End-To-End Cryptographic Verification: From Assembly to Security Theorems

Aymeric Fromherz
Inria Paris,
MPRI 2-30
Outline

• Last week:
  • Verification of stateful code
  • Application to HACL*: A verified C cryptographic library

• Today:
  • Verifying cryptographic code in assembly
  • Symbolic Analysis with Dolev-Yao*
  • End-to-End Verification: the Noise* example
Cryptographic Implementations in Assembly

• SIMD instructions
• More optimizations (instruction ordering, register allocation, clever loop unrolling, ...)
• Avoid compiler-induced vulnerabilities

Performance comparison in OpenSSL. Smaller is better.

Data from Zinzindohoué et al, CCS 17
The AES Instruction Set (AES-NI)

• Introduced in 2008, present on most Intel processors nowadays

• 6 instructions that speed up (and simplify) AES implementations:
  • AESENC: Perform one AES encryption round
  • AESENCLAST: Perform the last AES encryption round
  • AESDEC: Perform one AES decryption round
  • AESDECLAST: Perform the last AES decryption round
  • Also, AESKEYGENASSIST and AESIMC for parts of round key generation

• Some similar instructions for SHA (SHA-EXT since 2013)
Reminder: 256-bit Modular Multiplication

128 bit × 64 bit

reduce
Unsaturated 256-bit Modular Multiplication

More multiplications, but still faster because less carry propagation!

Add without carry

102 bit

×

51 bit 51 bit 51 bit 51 bit 51 bit

reduce
Saturated Arithmetic with Intel ADX

*How to (pre-)compute a ladder: Improving the Performance of X25519 and X448, Oliveira et al., SAC’ 2017*

- Intel ADX extension offers two new instructions for addition (ADCX and ADOX) with two distinct carry flags.
- Significantly reduces carry propagation, it can be delayed
- Saturated implementations can now outperform optimized unsaturated ones!
Efficient and Trustworthy?

Quoting "Jason A. Donenfeld" <jason at zx2c4.com>:
> Hi Armando,
>
> I've started importing your precomputation implementation into kernel
> space for use in kbench9000 (and in WireGuard and the kernel crypto
> library too, of course).
>
> - The first problem remains the license. The kernel requires
> GPLv2-compatible code. GPLv3 isn't compatible with GPLv2. This isn't
> up to me at all, unfortunately, so this stuff will have to be licensed
> differently in order to be useful.
>
The rfc7748_precomputed library is now released under LGPLv2.1.
We are happy to see our code integrated in more projects.

Quoting "Jason A. Donenfeld" <jason at zx2c4.com>:
> - It looks like the precomputation implementation is failing some unit
> tests! Perhaps it's not properly reducing incoming public points?
>
> There's the vector if you'd like to play with it. The other test
> vectors I have do pass, though, which is good I suppose.

Thanks, for this observation. The code was missing to handle some carry bits,
producing incorrect outputs for numbers between 2p and 2^256. Now, I have
rewritten some operations for GF(2^255 19) considering all of these cases.
More tests were added and fuzz test against HACL implementation.

Efficient, but very tricky code. We would like to establish its
correctness formally
How to Reason about Assembly

• Low* was a shallow embedding of C in F*: We reuse F* syntax, write F* programs and extract them to C

• Assembly differs heavily from F*:
  • No variables, only registers
  • Unstructured control-flow based on jumps
  • No types/abstraction, flat memory model mapping physical addresses to bytes

• The languages are too far, we need a deeper model of assembly in F*
Assembly Verification Plan

• Model the syntax of assembly programs as an F* datatype \textit{(deep embedding)}
• Define semantics for assembly programs
• Write a program to verify using our embedding
• Based on the semantics, establish its correctness in F*

\textit{Vale: Verifying High-Performance Cryptographic Assembly Code}, Bond et al., USENIX Security 17

\textit{A Verified, Efficient Embedding of a Verifiable Assembly Language}, Fromherz et al., POPL’ 19
Modeling Intel x64 Assembly Syntax

type reg = Rax | Rbx | Rcx | Rdx ...

type operand =
    | OConst: int -> operand
    | OReg: r: reg -> operand
    | OMem: m:mem_addr -> operand

type ins =
    | Mov64: dst:operand -> src:operand -> ins
    | Add64: dst:operand -> src:operand -> ins
    ...

Structured Assembly Control-Flow

• Even in assembly, cryptographic code usually follows some structured control-flow (branching, loops)
• We do not model unstructured control-flow (gotos/arbitrary jumps)

type cond =
  | Lt: o1: operand -> o2: operand -> cond
  | Eq: o1: operand -> o2: operand -> cond
...
type code =
  | Ins: ins:ins -> code
  | Block: block:list code -> code
  | ElseIf: ifCond:cond -> ifTrue:code -> ifFalse:code -> code
  | While: whileCond:cond -> whileBody:code -> code
Generating Executable Assembly Code

- A trusted printer transforms a value of type code into an ASM file

Block([ 
Ins(Mov64 (OReg rax) (OReg rbx)); 
Ins(Add64 (OReg rax) (OConst 1))]

mov %rax %rbx
add $1, %rax

IfElse (Eq (OReg rcx) (OReg rdx))
  (... //then branch)
  (... //else branch)

cmp %rcx %rdx
jne L1
... // then branch
jmp L2
L1:
... // else branch
L2:
Defining Assembly Semantics

• We want to define an interpreter for assembly code:

```ocaml
val eval (s: state) (c: code) : a * state
```

```ocaml
type state = {
  regs: reg → nat64;
  flags: nat64;
  mem: map int nat8;
  xmms: xmm → (nat32 * nat32 * nat32 * nat32);
  ok: bool;
}
```
Defining Assembly Semantics

let eval_operand (o:operand) (s:state) : nat64 = match o with
  | OReg r -> s.regs r
  | OConst n -> n
  ...

let valid_src_operand (o:operand) (s:state) =
  match o with
  | OMem addr -> forall p. p >= addr && p < addr + 8 => Map.contains s.mem p
  | _ -> true

let valid_dst_operand (o:operand) (s:state) =
  match o with
  | OConst _ -> false
  | OReg r -> r <> rsp
  ...

Defining Assembly Semantics

• Semantics in a monadic style to simplify notations
• Underspecify when possible to simplify model (e.g., flags)

```
let eval_ins (ins:ins) =
  s <- get;
  match ins with
  | Mov64 dst src -> . . .
  | Add64 dst src ->
    check (valid_src_operand src);; check (valid_dst_operand dst);;
    havoc flags;;
    let sum = eval_operand dst s + eval_operand src s in
    let new_carry = sum ≥ pow2_64 in
    set_operand dst ins (sum % pow2_64);;
    set_flags (update_cf s.flags new_carry)
```
The Vale Language

• Writing a full program as an AST is tedious (e.g., Block([Ins(Mov64 (OReg rax) (OReg rbx)); Ins(Add64 (OReg rax) (OConst 1))]) )
• Vale exposes a user-friendly language to simplify writing code
The Vale Language: Inlining

• Vale supports *inline if* statements, which are evaluated during **code generation**
• Useful for selecting instructions and for unrolling loops

```
inline if(platform == x86_AESNI) {
    ...
}
```

```
inline if (n > 0) {
    ...
    recurse(n - 1);
}
```
Vale: A Summary

Crypto code in Vale language → Vale Tool

Lemmas

AST + Proofs

Crypto Specification

F* Verifier

Machine Semantics (x86, x64, ARMv7)

Verified? (Yes / No)
Vale: A Summary

- Crypto code in Vale language
- Lemmas
- Crypto Specification
- Machine Semantics (x86, x64, ARMv7)

Vale Tool

AST + Proofs

F* Verifier

AST

Verified? (Yes / No)

Assembly Printer

Assembly Code

Assembler (e.g. GAS / MASM)
Vale: A Summary

- **Untrusted Components**
- **Verified Components**
- **Trusted Components**

- **Vale Tool**
  - Crypto code in Vale language
  - Lemmas
  - AST + Proofs

- **Crypto Specification**
  - Machine Semantics (x86, x64, ARMv7)

- **F* Verifier**
  - Verified? (Yes / No)

- **Handwritten Libraries**
- **Assembly Printer**
  - Assembler (e.g. GAS / MASM)
Information Leakage

Secrets should not leak through:

• **Digital side channels:** Observations of program behavior through cache usage, timing, memory accesses, ...

• **Residual Program State:** Secrets left in registers or memory after termination of program
Vale Taint Analysis

• Establish non-interference through a taint analysis (cf Lecture 2)
  \[
  \text{val taint\_analysis: c:code -> isPub:(loc -> bool) -> b:bool\{b \Rightarrow \text{isLeakageFree c isPub}\}}
  \]

• We can prove the correctness of the analysis based on the semantics
  • Possible because we can directly reason on the deeply embedded semantics

• For memory reasoning, we reuse memory aliasing information needed when proving functional correctness
  \[
  \text{OMem: m:mem\_addr -> t:taint -> operand}
  \]

• Taint is erased at runtime, only used for the analysis
Automatically Optimizing Assembly Code

• Handwritten assembly code is already manually optimized
• Some small changes can yield performance improvements on some architectures (depending on microarchitectural details)
• **Idea:** Try peephole optimizations to tweak code, while proving that the code transformations preserve semantics

*Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language*, Bosamiya et al., VSTTE’ 20
Semantically Equivalent Transformations

let semantically_equivalent (c1 c2: code) = 
  (forall (s1 s2:state). equiv_states s1 s2 ==> 
   equiv_states (eval_code c1 s1) (eval_code c2 s2))

type transform = c1: code -> c2:code{semantically_equivalent c1 c2}

• **Goal:** Define transformations satisfying the *transform* type
• Can be proven correct as an F* theorem thanks to our deep embedding of semantics
Transformation Example: Xor Rewriting

• Replace all occurrences of \texttt{mov \{reg\}, 0} by \texttt{xor \{reg\} \{reg\}}

• Semantically equivalent? Yes, \texttt{xor n n} is equal to 0, so this is equivalent to setting the value 0 in register \{reg\}
Instruction Reordering

• If we have two instructions A and B, we can swap them if there is no read-write or write-write conflict

• Formally, we can rewrite A; B into B; A if
  \[ \forall l \in \text{writes}(A). l \notin \text{reads}(B) \land l \notin \text{writes}(B) \]

add(r1, r2) is defined as \( r1 := r1 + r2 \)
Can add(rax, rbx); add(rcx, rdx) be rewritten into add(rcx, rdx); add(rax, rbx)?
Can add(rbx, rax); add(rcx, rdx) be rewritten into add(rcx, rdx); add(rbx, rax)?
Can add(rax, rbx); add(rcx, rbx) be rewritten into add(rcx, rbx); add(rax, rbx)?
Block Instruction Reordering

• Instruction reordering can be extended to **groups** of instructions

\[ \forall (X, Y) \in (A, B). \]
\[ \forall l \in \text{writes}(X). l \notin \text{reads}(Y) \land l \notin \text{writes}(Y) \]

Ex:  
\[ A = \text{add rax, 1; adc rbx, 1; } B = \text{add rcx, 1; adc rdx, 1} \]

• In each block, adc (add with carry) relies on the carry of the previous instruction

• We can swap blocks, but not individual instructions
Optimizing for Processor Generation

• So far, optimizations for an architecture (e.g., Intel x64 vs ARM)
• Transformations enable optimization for a processor generation (e.g., Intel’s i5-2500, i7-3770, i7-7600U, or i9-9900K)

• Workflow:
  • Start from verified assembly code
  • Try many verified transformations
  • Benchmark; if faster than previous fastest, keep this version

• Experimental results: Speedups of up to 27% compared to OpenSSL
Back to Curve25519

• We can implement efficient core modular arithmetic in assembly
  • Use ADX + BMI2 instructions
  • Prove correctness and side-channel resistance using Vale

• We would prefer to write the rest of the code in C
  • Add/Double formulae, Montgomery ladder
  • Implement and verify in Low*, retrieve executable C code

• How to interoperate between the two?
Interoperating between C and Assembly

Several questions

• How to relate memory models?
• How to enforce calling conventions across function calls?
• How to unify specifications?
• How to preserve security guarantees?
Interoperating between Vale and Low*

- We do not need a generic interoperation
  - For crypto, no callbacks from assembly to C, no allocation in assembly, ...

We call a Vale function from assembly, entirely execute it, and finally resume Low* execution
let call_assembly (c:vale_code) arg1 ... argn : Stack uint64
    (requires lift_pre P) (ensures lift_post Q)
= let h0 = get() in
  let s0 = initial_vale_state h0 arg1 ... argn in
  let s1 = eval c s0 in
  let rax, h1 = final_lowstar_state h0 s1 in
  put h1; rax

• Small, trusted model of interoperation
• Parametric in calling conventions (Windows, Linux, inline assembly, ...)
• Lifting of specifications is verified against the trusted model
• Lift_* is done generically
Interoperation: Calling Conventions

let initial_vale_state_linux_x64 h0 arg1 arg2 arg3 =
  let init_regs r =
    if r = rdi then arg1 else
    if r = rsi then arg2 else
    if r = rdx then arg3
  in let init_mem = lower h0 in ...
{ ok = true; regs = init_regs; mem = init_mem; ... }

• In practice: arity-generic to support an arbitrary number of arguments
• Stack spilling if too many arguments
• Calling conventions also require some registers to be preserved by callee (e.g., RBX, RSP, RBP, and R12–R15 on Linux x64)
Interoperation at Work: Optimizing Curve25519

let curve25519 (...) = ...
  fmul1 out f1 f2;
...

val fmul1 (dst:u256) (a:u256) (b:uint64{v f2 < pow2 17}) :
  Stack unit
  (requires fun h -> adx_enabled \ bmi2_enabled \ ...)
  (ensures ...)

procedure fmul1(...)...
  lets dst_ptr @ rdi; inA_ptr @ rsi; b @ rdx;
  requires adx_enabled && bmi2_enabled && ...
  ensures ...
  {
    fast_mul1(0, inA_b); ... Mov64(b, 38);
    carry_pass(false, 0, dst_b);
  }

## Interoperation at Work: Optimizing Curve25519

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Radix</th>
<th>Language</th>
<th>CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>donna64</td>
<td>51</td>
<td>C</td>
<td>159634</td>
</tr>
<tr>
<td>fiat-crypto</td>
<td>51</td>
<td>C</td>
<td>145248</td>
</tr>
<tr>
<td>amd64-64</td>
<td>51</td>
<td>Assembly</td>
<td>143302</td>
</tr>
<tr>
<td>sandy2x</td>
<td>25.5</td>
<td>Assembly + AVX</td>
<td>135660</td>
</tr>
<tr>
<td>HACL* + Vale (portable)</td>
<td>51</td>
<td>C</td>
<td>135636</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>64</td>
<td>Assembly + ADX</td>
<td>118604</td>
</tr>
<tr>
<td>Oliveira et al.</td>
<td>64</td>
<td>Assembly + ADX</td>
<td>115122</td>
</tr>
<tr>
<td>HACL* + Vale (targeted)</td>
<td>64</td>
<td>C + Assembly + ADX</td>
<td>113614</td>
</tr>
</tbody>
</table>

Verification code can reach state-of-the-art performance, sometimes outperforming the best existing unverified implementations.
Towards a Cryptographic Provider

• We focused so far on verifying individual implementations

• Clients expect a **cryptographic library** with user-friendly APIs, not a collection of primitives
  • APIs must be grouped by family (Agility)
  • Must allow to switch between implementations (Multiplexing)
  • Must cover all cryptographic needs (comprehensive)
EverCrypt, a Verified Cryptographic Provider

- Layer on top of HACL* + Vale
- Provides generic APIs for hashes, AEAD, ... with a single, unified specification
- Performs multiplexing between available implementations (depending on CPU features available, user preference, ...)
- Usable by verified and unverified clients alike
EverCrypt: Agility

• Verifies that multiple algorithms satisfy the same family of specifications
• Provides a unified API
• Makes switching from one algorithm to the other straightforward
EverCrypt: Multiplexing

• Several implementations with different levels of optimization (e.g., portable C, C with SIMD, Intel ASM with ADX+BMI2)

• Several versions require assumptions on CPU architecture (e.g., Intel x64, presence of AESNI instruction set)

• We want to use the fastest implementation available, but to avoid illegal instruction errors
EverCrypt: Multiplexing

• Mix of architecture requirements (has_avx/is_x64) and user preferences (wants_vale)
• Functions require as a precondition to run with the right extension set
• CPU instructions (cpuid) can inform about available extensions
• Leverage Low*/Vale interop to lift this information to the Low* level, and guarantee to avoid illegal instruction errors
### EverCrypt: Available Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>C version</th>
<th>ASM version</th>
<th>Agile API</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEAD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-GCM</td>
<td>✔</td>
<td>✔ (AESNI)</td>
<td>✔</td>
</tr>
<tr>
<td>ChachaPoly</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>ECDH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve25519</td>
<td>✔</td>
<td>✔ (BMI2 + ADX)</td>
<td>✔</td>
</tr>
<tr>
<td>P-256</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD5, SHA1</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SHA2</td>
<td>✔</td>
<td>✔ (SHAEXT)</td>
<td>✔</td>
</tr>
<tr>
<td>SHA3</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blake2</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Many functionalities, covering most of the standard cryptographic needs
End-to-End Verification

• So far, we saw different techniques for verifying the safety and correctness of low-level, efficient cryptographic implementations
• How to also preserve security guarantees at the protocol level?

• Case study: the Noise protocols


(Noise* slides from Son Ho, DY* slides from Karthik Bhargavan)
What is Noise?

• What does a handshake protocol do?
  • Exchange data to have a shared secret to communicate
  • Various use cases (one-way encryption, authenticated servers, mutual authentication, etc.)
  • Varying security
• Various protocols, some of them very advanced and complex (ex.: TLS):
  • Backward compatibility
  • Cipher suites negotiation
  • Session resumption
  • ...
• When advanced features not needed: Noise family of protocols
Noise Protocol Framework: Examples

\[ \text{X:} \]
- $s \leftarrow \ldots \rightarrow e, es, s, ss$

(one-way encryption: NaCl Box, HPKE...)

\[ \text{IK:} \quad \text{WhatsApp} \]
- $s \leftarrow \ldots \rightarrow e, es, s, ss$
- $e, ee, se \leftarrow e$

(mutual authentication and 0-RTT)

\[ \text{IKpsk2:} \quad \text{Wireguard VPN} \]
- $s \leftarrow \ldots \rightarrow e, es, s, ss$
- $e, ee, se \leftarrow e$

(authenticated server)

\[ \text{NX:} \]
- $e \rightarrow \ldots \leftarrow e, ee, s, es$

\[ \text{XX:} \]
- $e \rightarrow \ldots \leftarrow e, ee, s, es$
- $s, se \rightarrow s, es$

\[ \text{XK:} \quad \text{Lightning, I2P} \]
- $s \leftarrow \ldots \rightarrow e, es$
- $e, ee \leftarrow e$
- $s, se \rightarrow s, se$

Today: 59+ protocols (but might increase)
Noise Protocol Example: IKpsk2

<table>
<thead>
<tr>
<th>Initiator</th>
<th>Responder</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKpsk2:</td>
<td></td>
</tr>
<tr>
<td>$\leftarrow s$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$\rightarrow e, es, s, ss, [d0]$</td>
<td></td>
</tr>
<tr>
<td>$\leftarrow e, ee, se, psk, [d1]$</td>
<td></td>
</tr>
<tr>
<td>$\leftrightarrow [d2, d3, ...]$</td>
<td></td>
</tr>
</tbody>
</table>

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations...)
- Send/receive encrypted data
Noise Protocol Example: IKpsk2

Initiator

IKpsk2:

← s

... Exchange key material

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, ...]

The handshake describes how to:
• **Exchange key material**
• Use those to derive shared secrets (Diffie-Hellman operations...)
• Send/receive encrypted data
Noise Protocol Example: IKpsk2

The handshake describes how to:

- **Exchange key material**
- Use those to derive shared secrets (Diffie-Hellman operations...)
- Send/receive encrypted data

<table>
<thead>
<tr>
<th>Initiator</th>
<th>Responder</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IKpsk2:</strong></td>
<td></td>
</tr>
<tr>
<td>$\leftarrow s$</td>
<td>$s$: static</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$e$: ephemeral</td>
</tr>
<tr>
<td>$\rightarrow e, es, s, ss, [d0]$</td>
<td></td>
</tr>
<tr>
<td>$\leftarrow e, ee, se, psk, [d1]$</td>
<td></td>
</tr>
<tr>
<td>$\leftrightarrow [d2, d3, ...]$</td>
<td></td>
</tr>
</tbody>
</table>

The handshake describes how to:

- **Exchange key material**
### Noise Protocol Example: IKpsk2

<table>
<thead>
<tr>
<th>Initiator</th>
<th>Responder</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKpsk2:</td>
<td></td>
</tr>
<tr>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Derive shared secrets (Diffie-Hellman operations...)</td>
</tr>
<tr>
<td>$e, es, s, ss, [d0]$</td>
<td></td>
</tr>
<tr>
<td>$e, ee, se, psk [d1]$</td>
<td></td>
</tr>
<tr>
<td>$\leftrightarrow [d2, d3, ...]$</td>
<td></td>
</tr>
</tbody>
</table>

The handshake describes how to:
- Exchange key material
- **Use those to derive shared secrets (Diffie-Hellman operations...)**
- Send/receive encrypted data
Noise Protocol Example: IKpsk2

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations...)
- Send/receive encrypted data
Noise Protocol Example: IKpsk2

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations...)
- Send/receive encrypted data
Noise Protocol Example: IKpsk2

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

Secrets are **chained**:
- \(d0\) encrypted with a key derived from \(es, ss\)
- \(d1\) encrypted with a key derived from \(es, ss, ee, se, psk\)

⇒ The more the handshake progresses, the more secure the shared secrets are
Noise Protocol Example: IKpsk2

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 $e$, ss
• $d_1$ encrypted with a key derived from $e$, ss, ee, se, psk

$⇒$ **The more the handshake progresses, the more secure the shared secrets are**
What is Noise*?

Correctly implemented protocols?
• **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
• On top: complete, verified library stack exposed through a high-level, defensive API
• Complemented with a formal symbolic security analysis
What is Noise*?

Correctly implemented protocols?
- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified library stack exposed through a high-level, defensive API
- Complemented with a formal symbolic security analysis

![Diagram](image-url)

- **Low-Level Code**
  - Session API
  - Key Management
  - Messaging Code
  - Crypto Library

- **Formal Specification**
  - Secure Channel API
  - Abstract Device API
  - Noise Protocol Spec
  - Crypto Algorithms

- **Security Analysis**
  - Security Levels
  - PKI Assumptions
  - Secrecy Labels + Invariants
  - Symbolic Model

- **HACL* Verified Cryptographic Library**

- **Verified Noise Protocol Code**

- **Verified Noise Library Stack & High-Level API**

- **Compile protocol pattern to high performance C code**
  - Noise-AX C Code
  - Noise-AW C Code
  - Noise-LKpsk2 C Code

- **Verify (F*)**
  - Yes → Noise-... C Code
  - No → Bug?

- **Verify (D/*)**
  - Yes → Security Proof
  - No → Attack?
What is Noise\*?

Correctly implemented protocols?

- **Noise\* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified library stack exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**

![Diagram showing the structure of Noise\*, including Low-Level Code, Formal Specification, and Security Analysis sections.](image-url)
What is Noise*?

Correctly implemented protocols?
- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified library stack exposed through a high-level, defensive API
- Complemented with a formal symbolic security analysis

![Diagram of Noise* architecture]

- **Low-Level Code**
  - Session API
  - Key Management
  - Messaging Code
  - Crypto Library

- **Formal Specification**
  - Secure Channel API
  - Abstract Device API
  - Noise Protocol Spec
  - Crypto Algorithm

- **Security Analysis**
  - Security Levels
  - PKI Assumptions
  - Secrecy Labels + Invariants
  - Symbolic Model

- **Verifying Noise**
  - Verify (F*)
    - Yes → Compile protocol pattern to high performance C code
    - No → Bug?

- **Security Proof**
  - Verify (DY*)
    - Yes → Yes
    - No → Attack?
The Noise* protocol compiler
Formal Functional Specification of Noise

noiseprotocol.org:

- **message_patterns**: A sequence of message patterns. Each message pattern is a sequence of tokens from the set ("e", "s", "r", "re", "es", "es", "rs", "rs"). (An additional "psk" token is introduced in Section 9, but we defer its explanation until then.)

A **HandshakeState** responds to the following functions:

- **Initialize(handshake_pattern, initiator, prologue, s, e, rs, re)**: Takes a valid handshake_pattern (see Section 7) and an initiator boolean specifying this party's role as either initiator or responder.

- Takes a prologue byte sequence which may be zero-length, or which may contain context information that both parties want to confirm is identical (see Section 6).

- Takes a set of DH key pairs (s, e) and public keys (rs, re) for initializing local variables, any of which may be empty. Public keys are only passed in if the handshake_pattern uses pre-messages (see Section 7). The ephemeral values (s, rs) are typically left empty, since they are created and exchanged during the handshake, but there are exceptions (see Section 10).

Performs the following steps:

- Derives a protocol_name byte sequence by combining the names for the handshake pattern and crypto functions, as specified in Section 8. Calls "InitializeSymmetric(protoc0l_name).

- Calls MixHash(prologue).

- Sets the initiator, s, e, rs, and re variables to the corresponding arguments.

- Calls MixHash() once for each public key listed in the pre-messages from handshake_pattern, with the specified public key as input (see Section 7 for an explanation of pre-messages). If both initiator and responder have pre-messages, the initiator's public keys are hashed first. If multiple public keys are listed in either party's pre-message, the public keys are hashed in the order that they are listed.

- Sets message_patterns to the message patterns from handshake_pattern.

- **WriteMessage(payload, message_buffer)**: Takes a payload byte sequence which may be zero-length, and a message_buffer to write the output into. Performs the following steps, aborting if any EncryptAndHash() call returns an error:

**F** specification written as an interpreter:

```fsharp
// Process a message (without its payload)
let rec send_message_tokens nc initiator is_psk tokens
  (st : handshake_state) : result (bytes & handshake_state) =
  match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Res (msg2, st2) ->
      Res (msg1 @ msg2, st2)
```
Target code

Wireguard VPN (IKpsk2):

```c
/* First message: e, es, s, ss */
handshake_init(handshake->chaining_key, handshake->hash, handshake->remote_static);

/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral, handshake->ephemeral_private))
    goto out;
message_ephemeral(dst->unencrypted_ephemeral, dst->unencrypted_ephemeral, handshake->chaining_key, handshake->hash);

/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private, handshake->remote_static))
    goto out;

/* s */
message_encrypt(dst->encrypted_static, handshake->static_identity->static_public, NOISE_PUBLIC_KEY_LEN, key, handshake->hash);

/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key, handshake->precomputed_static_static))
    goto out;
```

Our Low* code follows the structure of the below spec.:

```plaintext
let rec send_message_tokens #nc initiator is_psk tokens st =
match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens_initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Success (msg2, st2) ->
      Res (msg1 @ msg2, st2)
```

Specialized, idiomatic C code: no recursion, no token lists, etc.

How to specialize an interpreter for a given input?
How to turn an interpreter into a compiler?
Hybrid Embeddings

Idea: use F* to meta-program as much as possible:
• Similar to super advanced C++ templates
• Write a meta-program once, specialize N times (⇒ 59 patterns)

• Large-scale, higher-level application of techniques seen on cryptographic primitives (Lecture 3)

With Noise*: complete, meta-programmed protocol stack
let send_IKpsk2_message0 (st : handshake_state) =
    send_message_tokens true true [E; ES; S; SS] st
let send_IKpsk2_message0 (st : handshake_state) =
  match [E; ES; S; SS] with
  | [] -> Res (empty, st)
  | tk :: tokens1 ->
    match send_message_token true true tk st with
     | Fail e -> Fail e
     | Res (msg1, st1) ->
       match send_message_tokens true true tokens1 st1 with
        | Fail e -> Fail e
        | Res (msg2, st2) ->
        Res (msg1 @ msg2, st2)
let send_IKpsk2_message0 (st : handshake_state) =
    match send_message_token true true E st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
        match send_message_tokens true true [ES; S; SS] st1 with
        | Fail e -> Fail e
        | Res (msg2, st2) ->
            Res (msg1 @ msg2, st2)
let send_IKpsk2_message0 (st : handshake_state) =
  match send_message_token true true E st with
  | Fail e -> Fail e
  | Res (msg1, st1) ->
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true SS st3 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st4)
let send_IKpsk2_message0 (st : handshake_state) = 
  match // E 
  begin match st.ephemeral with 
  | None -> Fail No_key 
  | Some k -> 
  let sym_st1 = mix_hash k.pub st.sym_state in 
  let sym_st2 = 
  if true // This is `is_psk` 
  then mix_key k.pub sym_st1 
  else sym_st1 
  in 
  let st1 = { st with sym_state = sym_st2; } in 
  let msg1 = k.pub in 
  Res (msg1, st1) 
  end 
  with 
  | Fail e -> Fail e 
  | Res (msg1, st1) -> // Other tokens: 
  match send_message_token true true ES st1 with 
  | Fail e -> Fail e 
  | Res (msg2, st2) -> 
  match send_message_token true true S st2 with 
  | Fail e -> Fail e 
  | Res (msg3, st3) ->
let send_IKpsk2_message0 (st : handshake_state) = 
  match // E 
  begin match st ephemeral with
    | None -> Fail No_key // Unreachable if proper precondition
    | Some k ->
      let sym_st1 = mix_hash k pub st sym_state in
      let sym_st2 = mix_key k pub sym_st1 in
      let st1 = { st with sym_state = sym_st2; } in
      let msg1 = k pub in
      Res (msg1, st1)
  end
  with
  | Fail e -> Fail e // Unreachable if proper precondition
  | Res (msg1, st1) -> // Other tokens:
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true S st2 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st3)

Embeddings in Low* are shallow: partial reduction applies!

// Simplified
let rec send_message_tokens_m =
  fun smi initiator is_psk tokens st outlen out ->
    match tokens with
    | Nil -> success _
    | tk :: tokens' ->
      let tk_outlen = token_message_vs nc smi tk in
      let tk_out = sub out 0 ul tk_outlen in
      let r1 = send_message_token_m smi initiator ...

⇒ Compilation through staging: first step with F* normalizer

E disappeared!
⇒ “meta” parameters (and computations) vs “runtime” parameters (and computations)
Tweaking Control-Flow and Types

// Spec
match send_message_token true true $ st with
  | Fail e -> Fail e
  | Res (msg, st') -> ...

// Low*
let r : error_code = send_message_token ... $ ... in
  if r = Success then
  ... // "if" branch
else
  ... // "else" branch

Unreachable!

Always succeeds!
Tweaking Control-Flow and Types

// Spec
match send_message_token true true S st with
| Fail e -> Fail e // Unreachable!
| Res (msg, st') -> ...

// Low
let r : error_code = send_message_token ... S ... in
if r = Success then
... // “if” branch
else
... // “else” branch
Always succeeds!

F* has dependent types!

type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true

let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...

Unreachable!

Always succeeds!
Tweaking Control-Flow and Types

// Spec
match send_message_token true true S st with
| Fail e -> Fail e Unreachable!
| Res (msg, st') -> ...

// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // “if” branch
else
  ... // “else” branch

// Low*
let r : error_code_or_unit false = send_message_token ... S ... in
if is_success #false r then
  ... // “if” branch
else
  ... // “else” branch

F* has dependent types!

type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true

let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...

After partial reduction
Tweaking Control-Flow and Types

Spec

match send_message_token true true S st with
| Fail e -> Fail e  [Unreachable!]
| Res (msg, st') -> ...

Low*

let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
... // "if" branch
else
... // "else" branch

let r : unit = send_message_token ... S ... in
if is_success false r then
... // "if" branch
else
... // "else" branch

F* has dependent types!

type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true

let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
Tweaking Control-Flow and Types

// Spec
match send_message_token true true S st with
| Fail e -> Fail e  // Unreachable!
| Res (msg, st’) -> ...

// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // “if” branch
else
  ... // “else” branch

// Low*
let r : unit = send_message_token ... S ... in
if true then
  ... // “if” branch
else
  ... // “else” branch

F* has dependent types!

type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true

let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...

After partial reduction

Unreachable!
Tweaking Control-Flow and Types

Write **general dependent types** which reduce to **precise non-dependent types**:  
- Drastically improve code quality (make it smaller, more readable, more idiomatic)  
- Make extracted types (structures, etc.) more precise  
- Make **function signatures** more informative (unit elimination)

```c
val f (x : uint32_t) (y : unit) : unit // Low*
void f (x : uint32_t); // Generated C
```
Tweaking Control-Flow and Types

Write general dependent types which reduce to precise non-dependent types:
- Drastically improve code quality (make it smaller, more readable, more idiomatic)
- Make extracted types (structures, etc.) more precise
- Make function signatures more informative (unit elimination)

• We don’t have to choose between genericity and efficiency
Generated Code (IKpsk2)

Noise*

```c
/* e */
Impl_Noise_Instances_mix_hash(ms_h, (uint32_t)32U, mepub);
Impl_Noise_Instances_kdf(ms_ck, (uint32_t)32U, mepub, ms_ck, mc_state, NULL);
memcpy(tk_out, mepub, (uint32_t)32U * sizeof (uint8_t));
/* es */

uint8_t* out = pat_out + (uint32_t)32U;
Impl_Noise_Instances_error_code r11 = Impl_Noise_Instances_mix_dh(mepriv, mremote_static, mc_state, ms_ck, ms_h);
Impl_Noise_Instances_error_code r2;
if (r11 == Impl_Noise_Instances_CSuccess) {
    /* s */
    uint8_t* out1 = out +
    uint8_t* tk_out2 = out1;
    Impl_Noise_Instances_encrypt_and_hash((uint32_t)32U,
        mepub,
        tk_out2,
        mc_state,
        ms_h,
        (uint64_t)32U);
    /* ss */
    Impl_Noise_Instances_error_code r = Impl_Noise_Instances_mix_dh(mspriv, mremote_static, mc_state, ms_ck, ms_h);
    Impl_Noise_Instances_error_code r20 = r;
    Impl_Noise_Instances_error_code r21 = r20;
    r2 = r21;
} else
    r2 = r11;
```

Wireguard VPN (for reference):

```c
/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral,
        handshake->ephemeral_private))
    goto out;
message_ephemeral(dst->unencrypted_ephemeral,
        dst->unencrypted_ephemeral, handshake->chaining_key,
        handshake->hash);
/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private, handshake->remote_static))
    goto out;
/* s */
message_encrypt(dst->encrypted_static,
        handshake->static_identity->static_public,
        NOISE_PUBLIC_KEY_LEN, key, handshake->hash);
/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key,
        handshake->precomputed_static_static))
    goto out;
```
What does the high-level API give us?

- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation
High-Level API

IKpsk2:

$\leftarrow s$

$\ldots$

$\rightarrow e, es, s, ss, [d0]$

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

$\leftrightarrow [d2, d3, \ldots]$
High-Level API

IKpsk2:

\[
\begin{align*}
\text{← } & S \\
\text{...} & \\
\rightarrow & e, es, S, ss, [d0] \\
\leftarrow & e, ee, se, psk, [d1] \\
\leftrightarrow & [d2, d3, ...]
\end{align*}
\]

- Initiator and responder must remember which key belongs to whom

Peer Management
High-Level API

IKpsk2:

\[
\begin{align*}
\leftarrow & \; s \\
\ldots & \\
\rightarrow & \; e, \; es, \; s, \; ss, \; [d0] \\
\leftarrow & \; e, \; ee, \; se, \; psk, \; [d1] \\
\leftrightarrow & \; [d2, \; d3, \; ...]
\end{align*}
\]

- 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

**Peer Management**

**Key Validation**
High-Level API

**IKpsk2:**

\[
\begin{align*}
\leftarrow & \ s \\
\ldots \\
\rightarrow & \ e, \ es, \ s, \ ss, \ [d0] \\
\leftarrow & \ e, \ ee, \ se, \ psk, \ [d1] \\
\leftrightarrow & \ [d2, \ d3, \ ...] 
\end{align*}
\]

- 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
High-Level API

\[ \text{IKpsk2:} \]

\[ \leftarrow s \]

\[ \ldots \]

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

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

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

- 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)
High-Level API

IKpsk2:

\[ \leftarrow s \]

\[ \ldots \]

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

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

\[ \leftrightarrow [d2, d3, \ldots] \]

- 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?

Peer Management
Key Validation
Key Storage
State Machine
Message Encapsulation
What does the high-level API give us?

- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation
Meta-Programmed State Machine
With 3 messages (ex.: XX):

```
//
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...);
  else if (step == 1)
    return send_message1(...);
  else if (step == 2)
    return send_message2(...);
  else
    ... // Unreachable!!
}
```
Meta-Programmed State Machine
With 3 messages (ex.: XX):

```c
// With precondition: step <= 2
error_code handshake_send(..., uint step, ...) {
    if (step == 0)
        return send_message0(...);
    else if (step == 1)
        return send_message1(...);
    else // No check - step == 2
        return send_message2(...);
}
```
Meta-Programmed State Machine
With 3 messages (ex.: XX):

```c
// With precondition: step <= 2
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...); // initiator state
  else if (step == 1)
    return send_message1(...); // responder state!
  else // No check - step == 2
    return send_message2(...); // initiator state
}
```

state is a dependent type, reduced and monomorphized at extraction time!
Meta-Programmed State Machine
With 3 messages (ex.: XX):

```c
// With precondition: step <= 2 \ (step % 2) == 0
error_code initiator_handshake_send(..., uint step, ..., initiator_state st) {
    if (step == 0) {
        return send_message0(...);
    } else { // No check - step == 2
        return send_message2(...);
    }
}

// With precondition: step <= 2 \ (step % 2) == 1
error_code responder_handshake_send(..., uint step, ..., responder_state st) {
    return send_message1(...);
}
```

state is a dependent type, reduced and monomorphized at extraction time!
Meta-Programmed State Machine

With 3 messages (ex.: XX):

```c
// With precondition: step <= 2 /\ (step % 2) == 0
error_code initiator_handshake_send(..., uint step, ..., initiator_state st) {
    if (step == 0) {
        return send_message0(...);
    } else /* No check - step == 2 */
    return send_message2(...);
}
```

```c
// With precondition: step <= 2 /\ (step % 2) == 1
error_code responder_handshake_send(..., uint step, ..., responder_state st) {
    return send_message1(...);
}
```

```c
// Generated from an F* inductive
struct state {
    int tag;
    union {
        struct initiator_state;
        struct responder_state;
    } val;
}
```

state is a dependent type, reduced and monomorphized at extraction time!
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

**Target C code:**

```c
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  } else {
    return send_message2(...);
  }
}
```

```c
error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

**F* code:**

```fsharp
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

**Target C code:**

```c
error_code initiator_handshake_send(...) {
    if (step == 0) {
        return send_message0(...);
    } else {
        return send_message2(...);
    }
}

error_code responder_handshake_send(...) {
    return send_message1(...);
}
```

**F* code:**

```fsharp
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

---

**Meta parameter**

\( i \in \{0, 1, \ldots\} \)

**Runtime parameter**

\( i \in \{0, 1, \ldots\} \)
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

**Target C code:**

```c
error_code initiator_handshake_send(...) {
    if (step == 0) {
        return send_message0(...);
    } else {
        return send_message2(...);
    }
}

error_code responder_handshake_send(...) {
    return send_message1(...);
}
```

**F* code:**

```fsharp
if (step == 0) {
    return send_message0(...);
} else {
    return send_message2(...);
}
```

**Meta parameter**

\( i \in \{0, 1, \ldots\} \)

**Runtime parameter**

\( i \in \{0, 1, \ldots\} \)
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```c
error_code initiator_handshake_send(...) {
    if (step == 0) {
        return send_message0(...);
    } else
        return send_message2(...);
}
error_code responder_handshake_send(...) {
    return send_message1(...);
}
```

F* code:

```fsharp
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat)(step:UInt32.t) =
    if i+2 >= num_handshake_messages then...
    else if step = size i then...
    else handshake_send ... (i+2) step // Increment i by 2
```

Meta parameter
(i ∈ {0, 1, ...})

Runtime parameter

Meta parameter

Target C code:

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat)(step:UInt32.t) =
    if i+2 >= num_handshake_messages then...
    else if step = size i then...
    else handshake_send ... (i+2) step // Increment i by 2
```
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:
```c
error_code initiator_handshake_send(...) {
    if (step == 0) {
        return send_message0(...);
    } else {
        return send_message2(...);
    }
}

error_code responder_handshake_send(...) {
    return send_message1(...);
}
```

F* code:
```fstar
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat)(step:UInt32.t) =
    if i+2 >= num_handshake_messages then
        ... // last possible send_message function
    else if step = size i then
        ... // instantiated send_message function
    else
        handshake_send ... (i+2) step // Increment i by 2
```
Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

**Target C code:**
```c
error_code initiator_handshake_send(...) {
    if (step == 0) {
        return send_message0(...);
    } else {
        return send_message2(...);
    }
}

error_code responder_handshake_send(...) {
    return send_message1(...);
}
```

**F* code:**
```fsharp
// Pre: initiator==((i%2)==0) \ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... [(i:nat)][(step:UInt32.t) =
    if i+2 >= num_handshake_messages then
        ... // last possible send_message function
    else if step = size i then
        ... // instantiated send_message function
    else
        handshake_send ... (i+2) step // Increment i by 2

let initiator_handshake_send ... step = handshake_send true ... 0 step
let responder_handshake_send ... step = handshake_send false ... 1 step
```

**Meta parameter**
\(i \in \{0, 1, \ldots\}\)

**Runtime parameter**
\(i \in \{0, 1, \ldots\}\)
What does the high-level API give us?

- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation
Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

**Initialization and premessages phase:**

```c
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                         charlie_public_key,
                         alice_charlie_psk);
...
```

**Handshake phase:**

```c
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```c
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);
...
```

```c
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
...
```

```
// Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```
Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```c
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
    charlie_public_key,
    alice_charlie_psk);
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);
```

```
IKpsk2:

\[ \leftarrow s \quad \text{initiator knows responder from beginning} \]
```

```
\rightarrow e, es, s, ss
\leftarrow e, ee, se, psk
```

**Handshake** phase:

```c
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```c
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```
Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```c
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                           charlie_public_key,
                           alice_charlie_psk);
```

```c
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);
```

...  

IKpsk2:

\[ \rightarrow s \text{ initiator knows responder from beginning} \]

\[ \cdots \quad \text{Responder learns initiator’s identity} \]

\[ \rightarrow e, es, s, ss \]

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

**Handshake phase:**

```c
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
```

```c
// On Bob’s side
session *sn;
sn = create_responder(dv); // We don’t know who we talk to yet
uint8_t msg[...];
```

```c
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```
Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialiation and **premessages** phase:

```c
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
               charlie_public_key,
               alice_charlie_psk);
```

```c
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...IKpsk2:

<table>
<thead>
<tr>
<th>Action</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td>Initiator knows responder from beginning</td>
</tr>
<tr>
<td>→ e, es, s, ss</td>
<td>Responder learns initiator’s identity</td>
</tr>
<tr>
<td>← e, ee, se, psk</td>
<td></td>
</tr>
</tbody>
</table>
```

**Handshake** phase:

```c
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```c
// On Bob’s side
session *sn;
sn = create_responder(dv); // We don’t know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

**peer_id** parameter: varies with pattern and role
Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

**Initialization and premessages phase:**

```c
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
    charlie_public_key, alice_charlie_psk);
...
```

**Handshake phase:**

```c
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
...
```

```c
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);
...
```

**IKpsk2:**

 resembls initiators knowledge responder from beginning

```c
← s
...             Responder learns initiator's identity
→ e, es, $, ss
← e, ee, se, psk
```

**peer_id parameter: varies with pattern and role**

peer_id parameter: varies with pattern and role
What does the high-level API give us?

- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation
Key Storage and Validation

IKpsk2:

\[
\begin{align*}
&\leftarrow s \\
&\ldots \\
&\rightarrow e, es, s, ss \\
&\leftarrow e, ee, se, psk
\end{align*}
\]

XX:

\[
\begin{align*}
&\rightarrow e \\
&\leftarrow e, ee, s, es \\
&\rightarrow s, se
\end{align*}
\]

Wireguard VPN: all remote static keys must have been registered in the device before

WhatsApp: we actually transmit keys, which must be validated by some external mean

We parameterize our implementation with:

• **Policy** (bool): can we accept unknown remote keys? (Wireguard: false / WhatsApp: true)

• **Certification** function: certification_state \(\rightarrow\) public_key \(\rightarrow\) payload \(\rightarrow\) option peer_name

**Long-term keys storage** (on disk): serialization/deserialization functions for device static identity and peers (random nonces + device/peer name as authentication data).
What does the high-level API give us?

- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation
## Message Encapsulation – Security Levels

Every payload has an **authentication level** (≤ 2) and a **confidentiality level** (≤ 5):

<table>
<thead>
<tr>
<th>IKpsk2</th>
<th>Payload Conf. Level</th>
<th>XX</th>
<th>Payload Conf. Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td>→</td>
<td>→ e</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>← e, ee, s, es</td>
<td></td>
</tr>
<tr>
<td>→ e, es, s, ss</td>
<td>2</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>← e, ee, se, psk</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>→</td>
<td>5</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>←</td>
<td>-</td>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>

We protect the user from sending secret data/trusting received data **too early** (dynamic checks on **user-friendly auth./conf. levels**):

```c
encap_message_t *pack_with_conf_level(
    uint8_t requested_conf_level, // ←--- confidentiality
    const char *session_name,     const char *peer_name, uint32_t msg_len, uint8_t *msg);

bool unpack_message_with_auth_level(
    uint32_t *out_msg_len, uint8_t **out_msg, char *session_name, char *peer_name,
    uint8_t requested_auth_level, // ←--- authentication
    encap_message_t *msg);
```
Generated Code & Performance

- Security Levels
- PKI Assumptions
- Secrecy Labels + Invariants
- Symbolic Model

Low-Level Code:
- Session API
- Key Management
- Messaging Code
- Crypto Library

Formal Specification:
- Secure Channel API
- Abstract Device API
- Noise Protocol Spec
- Crypto Algorithms

Security Analysis:
- Security Levels
- PKI Assumptions
- Secrecy Labels + Invariants
- Symbolic Model
Some signatures:

```c
Noise_peer_t *Noise_device_add_peer(Noise_device_t *dvp, uint8_t *pinfo, uint8_t *rs, uint8_t *psk);
void Noise_device_remove_peer(Noise_device_t *dvp, uint8_t *pinfo, uint8_t *rs, uint8_t *psk);
Noise_peer_t *Noise_device_lookup_peer_by_id(Noise_device_t *dvp, uint32_t pid);
Noise_peer_t *Noise_device_lookup_peer_by_static(Noise_device_t *dvp, uint8_t *s);
Noise_session_t *Noise_session_create_initiator(Noise_device_t *dvp);
Noise_session_t *Noise_session_createResponder(Noise_device_t *dvp);
void Noise_session_free(Noise_session_t *sn);
Noise_rcode Noise_session_write(Noise_encap_message_t *payload, Noise_session_t *sn_p, uint32_t *out_len, uint8_t **out);
Noise_rcode Noise_session_read(Noise_encap_message_t **payload_out, Noise_session_t *sn_p, uint32_t inlen, uint8_t *input);
```
Performance

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Noise*</th>
<th>Custom</th>
<th>Cacophony</th>
<th>NoiseExpl.</th>
<th>Noise-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>6677</td>
<td>N/A</td>
<td>2272</td>
<td>4955</td>
<td>5603</td>
</tr>
<tr>
<td>NX</td>
<td>5385</td>
<td>N/A</td>
<td>2392</td>
<td>4046</td>
<td>5065</td>
</tr>
<tr>
<td>XX</td>
<td>3917</td>
<td>N/A</td>
<td>1593</td>
<td>3149</td>
<td>3577</td>
</tr>
<tr>
<td>IK</td>
<td>3143</td>
<td>N/A</td>
<td>1357</td>
<td>2459</td>
<td>2822</td>
</tr>
<tr>
<td>IKpsk2</td>
<td>3138</td>
<td>3756</td>
<td>1194</td>
<td>2431</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Performance Comparison**, in handshakes / second. Benchmark performed on a Dell XPS13 laptop (Intel Core i7-10510U) with Ubuntu 18.04
Security Analysis
Security Analysis – Dolev-Yao*

• Dolev-Yao* (abbreviated into DY*) is a symbolic analysis framework in F*

• Successfully used for the symbolic analysis of several protocols (ACME standard, part of MLS, Signal, ...)

DY*: Symbolic Bitstring Model

• DY* relies on a symbolic model of bitstrings

type bytes =
  | Constant: string -> bytes
  | Fresh: nat -> bytes
  | Concat: bytes -> bytes -> bytes
  | AEnc: bytes -> bytes -> bytes -> bytes
  | PK: bytes -> bytes
  | PEnc: bytes -> bytes -> bytes
  | VK: bytes -> bytes
  | Sig: bytes -> bytes -> bytes
DY*: Symbolic Model

- Bytes with different constructors are considered disjoint

```ocaml
let pke_enc pk m = PEnc pk m
let pke_dec sk c = match c with
  | PEnc p m -> if p = PK sk then Some m else None
  | _ -> None

let sign sk m = Sig sk m
let verify vk m sg = match sg with
  | Sig sk m' -> if vk = VK sk && m = m' then true else false
  | _ -> false
```
DY*: Global Protocol Trace

• The execution of a protocol is expressed as a trace of events

```haskell
type principal = string

type entry =
  | FreshGen: p: principal -> entry
  | Send: from: principal -> to: principal -> msg: bytes -> entry
  | Store: at: principal -> state: bytes -> entry
  | Event: p: principal -> ev: bytes -> entry
  | Compromise: p: principal -> entry

type trace = list entry
```
DY*: Executing Protocol Actions

• Each action extends the protocol trace (or uses it if it depends on past events)

```ocaml
let gen p : trace -> trace = fun tr -> FreshGen p :: tr

let recv p : trace -> option bytes = in recv_aux p
```
DY*: Executing Attacker Actions

let compromise p : trace -> trace = fun tr -> Compromise p :: tr

• Attacker can call `compromise p` to gain control of `p`
• Attacker can call `gen p` (for compromised `p`) to get fresh bytes
• Attacker can call `recv p` (to read any message)
• Attacker can call `retrieve p` (for compromised `p`) to read its state
• Attacker can call `send p1 p2 m` (for any message `m` it knows)
• Attacker **cannot** call `event` or `store`
DY*: Attacker Knowledge

```val attacker_knows: trace -> bytes -> prop```

- Attacker always knows Constant s
- Attacker learns msg from each Send from to msg in trace
- Attacker learns st from each Store p st (for compromised p)
- Attacker can call any crypto function with values it already knows (concat, split, pk_enc, pk_dec, sign, ...)

DY*: Reachable Traces

• Defines “well-formed” execution traces according to attacker capabilities

• Assume some protocol:
  
  ```
  val sendMsg1: principal -> principal -> trace -> trace
  val recvMsg1: principal -> trace -> trace
  ```

  ```
  let rec reachable (tr: trace) : prop =
    (exists p1 p2 tr’. tr == sendMsg1 p1 p2 tr’ ∧ reachable tr’) ∨
    (exists p tr’. tr == recvMsg1 p tr’ ∧ reachable tr’) ∨
    (match tr with
      | [] -> True
      | FreshGen p :: tr’ -> List.mem (Compromise p) tr’ ∧ reachable tr’
      | Send p1 p2 m :: tr’ -> attacker_knows tr’ m ∧ reachable tr’
      | Compromise p :: tr’ -> reachable tr’
      | _ -> False
  ```
DY*: Stating Confidentiality Goals

let protocol_sent p secret tr = List.mem (Event p (concat (literal "Send") secret)) tr

let compromised p tr = List.mem (Compromise p) tr

val confidentiality_lemma () : Lemma (forall tr p m. reachable tr \ protocol_sent p m tr \ attacker_knows tr m => compromised p tr)

- Case analysis on all reachable traces (by induction on length of trace)
- Reason about all possible interleavings of attacker and protocol actions
DY*: Stating Authentication Goals

let protocol_sent p1 p2 secret tr = ...
let protocol_received p1 p2 secret tr = ...

val authentication_lemma () : Lemma (forall tr p1 p2 m.
  reachable tr \& protocol_received p1 p2 m tr =>
  protocol_sent p1 p2 m tr V compromised p1 tr
)

- **Correspondance Assertion**: Received p1 p2 m \implies Sent p1 p2 m
- Again, proved for all possible interleavings
Instead of proving each property by induction on traces, DY* relies on security labels

Labels for the data-types:
- CanRead $[P \ "Alice"]$ : static data that can only be read by principal "Alice"
- CanRead $[S \ "Bob\"\text{ sid}]$ : 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 1 : private key of label 1
- dh_public_key 1 : public key associated to a private key of label 1

```plaintext
// 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
```
Security Analysis - Example

```
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
```

Alice  Bob
→ e, es, s, ss, [d]

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es

l_ss := ((CanRead [P "Alice"]) ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss

ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)

val aead_encrypt
  (#1 : label)
  (sk : aead_key l) // encryption key
  (iv : msg public) // nonce
  (plain : msg l) // plaintext
  (ad : msg public) : // authentication data
  msg public
```

We can then send the encrypted message: register a Send event in a global trace
Security Analysis: \texttt{can\_flow}

- Labels are purely \texttt{syntactic}.
- \textbf{Semantics} of \texttt{DY*} are given through a \texttt{can\_flow} predicate which states properties about a global trace of events.
- The content of a message sent over the network is \texttt{compromised} if its label flows to \texttt{public}.

- Labels can flow to more secret labels (i is a timestamp):

\[
can\_flow\ i (\text{CanRead}[P\ p1])\ (\text{CanRead}[P\ p1] \sqcap \text{CanRead}[P\ p2])
\]

- The attacker can \texttt{dynamically compromise} a participant's current state: event \texttt{Compromise p} ...
- A label is compromised (and data with this label) if it flows to \texttt{public}:

\[
\text{compromised\_before\ i\ (P\ p)}\implies\text{can\_flow\ i\ (CanRead}[P\ p])\ \text{public}
\]

\[
\text{compromised\_before\ i\ (S\ p\ sid)}\implies\text{can\_flow\ i\ (CanRead}[S\ p\ sid])\ \text{public}
\]

- If a label flows to \texttt{public} we can deduce the existence of compromise events:

\[
can\_flow\ i\ (\text{CanRead}[P\ p])\ \text{public}\implies\text{compromised\_before\ i\ (P\ p)}
\]
We do the security analysis once and for all.

We formalize the Noise security levels with predicates, and prove that those predicates are satisfied at the proper steps of the proper handshakes:

<table>
<thead>
<tr>
<th>Level</th>
<th>Confidentiality Predicate (over i, idx, and l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\top$</td>
</tr>
<tr>
<td>1</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid] \cup idx.peer_eph_label\ l)}$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ P\ idx.peer]\ l)}$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ P\ idx.peer]\ l)} \wedge$</td>
</tr>
<tr>
<td></td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ idx.peer_eph_label]\ l)}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ P\ idx.peer]\ l)} \wedge$</td>
</tr>
<tr>
<td></td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid] \cup idx.peer_eph_label\ l)}$</td>
</tr>
<tr>
<td></td>
<td>$(\text{compromised_before_i\ (P\ idx.p) \lor \text{compromised_before_i\ (P\ idx.peer) \lor}$</td>
</tr>
<tr>
<td></td>
<td>$(\exists\text{sid}.\ peer_eph_label == \text{CanRead\ [S\ idx.peer_sid']})$)</td>
</tr>
<tr>
<td>5</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ P\ idx.peer]\ l)} \wedge$</td>
</tr>
<tr>
<td></td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid] \cup idx.peer_eph_label\ l)}$</td>
</tr>
<tr>
<td></td>
<td>$(\text{compromised_before_i\ (S\ idx.p\ idx.sid) \lor \text{compromised_before_i\ (P\ idx.peer) \lor}$</td>
</tr>
<tr>
<td></td>
<td>$(\exists\text{sid'}.\ peer_eph_label == \text{CanRead\ [S\ idx.peer_sid']})$)</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>$\top$</td>
</tr>
<tr>
<td>1</td>
<td>$\text{can_flow_i\ (CanRead\ [P\ idx.p;\ P\ idx.peer]\ l)}$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{can_flow_i\ (CanRead\ [S\ idx.p\ idx.sid;\ P\ idx.peer]\ l)}$</td>
</tr>
</tbody>
</table>

Strong forward-secrecy
Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**), from the sender’s perspective:

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

Handshake secrets are only readable by the peer and the current session `sid` at `p`.

Handshake secrets are also bound to some peer ephemeral key `re`.

Unless the peer’s long-term keys and the specific session `S p sid` were compromised before the session is complete, the peer ephemeral key must have label to `S peer sid’`.

**Certification** of remote static key gives:

```plaintext
\[
\text{get\_dh\_label\ rs} = \text{CanRead\ [P\ peer]}
\]
```
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:

   ```
   // 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:

   IKpks2 (from the responder’s point of view)

   ```
   ← s
   ...
   → e, es, s, ss, [d]
   l1 = (peer_eph_label ⊔ CanRead [P p]) \ (…)
   ← e, ee, se, psk, [d]
   l2 = (peer_eph_label ⊔ CanRead [P p]) \ (…) \ (peer_eph_label ⊔ CanRead [S p sid]) \ (…)
   ...
   ```

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

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

   ```
   Level | Confidentiality Predicate (over i, idx, and l)
   0     | T
   1     | can_flow i (CanRead [S idx.p idx.sid] \ idx.peer_eph_label) l
   2     | can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
   3     | can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l \ (compromised_before i (P idx.p) \ compromised_before i (P idx.peer) \ (3sid’_peer_eph_label == CanRead [S idx.peer sid’]))
   4     | can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l \ (compromised_before i (S idx.p idx.sid) \ compromised_before i (P idx.peer) \ (3sid’_peer_eph_label == CanRead [S idx.peer sid’]))
   ```

5. We prove that those **security predicates are satisfied** by the target labels.

---

**Security Analysis - Summary**

---

124

**Strong forward-secrecy**
Main Takeaways

• **Do not roll your own crypto**
  • Implementing cryptography is error-prone, and mistakes can have disastrous consequences
• But if you do, formally verify it
  • Successful verification tools exist for both C and Assembly
  • Verification can also help with code maintenance, and extending to new architectures/variants at a lower cost
• Many tools and techniques allow to reason about the security of protocol models
• End-to-end verification is still tricky, however several recent projects offer promising solutions