Formal Verification for Cryptography and Protocols

Jonathan Protzenko, Project Everest, and many other awesome collaborators including, but not limited to: Son Ho, Théophile Wallez, Karthikeyan Bhargavan, Abhishek Bichhawat, Marina Polubelova, Aymeric Fromherz, Natalia Kulatova, Santiago Zanella, Benjamin Beurdouche, and many more
Brief background

- **What I work on:** verifying critical software
- **What this means:** showing with mathematical certainty that the code is “doing the right thing”
- **Where I work:** Microsoft Azure Research, USA (previously: INRIA, Paris)
- **Fun fact:** I studied at NUS for a semester in 2008-2009 (had a fantastic time)
This lesson, in a nutshell

Cryptography & protocols

Protocols:
- TLS (as in: HTTPS)
- Signal (as in: Facebook Messenger)
- Noise (as in: Wireguard VPN)

Cryptography:
- Kyber (post-quantum key-exchange)
- SHA3 (cryptographically secure hash)
- Ed25519 (elliptic curve digital signatures)

Properties:
- functional correctness (it’s doing the right thing!)
- protocol security (authenticity, confidentiality, forward secrecy...)

Tools:
- F*, a dependently typed proof assistant
- Aeneas, translating Rust into a pure functional equivalent
My research: formally verified cryptography and protocols, in real-world software

 NSS verified crypto library
 Piecewise approach, totaling 50% of their LoC

 via Wireguard VPN
 Currently only Curve25519

 via Ocaml bindings
 taking all of EverCrypt (esp. P256)

 Curve25519 and others
 Piecewise approach

 MSQUIC handshake + crypto

 CCF (aka Entreprise Blockchain)
 EverCrypt hashes + MerkleTree

 Kaizala (enterprise messaging)
 QUIC + whole stack

 hashlib – MD5, SHA1, SHA2,
 SHA3 and soon Blake2
Goals of this lesson

1. **Background on cryptographic algorithms**
   - A brief, incomplete *taxonomy* of algorithms
   - Understanding their expected *properties*
   - The case for crypto algorithms *verification*

2. **From algorithms to protocols**
   - What *is* a cryptographic protocol
   - Expected *properties* of such protocols
   - How does one go about *proving* those?

3. **Formal verification, in a nutshell**
   - Brief background on F*
   - How to leverage SMT-based verification for crypto & protocols

4. **Connection to real-world implementations**
   - Legacy: F* → C
   - Future: Rust → F*, Lean and more!
Many other projects in this space

- Fiat-Crypto, Jasmine, Cryptoline...
- Proverif, CryptoVerif, Tamarin, Squirrel...
+ many other projects I’ll mention throughout
Background on cryptographic algorithms

Taxonomy, properties and verification
At a very high level

• Cryptography is a set of techniques aiming to achieve information security

• Information security, broadly, boils down to specific goals, e.g.:
  • confidentiality: no one else can decrypt the data
  • authenticity: the data is really coming from the right person
  • integrity: the data wasn’t tampered with
  • ...

• Example: file encryption, TLS, instant messaging...
Examples of primitives

• We start with **primitives**, before moving on to protocols
• **Examples** of primitives: SHA256, Poly1305, HKDF, HPKE, AES, GCM...
  • Essentially, functions with *inputs* and *outputs*
  • Contrast with protocols, which have a notion of time, participants, state machines, etc.

```
jonathan@absinthe:~/Seafile/My Library/Papers $ sha256sum 20240112\ NUS.pptx
55eec82ec1b84d1f5db85d35a7a8cf56d78fa2b39fb2596faf524bd694b64f3e   20240112 NUS.pptx
```
How primitives are defined

Here is a quantitative form of the guarantee. Assume that the attacker sees at most $C$ authenticated messages and attempts at most $D$ forgeries. Assume that the attacker has probability at most $\delta$ of distinguishing $AES_k$ from a uniform random permutation after $C + D$ queries. Assume that all messages have length at most $L$. Then, with probability at least

$$1 - \delta - \frac{(1 - C/2^{128})^{(C+1)/2}D[L/16]}{2^{106}},$$

all of the attacker’s forgeries are discarded. In particular, if $C \leq 2^{64}$, then

New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba

Jean-Philippe Aumasson$^1$, Simon Fischer$^1$, Shahram Khazaei$^2$, Willi Meier$^3$, and Christian Rechberger$^3$

$^1$ FHNW, Windisch, Switzerland  
$^2$ EPFL, Lausanne, Switzerland  
$^3$ IAIK, Graz, Austria

Abstract. The stream cipher Salsa20 was introduced by Bernstein in 2005 as a candidate in the eSTREAM project, accompanied by the reduced versions Salsa20/8 and Salsa20/12. ChaCha is a variant of Salsa20 aiming at bringing better diffusion for similar performance. Variants of Salsa20 with up to 7 rounds (instead of 20) have been broken by differential cryptanalysis, while ChaCha has not been analyzed yet. We introduce a novel method for differential cryptanalysis of Salsa20 and ChaCha, inspired by correlation attacks and related to the notion of neutral bits. This is the first application of neutral bits in stream cipher cryptanalysis. It allows us to break the 256-bit version of Salsa20/8, to bring faster attacks on the 7-round variant, and to break 6- and 7-round ChaCha. In a second part, we analyze the compression function Rumba, built
How to think of primitives, practically

• Finding good / better primitives = job of the cryptographer
  • differential cryptanalysis (DES)
  • faster algorithms (ECDH vs RSA)
  • more computationally difficult algorithms (post-quantum algorithms)
  • fewer implementation pitfalls (e.g. Curve25519)

• As a user, one can think it terms of functionality
  • Key derivation: generate from key from input key material (HKDF)
  • Key agreement: compute shared secret (ECDH)
  • Hash: capture integrity with a digest (SHA256)
  • Stream cipher: pseudo-random bytes (AES)
  • Message-Authentication Code (Poly1305)
How to think of primitives, practically (2)

- **Implementors** do not start from the math specification
- Algorithms are distilled as **RFCs**, NIST standards, and other reference documents
- Curve25519 example

To implement the X25519(k, u) function... first decode k and u and then perform the following procedure... which is based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p. The constant a24 is \((486662 - 2) / 4 = 121665\).

\[
\begin{align*}
    x_1 &= u \\
    x_2 &= 1 \\
    z_2 &= 0 \\
    x_3 &= u \\
    z_3 &= 1 \\
    swap &= 0
\end{align*}
\]

For \(t = \text{bits}-1\) down to 0:
\[
\begin{align*}
    k_t &= (k >> t) & 1 \\
    \text{swap} &= \text{swap} \oplus k_t \\
    &\text{// Conditional swap; see text below.}
\end{align*}
\]
\[
\begin{align*}
    (x_2, x_3) &= \text{cswap}(\text{swap, } x_2, x_3) \\
    (z_2, z_3) &= \text{cswap}(\text{swap, } z_2, z_3) \\
    \text{swap} &= k_t
\end{align*}
\]
\[
\begin{align*}
    A &= x_2 + z_2 \\
    AA &= A^2 \\
    B &= x_2 - z_2 \\
    BB &= B^2 \\
    E &= AA - BB \\
    C &= x_3 + z_3 \\
    D &= x_3 - z_3 \\
    DA &= D \times A \\
    CB &= C \times B \\
    x_3 &= (DA + CB)^2 \\
    z_3 &= x_1 \times (DA - CB)^2 \\
    x_2 &= AA \times BB \\
    z_2 &= E \times (AA + a24 \times E) \\
    &\text{// Conditional swap; see text below.}
\end{align*}
\]
\[
\begin{align*}
    (x_2, x_3) &= \text{cswap}(\text{swap, } x_2, x_3) \\
    (z_2, z_3) &= \text{cswap}(\text{swap, } z_2, z_3) \\
    \text{Return } x_2 \times (z_2^{(p - 2)})
\end{align*}
\]

(Note that these formulas are slightly different from Montgomery’s original paper. Implementations are free to use any correct formulas.)
How to think of primitives, practically (3)

• The task then becomes: how to optimize this reference implementation

• There are plenty of implementation and mathematical tricks that can be played to speed things up

• Example: Curve25519
/**
Compute the scalar multiple of a point.

@param out Pointer to 32 bytes of memory, allocated by the caller, where the resulting point is written to.
@param priv Pointer to 32 bytes of memory where the secret/private key is read from.
@param pub Pointer to 32 bytes of memory where the public point is read from. */
void Hacl_Curve25519_51_scalarmult(uint8_t *out, uint8_t *priv, uint8_t *pub);

/**
Calculate a public point from a secret/private key.

This computes a scalar multiplication of the secret/private key with the curve's basepoint.

@param pub Pointer to 32 bytes of memory, allocated by the caller, where the resulting point is written to.
@param priv Pointer to 32 bytes of memory where the secret/private key is read from. */
void Hacl_Curve25519_51_secret_to_public(uint8_t *pub, uint8_t *priv);

/**
Execute the diffie-hellmann key exchange.

@param out Pointer to 32 bytes of memory, allocated by the caller, where the resulting point is written to.
@param priv Pointer to 32 bytes of memory where our secret/private key is read from.
@param pub Pointer to 32 bytes of memory where their public point is read from. */
bool Hacl_Curve25519_51_ecdh(uint8_t *out, uint8_t *priv, uint8_t *pub);
Key exchange example
(Curve25519 = ECDH, for Elliptic Curve Diffie-Hellman)

- Participant A, generate secret key = x, compute public key = $g^x$
- Participant B, generate secret key = y, compute public key = $g^y$
- A and B exchange their public keys (using a protocol? see part 2)
- A computes $(g^x)^y = g^{xy}$, B computes $(g^y)^x = g^{xy}$
- Shared secret $g^{xy}$ has been established
- Relies on the fact that finding $g^{xy}$ from $g^x$ and $g^y$ is difficult
How things *should be*: correct primitives

- Proceed step by step using *proofs by refinement*
- Mechanized or pen-and-paper
- The lower-level version exhibits the same (a subset of the) behavior of the high-level version

```
Optimized implementation (crazy tricks!)  refines  RFC pseudo-code  refines  Math version

math / crypto robustness (out of scope)
```
How things *actually* are

Correct bounds in 32-bit code.
The 32-bit code was illustrative of the tricks used in the original curve25519 paper rather than rigorous. However, it has proven quite popular.

This change fixes an issue that Robert Ransom found where outputs between $2^{255} - 1$ and $2^{255} - 2$ weren't correctly reduced in `fcontract`. This appears to leak a small fraction of a bit of security of private keys.

Additionally, the code has been cleaned up to reflect the real-world needs. The ref10 code also exists for 32-bit, generic C but is somewhat slower and objections around the lack of qasm availability have been raised.

---

**Curve25519-donna**
Why are primitives so broken?

• Many, many **implementations** of the same **algorithm**
  • Different architectures (ARM, x64...)
  • Different microarchitectures (Haswell, Sandy Bridge, ...)
  • Different instruction support (SHA-NI, AES-NI, ...)
  • Different implementation strategies (precomputations, limb representations...)
• Written in extremely **low-level languages** (C, ASM) for perf
• Inherent **complexity** + impossibility to test the **search space**
• New implementations **keep coming**
Why are primitives so broken (2)?

• Added twist: **side-channels**
• Discovered late 1990s (Kocher et. al.) in the context of smart cards
  • Power consumption reveals what’s happening in DES (very fun paper)
• Root of the problem: a discrepancy between the **model** of the machine and the **reality** of modern systems
  • Model: Von Neumann architecture (processor + memory in a vacuum)
  • Reality: speculation, out-of-order execution, fast paths, microcoded instructions, electromagnetic emissions, timing differences, power consumption
• Gives you e.g. **Spectre & Meltdown**
• Cryptanalysts find side-channels: **a correct implementation avoids them** (or at least the known ones)
It’s about to get even worse!

• Ongoing revolution in cryptography: many existing algorithms are simply broken in a **post-quantum world**

• Very active cryptology research area: designing and proving the robustness of quantum-safe algorithms
  • standardization ongoing at NIST following competition

• New cryptographic tools: lattices, isogenies, learning with errors...

• New challenges
  • harder to prove that the math is correct (e.g. Kyber)
  • harder to prove that the implementation is correct
What can you do about it?

- Choose primitives wisely (easier said than done)
- Formally verify your algorithms (part 3 of this lecture)
- Write implementations in Rust (part 4 of this lecture)

None of this should impact performance, user experience, or ease of integration.

A true research challenge!
Background on cryptographic protocols
Design, security, attacker model & proof framework
Quick recap

• Primitives are **individual building-blocks of cryptography**
• They provide a given **functionality**
  • **Diffie-Hellman**: computing key agreement (just saw that)
  • **Encryption**: given a key, encrypt or decrypt a stream of data
  • **Key derivation**: given input key material (label + key), generate new pseudo random keys
    • And more...
• But primitives can also be **combined** to form protocols... such as lKpsk2 (Wireguard)
Protocol Example: IKpsk2 (a Noise protocol)

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations...)
- Send/receive encrypted data

Secrets are chained:
- $d_0$ encrypted with a key derived from $es, ss$
- $d_1$ encrypted with a key derived from $es, ss, ee, se, psk$

⇒ The more the handshake progresses, the more secure the shared secrets are
What happens in IKpsk2

Initiator

Responder

New chaining key
Symmetric encrypt/decrypt key

ck1, sk1: DH(E\_i, S\_r)
ck2, sk2: DH(E\_i, S\_r), DH(S\_i, S\_r)

dh0 (initiator) == dh0 (responder)
• we exchange keys
• we derive same values for ck, sk at every step

Uses authentication data: may fail (if message is corrupted or if not same symmetric key)

responder sends S\_r\_pub before handshake (uses Public Key Infrastructure...)

Uses authentication data: may fail (if message is corrupted or if not same symmetric key)
A protocol does a whole lot of stuff

IKpsk2:

\[ s \]

\[ e, \ es, \ ss, [d0] \]

\[ e, \ ee, \ se, \ psk, [d1] \]

\[ [d2, d3, ...] \]

- Initiator and responder must remember which key belongs to whom
  
- Responder receives a static key during the handshake
  
  - Peer lookup (if key already registered)
  
  - Unknown key validation
  
- Long-term key storage

- Transitions are low-level
  
  - State Machine
  
  - Message lengths
  
  - Invalid states (if failure)

- Early data
  
  - when is it safe to send secret data?
  
  - when can we trust the data we received?
Compared to primitives, a protocol...

- Is a **long-lived** piece of code – messages exchanged over time
- Requires a lot **more components** to implement: state machine, API, parsing of formats, data structures
- Demands that **additional properties** be proven
- Thus offers more **opportunities to get things wrong**
And things **do** get wrong

- There’s an entire RFC of TLS attacks
- Recently in the news: SSH (“Terrapin”)
- Also well known: WPA (WiFi)
- Bluetooth, LTE, and many more

We need a reasoning framework for protocols
Properties we want from a protocol

- **Authenticity**: message sent by A and successfully decrypted by B must come from A
  - unless keys (which?) have been compromised by *the attacker*
- **Confidentiality**: contents of the message cannot be obtained by the attacker
  - unless keys (which?) have been compromised by *the attacker*
  - bonus: past messages are still safe ("forward secrecy")
  - bonus: future messages will eventually be safe ("post-compromise recovery")

Who exactly is the attacker?
In order to prove properties, you need an attacker model

**Q1: How does the attacker interact with the network?**
- Can only read messages (passive attacker)
- Can intercept and tamper messages (active attacker)

**Q2: What can the attacker do with cryptography?**
- Use it in abstract ways: call primitives, recombine data
- Perform cryptanalysis: break cryptography with a probability
Computational model (briefly)

• Designed in the 1980s (Goldwasser, Micali: Probabilistic encryption.)
• Attacker can, with some probability, **break primitives**
• Attacker operates in **polynomial time** (for now)
• The usual way to think about this: **game semantics**

\[
\text{Game } \text{Ae}(\mathcal{A}, \text{AE}) \\
\quad b \leftarrow \{0, 1\}; \ L \leftarrow \emptyset; \ k \leftarrow \text{AE.keygen}() \;
\quad b' \leftarrow \mathcal{A}^{\text{Encrypt,Decrypt}}(); \ \text{return } (b \overset{?}{=} b')
\]

<table>
<thead>
<tr>
<th>Oracle Encrypt(p)</th>
<th>Oracle Decrypt(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( b ) then ( c \leftarrow \text{byte}^\ell_c; \ L[c] \leftarrow p ) else ( c \leftarrow \text{AE.encrypt} \ k \ p ) return ( c )</td>
<td>if ( b ) then ( p \leftarrow L[c] ) else ( p \leftarrow \text{AE.decrypt} \ k \ c ) return ( p )</td>
</tr>
</tbody>
</table>
Computational model (briefly)

- A rich area of research with multiple tools: EasyCrypt, CryptoVerif, Squirrel
- **Research challenge**: starting with games over primitives, lift those to a complete protocol
  - see e.g. “Proving the TLS 1.3 record layer”
- Techniques developed for **composability**
  - SSPs, state-separating proofs
  - UC, universal composability

---

Modular Code-Based Cryptographic Verification, *Fournet, Kohlweiss, Strub*

State Separation for Code-Based Game-Playing Proofs, *Brzuska, Delignat-Lavaud, Fournet, Kohbrok, Kohlweiss*
Computational model (briefly)

• Allows finding computational attacks, such as
  • if you forget to rekey, or to derive sufficiently fresh keys in your protocol, the adversary might start to be able to break your crypto with non-negligible protocol (i.e., establishing bounds)

• Precise, but requires deeper cryptographic expertise to use
  • Games are delicate to establish (IND-CPA, IND-CCA, IND-UF-1CMA, ...)
  • Lends itself poorly to automation (active research area, e.g. Squirrel)
In order to prove properties, you need an attacker model

Q1: How does the attacker interact with the network?
- Can only read messages (passive attacker)
- Can intercept and tamper messages (active attacker)

Q2: What can the attacker do with cryptography?
- Use it in abstract ways: call primitives, recombine data
- Perform cryptanalysis: break cryptography with a probability
The Dolev-Yao model

• Invented in 1983, but still very relevant today
• Also known as the “symbolic model”
• Abstract usage of crypto primitives
• Representation: symbolic terms, e.g. an AST with rewriting rules

Example: \texttt{dec(k, enc(k, msg)) == msg}

On the security of public key protocols, Dolev, Yao
The Dolev-Yao model (2)

- We model the network as a log of events (messages)
- Attacker is an **active network attacker**, meaning it can:
  - **learn** bytes that are sent over the network
  - **concatenate** bytes
  - **call cryptographic** functions to create new bytes
  - **interact** with participants by sending them messages

Compared with the computational model, the symbolic models allows finding **logical flaws in protocols**, e.g. “I forgot to authenticate this piece of data”.
The Dolev-Yao model (3)

- Popular model because it lends itself well to **automation**
- Systems of **rewriting rules**, allows bounded exploration of protocol sessions, but also general-purpose proof assistants along with custom tactics and proof frameworks
- **Software**: ProVerif, Tamarin, DY*
- Used to analyze TLS, Signal, Bluetooth, MLS, and many more

What we shall be using in Part 3!
Supplemental reading

• For an excellent survey of computation vs symbolic model, their history, and the link between the two, see:

Verifying primitives & protocols

Formal verification (i.e., machine-checked) of specifications: correctness and security.
A quick recap

• **Primitives** (basic building blocks) = hard to get right
• **Protocols** (built of out primitives) = also hard to get right

Each of these two has **properties of interest**: functional correctness, side-channel resistance, confidentiality, authenticity

This section: **formally verifying** those properties
Formal verification, in a nutshell

• Prove theorems about your code, using a proof assistant
• Establishes the theorem all the time, for any of the inputs (≠ test, ≠ model-checking)
• A variety of proof assistants: Coq, Lean, Isabelle, F*, HOL, each with different strengths and backing theories
• Large pieces of code have been proven correct: compilers (CompCert), OSes (SEL4), cryptographic libraries (HACL*, Fiat-Crypto), protocols (TLS)

The new frontier: verifying efficient, secure, real-world code!
A Coq proof of the correctness of X25519 in TweetNaCl

Peter Schwabe  
MPI-SP, Germany &  
Radboud University, The Netherlands

Benoît Viguier  
Radboud University, The Netherlands

Timmy Weerag  
Radboud University & Open University of the Netherlands

Freek Wiedijk  
Radboud University, The Netherlands

Abstract—We formally prove that the C implementation of the X25519 key-exchange protocol in the TweetNaCl library is correct. We prove both that it correctly implements the protocol from Bernstein’s 2006 paper, as standardized in RFC 7748, as well as the absence of undefined behavior like arithmetic overflows and array out-of-bounds errors. We also formally prove, based on the work of Bartzia and Strub, that X25519 is mathematically correct, i.e., that it correctly computes scalar multiplication on the elliptic curve Curve25519.

The proofs are all computer-verified using the Coq theorem prover. To establish the link between C and Coq we use the Verified Software Toolchain (VST).

Implementation refines RFC pseudo-code refines Math version

math / crypto robustness (out of scope)
Our strategy: proof by refinement

• Use the F* proof assistant
• Proceed step-wise, a.k.a. refinements
• Generalize the methodology: implementation, spec, properties
• Apply to primitives & protocols
F*: the workhorse of this research

A programming language
A proof assistant
A program verification tool

Dijkstra Monads for Free. Ahman, Hrițcu, Martínez, Plotkin, Protzenko, Rastogi, Swamy

Dependent Types and Multi-monadic Effects in F*
Swamy, Hrițcu, Keller, Rastogi, Delignat-Lavaud, Forest, Bhargavan, Fournet, Strub, Kohlweiss, Zinzindohoue, Zanella-Béguelin
## Two camps of program verification tools

<table>
<thead>
<tr>
<th>Interactive proof assistants</th>
<th>Semi-automated verifiers of imperative programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq, Isabelle, Agda, Lean, PVS,...</td>
<td>Dafny, FramaC, Why3, Verve, IronClad, miTLS, Vale</td>
</tr>
<tr>
<td>CompCert, seL4, Bedrock, 4 colors</td>
<td>(\textit{air}) gap (\textit{gap})</td>
</tr>
</tbody>
</table>

- **In the left corner**: Very expressive dependently-typed logics, but only purely functional programming
- **In the right**: effectful programming, SMT-based automation, but only first-order logic
F*: Bridging the gap

- Functional programming language with effects
  - Like OCaml, Haskell, F#, ...
  - Compiles to OCaml or F#
    - A subset of F* compiled to C (with manual control over memory management)

- With an expressive core dependent type theory
  - Like Coq, Agda, Lean, ...

- Semi-automated verification using SMT
  - Like Dafny, Vcc, Liquid Haskell, ...

- In-language extensibility and proof automation using metaprograms
A first taste

• Write ML-like code

```plaintext
let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1)
```

• Give it a specification, claiming that `factorial` is a total function from non-negative to positive integers.

```plaintext
val factorial : n:int{n >= 0} -> Tot (i:int{i >= 1})
```

• Ask F* to check it

```
fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully
```
F* builds a typing derivation of the form:

\[ \Gamma_{\text{prelude}} \vdash \text{let factorial n = e : t} \leftarrow \phi \]

- In a context \( \Gamma_{\text{prelude}} \) including definitions of F* primitives
- The program \text{let factorial n = e} has type \( t \), given the validity of a logical formula \( \phi \)
- \( \phi \) is passed to Z3 (an automated theorem prover/SMT solver) to check for validity
- If the check succeeds, then, from the metatheory of F*, the program is safe at type \( t \)
The functional core of F*

• Recursive functions

```ocaml
code block
```

• Inductive datatypes (immutable) and pattern matching

```ocaml
let rec factorial n = (if n = 0 then 1 else n * (factorial (n - 1)))
```

```ocaml

```ocaml
val factorial : int -> int
```

```ocaml
let rec factorial n = (if n = 0 then 1 else n * (factorial (n - 1)))
```

```ocaml
type list (a:Type) =
    | Nil : list a
    | Cons : hd:a -> tl:list a -> list a
```

```ocaml
val map : ('a -> 'b) -> list 'a -> list 'b
```

```ocaml
let rec map f x = match x with
    | [] -> []
    | h :: t -> f h :: map f t
```

• Lambdas (unnamed, first-class functions)

```ocaml
map (fun x -> x + 42) [1;2;3]
```

```ocaml
map (fun x -> x + 42) [1;2;3]
```
Refinement types

```ocaml
type nat = x:int{x>=0}

val factorial : nat -> nat
let rec factorial n = (if n = 0 then 1 else n * (factorial (n - 1)))
```

- **Refinements introduced by type annotations (code unchanged)**

```ocaml
n >= 0, n <> 0 |= n - 1 >= 0
n >= 0, n <> 0, factorial (n - 1) >= 0 |= n * (factorial (n - 1)) >= 0
```

- **Logical obligations discharged by SMT (simplified)**

- **Refinements eliminated by subtyping: nat::<int**

```ocaml
let i : int = factorial 42
let f : x:nat{x>0} -> int = factorial
```
DEMO !!
Verifying a primitive: Poly1305

• Proposed in 2005, by Daniel J Berstein

• **Message authentication code** (MAC)
  • given a key and a message, compute an authentication code

• Mathematical basis (more on that shortly) defined in paper

• Implementation defined in RFC7539

---

Poly1305: the math

- Input: key (32 bytes) = s || r
- Input: message = m0 || ... || mn (mn padded with zeroes)
- mi, s, r = 16 bytes a.k.a. 128-bit integers

The main computation in the Poly1305 MAC evaluates the following polynomial in the prime field $\mathbb{Z}_p$, where $p = 2^{130} - 5$:

$$a = (m_1 \cdot r^n + m_2 \cdot r^{n-1} + \ldots + m_n \cdot r) \mod p.$$
In practice, this polynomial is evaluated block by block, by applying Horner’s method to rearrange the polynomial as follows:

\[ a = (((0 + m_1) \cdot r + m_2) \cdot r + \ldots + m_n) \cdot r \mod p. \]
Poly1305: implementation specification

• Problem: this needs to be FAST
• Core operation: modulo-reduction (% p)
• Chosen carefully: $2^{130} - 5$ makes for efficient implementations
• Common representation in crypto: limbs

Field arithmetic with a radix-$2^{26}$ representation. We encode an element of the Poly1305 field as a sequence of five 64-bit unsigned integers $[e_0; e_1; e_2; e_3; e_4]$, where each $e_i$ is below $2^{26}$, with the following evaluation function:

$$e_4 \cdot 2^{104} + e_3 \cdot 2^{78} + e_2 \cdot 2^{52} + e_1 \cdot 2^{26} + e_0.$$
Poly1305: implementation specification

\[ (2^{130} - 5) \; \% \; p = 0, \text{ hence, } 2^{130} \; \% \; p = 5 \]

\[ (bn_v b) \; \% \; p = (((b_8 \cdot 2^{78} + b_7 \cdot 2^{52} + b_6 \cdot 2^{26} + b_5) \cdot 2^{130}) + (b_4 \cdot 2^{104} + b_3 \cdot 2^{78} + b_2 \cdot 2^{52} + b_1 \cdot 2^{26} + b_0)) \; \% \; p. \]
Poly1305: implementation specification

Finally, the carry propagation is performed for the result of modular multiplication. It returns an integer $e = r \cdot a \mod p = [e_0; e_1; e_2; e_3; e_4]$ less than the prime $p$, where each coefficient $e_i$ is less than $2^{26}$.

→ back to normal form (“the invariant”)
Poly1305: implementation specification

• This is where it gets hard!
• Final carry propagation step is expensive
• Developers are tempted to skip some of them ("it fits")


Poly1305: proof infrastructure

Actual implementation (part 4) \(\xrightarrow{\text{refines}}\) Implementation specification (this) \(\xrightarrow{\text{implements}}\) Mathematical construction (earlier, slide 53)

"implementation sketch" (machine integers, not really executable)

\[
\text{let } \text{felem5} = (\text{uint64} & \text{uint64} & \text{uint64} & \text{uint64} & \text{uint64})
\]

"mathematical truth" (math integers)

\[
\text{let } \text{prime} = \text{pow2} 130 - 5 \\
\text{let } \text{felem} = x : \text{nat}\{x < \text{prime}\}
\]

\[
\text{let } \text{as_n\_nat5} (e : \text{felem5}) : \text{nat} = \text{let } (e0, e1, e2, e3, e4) = e \text{ in} \\
\quad v \ e0 + v \ e1 * \text{pow2} 26 + v \ e2 * \text{pow2} 52 + v \ e3 * \text{pow2} 78 + v \ e4 * \text{pow2} 104 \\
\text{let } \text{feval5} (e : \text{felem5}) : \text{felem} = \text{as\_nat5} e \mod \text{prime}
\]
Poly1305: proof infrastructure

1. Full-fledged implementation-specification
2. Introduce invariants + helper lemmas
3. Show that each of the key functions matches the math spec
4. Assemble top-level API

Step 1: full description of the implementation (here, multiplication)
Poly1305: proof infrastructure

1. Full-fledged implementation-specification
2. Introduce invariants + helper lemmas
3. Show that each of the key functions matches the math spec
4. Assemble top-level API

Step 2, invariant: \texttt{felem\_fits5}, which measures “how far” above $2^{26}$ \texttt{ei} is

\[
e_0 \leq f_0 \cdot M \land e_1 \leq f_1 \cdot M \land e_2 \leq f_2 \cdot M \land e_3 \leq f_3 \cdot M \land e_4 \leq f_4 \cdot M
\]

We restrict \( f_i \) to be less than or equal to \( 2^{32-26} = 2^6 \).

Step 2, helper lemma: addition relates to \texttt{felem\_fits}
Poly1305: proof infrastructure

1. Full-fledged implementation-specification
2. Introduce invariants + helper lemmas
3. Show that each of the key functions matches the math spec
4. Assemble top-level API

```
val fadd5_eval_lemma:
  # w: lanes
  -> f1:felem5 w{felem_fits5 f1 (2,2,2,2,2)}
  -> f2:felem5 w{felem_fits5 f2 (1,1,1,1,1)} ->
  Lemma (feval5 (fadd5 f1 f2) == map2 Vec pfadd (feval5 f1) (feval5 f2))
[SMTPat (fadd5 f1 f2)]
```

Step 3: eval (add f1 f2) == add (eval f1) (eval f2)
Poly1305: final remarks on the implementation strategy

• It’s even more complicated than shown, because
  • vectorized implementations: AVX/NEON, AVX2, AVX512 process multiple field elements
  • implementation optimizations require big math proofs
• The specification (slide 53) is not the final API
  • processes everything in one go or block by block
  • instead, you want a streaming API with internal buffering
  • long-lived persistent heap-allocated state = different kinds of proofs
• You want to be smart about these things
  • generic vectorization patterns for other algorithms
  • generic streaming patterns for all block algorithms

Hard to get right – famous SHA3 bug and others
Integrated into Python
Next: proofs of protocol security

- Now focusing on the next level up the stack: **protocols**
- How does one go about formally verifying **protocol security**
- **Capturing with formal language** notions such as: confidentiality, authenticity, forward secrecy, post-compromise recovery
- Our example: **IKpsk2** (as before)
A recap on Dolev-Yao

• A symbolic model, i.e. primitives are “perfect”
  • As opposed to the computational model, which bounds the probability that an attacker can break a cryptographic construction

• Trace-based model
  • Record global events (for all sessions), e.g. key generation, sending a message, random number generation, etc.

• Attacker model
  • Active attacker model, can intercept, re-order, send, learn from messages
  • Cannot guess randoms, cannot invert encryption
Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.

- Connects to all of the proofs introduced before ("one big qed")
- Less automation than e.g. Tamarin or ProVerif, but
  - can deal with unbounded protocol trace executions
  - can be part of a meta-programmed framework ("protocol implementations for free")

Ho, Protzenko, Bichhawat, Bhargavan.
Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations

Bhargavan, Bichhawat, Do, Hosseyni, Küsters, Schmitz, Würtele
Security Analysis - Dolev-Yao*

Define labels for the data-types:
• CanRead [P "Alice"] : static data that can only be read by principal "Alice"
• CanRead [S "Bob" sid]: ephemeral data that can only be read by principal "Bob" at session sid

Annotate the data types to give them usages and labels:
• dh_private_key l : private key of label l
• dh_public_key l : public key associated to a private key of label l

// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1) (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // l1 ⊔ l2

For now: those annotations are purely syntactic
Security Analysis – Target Labels

\[ \text{IKpsk2 (from the responder’s point of view)} \]
\[ \leftarrow s \]
\[ \cdots \]
\[ \rightarrow e, es, s, ss, [d] \]
\[ \text{Responder learns information about the initiator’s ephemeral by linking it to the initiator’s static key} \]
\[ \text{(peer_eph_label} \cup \text{CanRead [P p])} \cap (\text{CanRead [P peer} \cup \text{CanRead [P p])} \]
\[ \leftarrow e, ee, se, psk, [d] \]
\[ \text{(peer_eph_label} \cup \text{CanRead [P p])} \cap (\text{CanRead [P peer} \cup \text{CanRead [P p])} \]
\[ \leftarrow \text{[d]} \]
\[ \rightarrow [d] \]
\[ \leftarrow [d] \]

\[ \text{Authentication invariant gives us information upon receiving messages} \]

Upon receiving a message: we get information from the \textbf{current label} and previously used keys. The authentication invariant is verified whenever encrypting data (and retrieved when decrypting).
let ck0 = hash "Noise_IKpsk2..." in
  // e
  ...
  // es
  let dh_es = dh e rs in
  let ck1, sk1 = kdf2 ck0 dh_es in
  // s
  ...
  // ss
  let dh_ss = dh s rs in
  let ck2, sk2 = kdf2 ck1 dh_ss in
  // d (plain text)
  let cipher =
    aead_encrypt sk2 ... plain
  in
  ...
  // Output
  concat ... cipher

We can then send the encrypted message: register a Send event in a global trace
Security Analysis: can_flow

- Labels are purely syntactic
- **Semantics** of DY* are given through a can_flow predicate which states properties about a global trace of events
- The content of a message sent over the network is **compromised** if its label flows to public
- Labels can flow to more secret labels (i is a timestamp):
  
  ```
  can_flow i (CanRead [P p1]) (CanRead [P p1] ∩ CanRead [P p2])
  ```

- The attacker can **dynamically compromise** a participant’s current state: event `Compromise p` ...
- A label is compromised (and data with this label) if it flows to public:
  
  ```
  compromised_before i (P p) ==> can_flow i (CanRead [P p]) public
  compromised_before i (S p sid) ==> can_flow i (CanRead [S p sid]) public
  ```

- If a label flows to public we can deduce the existence of compromise events:
  
  ```
  can_flow i (CanRead [P p]) public ==> compromised_before i (P p)
  ```
Security Analysis – Security Predicates

Confidentiality level 5 (strong forward secrecy):

```
\text{can\_flow\ i\ (CanRead\ [S\ p\ sid] \sqcup\ CanRead\ [P\ peer])\ l\ /\}
\text{can\_flow\ i\ (CanRead\ [S\ p\ sid] \sqcup\ get\_dh\_label\ re)\ l\ /\}
(\text{compromised\_before\ i\ (S\ p\ sid)\ /\ compromised\_before\ i\ (P\ peer)\ /\}
(\exists\ sid'.\ get\_dh\_label\ re\ ==\ CanRead\ [S\ peer\ sid']))
```

We initially have no information about re: we link it to the remote static key (which has been certified)

**Authentication invariant** (an aead encrypt/decrypt):

```
\text{can\_flow\ i\ l\ public\ /\}
\text{begin\ match\ opt\_rs,\ opt\_re\ with}
|\ \text{Some\ rs,\ Some\ re}\ ->
  \exists\ peer'\ sid'.\ get\_dh\_label\ rs\ =\ CanRead\ [P\ peer']\ /\get\_dh\_label\ re\ =\ CanRead\ [S\ peer'\ sid']
|\ _\ ->\ True
\text{end}
```

**Certification** of remote static key gives:

```
get\_dh\_label\ rs\ =\ CanRead\ [P\ peer]
```
Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*. We do the security analysis once and for all.

1. We add annotations to types to reflect security properties:

```ocaml
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1) (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We generate target labels for every step of the handshake:

IKpsk2 (from the responder’s point of view)

\[ \begin{array}{c}
\downarrow s \\
\vdots \\
\downarrow e, e, s, s, [d]
\end{array} \]

3. We prove that the handshake state meets at each stage of the protocol the corresponding security label

4. We prove an ephemeral authentication invariant to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We formalize the Noise security levels with predicates over labels:

```text
<table>
<thead>
<tr>
<th>Level</th>
<th>Confidentiality Predicate (over i, idx, and l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( T )</td>
</tr>
<tr>
<td>1</td>
<td>can_flow (CanRead [S idxp idx.sid (\sqcup) idxp.peer_eph_label] l)</td>
</tr>
<tr>
<td>2</td>
<td>can_flow (CanRead [S idxp idx.sid; P idxp.peer]) l</td>
</tr>
<tr>
<td>3</td>
<td>can_flow (CanRead [S idxp idx.sid; P idxppeer]) l (\land) can_flow (CanRead [S idxp idx.sid; (\sqcup) idxppeer_eph_label] l)</td>
</tr>
<tr>
<td>4</td>
<td>can_flow (CanRead [S idxp idx.sid; P idxppeer]) l (\land) can_flow (CanRead [S idxp idx.sid; (\sqcup) idxppeer_eph_label] l) (\land) (compromised_before_i (P idxp) (\lor) compromised_before_i (P idxppeer) (\lor) ((\text{sid}^{`}) peer_eph_label (=) CanRead [S idxppeer sid'])])</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Level</th>
<th>Authentication Predicate (over i, idx, and l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( T )</td>
</tr>
<tr>
<td>1</td>
<td>can_flow (CanRead [P idxp; P idxppeer]) l</td>
</tr>
<tr>
<td>2</td>
<td>can_flow (CanRead [S idxp idx.sid; P idxppeer]) l</td>
</tr>
</tbody>
</table>
```

6. We prove that those security predicates are satisfied by the target labels by combining 3., and 4.

Strong forward-secrecy
Connecting to low-level implementations (actually executable!)

C: the past
Rust: the future
Now: actually running your code

• What we have seen before can **extract and run, slowly**
• But OCaml ≠ what people want
• What people want: real-world code for Python, Mozilla, etc.

```
| Real-world code! | refines | Implementation specification | exhibits | Desired properties |
```
Let \( \text{multiply_by_9} (a:\text{uint32}) : \text{Pure uint32} \)

\[
\begin{align*}
&\text{(requires)} & 9 \cdot a & \leq \text{MAX\_UINT\_32} \\
&\text{(ensures)} & \lambda \text{result} \to \text{result} & = 9 \cdot a \\
&\text{=} & \text{let } b = a \ll 3ul\text{ in} & a + b
\end{align*}
\]

\[
\begin{align*}
\text{uint32\_t multiply\_by\_9 (uint32\_t a)} \\
\{ \\
\text{uint32\_t b = a } \ll (\text{uint32\_t}3) \\
\text{return a + b;}
\}
\end{align*}
\]

Verified Low-Level Programming Embedded in F*.
(Legacy) Low* by example: a bit of HACL*

Math spec in F*

Double round takes an array, of fixed size, and shuffles it per the spec.

Memory reasoning + functional correctness

Efficient C implementation

Verification imposes no runtime performance overhead

Partial evaluation, metaprogramming, etc.

C

Low*

KaRaMeL Compiler from Low* to C

```
val double_round:
  st:state ->
  Stack unit
  (requires fun h -> live h st)
  (ensures fun h0 _ h1 -> modifies (loc st) h0 h1 /
    as_seq h1 st == Spec.double_round (as_seq h0 st))

[@ CInline]
let double_round st =
  quarter_round st (size 0) (size 4) (size 8) (size 12);
  quarter_round st (size 1) (size 5) (size 9) (size 13);
  quarter_round st (size 2) (size 6) (size 10) (size 14);
  quarter_round st (size 3) (size 7) (size 11) (size 15);
...
```

```
static inline void double_round_32(uint32_t *st) {
  st[0U] = st[0U] + st[4U];
  uint32_t std = st[12U] ^ st[0U];
  st[12U] = std << 16U | std >> 16U;
  st[8U] = st[8U] + st[12U];
  uint32_t std0 = st[4U] ^ st[8U];
  st[4U] = std0 << 12U | std0 >> 20U;
  st[0U] = st[0U] + st[4U];
  uint32_t std1 = st[12U] ^ st[0U];
  st[12U] = std1 << 8U | std1 >> 24U;
  st[8U] = st[8U] + st[12U];
  ...
}```
C code is (I hope) going the way of the dodo

• C is just hard
  • hard to reason about – arbitrary aliasing
  • hard to use – constant mine field
  • hard to evolve – frustration with wg14
• Low* = shallow embedding of C
  • no one likes generated code
  • inherits quirks of the host language
  • difficult memory reasoning with arbitrary aliasing
From the StackOverflow developer survey:

“Rust is on its seventh year as the most loved language”
Because you’re going to ask: the bottom of the list
2023: Year of Verified Rust?

• An explosion of projects
  • Verus (direct-to-SMT)
  • Prusti (atop Viper)
  • Creusot (atop Why3)
  • Liquid Rust (atop Liquid Types)
  • and many more beyond formal verification
AENEAS

from Rust Programs

to Pure Lambda Calculus

Son Ho (Inria),
Jonathan Protzenko (MSR)
Rust: low-level, memory safe language

In C:
```c
uint32_t *choose(bool b, int32_t *x, int32_t *y) {
    if (b) { return x; }
    else { return y; }
}
```

```c
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

In Rust:
```rust
fn choose<'a>(b: bool, x: &'a mut i32, y: &'a mut i32) -> &'a mut i32 {
    if b { return x; }
    else { return y; }
}
```

```rust
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

Leverage Rust’s type system to ease verification (system programming, etc.)?

- **Lifetime `'a`**
- **Exclusive access**
- **Null? Dangling? Aliased?**
- **Aliasing?**

'a ends here (borrow checker)
Leveraging Safe Rust in Verification

Explored design space:
- Rust program with annotations
  - Prusti
  - Creusot
  - Verus
  - ... (Intrinsic proofs, High-automation)
- SMT formula
- SMT solver
- Yes ✓
- No ✗
- Don’t know

Our work:
- Rust program (no annotations)
  - Aeneas
  - Pure, executable model
  - Write and prove lemmas (panic freedom, functional correctness)
  - HOL4
  - ... (Extrinsic proofs, Interaction with tactics)

(similar to Electrolysis)
Translating safe Rust to pure

Rust:

```rust
def choose<'a>
(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
{
    if b { return x; }
    else { return y; }
}

let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);
*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

```rust
let choose_fwd (b : bool) (x : i32) (y : i32) : i32 =
    if b then x else y

let choose_back (b : bool) (x : i32) (y : i32) (z : i32) :
    i32 * i32 =
    if b then (z, y) else (x, z)

let x = 0 in
let y = 1 in
let z = choose_fwd true x y in
let z = 2 in
let (x, y) = choose_back true x y z in
...
```

**Modular** translation with *forward* and *backward* functions
Recursive Functions

Rust:

```rust
pub enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn nth<'a, T>(l: &'a mut List<T>, i: u32) -> &'a mut T {
    match l {
        List::Cons(x, tl) => {
            if i == 0 {
                return x;
            } else {
                return nth(tl, i - 1);
            }
        }
        List::Nil => { panic!() }
    }
}
```

Translation:

```rust
let rec nth_fwd (t : Type) (l : list_t t) (i : u32) : result t =
    begin match l with
    | ListCons x tl ->
        if i = 0
        then Return x
        else begin i0 <-- u32_sub i 1; nth_fwd t tl i0 end
    | ListNil -> Fail Failure
    end

let rec nth_back (t : Type) (l : list_t t) (i : u32) (ret : t) :
    result (list_t t) =
    begin match l with
    | ListCons x tl ->
        if i = 0
        then Return (ListCons ret tl)
        else begin
            i0 <-- u32_sub i 1;
            tl0 <-- nth_back t tl i0 ret;
            Return (ListCons x tl0) end
    | ListNil -> Fail Failure
    end
```

Forward and backward functions behave like lenses.
Opaque (External) Functions

Rust (external dependency):

```rust
struct S { x: i32 /* private field */ }  
fn create(x: i32) -> S;  
fn get_field<'a>(s : &'a mut S) -> &'a mut i32;
```

Translation (in an interface file):

```plaintext
type S  
val create_fwd : i32 -> result S  
val get_field_fwd : S -> result i32  
val get_field_back : S -> i32 -> result S
```

Rust (local crate):

```rust
fn f() {
    let mut s = create(0);
    let x = get_field(&mut s);
    *x += 1;
}
```

Translation (in an implementation file):

```plaintext
let f_fwd =
    s <-- create_fwd 0;  
x <-- get_field_fwd s;  
x0 <-- i32_add x 1;  
s <-- get_field_back s x0;  
Return ()
```

Rust signatures efficiently capture the **effectful behavior**
Loops

Rust:

```rust
code
pub enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32) -> &mut T {
    loop {
        match ls {
            List::Cons(x, tl) => {
                if i == 0 { return x; }
                else {
                    ls = tl;
                    i -= 1;
                    continue;
                }
            }
            List::Nil => { panic!() }
        }
    }
}
```

Translation:

```rust
code
let rec nth_loop_fwd
    (t : Type) (ls : list_t t) (i : u32) : result t =
begin match ls with
    | ListCons x tl ->
        if i = 0 then Return x
    else begin i0 <-- u32_sub i 1; nth_loop_fwd t tl i0 end
    | ListNil -> Fail Failure end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

- Translated functions are similar to the recursive case

```rust
code
let rec nth_loop_back
    (t : Type) (ls : list_t t) (i : u32) (ret : t) : result (list_t t) =
begin match ls with
    | ListCons x tl ->
        if i = 0 then Return (ListCons ret tl)
    else begin
        i0 <-- u32_sub i 1;
        tl0 <-- nth_loop_back t tl i0 ret;
        Return (ListCons x tl0) end
    | ListNil -> Fail Failure end

let nth_back t ls i ret = nth_loop_back t ls i ret
```
**Infrastructure**

(charon) → LLBC (Low-Level Borrow Calculus) → Aeneas (OCaml implementation) → Pure model

**Limitations:**
- safe Rust
- no traits
- no nested loops
- no nested borrows in function signatures
- no interior mutability

Computes the borrow graph:
- don’t trust borrow checker
- operational semantics for Rust!

- Hol4
- Isabelle
- son.ho@inria.fr
github.com/AeneasVerif

*Oldes: Rust Verification by Functional Translation, ICFP 2022*
How does that work?

What do we need?
• Compute borrow graph at each point of the execution
• Precisely abstract function calls
Operational Semantics for Borrows - Mutable

```rust
let mut x = 0;              // (i)
let mut px1 = &mut x;       // (ii)
*px1 = 1;                   // (iii)
let mut px2 = &mut (*px1);  // (iv)
assert!(x == 1);            // (v)
```

// (i)
x -> (0 : u32)

// (ii)
x  -> (0 : u32)
px1 -> ?

// (ii)
x  -> ?
px1 -> mut_borrow .. (0 : u32)

// (ii)
x  -> mut_borrow .. (0 : u32)
px1 -> mut_borrow .. (0 : u32)

// (ii)
x  -> mut_borrow .. (0 : u32)
px1 -> mut_borrow .. (0 : u32)

// (ii)
x  -> mut_borrow l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x  -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x  -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x  -> (1 : u32)
px1 -> ⊥
px2 -> ⊥
```

Lazy semantics of borrows
Gives us a **borrow checker**
Abstracting function calls

```rust
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32

let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);

let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);

// Env
x -> (s2 : i32)
y -> (s3 : i32)
px -> ⊥
py -> ⊥
z -> ⊥
r0 {
    ⊥ // gave back: s2
    ⊥ // gave back: s3
    (s1 : i32)
}
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32

// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);

// Env
x -> (s2 : i32)
y -> (s3 : i32)
px -> ⊥
py -> ⊥
z -> ⊥
r0 {
   ⊥ // gave back: s2
   ⊥ // gave back: s3
   (s1 : i32)
}

// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
(s2, s3) <-- choose_back true 0 1 s1;
[.]
let x = 0;    // (i)
let px1 = &x;   // (ii)
let px2 = &x;   // (iii)

// (i)
x -> (0 : i32)

// (ii)
x  -> shared_loan {l0} (0 : i32)
px1 -> shared_borrow l0

// (iii)
x  -> shared_loan {l0, l1} (0 : i32)
px1 -> shared_borrow l0
px2 -> shared_borrow l1
Tying it all together

• A complete and general **end-to-end toolchain** for verifying cryptography, protocols, and more

• User writes **Rust**, verification expert proves **properties of interest** via a series of refinements

![Diagram](image-url)

- crypto is correct
- protocol is secure
- etc.
A few parting thoughts

• Crypto & protocols is a fertile research area
• **Formidable challenges**: math, attacker models, security
  • see e.g. MLS
• Opportunity to have real-world impact on widely-used software
  • see e.g. MLS
• Aligns with modern software development and verification in Rust