Formal Methods in Cryptography

Zach Flores

Presented at the NIST Crypto Reading Club, 2024-April-17 (virtual)

Outline

Is the Code you Wrote Correct?

Whenever we program something, how can we be sure what it returns is correct? Can we always trust that every programmer everywhere has done their due diligence and written bug-free code?

Is the Code you Wrote Correct?

Whenever we program something, how can we be sure what it returns is correct? Can we always trust that every programmer everywhere has done their due diligence and written bug-free code?

Is the Code you Wrote Correct?

Whenever we program something, how can we be sure what it returns is correct? Can we always trust that every programmer everywhere has done their due diligence and written bug-free code?

Of course not!

Is the Code you Wrote Correct?

We de nitely need to be sure implementations are correct when it comes to critical infrastructure that people's lives, safety, and security depend on. This brings us to the case study of the Therac-25 in radiation oncology.

Is the Code you Wrote Correct?

We de nitely need to be sure implementations are correct when it comes to critical infrastructure that people's lives, safety, and security depend on. This brings us to the case study of the Therac-25 in radiation oncology.

The Therac-25

The Therac-25 is a computer-controlled radiation therapy for cancer patients produced by Atomic Energy of Canada Limited (AECL).

The Therac-25

The Therac-25 is a computer-controlled radiation therapy for cancer patients produced by Atomic Energy of Canada Limited (AECL).

There were 6 incidents from 1985 to 1987 in which patients were exposed to extremely high levels of radiation, and ended up dying as a result or were severely injured. A commission found several issues with the construction of the Therac-25, some of which we list on the next slide.

Problems with the Therac-25

1. AECL did not consider the design of the software during its assessment of how the machine might produce the desired results and what failure modes existed, focusing purely on hardware and asserting that the software was free of bugs.

Problems with the Therac-25

1. AECL did not consider the design of the software during its assessment of how the machine might produce the desired results and what failure modes existed, focusing purely on hardware and asserting that the software was free of bugs.
2. Several error messages merely displayed the word\MALFUNCTION" followed by a number from 1 to 64. The user manual did not explain or even address the error codes, nor give any indication that these errors could pose a threat to patient safety.

Problems with the Therac-25

1. AECL did not consider the design of the software during its assessment of how the machine might produce the desired results and what failure modes existed, focusing purely on hardware and asserting that the software was free of bugs.
2. Several error messages merely displayed the word\MALFUNCTION" followed by a number from 1 to 64. The user manual did not explain or even address the error codes, nor give any indication that these errors could pose a threat to patient safety.
3. The software set a ag variable by incrementing it, rather than by setting it to a xed non-zero value. Occasionally an arithmetic over ow occurred, causing the ag to return to zero and the software to bypass safety checks.

How to Avoid Bugs in the First Place

In 1987, the FDA placed a shutdown on all Therac-25 machines until permanent repairs could be made. Eventually, these repairs were made and the Therac-25 was recommissioned and still remains in use today.

How to Avoid Bugs in the First Place

In 1987, the FDA placed a shutdown on all Therac-25 machines until permanent repairs could be made. Eventually, these repairs were made and the Therac-25 was recommissioned and still remains in use today.

Glad these mistakes could be xed, but how could they have been avoided in the rst place?

How to Avoid Bugs in the First Place

In 1987, the FDA placed a shutdown on all Therac-25 machines until permanent repairs could be made. Eventually, these repairs were made and the Therac-25 was recommissioned and still remains in use today.

Glad these mistakes could be xed, but how could they have been avoided in the rst place?

One framework to avoid these bugs is through the use of formal methods.

How to Avoid Bugs in the First Place

In 1987, the FDA placed a shutdown on all Therac-25 machines until permanent repairs could be made. Eventually, these repairs were made and the Therac-25 was recommissioned and still remains in use today.

Glad these mistakes could be xed, but how could they have been avoided in the rst place?

One framework to avoid these bugs is through the use of formal methods.

What are Formal Methods?

Formal methods are tools and techniques based on mathematics and formal logic to solve software and hardware problems at all levels of the development process.

What are Formal Methods?

Formal methods are tools and techniques based on mathematics and formal logic to solve software and hardware problems at all levels of the development process.

What is a situation where the federal government might be strongly interested in using formal methods? AES is used to protect data as all levels of classi cation in the DoD, so an implementation mistake could be costly to national security, so we should have a way to ensure that there is a literal mathematical proof that the implementation is correct.

What are Formal Methods?

Formal methods are tools and techniques based on mathematics and formal logic to solve software and hardware problems at all levels of the development process.

What is a situation where the federal government might be strongly interested in using formal methods? AES is used to protect data as all levels of classi cation in the DoD, so an implementation mistake could be costly to national security, so we should have a way to ensure that there is a literal mathematical proof that the implementation is correct.

What are tools we can use for that?

Types of Tools

There are several types of tools within formal methods we can use.

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)
2. Proof-oriented programming langauges

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)
2. Proof-oriented programming langauges
3. SMT (Satis ability modulo theories) Solvers

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)
2. Proof-oriented programming langauges
3. SMT (Satis ability modulo theories) Solvers
4. Speci cation and veri cation software

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)
2. Proof-oriented programming langauges
3. SMT (Satis ability modulo theories) Solvers
4. Speci cation and veri cation software
5. Protocol analyzers

Types of Tools

There are several types of tools within formal methods we can use.

1. Interactive theorem provers (ITPs)
2. Proof-oriented programming langauges
3. SMT (Satis ability modulo theories) Solvers
4. Speci cation and veri cation software
5. Protocol analyzers

We discuss what these are and examples of them next.

Interactive Theorem Provers

An interactive theorem prover a is software that assists in translating mathematics into formal logic to verify correctness. Code that represents the mathematics is written in an interface, and at each step, the software veri es whether the step is valid within the logical framework.

Interactive Theorem Provers

An interactive theorem prover a is software that assists in translating mathematics into formal logic to verify correctness. Code that represents the mathematics is written in an interface, and at each step, the software veri es whether the step is valid within the logical framework.

Some examples of interactive theorem provers include Coq, Lean, Agda, and Isabelle. Some of these ITPs di er signi cantly in the logic they employ. For example, Isabelle employs higher-order logic, and Coq is built on the Calculus of Inductive Constructions, which is a higher-order typed lambda calculus.

Interactive Theorem Provers

An interactive theorem prover a is software that assists in translating mathematics into formal logic to verify correctness. Code that represents the mathematics is written in an interface, and at each step, the software veri es whether the step is valid within the logical framework.

Some examples of interactive theorem provers include Coq, Lean, Agda, and Isabelle. Some of these ITPs di er signi cantly in the logic they employ. For example, Isabelle employs higher-order logic, and Coq is built on the Calculus of Inductive Constructions, which is a higher-order typed lambda calculus.

Within Isabelle, the Law of Excluded Middle holds, but it does not within Coq{this creates a very di erent method for proofs.

Some Coq Code

A proof that is fairly easy (maybe not at rst glance though!) in Coq is that given two types (which we can think of as sets), d and d^0, is that there is a bijection between d d^0 and d^0 d.

Some Coq Code

A proof that is fairly easy (maybe not at rst glance though!) in Coq is that given two types (which we can think of as sets), d and d^0, is that there is a bijection between d d^0 and d^0 d.

Proof-oriented programming languages

We can think of a proof-oriented programming language as a middle ground between a general purpose programming language and an ITP.

Proof-oriented programming languages

We can think of a proof-oriented programming language as a middle ground between a general purpose programming language and an ITP.

The language F* is really the go-to example here. The F* type-checker aims to prove that programs meet their speci cations using a combination of SMT solving and manual proofs. This is similar to other tools we will see.

Proof-oriented programming languages

We can think of a proof-oriented programming language as a middle ground between a general purpose programming language and an ITP.

The language F* is really the go-to example here. The F* type-checker aims to prove that programs meet their speci cations using a combination of SMT solving and manual proofs. This is similar to other tools we will see.

Similar to Coq, F* is depdently-typed, and if one wants to de ne vectors of type a in F*, you would need the following code.

Proof-oriented programming languages

We can think of a proof-oriented programming language as a middle ground between a general purpose programming language and an ITP.

The language F* is really the go-to example here. The F* type-checker aims to prove that programs meet their speci cations using a combination of SMT solving and manual proofs. This is similar to other tools we will see.

Similar to Coq, F* is depdently-typed, and if one wants to de ne vectors of type a in F*, you would need the following code.

SMT Solvers

Satis ability Modulo Theories (SMT) is a the problem of determining whether or not a mathematical formula is satis able. The name comes from the fact that these formulas are interpreted within (\modulo") a certain formal theory in rst-order logic with equality. Then SMT solvers aim to solve SMT for a practical subset of inputs.

SMT Solvers

Satis ability Modulo Theories (SMT) is a the problem of determining whether or not a mathematical formula is satis able. The name comes from the fact that these formulas are interpreted within (\modulo") a certain formal theory in rst-order logic with equality. Then SMT solvers aim to solve SMT for a practical subset of inputs.

For example, if we are working in SMT for a subset of Z, and we want to know if the inequality $x + y + 3z \quad 4a$ holds, then the SMT solver will return a \YES" or \NO".

SMT Solvers

Satis ability Modulo Theories (SMT) is a the problem of determining whether or not a mathematical formula is satis able. The name comes from the fact that these formulas are interpreted within (\modulo") a certain formal theory in rst-order logic with equality. Then SMT solvers aim to solve SMT for a practical subset of inputs.

For example, if we are working in SMT for a subset of Z, and we want to know if the inequality $x + y + 3z \quad 4a$ holds, then the SMT solver will return a \YES" or \NO".

Examples of commonly used SMT solvers include Z3, CVC5, Yices, and Boolector. These are integrated into the next tool we will talk about.

Specification and Verification Software

Software that can be used to write clear mathematical specifications of algorithms that can be used to provide a proof that an implementation is correct is a common tool in formal methods.

Speci cation and Veri cation Software

Software that can be used to write clear mathematical speci cations of algorithms that can be used to provide a proof that an implementation is correct is a common tool in formal methods.

A very common pair of tools used in the cryptographic community is Cryptol and the Software Analysis Workbench (SAW). Cryptol and SAW really rely on SMT solvers to do the heavy lifting, but provide a clear space to work in: Cryptol is a domain-speci c language to write clear cryptographic speci cations and SAW can take these speci cations to produce proofs that an implementation is correct.

Multiplication in $F_{2^{12}}$ in C for mceliece38864

An example of code we might want to verify is correct is the C code for multiplication in $F_{2^{12}}$ (represented as the quotient ring $F_2[z]=(z^{12}+z^3+1)$) for mceliece38864 in the NIST reference implementation. This code is us everywhere in mceliece38864, and correctness of multiplication in $F_{2^{12}}$ is essential to correctness of the algorithm.

Multiplication in $F_{2^{12}}$ in C for mceliece38864

Protocol Analyzers

A protocol analyzer is software that provides a proof a cryptographic protocol is secure. Why does this matter? An algorithm could provide the best possible security, but the way it is used can be set up in such a way that makes the user very vulnerable.

Protocol Analyzers

A protocol analyzer is software that provides a proof a cryptographic protocol is secure. Why does this matter? An algorithm could provide the best possible security, but the way it is used can be set up in such a way that makes the user very vulnerable.

Examples of these tools include Tamarin and Verifpal. An example of a protocol these tools can verify is the following message that was exchanged using Di e-Hellman.

Protocol Analyzers

How are these Tools Used?

There are several ways in which these tools can be used in cryptography. Some of them include the following:

How are these Tools Used?

There are several ways in which these tools can be used in cryptography. Some of them include the following:

1. Providing a formal proof that an implementation of a cryptographic algorithm is correct

How are these Tools Used?

There are several ways in which these tools can be used in cryptography. Some of them include the following:

1. Providing a formal proof that an implementation of a cryptographic algorithm is correct
2. Providing a formal proof a cryptographic protocol is correct

How are these Tools Used?

There are several ways in which these tools can be used in cryptography. Some of them include the following:

1. Providing a formal proof that an implementation of a cryptographic algorithm is correct
2. Providing a formal proof a cryptographic protocol is correct
3. Translating mathematics into the language of an interactive theorem prover to provide a formal proof it is correct

How are these Tools Used?

There are several ways in which these tools can be used in cryptography. Some of them include the following:

1. Providing a formal proof that an implementation of a cryptographic algorithm is correct
2. Providing a formal proof a cryptographic protocol is correct
3. Translating mathematics into the language of an interactive theorem prover to provide a formal proof it is correct

We look at some examples of these.

Correctness of an implementation

We saw how multiplication in the eld $F_{2^{12}}$ was implemented in C for mceliece38864, and multiplication in this eld is used everywhere within this algorithm, so how sure are we it's correct? Cryptol and SAW!

Correctness of an implementation

We saw how multiplication in the eld $F_{2^{12}}$ was implemented in C for mceliece38864, and multiplication in this eld is used everywhere within this algorithm, so how sure are we it's correct? Cryptol and SAW!

There has been recent work by a team in the UK to verify eld arithmetic in mceliece38864. The process was as follows: First, translate the C code into Cryptol.

Correctness of an implementation

We saw how multiplication in the eld $F_{2^{12}}$ was implemented in C for mceliece38864, and multiplication in this eld is used everywhere within this algorithm, so how sure are we it's correct? Cryptol and SAW!

There has been recent work by a team in the UK to verify eld arithmetic in mceliece38864. The process was as follows: First, translate the C code into Cryptol.

Correctness of an Implementation

This new Cryptol code fairly similar to the C code, so perhaps not really a lot of assurance it's really doing what it should do? Well, we can also write high-level code for this multiplication operation in Cryptol:

Correctness of an Implementation

This new Cryptol code fairly similar to the C code, so perhaps not really a lot of assurance it's really doing what it should do? Well, we can also write high-level code for this multiplication operation in Cryptol:

Correctness of an Implementation

This new Cryptol code fairly similar to the C code, so perhaps not really a lot of assurance it's really doing what it should do? Well, we can also write high-level code for this multiplication operation in Cryptol:

And the really nice thing about Cryptol is that we can the built-in SMT solvers to give a mathematical proof (we are not just testing!) that these two two functions are equal!

Correctness of an Implementation

This new Cryptol code fairly similar to the C code, so perhaps not really a lot of assurance it's really doing what it should do? Well, we can also write high-level code for this multiplication operation in Cryptol:

And the really nice thing about Cryptol is that we can the built-in SMT solvers to give a mathematical proof (we are not just testing!) that these two two functions are equal!

How do we verify the implementation in C is correct? That's where we use the lower level Cryptol code! We use the C code, the low-level Cryptol code, and use SAW to produce a proof artifact that the implementation is correct.

Correctness of a Protocol

As we mentioned, Verifpal is a tool to help verify that a cryptographic protocol is correct. The protocol we saw a few slides ago can be modeled in Verifpal as follows:

Correctness of a Protocol

As we mentioned, Verifpal is a tool to help verify that a cryptographic protocol is correct. The protocol we saw a few slides ago can be modeled in Verifpal as follows:

Correctness of a Protocol

In the last line, we are asking Verifpal if everything goes as planned when we decrypt our encrypted message with the correct parameters. Verifpal can also tell us when protocols will be susceptibile to things like a man-in-the-middle attack.

Is your Math Correct?

Most of mathematics that we are accustomed to is what we call \informal". There are times when it's of strong interest to translate this \informal" math to the formal logic of an ITP.

Is your Math Correct?

Most of mathematics that we are accustomed to is what we call \informal". There are times when it's of strong interest to translate this \informal" math to the formal logic of an ITP.

A very important example of this is the CompCert project. CompCert is a formally veri ed optimizing compiler for a large subset of the C99 programming language which can currently be used for PowerPC, ARM, RISC-V, x86 and x86-64 architectures. The veri cation was done using Coq, and it was not done quickly.

Is your Math Correct?

Most of mathematics that we are accustomed to is what we call \informal". There are times when it's of strong interest to translate this \informal" math to the formal logic of an ITP.

A very important example of this is the CompCert project. CompCert is a formally veri ed optimizing compiler for a large subset of the C99 programming language which can currently be used for PowerPC, ARM, RISC-V, x86 and x86-64 architectures. The veri cation was done using Coq, and it was not done quickly.

There is also projects where people have translated the informally written mathematics of cryptographic algorithms, such as CRYSTALS-Kyber, into an ITP, such as Isabelle.

Are Formal Methods Used in Cryptographic Algorithm Development?

The short answer to this is no, at least publicly. There have been some attempts by the NTRU Prime and the Classic McEliece teams to either make their specifications and implementations more amenable to formal verification, but there is no large scale effort in cryptography to include formal methods, or even consider its use after development.

The End

Thank you!