MIT CSAIL Alliances | Adam Chlipala Podcast Export 1

Welcome to MIT's Computer Science and Artificial Intelligence Labs Alliance's podcast series. My name is Steve Lewis. I'm the Assistant Director of Global Strategic Alliances for CSAIL at MIT. In this podcast series, I will interview principal researchers at CSAIL to discover what they're working on and how it will impact society.

Adam Chlipala has been on the faculty in computer science at MIT since 2011. He did his undergrad at Carnegie Mellon and his PhD at Berkeley. And his research focuses on clean slate redesign of computer systems infrastructure, typically taking advantage of machine check proofs of functional correctness.

Much of his work uses the Coq Proof Assistant, about which he has written a popular book*Certified Programming with Dependent Types.* He most enjoys finding opportunities for drastic simplification of over incumbent abstractions in computer systems. And some of his favorite tools toward that end are object capability systems, transactions, proof carrying code, and high level languages with whole program optimizing compilers.

Some projects particularly, far along in the real world adoption curve are fiat cryptography for proof producing generation of low level cryptographic code. Today, run by Chrome for most HTTPS connections and Ur/Web, a production quality domain specific language for web applications. Adam, thanks for your time today. What prompted the development of a Fiat cryptography?

So the story for me begins with attending some workshops that were run to bring together the people who build and maintain the most popular cryptographic software with experts in academia and elsewhere on formal methods, and programming languages, and other research topics like that just to try to see what some of the potential biggest impact areas were for creating new connections between those groups. And at those workshops, I heard several people who are deep in the guts of implementing cryptographic libraries say, that one of the parts of the code they were least confident in was the big integer arithmetic that is behind something called elliptic curve cryptography, which is preferentially used by web browsers and a lot of other places to create secure connections. So I was surprised at the time to hear that arithmetic was this cutting edge implementation challenge. But it turned out to be true when you're super concerned with getting every last bit of performance out of your implementation.

So I came back to MIT thinking about that problem. And it turned out I was just starting to work with an undergraduate who approached me about doing research. And he'd been thinking about this problem too from his experience implementing related code. And we decided it was a good sign. And we should start working on it. And the rest is history.

Very good. So how does Fiat cryptography differ from what came before and what are the benefits of automating this process?

Yeah, so let me distinguish between two different levels of what came before. One is just the mainstream process of building cryptographic libraries or other cryptographic software in say, OpenSSL. That's the most popular library in the space, where say, looking back 10 years, no very strong mathematical assurance that the code actually does what it's supposed to do. They rely on testing to run their routines through many different inputs and check that the right outputs come out. But the set of possible inputs is so large, testing can only cover a small portion of that space.

So there could easily be bugs lurking in tricky corner cases that actually have significant security consequences. And the tricky thing about security is, even if there's only one input that makes your program do something terrible, you have to assume that the bad guys out there are going to be smart enough to find that input. So that's how it's very different from traditional, let's say, reliability concerns as opposed to security concerns. So a number of projects then started thinking about how to get stronger assurance about the correct behavior of this code. And some of them had even produced, essentially, machine checked mathematical proofs that particular programs do the right thing.

But someone still needed to write the code that was going to be checked. And someone still needed to write a program specific proof. So we thought we could do things differently and actually generate the code automatically along with its proof by identifying families of related program functionality that were subject to unified correctness arguments saying, any program in this family is correct. And so we then embody a particular implementation strategy. Pick an algorithm, say, one of the algorithms behind elliptic curve cryptography. And then say, tell me which large prime number you want to use as a parameter for this algorithm.

And I'm going to generate the fast code for you along with a proof that it implements the traditional whiteboard level math. And that was the really interesting thing about this area, namely that the state of the art before we got involved was that every time you changed that prime number, that's a parameter to your algorithm. One of a pretty small set of experts in the world rewrites the code from scratch in C, your assembly because that was viewed as essential to get good enough performance. So we showed both that, you could automate that process, and that you could do it in a way that generates the highest level of formal assurance with a proof that can be independently checked that the code that comes out really implements the original algorithms pick.

That's fascinating. Are there any potential downsides to automating cryptography.

Probably the biggest downside is that it's still a pretty new idea. A lot of practitioners are skeptical about it. They really want to read the code that comes out of this process and separately convince themselves that it's reasonable. And it takes a lot of work to build an automatic program writer that creates code that actually looks nice and is easy to understand. And we're at a medium stage on the spectrum of how hard we work on that and how successful we are at creating code that can be read if the person reading it has enough patience. Another potential downside is, you have to be very clear on what makes a generated program good enough.

You need a completely formal mathematical characterization of the properties the program is supposed to have. And once you agree on that, you can trust a tool that's been proved to only generate code that meets your requirements. But if you forgot something from the requirements list, then you have no guarantee about the behavior of the code that's going to pop out. And as a result, it could surprise you, say, you forgot to think about a very important property in cryptography, which is having the timing of your code not leak your secrets to someone who's observing the execution. Well, we've never formalized that property in our formal specifications.

Though by focusing on a particular class of algorithms, we're able to use an output language that just by construction doesn't have say, conditionals in it. So it avoids timing leaks in a certain low tech way. But in general, you have to be very careful that your formal guarantees cover what people in the real world actually want. And if they don't, then maybe someone still needs to inspect the output of the process. And it could be pretty big, and intimidating, and hard to read.

I see, so why is guaranteeing correctness so important to cryptographic code?

I would say it's because, you've had people who are essentially mathematicians think really hard about which algorithms are actually going to present the cryptographic properties that motivate whatever piece of code it is you're talking about, say that hide your secret information being transmitted to a website, or something like that. And the details really matter here. And if you've actually implemented a subtly different algorithm from the one that the theorists spend a lot of time designing improving, then all bets are off in terms of protecting your secret data, preventing bad actors from manipulating your data, all those kind of properties that cryptographic systems protect.

And at the same time, the code that's written for the low level arithmetic that our project focuses on, is pretty long and intricate, easily hundreds or thousands of lines of fairly repetitive structure. There's no obvious rhyme or reason that non-specialist perceives looking at it for the first time. So you can imagine that just auditing it manually is imperfect and likely to lead to say, some classic errors that showed up here are the wrong variable name is referenced on one line out of 10,000. In other words, the program is correct. But who knows if this one change will allow your adversary to get access to all your secrets. So the combination of high stakes for correctness issues, voiding the warranty, so to speak, and long and complex code make this an appealing place to get strong mathematical guarantees.

And what are some of the potential industry uses for Fiat cryptography?

Probably easiest to start with the uses that are already in production today. So I think all of Chrome, Firefox, and we think probably, Safari, though we're not sure, are running our code to basically help the web browser properly initiate a secure connection to whatever website it's talking to. So that involves, for instance, helping your web browser realize, when you do a Google search, it's actually Google you're talking to, as well as initiating the creation of a secure channel between your browser and Google using a protocol called key exchange. So both of those, when the elliptic curve style of secure web browsing is activated, which most web browsers and most web servers are trying to do by default these days. If you're running Chrome, then it's probably running our code that was generated with proof of correctness.

And I think, 32-bit Firefox uses our code with 64-bit, 1,000, or something like that. So we're also on Google servers, we're on, I think, all new Android distributions. In the Linux kernel, we're in a few different blockchain implementations. So these are all people who need elliptic curve primitives to do a variety of different things. But the nice thing is these primitives are standardized enough that we can do the work once and just produce what looks almost like a special compiler, just like, GCC, that you can call with a description of your cryptographic scenario and then it just prints out the C code, or the rest code, or go code, we have a few different back ends that you can bring into your project as opposed to having to write all that by hand or copy it from some other source.

That's quite a large install base for sure. That's awesome. And with Fiat cryptography providing such potential seismic shifts to the internet landscape, what are the next challenges that you are excited to tackle?

Well, there are other parts of the-- let's call it the stack of cryptographic systems, that Fiat cryptography project doesn't tackle. It's actually a pretty narrowly focused project in that it traditionally just looked at the big integer arithmetic that's at the heart of these algorithms. And there's other stuff around the core integer arithmetic, both, let's say, higher level and lower level parts. The higher level parts would include say, a whole cryptographic algorithm like key exchange, which uses the arithmetic as subroutines. But involves a lot more to it.

So we're looking into how to extend our approach to that additional code where we are still starting from nice high level presentations of algorithms. And then, correct by construction way, deriving the efficient low level code that realizes those ideas. And we're also looking at how to support the cryptographic protocols that are defined on top of those primitives, which basically lay out the complete dance between two or more parties who sends messages to whom, when, and the right way to process messages that are coming in and change your state. So we have a paper coming up at the Principles of Programming Languages Conference in January, where we show a new approach to doing that using the same theorem Coq that is behind fiat cryptography. And we're looking forward to integrating those two projects at some point eventually.

And then at lower levels of the stack, fiat cryptography has traditionally bottomed out in languages like C, where you're still going to run a conventional compiler. And you're going to trust it to compile correctly and not introduce any new security issues. So that makes it appealing to try to get guarantees of the level of assembly code or machine code instead. And we're looking into ways to do that by doing some automatic checking of equivalence of a particular assembly program with the higher level outputs that we've previously been producing. So you can get the performance benefits of working with even lower level code without sacrificing any of the formal guarantees.

For those unfamiliar, what is-- is it Coq? Is that how you pronounce it?

So it's a French word for rooster and I pronounce it "cock." And it is a general theorem improving environment. And it basically has a language, just like a programming language for defining systems and writing down specifications of how they're meant to behave. As well as, writing out formal mathematical proofs establishing that those systems actually meet the specifications that you've written down. And the nice thing about this system, which is called a proof assistant, is that you can have completely automatic checking of proofs. So if you've received say, a piece of software from someone you don't trust at all, you can read the specification that this person claims to have approved. And decide if the software really met that property, would that be good enough for me to be happy to run it.

And then you can run the proof checker using the proof provided by the author of the system. And if the proof checker gives a thumbs up, then the system meets its specification, you might be OK running it even if you haven't looked at a single line of its source code. So that's the real promise of this technology, which I think will be increasingly relevant for all scenarios. And people piecing together components from different providers and still wanting to have a lot of confidence in their correctness and security of the final product.

Let's switch gears a little bit and talk about Verified IoT Lightbulb. What is that?

Yeah, so that is basically, a proof of concept we created to start learning about how to apply these formal methods techniques to certify the correctness of complete hardware software systems. And to begin with, we're focused on what you either call embedded systems, or internet of things systems depending on which decade's terminology you're into I suppose.

So the idea is, systems that are simple enough that you can get a bunch of grad students re-implementing all the software, and all the digital hardware from scratch. And you still get an output that actually does something at least somewhat useful. And the concrete example we chose there was, an internet connected light bulb, which has a traditional wired ethernet network card which can receive packets that basically get to say turn the light bulb off or on. And there's a light bulb plugged into the processor that's receiving those packets.

So our prototype is using an FPGA or a reconfigurable hardware platform to execute the processor whose hardware design we built. And that's the only big qualitative difference from a real IoT system here. And it's not so much that we thought that the light bulb example itself was such a grand challenge problem. It's just a nice initial test case, building up tools that can be used to prove the correctness and security of all the hardware and software in a particular system. So we're much more interested in, for instance, we developed a new low level systems programming language called Bedrock too. We created a compiler to RISC-V machine code for it.

RISC V being a popular open hardware instruction set, which is a competitor to x86 and ARM. So we have that compiler, we've proved that correct. We have tools for proving correctness of individual programs written in this language, Bedrock too. And we have also created a RISC-V computer processor with a formal proof of correctness. And connected that proof to the compiler proofs, so you get what you could call, a full stack correctness theorem. And the reason that that's valuable is that computer systems typically have multiple complex layers.

And the interfaces that the layers expose to others can be pretty complicated. One example of a complicated interface is the C programming language or one of the x86 machine code formats. And when you do a formal verification, if you say, look just at the compiler or just at the processor, you might worry that you made a mistake formalizing what C programs mean or what x86 machine programs-- machine code programs mean. But when you bring everything together into one formerly verified stack, you no longer have to worry about making mistakes in those places. Because the machine code language and the system's programming language become purely internal interfaces within the complete system you've built. That let's say, assumes the details of a hardware description language at the bottom, and assumes some particular application level specification at the top.

And then, if you made a mistake in one of those intermediate layers, like you messed up the description of how a particular assembly instruction runs. You must run into that bug in the course of doing your proof. You won't be able to finish the end to end proof unless you got all those details right to the extent needed for your particular system configuration. There might be an instruction that's misimplemented in your processor, but actually your software never uses it. Then you'd be able to finish the proof potentially.

But you have to catch any bugs that are directly relevant to the behavior of your system. And the top level behavioral theorem that we chose to use here was, basically like a regular expression style characterization of considering all the ones and zeros coming in on the input wires of the system through its execution. What ones and zeros are allowed on the output wires at each instant. So this completely characterizes when the light bulb is supposed to be on or not based on what shapes of packets have come in so far. So if there was any buffer overflow problem in the system that let an attacker overwrite the code and do something completely different, you'd have to catch it doing this proof. Because then, there's no way the system would always have the light bulb on or off at the right moment, because the attacker could override the code that controls that behavior.

And likewise, if there was some debugging mode in the processor that you didn't realize was there to let anyone send some special signal and take over the processor and make do arbitrary things. You'd have to catch that in your proof too. Because that also would completely circumvent the intent of the system.

And so talking about the broader goals of the Verified IoT Lightbulb, what industry applications do you see for this type of system? Are you developing a reference architecture for chip manufacturers or something similar?

Well, we'd like to see this approach to thinking about specifications and proofs be used broadly throughout all the different industry players who contribute to real computer systems, whether they're labeled as IoT or not. So it would be great to see. And I think we will see more and more of open source computer processor designs. There are already some popular ones associated with the RISC V architecture family that I mentioned we're using.

And once you have those designs, we hope it will be feasible to distribute the design with the proof of its correctness in the sense that we've used in this project. And therefore, enable people whether they're hobbyists, or gigantic companies to integrate those processors into larger projects to create systems that have complete formal proofs of correctness. And IoT seems like a great place to focus because all the code behind a system can be relatively short.

And it's still relatively young field where we hope there's more room to influence best practices. And so we're doing our best to develop reusable tools to make it possible to certify new systems that remix existing components and add a few new ones, where hopefully, the new proof effort is proportional to the amount of truly new stuff that you've introduced. You don't have to repeat work for components that have been handled in past projects and have had their nice specifications and proofs created and say, made available on GitHub, or however people are going to be distributing hardware and software components for this ecosystem.

That's fascinating. Do you have any other research or cryptography news that you're excited to share?

Let's see, I may have unintentionally got into some of this one in response to an earlier question. I can mention one other at least security-related project, which is thinking about the topic of hardware enclave systems and other approaches to get help from your processor in running a bunch of pieces of code simultaneously, where the authors of those different pieces of code don't trust each other and don't want one of these simultaneously running programs to eavesdrop on them or influence their execution. And there are well-known systems like, Intel's SGX, that provide some features along these lines. But then there was also prior research by my collaborators in CSAIL like Professor Srini Devadas that showed some security issues in SGX like how it could actually leak some secrets through the timing of code, not just the direct traditional input and output channels.

So that called for some redesign of enclave systems. And especially when some security attacks known as Specter and Meltdown became known in 2018, then those revealed other problems that these systems can have, or they don't actually provide the information leakage protections that folks thought they did. So we've been looking at is, regardless of what implementation strategy you use for a system that provides process isolation and hardware, how can you prove that it truly accomplishes that? What does process isolation actually mean here, and what are the pragmatics in demonstrating it for particular systems? So we've been looking at-- to begin with, creating really simple enclave systems. But in actual hybrid designs that you can at least run on an FPGA and probably fabricate in Silicon eventually.

And then, building up all the machinery. And proving that those designs actually do enforce this strong isolation where different simultaneously running processes, can't learn anything about each other by observing timing. And this has implications up through the software stack because coming full circle here, in cryptography, there's a lot of concern around leaking information through timing. There are even special software coding guidelines that you can follow to be able to count on hardware providing a really strong timing independence for your code. So we've done a little research on that as well.

And we'd hope to be able to bring all this together to guarantee that when your cryptographic software passes particular tests about coding guidelines for timing independence, then when you run it on this hardware, even if this other process is running at the same time, that we can really provide complete isolation. We don't leak any of your secret message, or your secret key through timing that can be observed between processes.

That's great. Well, Adam, is there a website that you can direct our listeners to if they're interested in learning more about your research?

A good starting point should just be my home page, which is my first name dot my last name dot net, adam.chlipala.net. And a bunch of our projects are on GitHub. So you can find fiat-crypto, for instance, on the MIT-plv organization on GitHub. And the project is fiat-crypto.

Great, very, very fascinating topic. And good luck with the rest of your research. And I appreciate your time today, Adam. Thank you.

Thank you.

If you're interested in learning more about the CSAIL alliance program and the latest research at CSAIL, please visit our website at cap.csail.mit.edu. And listen to our podcast series on Spotify, Apple Music, or wherever you listen to your podcasts. Tune in next month for a brand new edition of the CSAIL Alliance's podcast and stay ahead of the curve.