Hacker-Proof Code Confirmed

Computer scientists can prove certain programs to be error-free with the same certainty that mathematicians prove theorems. The advances are being used to secure everything from unmanned drones to the internet.

By Kevin Hartnett

In the summer of 2015 a team of hackers attempted to take control of an unmanned military helicopter known as Little Bird. The helicopter, which is similar to the piloted version long-favored for U.S. special operations missions, was stationed at a Boeing facility in Arizona. The hackers had a head start: At the time they began the operation, they already had access to one part of the drone's computer system. From there, all they needed to do was hack into Little Bird's onboard flight-control computer, and the drone was theirs.

When the project started, a "Red Team" of hackers could have taken over the helicopter almost as easily as it could break into your home Wi-Fi. But in the intervening months, engineers from the Defense Advanced Research Projects Agency (DARPA) had implemented a new kind of security mechanism — a software system that couldn't be commandeered. Key parts of Little Bird's computer system were unhackable with existing technology, its code as trustworthy as a mathematical proof. Even though the Red Team was given six weeks with the drone and more access to its computing network than genuine bad actors could ever expect to attain, they failed to crack Little Bird's defenses.

"They were not able to break out and disrupt the operation in any way," said Kathleen Fisher, a professor of computer science at Tufts University and the founding program manager of the High-Assurance Cyber Military Systems (HACMS) project**.** "That result made all of DARPA stand up and say, oh my goodness, we can actually use this technology in systems we care about."

The technology that repelled the hackers was a style of software programming known as formal verification. Unlike most computer code, which is written informally and evaluated based mainly on whether it works, formally verified software reads like a mathematical proof: Each statement follows logically from the preceding one. An entire program can be tested with the same certainty that mathematicians prove theorems.

"You're writing down a mathematical formula that describes the program's behavior and using some sort of proof checker that's going to check the correctness of that statement," said Bryan Parno, who does research on formal verification and security at Microsoft Research.

The aspiration to create formally verified software has existed nearly as long as the field of computer science. For a long time it seemed hopelessly out of reach, but advances over the past decade in so-called "formal methods" have inched the approach closer to mainstream practice. Today formal

software verification is being explored in well-funded academic collaborations, the U.S. military and technology companies such as Microsoft and Amazon.

The interest occurs as an increasing number of vital social tasks are transacted online. Previously, when computers were isolated in homes and offices, programming bugs were merely inconvenient. Now those same small coding errors open massive security vulnerabilities on networked machines that allow anyone with the know-how free rein inside a computer system.

“Back in the 20th century, if a program had a bug, that was bad, the program might crash, so be it,” said Andrew Appel, professor of computer science at Princeton University and a leader in the program verification field. But in the 21st century, a bug could create “an avenue for hackers to take control of the program and steal all your data. It’s gone from being a bug that’s bad but tolerable to a vulnerability, which is much worse,” he said.

The Dream of Perfect Programs

Kathleen Fisher, a computer scientist at Tufts University, leads the High-Assurance Cyber Military Systems (HACMS) project.

In October 1973 Edsger Dijkstra came up with an idea for creating error-free code. While staying in a hotel at a conference, he found himself seized in the middle of the night by the idea of making programming more mathematical. As he explained in a later reflection, “With my brain burning, I left my bed at 2:30 a.m. and wrote for more than an hour.” That material served as the starting point for his seminal 1976 book, “A Discipline of Programming,” which, together with work by Tony Hoare (who, like Dijkstra, received the Turing Award, computer science’s highest honor), established a vision for incorporating proofs of correctness into how computer programs are written.

It’s not a vision that computer science followed, largely because for many years afterward it seemed

impractical — if not impossible — to specify a program's function using the rules of formal logic.

A formal specification is a way of defining what, exactly, a computer program does. And a formal verification is a way of proving beyond a doubt that a program's code perfectly achieves that specification. To see how this works, imagine writing a computer program for a robot car that drives you to the grocery store. At the operational level, you'd define the moves the car has at its disposal to achieve the trip — it can turn left or right, brake or accelerate, turn on or off at either end of the trip. Your program, as it were, would be a compilation of those basic operations arranged in the appropriate order so that at the end, you arrived at the grocery store and not the airport.

The traditional, simple way to see if a program works is to test it. Coders submit their programs to a wide range of inputs (or unit tests) to ensure they behave as designed. If your program were an algorithm that routed a robot car, for example, you might test it between many different sets of points. This testing approach produces software that works correctly, most of the time, which is all we really need for most applications. But unit testing can't guarantee that software will always work correctly because there's no way to run a program through every conceivable input. Even if your driving algorithm works for every destination you test it against, there's always the possibility that it will malfunction under some rare conditions — or "corner cases," as they're called — and open a security gap. In actual programs, these malfunctions could be as simple as a buffer overflow error, where a program copies a little more data than it should and overwrites a small piece of the computer's memory. It's a seemingly innocuous error that's hard to eliminate and provides an opening for hackers to attack a system — a weak hinge that becomes the gateway to the castle.

"One flaw anywhere in your software, and that's the security vulnerability. It's hard to test every possible path of every possible input," Parno said.

Actual specifications are subtler than a trip to the grocery store. Programmers may want to write a program that notarizes and time-stamps documents in the order in which they're received (a useful tool in, say, a patent office). In this case the specification would need to explain that the counter always increases (so that a document received later always has a higher number than a document received earlier) and that the program will never leak the key it uses to sign the documents.

This is easy enough to state in plain English. Translating the specification into formal language that a computer can apply is much harder — and accounts for a main challenge when writing any piece of software in this way.

"Coming up with a formal machine-readable specification or goal is conceptually tricky," Parno said. "It's easy to say at a high level 'don't leak my password,' but turning that into a mathematical definition takes some thinking."

To take another example, consider a program for sorting a list of numbers. A programmer trying to formalize a specification for a sort program might come up with something like this:

> For every item j in a list, ensure that the element $j \leq j+1$

Yet this formal specification — ensure that every element in a list is less than or equal to the element that follows it — contains a bug: The programmer assumes that the output will be a permutation of the input. That is, given the list [7, 3, 5], she expects that the program will return [3, 5, 7] and satisfy the definition. Yet the list [1, 2] also satisfies the definition since "it is *a* sorted list, just not the sorted list we were probably hoping for," Parno said.

In other words, it's hard to translate an idea you have for what a program should do into a formal specification that forecloses every possible (but incorrect) interpretation of what you want the program to do. And the example above is for something as simple as a sort program. Now imagine taking something much more abstract than sorting, such as protecting a password. "What does that mean mathematically? Defining it may involve writing down a mathematical description of what it means to keep a secret, or what it means for an encryption algorithm to be secure," Parno said. "These are all questions we, and many others, have looked at and made progress on, but they can be quite subtle to get right."

Block-Based Security

Between the lines it takes to write both the specification and the extra annotations needed to help the programming software reason about the code, a program that includes its formal verification information can be five times as long as a traditional program that was written to achieve the same end.

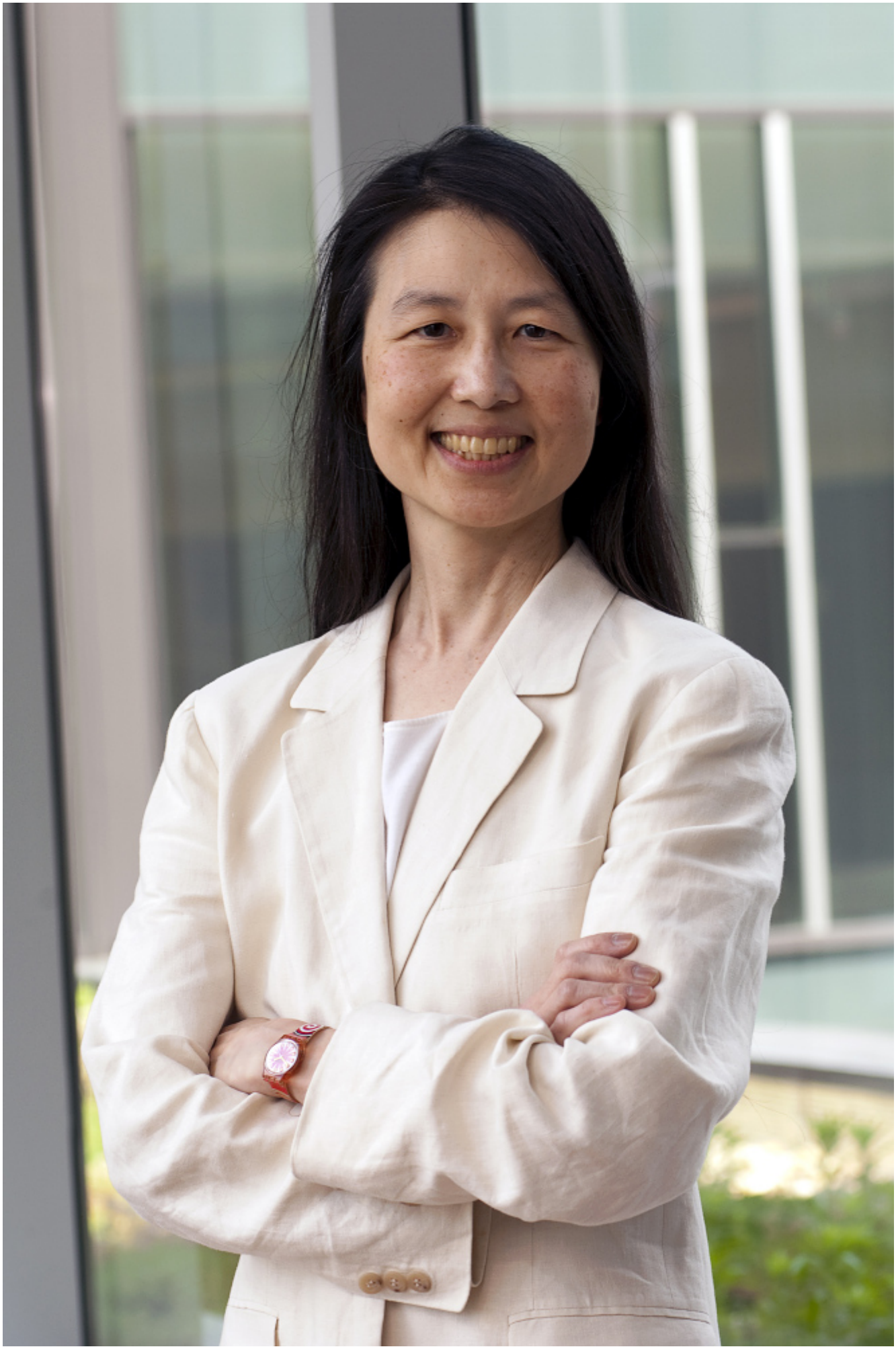
This burden can be alleviated somewhat with the right tools — programming languages and proof-assistant programs designed to help software engineers construct bombproof code. But those didn't exist in the 1970s. "There were many parts of science and technology that just weren't mature enough to make that work, and so around 1980, many parts of the computer science field lost interest in it," said Appel, who is the lead principal investigator of a research group called DeepSpec that's developing formally verified computer systems.

Even as the tools improved, another hurdle stood in the way of program verification: No one was sure whether it was even necessary. While formal methods enthusiasts talked of small coding errors manifesting as catastrophic bugs, everyone else looked around and saw computer programs that pretty much worked fine. Sure, they crashed sometimes, but losing a little unsaved work or having to restart occasionally seemed like a small price to pay for not having to tediously spell out every little piece of a program in the language of a formal logical system. In time, even program verification's earliest champions began to doubt its usefulness. In the 1990s Hoare — whose "Hoare logic" was one of the first formal systems for reasoning about the correctness of a computer program — acknowledged that maybe specification was a labor-intensive solution to a problem that didn't exist. As he wrote in 1995:

> Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalization…. It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve.

Then came the internet, which did for coding errors what air travel did for the spread of infectious diseases: When every computer is connected to every other one, inconvenient but tolerable software bugs can lead to a cascade of security failures.

"Here's the thing we didn't quite fully understand," Appel said. "It's that there are certain kinds of software that are outward-facing to all hackers in the internet, so that if there is a bug in that software, it might well be a security vulnerability."



Jeannette Wing, a computer scientist at Microsoft Research, is developing a formally verified protocol for the internet.

By the time researchers began to understand the critical threats to computer security posed by the internet, program verification was ready for a comeback. To start, researchers had made big advances in the technology that undergirds formal methods: improvements in proof-assistant programs like Coq and Isabelle that support formal methods; the development of new logical systems (called dependent-type theories) that provide a framework for computers to reason about code; and improvements in what's called "operational semantics" — in essence, a language that has the right words to express what a program is supposed to do.

"If you start with an English-language specification, you're inherently starting with an ambiguous specification," said Jeannette Wing, corporate vice president at Microsoft Research. "Any natural language is inherently ambiguous. In a formal specification you're writing down a precise specification based on mathematics to explain what it is you want the program to do."

In addition, researchers in formal methods also moderated their goals. In the 1970s and early 1980s, they envisioned creating entire fully verified computer systems, from the circuit all the way to the programs. Today most formal methods researchers focus instead on verifying smaller but especially vulnerable or critical pieces of a system, like operating systems or cryptographic protocols.

"We're not claiming we're going to prove an entire system is correct, 100 percent reliable in every bit, down to the circuit level," Wing said. "That's ridiculous to make those claims. We are much more clear about what we can and cannot do."

The HACMS project illustrates how it's possible to generate big security guarantees by specifying one small part of a computer system. The project's first goal was to create an unhackable recreational quadcopter. The off-the-shelf software that ran the quadcopter was monolithic, meaning that if an attacker broke into one piece of it, he had access to all of it. So, over the next two years, the HACMS team set about dividing the code on the quadcopter's mission-control computer into partitions.

The team also rewrote the software architecture, using what Fisher, the HACMS founding program manager, calls "high-assurance building blocks" — tools that allow programmers to prove the fidelity of their code. One of those verified building blocks comes with a proof guaranteeing that someone with access inside one partition won't be able to escalate their privileges and get inside other partitions.

Later the HACMS programmers installed this partitioned software on Little Bird. In the test against the Red Team hackers, they provided the Red Team access inside a partition that controlled aspects of the drone helicopter, like the camera, but not essential functions. The hackers were mathematically guaranteed to get stuck. "They proved in a machine-checked way that the Red Team would not be able to break out of the partition, so it's not surprising" that they couldn't, Fisher said. "It's consistent with the theorem, but it's good to check."

In the year since the Little Bird test, DARPA has been applying the tools and techniques from the HACMS project to other areas of military technology, like satellites and self-driving convoy trucks. The new initiatives are consistent with the way formal verification has spread over the last decade: Each successful project emboldens the next. "People can't really have the excuse anymore that it's too hard," Fisher said.

Verifying the Internet

Security and reliability are the two main goals that motivate formal methods. And with each passing day the need for improvements in both is more apparent. In 2014 a small coding error that would have been caught by formal specification opened the way for the Heartbleed bug, which threatened to bring down the internet. A year later a pair of white-hat hackers confirmed perhaps the biggest fears we have about internet-connected cars when they successfully took control of someone else's Jeep Cherokee.

As the stakes rise, researchers in formal methods are pushing into more ambitious places. In a return to the spirit that animated early verification efforts in the 1970s, the DeepSpec collaboration led by Appel (who also worked on HACMS) is attempting to build a fully verified end-to-end system like a web server. If successful, the effort, which is funded by a $10 million grant from the National Science Foundation, would stitch together many of the smaller-scale verification successes of the last decade. Researchers have built a number of provably secure components, such as the core, or kernel, of an operating system. "What hadn't been done, and is the challenge DeepSpec is focusing on, is how to connect those components together at specification interfaces," Appel said.

Over at Microsoft Research, software engineers have two ambitious formal verification projects under way. The first, named Everest, is to create a verified version of HTTPS, the protocol that secures web browsers and that Wing refers to as the "Achilles heel of the internet."

The second is to create verified specifications for complex cyber-physical systems such as drones. Here the challenge is considerable. Where typical software follows discrete, unambiguous steps, the programs that tell a drone how to move use machine learning to make probabilistic decisions based on a continuous stream of environmental data. It's far from obvious how to reason about that kind of uncertainty or pin it down in a formal specification. But formal methods have advanced a lot even in the last decade, and Wing, who oversees this work, is optimistic formal methods researchers are going to figure it out.

Correction: This article was revised on September 21 to clarify that in formally verified software, each statement follows logically from the preceding one, not from the next.

This article was reprinted on Wired.com.