code that is faster than all prior implementations.

**CHLIPALA:** I don’t think there’s any inherent performance or portability penalty, and, indeed, I expect that, longer term, the adoption of formal methods will improve performance. Yes, as new implementations are written to better support formal methods, they will start out less optimized, and we still need to come up with clever ideas to make some well-known optimizations compatible with tractable correctness proofs. However, I’m confident that the world of formally verified implementations will catch up with the mainstream in the next few years.

At that point, developers will feel freer to experiment with new optimizations since they will be able to relatively quickly patch their old proofs to apply to new code. Don’t underestimate how even the experts can be afraid to modify dusty code bases! For instance, we worked with Google to adopt Fiat Cryptography in the BoringSSL library used in Chrome and elsewhere. They had an idea for a new optimization (based on lookup tables) for the Curve25519 elliptic curve but had been hesitant to touch the AMD64 assembly code for it. Armed with our tool, they generated a C version that, linked with handwritten lookup table code, was actually twice as fast as the original—leaving them happy to retire the largely inscrutable assembly code.

**PROTZENKO:** Recent work by many teams (including our work on EverCrypt) shows that fully verified implementations match or exceed the performance of state-of-the-art unverified implementations. The compromise is no longer about speed but about the effort required to get there and the loss of portability that may result.

I see two compromises emerging. If your goal is to get the fastest implementation at any cost, then this is achievable with sufficient manpower. However, the code may not be reusable for other architectures or instruction sets and will thus have to be duplicated, which creates an

additional maintenance burden in the long run. (Side note: maintaining verified code is something that is currently not discussed enough in the community.) If being within a few percentage points of the best performance is acceptable, then a relatively modest effort may get you a long way. The idea is to stay at a somewhat higher level of abstraction and leave it up to the rest of your toolchain to automatically synthesize, meta-evaluate away, or simply compile this high-level code down to specific target architectures or instruction sets. Such approaches have been advocated by Fiat Cryptography and, in a different context, by our latest work on vectorized HACL\*.

**WANG:** The answers to both questions depend on the implementations under verification. As an extreme case, formal methods have been applied to prove properties of assembly implementations in open source crypto libraries. Such implementations are manually optimized and very efficient. Different implementations are needed to exploit assembly instructions from various architectures. They are hence not portable. Every implementation must be separately verified. At the other extreme, formal methods have been used to verify portable C implementations without any compiler extension. Such implementations are very portable but may not be as efficient as assembly implementations. A number of formal methods are available. I guess it is for cryptographic programmers to decide the tradeoff between efficiency and portability, not formal methods.

**COMPUTER:** In cryptographic applications, do some programming languages lend themselves better to formal methods? How do formal methods interact with manual and compiler optimizations?

**BHARGAVAN:** Cryptographic applications used to always be written in a mix of C and assembly, but more recently,

programmers have started to use higher-level languages such as Java, Rust, and Go. The benefit of these languages is that it becomes easy to eliminate common programming errors, including buffer overruns, using static or dynamic type systems. To apply stronger formal methods, one typically ends up targeting even higher-level verification-oriented languages such as OCaml or F\*. The main disadvantage of using high-level languages is that we now have to verify or trust the compiler. Projects such as the CompCert verified C compiler<sup>15</sup> can help close this trust gap by using only verified optimization, but this comes at some loss in performance. An alternative is to develop a domain-specific crypto-oriented language, such as Cryptol<sup>16</sup> or Jasmin, and build a targeted verified compiler for it.

**CHLIPALA:** Yes, it tends to be more pleasant to do rigorous reasoning about higher-level languages. Proof tools are often built around purely functional languages—think Haskell but even purer! It is especially straightforward to state and prove correctness properties on code written in similar languages. At the same time, it is possible to build up libraries supporting effective and rather automated reasoning about languages as diverse as C, Verilog, and Structured Query Language (SQL)—with all arguments justified from first principles, using the same proof-checking algorithm. Many formal method approaches work well as foundations for ecosystems of verified tools, where we expect most work to be done on programs in high-level languages but where it is worthwhile and feasible to invest in more involved proofs of programs in lower-level languages, and all the proofs fit together in the end.

Optimizations are an interesting question. Like my previous answer highlighted, the chance to adapt an existing correctness proof can make manual optimization much less stressful, especially in security-critical components. Another headache for security-conscious engineers in recent years

is compilers that detect undefined behavior and then feel free to arbitrarily change program behavior. Almost any correctness proof rules out undefined behavior, so we can stop worrying about “rogue compilers” when we commit to a proof of our code, even in grungy languages like C! By the way, there are great applications of formal methods to compilers (for example, the CompCert C compiler<sup>15</sup>), so we can even stop worrying about compiler bugs.

**PROTZENKO:** Programming languages designed with a formal semantics from the get-go generally lend themselves much better to formal methods. This is one of the reasons why the C language remains, to this day, so hard to analyze: debates regularly spring up about fine points of the standard and about the legality or semantics of some particularly vicious programs. But more specifically, functional programming languages, which emphasize values over mutation, lend themselves to much easier verification. Sadly, this is sometimes at odds with maximal performance requirements for cryptography.

Compiler optimizations remain a long-standing problem because formally verified compilers have not yet been adopted by the mainstream. This means that if you are rubber-stamping a piece of C code as “correct,” all your effort may be ruined by a bad compiler optimization. On the other hand, we have grown to depend even more on C compiler optimizations: it is now very easy to rewrite a convoluted piece of C code into a simpler version that better lends itself to formal verification, knowing that any modern compiler will generate code that’s just as efficient.

**WANG:** Certainly, some programming languages enable programmers to prove their programs during software development. In such languages, programmers can be forced to apply formal methods. It is easier to adopt formal methods in such languages. As for manual and compiler optimizations, there are techniques for checking

program equivalence before and after optimizations. The idea is to start with a correct but inefficient implementation. Formal methods can be used to establish equivalence between the correct and optimized implementations. Such techniques have been applied to verify cryptographic programs.

**COMPUTER:** Do formal methods impact the readability of software code? How do you see the value of source code review?

**BHARGAVAN:** The way you write code for verification is sometimes quite

to guarantee correctness. The maintainers of established code bases often want to be able to audit the generated code, even though it has been proved correct. However, they tend to be OK with not reading the machine code their compilers generate or the Verilog for the processors the code runs on! I think we need to shift cultural norms to “audit the lowest-level code that doesn’t sit on top of formally verified components.” So, for instance, when using a formally verified compiler, audit the source code it receives, not the code it outputs in C or assembly or Verilog or whatever. I mean, do we really

another, then we still have work to do to get there. We cannot expect our code to be accepted with no questions asked!

**WANG:** The simplest impact on readability would be documented specifications. Formal methods require clearly specified properties. For instance, input and output ranges for field arithmetic may be slightly relaxed to save a few reductions during a sequence of computation. To apply formal methods, such specifications need to be documented by programmers and hence improve readability. Additionally, more properties can be found and proved during verification. Programs can be annotated with such properties to improve their readability. For example, formal methods may prove that a carry bit is always zero. Such information explains why carry propagation is redundant and hence improves readability. Source code review can bring new insight to the correctness and efficiency of programs. Formal methods can then be applied to justify the insight and improve implementations. I think code review is valuable and independent of formal methods.

Formal methods should not be seen as an absolute, elitist answer to the bugs that plague cryptographic code.

different and counterintuitive for programmers. This can result in code that is not as readable. This is especially the case when the code is synthesized from a higher-level language. In the HACL\* project, we use the KreMLin compiler for F\*,<sup>17</sup> and a lot of engineering effort goes into making the generated C code readable. This code is then manually reviewed by engineers at Mozilla and Linux, and we often have to modify the tool to generate code that is acceptable to these projects. In my opinion, formal verification is only one component of high-assurance cryptographic software. It is still important for the code to be reviewable so that it can be easily understood by programmers who may need to make future modifications. An open problem is how one can reflect the verified invariants in a piece of code in a way that a programmer learns to read and obey when making modifications.

**CHLIPALA:** I think this question gives me the best soap box to spread a public service message, so thanks for asking it! A number of projects are automatically generating low-level cryptographic code, using formal methods

trust human auditors to catch bugs in thousands of lines of assembly? Code review remains invaluable, but let’s do it on code as high level as we can manage, where mistakes tend to jump out at the reader.

**PROTZENKO:** This has historically been a problem, as formal methods use logic, predicates, and syntax that may not be familiar to programmers who have no training in this field. Fortunately, this can be mitigated in a variety of ways: if the source (verification) language is “unfamiliar,” generating code that can be audited helps tremendously. This is the approach we use with Low\*, which generates C code from F\* sources. If the source language is merely annotated (for example, Frama-C), then this means programmers can still understand the code, but verification is harder because the source is not constrained as much. Formal methods should not be seen as an absolute, elitist answer to the bugs that plague cryptographic code: if existing maintainers of open source libraries cannot figure out what it is that we’re doing and cannot review the code one way or

**COMPUTER:** What are the challenges to apply formal methods to software projects that use continuous integration (CI)?

**BHARGAVAN:** Beyond a certain size of a project and a certain number of users, CI becomes an essential tool, but responding to CI failures is a time-consuming task that does not always work well for formal methods. The problem is both technical and cultural. Verification tools take much longer than functional tests, so a CI run can easily go on for hours. Verification tools often use heuristics, so small changes in the code or in the version of a verification tool can sometimes cause a verification failure that is easy to fix but hard for programmers to understand. Finally, the failure of some verification goals can be understood only by Ph.D.-level experts in the verification technique. Consequently, the job of formal methods is



not done with the development of a verified artifact. Verification engineers and software developers need to continue to collaborate to maintain the artifact as it evolves. This is an ongoing challenge for us in the HACLS\* project and, more generally, an open problem for formal verification tools.

**CHLIPALA:** I wouldn't say there are distinctive challenges here. Formal tools fit very well into CI. Many of my projects use Travis CI<sup>18</sup> to recheck proofs on every code check in. Sometimes that rechecking can run for longer than developers are used to, but I expect engineering effort to dramatically reduce those overheads during the coming years.

**PROTZENKO:** It really varies based on the kind of verification you apply, whether you verify existing code in someone else's repository or produce verified code to be consumed by some other project under CI. In my experience producing verified C code, the challenge has been to distribute the verified code in a way that can be easily consumed by downstream users. It's one thing to send a verified piece of code through e-mail for a quick experiment, but making sure the code remains usable at all times and is packaged in a way that requires no manual tweaks is the real challenge. Once this goal has been met, consumers can choose from a variety of options, ranging from rerunning the whole verification pipeline as part of their builds to always using the latest code or manually refreshing it periodically.

**WANG:** Based on my (very) limited knowledge about CI, I believe specifications and scalability would be the main challenges. Interfaces between components in software projects need to be specified for formal methods. Based on these specifications, formal methods can be used to prove the correctness of components or even to synthesize correct components. Applying formal methods still requires significant effort. If components change too often, it

does not appear feasible to apply formal methods for every update. Moreover, interface specifications can be very tedious and prone to errors. If interface specifications also change during software development, errors could be introduced in specifications and thus nullify formal methods. In a very dynamic programming paradigm such as CI, it would not be easy to find correct specifications and verify every update during integration.

**COMPUTER:** How can we ensure that the formally verified source code is also the one that is deployed? Is there value in using reproducible builds?

**BHARGAVAN:** Yes, this is definitely an issue, and reproducible builds as well as software attestation can be a solution.

**CHLIPALA:** Yes, reproducible builds show their value here as elsewhere. However, in the setting of formal methods, it's interesting to consider reproducible builds as a kind of performance optimization of a more fundamental process. When your applications, libraries, compilers, and processors are all proved mechanically, it's possible to formally characterize the build process with a theorem saying, "The output of the following build process is low-level code that meets the following correctness and security properties." Then, any skeptics can run the build process themselves and feel confident that the resulting code is legit. In fact, that method even works for nondeterministic build processes! To save end users the trouble of rerunning builds, trusted authorities can do cryptographic signing of the results of their own builds, promising that they ran the recipes that were proved. Then, if you trust the authority, you can read the theorem statement and feel confident in it, even if you didn't run the build yourself. This workflow is certainly streamlined if every build generates the same bits, but it's not essential.

**PROTZENKO:** I believe a lot of standard practices used elsewhere in


software development should also be applied to projects that perform formal verification. If a verification project is not doing reproducible builds (for example, Docker, Vagrant, or others), has no CI, or cannot be tried out easily by a first-timer, then we are doing ourselves a disservice, and we won't look good from the point of view of the very people we are trying to convince!

**WANG:** Technically, formally verified source codes are never deployed. They have to be compiled into executable binaries for deployment. From source codes to binary executables, many complicated transformations are required. It is hence extremely difficult to ensure that formally verified source codes are always correctly compiled into executable binaries. There are, indeed, certified C compilers with formally verified compilation. Such compilers still miss commonly used language extensions and generate less-efficient binary codes, and hence they are not yet widely adopted by developers. I think the best way to ensure correct binary codes is to verify assembly codes from programmers or compilers. Even so, assemblers can introduce errors. It is never easy to obtain correct binary codes before deployment.

Assume correct binary codes are available. Formal techniques have been developed to ensure correct deployment. In proof-carrying code, low-level codes are shipped with their proofs of correctness. Shipped codes will not be executed until their proofs are verified. I am not familiar with reproducible builds. But the technology appears to assume the correctness of compilation. I fail to see why reproducible errors introduced by compilers can ensure the correct deployment of formally verified source codes.

**T**his concludes the questions that we had for our panelists. But *Computer* welcomes your input, so we still have a few questions left for you.

- › Have you heard about formal methods before, and did the panelists' answers change your understanding?
- › Did you know that you may already be using formally verified cryptography if you are reading this article online using Google Chrome or Mozilla Firefox?
- › What are your thoughts about the challenges and opportunities when formal methods are applied to real-world applications?

Feel free to let us know. We hope you enjoyed the discussion and that you agree that it will be interesting to keep an eye on future developments in this area. 

## REFERENCES

1. M. Barbosa et al., "SoK: Computer-aided cryptography," *IACR Cryptol. ePrint Arch.*, vol. 2019, Art. No. 1393. Dec. 2019.
2. K. Bhargavan et al., "Everest: Towards a verified, drop-in replacement of HTTPS," in *Proc. 2nd Summit Adv. Program. Lang. (SNAPL 2017)*, 2017, no. 1, pp. 1:1–1:12.
3. J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL\*: A verified modern cryptographic library," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 1789–1806.
4. A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic – With proofs, without compromises," in *Proc. IEEE Symp. Security Privacy (SP)*, 2019, pp. 1202–1219. doi: 10.1109/SP.2019.00005.
5. J. B. Almeida et al., "Jasmin: High-assurance and high-speed cryptography," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 1807–1823.
6. Y.-F. Fu, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Signed cryptographic program verification with typed CryptoLine," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 1591–1606.
7. M. Zalewski. "American Fuzzy Lop (AFL)." Accessed: Aug. 10, 2020. [Online]. Available: <https://lcamtuf.coredump.cx/afl/>
8. A. W. Appel, "Verified software toolchain," in *European Symposium on Programming, LNCS 6602*, G. Barthe, Ed. Berlin: Springer-Verlag, 2011, pp. 1–17.
9. B. Bond et al., "Vale: Verifying high-performance cryptographic assembly code," in *Proc. 26th USENIX Security Symp. (USENIX Security 17)*, 2017, pp. 917–934.
10. L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *Proc. 24th USENIX Security Symp. (USENIX Security 15)*, 2015, pp. 207–221.
11. G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic 'constant-time'," in *Proc. IEEE 31st Comput. Security Found. Symp. (CSF)*, 2018, pp. 328–343. doi: 10.1109/CSF.2018.00031.
12. A. Delignat-Lavaud et al., "Implementing and proving the TLS 1.3 record layer," in *Proc. IEEE Symp. Security Privacy (SP)*, 2017, pp. 463–482. doi: 10.1109/SP.2017.58.
13. M. Polubelova et al., "HACL×N: Verified Generic SIMD crypto (for all your favorite platforms)," *IACR Cryptol. ePrint Arch.*, vol. 2020, Art. No. 572. May 2020.
14. J. Protzenko et al., "EverCrypt: A fast, verified, cross-platform cryptographic provider," in *Proc. IEEE Symp. Security Privacy (SP)*, 2020, pp. 983–1002. doi: 10.1109/SP40000.2020.00114.
15. X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. doi: 10.1145/1538788.1538814.
16. R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench," in *Proc. Working Conf. Verified Softw.: Theories, Tools, Exper., LNCS 9971*, 2016, pp. 56–72.
17. J. Protzenko et al., "Verified low-level programming embedded in F\*," in *Proc. ACM Program. Lang.*, vol. 1, Aug. 2017, Art. No. 17. doi: 10.1145/3110261.
18. K. Palmkog, A. Celik, and M. Gligoric, "piCoq: Parallel regression proving for large-scale verification projects," in *Proc. 27th ACM SIGSOFT Int. Symp. Software Testing Anal.*, 2018, pp. 344–355. doi: 10.1145/3213846.3213877.

**NICKY MOUHA** is a contractor at the National Institute of Standards and Technology, Gaithersburg, Maryland, USA, employed by Stratavia, USA. Contact him at [nicky.mouha@nist.gov](mailto:nicky.mouha@nist.gov).

**ASMAA HAILANE** is a guest researcher at the National Institute of Standards and Technology, Gaithersburg, Maryland, USA. Contact her at [asmaa.hailane@nist.gov](mailto:asmaa.hailane@nist.gov).

**75 YEARS** IEEE COMPUTER SOCIETY  
**DIGITAL LIBRARY**

Access all your IEEE Computer Society subscriptions at  
**computer.org**  
**/mysubscriptions**