INRIA – Séminaire Gallium
3 octobre 2019

EverpArse

verified zero-copy parsing and serialization for data-exchange formats

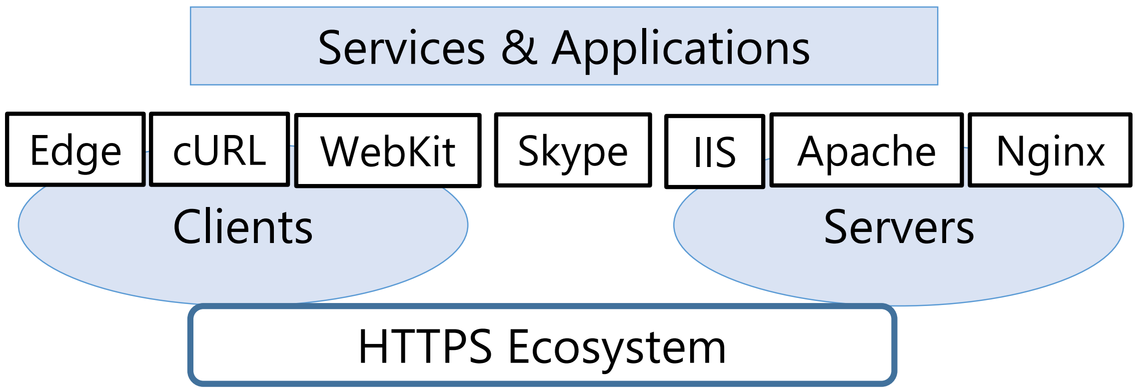
Tahina Ramananandro (RiSE @MSR Redmond)
Antoine Delignat-Lavaud, Cédric Fournet (MSR Cambridge)
Nikhil Swamy, Jonathan Protzenko (RiSE @MSR Redmond)
Nadim Kobeissi (NYC Paris, Symbolic Software), Tej Chajed (MIT)

Microsoft

RiSE

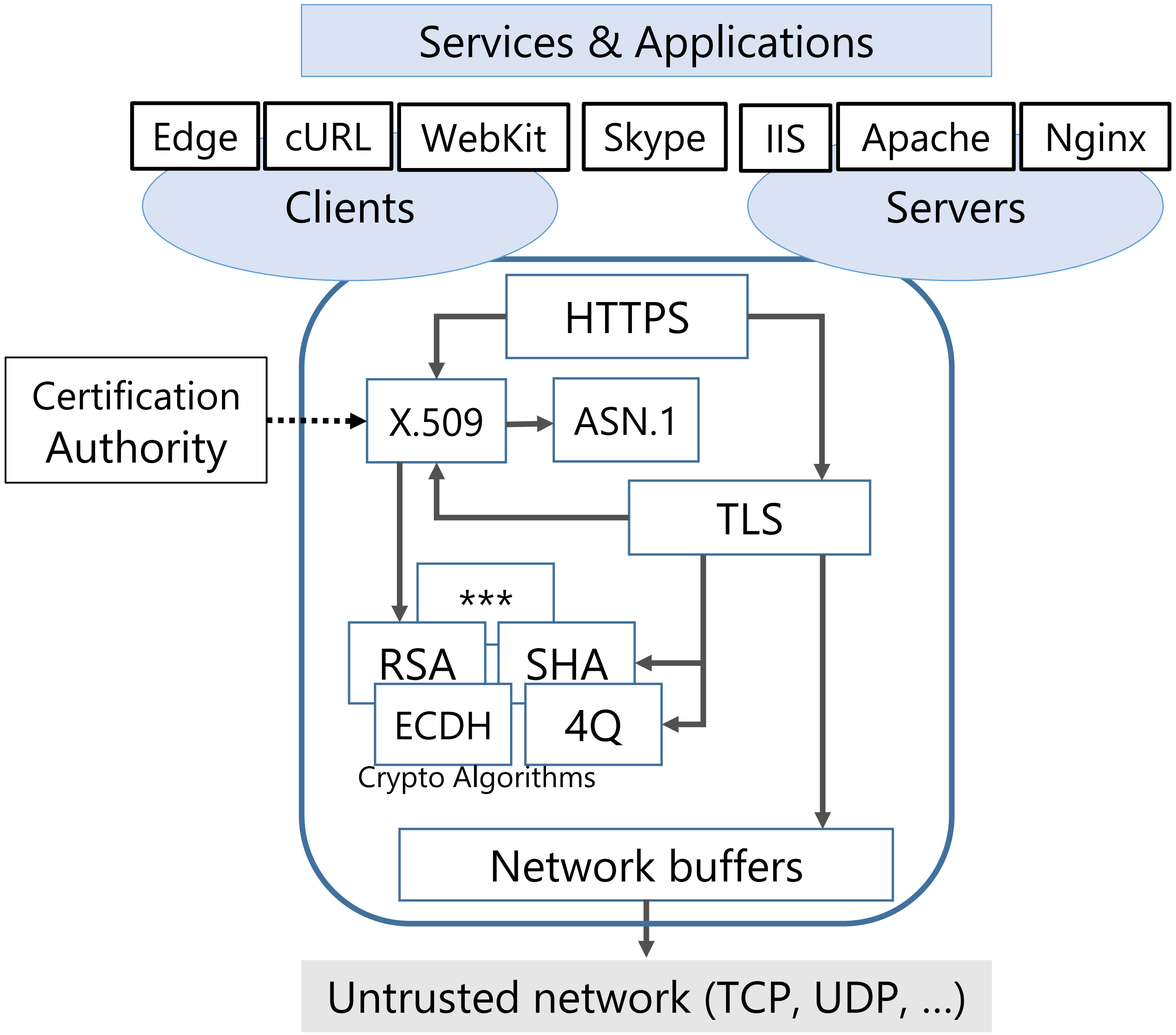
everest
Verified End-to-End Secure Transport

The HTTPS Ecosystem is critical



- Most widely deployed security?
  ½ Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, …

The HTTPS Ecosystem is complex



The HTTPS Ecosystem is broken

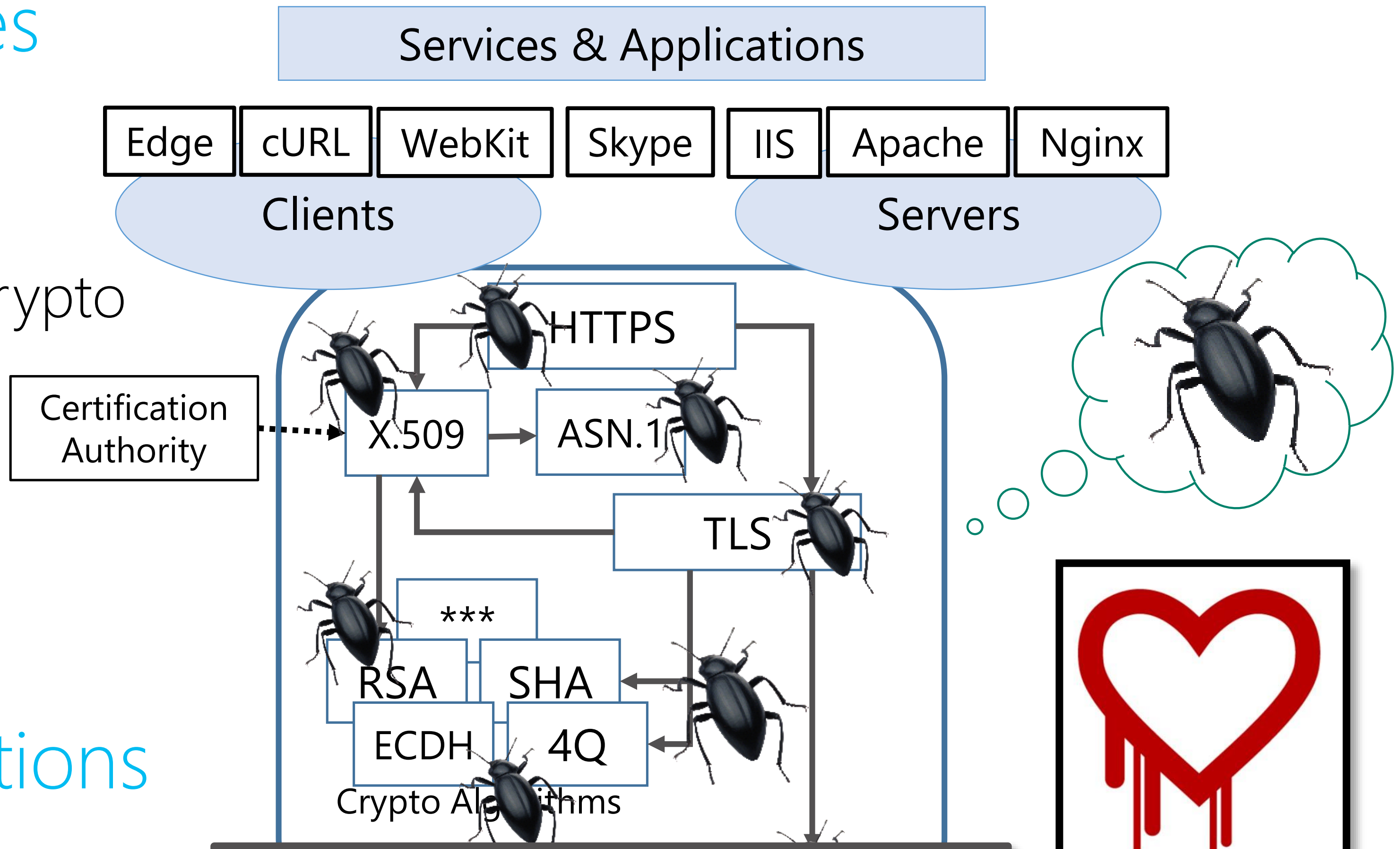
- 20 years of attacks & fixes

  Buffer overflows
  Incorrect state machines
  Lax certificate parsing
  Weak or poorly implemented crypto
  Side channels

  Informal security goals
  Dangerous APIs
  Flawed standards

- Mainstream implementations

  OpenSSL, SChannel, NSS, …
  Still patched every month!

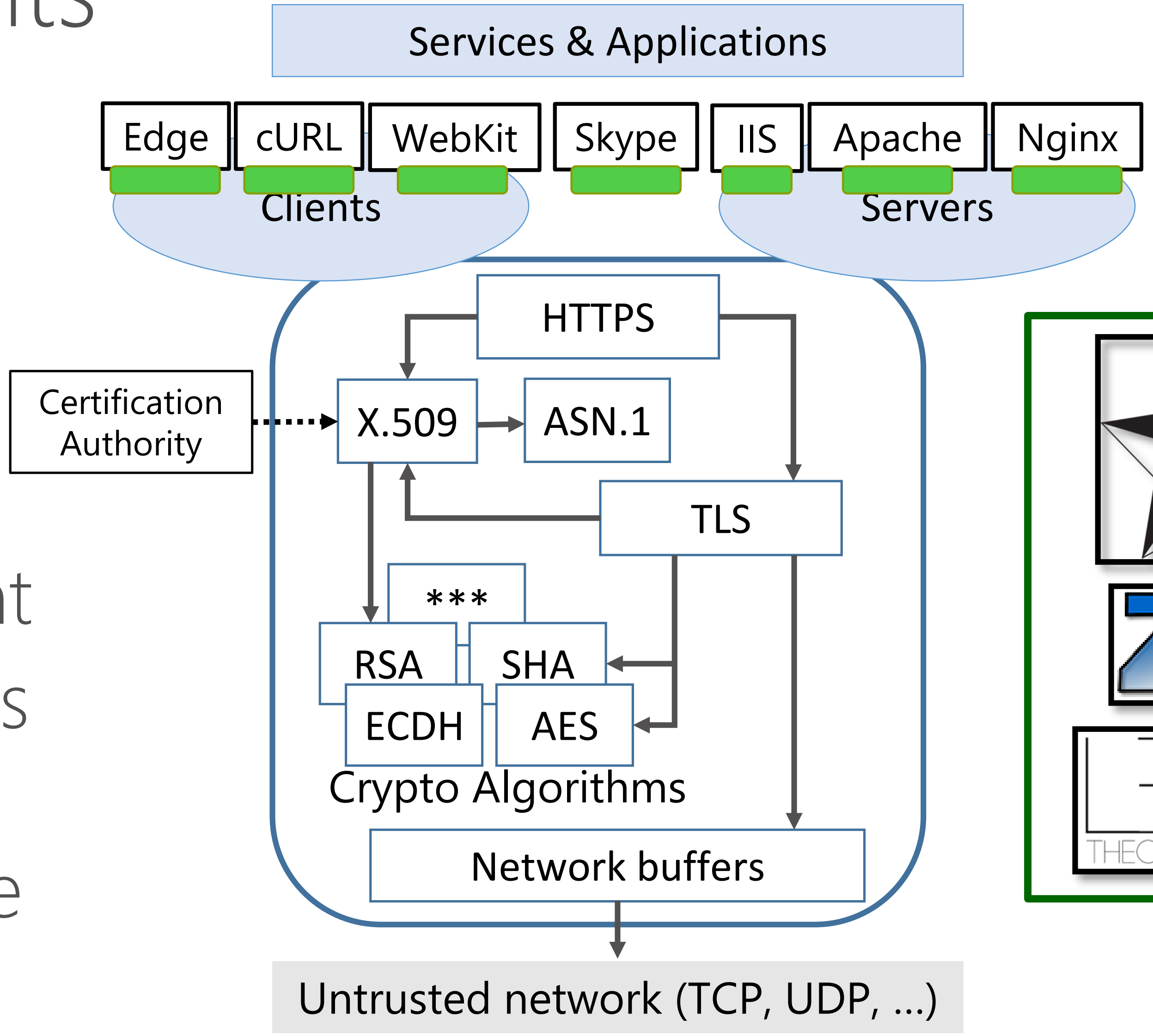


The Washington Post

The Switch

'FREAK' flaw undermines security for Apple and Google users, researchers discover

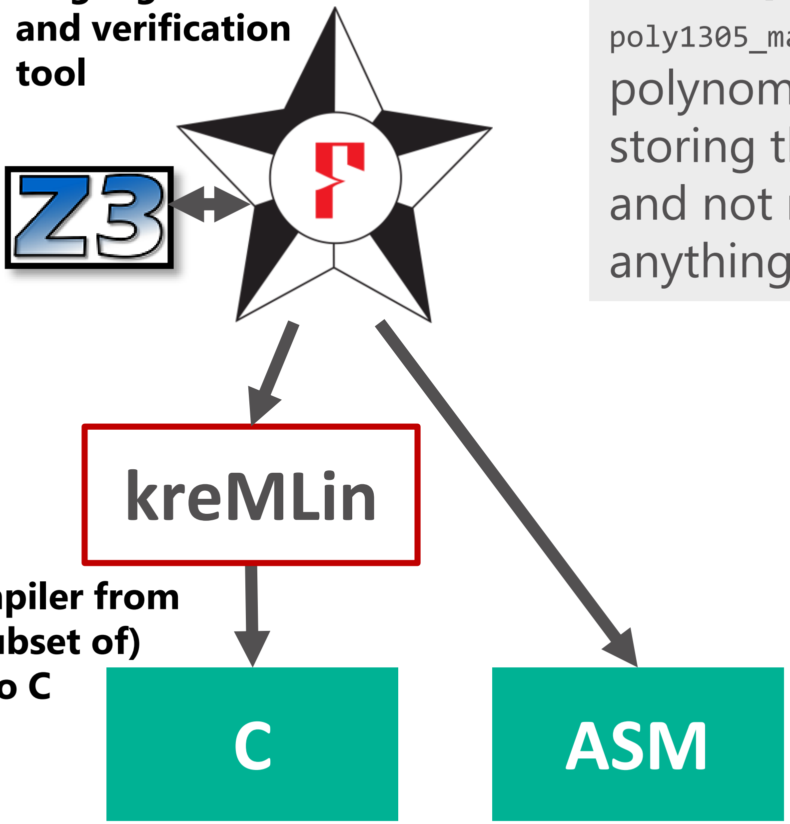
Everest 2016—2021: Verified Components for the HTTPS Ecosystem



- Strong verified security
- Widespread deployment
- Trustworthy, usable tools
- Growing expertise in high-assurance software development

Verification Tools and Methodology

F*: A general purpose programming language and verification tool



Compiler from (a subset of) F* to C

Math spec in F*

`poly1305_mac` computes a polynomial in GF(2^{130}-5), storing the result in `tag`, and not modifying anything else

```
val poly1305_mac: tag:nbytes 16 →
                  len:u32 →
                  msg:nbytes len{disjoint tag msg} →
                  key:nbytes 32 {disjoint msg key ∧ disjoint tag key} →
                    ST unit
    (requires (λ h → msg ∈ h ∧ key ∈ h ∧ tag ∈ h))
    (ensures (λ h0 _ h1 →
        let r=Spec.clamp h0.[sub key 0 16] in
        let s=h0.[sub key 16 16] in
        modifies {tag} h0 h1 ∧
        h1.[tag] == Spec.mac_1305 (encode_bytes h0.[msg]) r s))
```

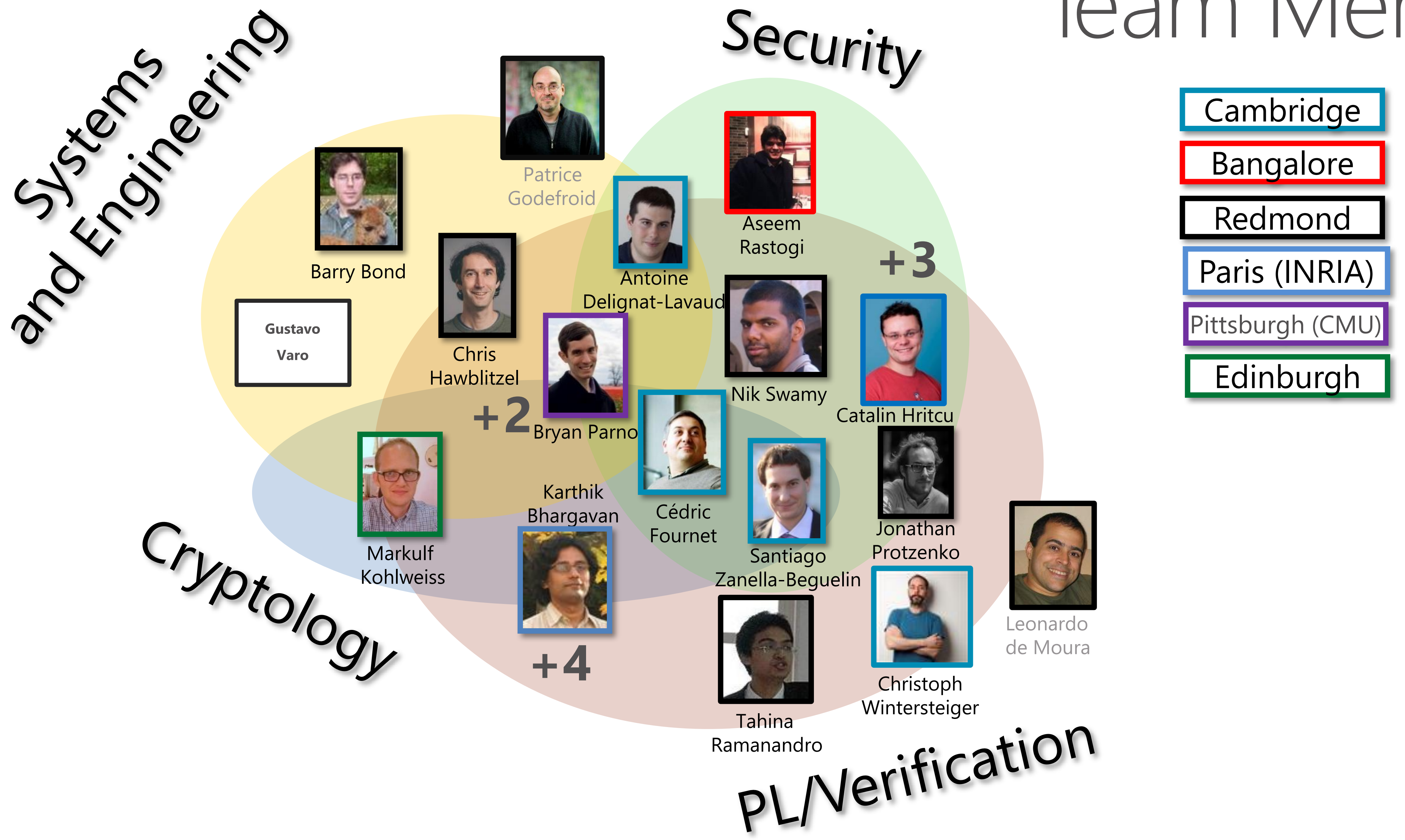
Efficient ASM implementation

Verification imposes no runtime performance overhead

```
procedure{:quick}{:public}{:exportSpecs} Poly1305(
    ghost ctx_b:buffer64,
    ghost inp_b:buffer64,
    ghost len_in:nat64,
    ghost finish_in:nat64)
    lets
        ctx @= rdi; inp @= rsi; len @= rdx; finish @= rcx;
        h0 @= r14; h1 @= rbx; h2 @= rbp;
        ctx_in := (if win then rcx else ctx);
        inp_in := (if win then rdx else inp);
        n := 0x1_0000_0000_0000_0000;
        p := n * n * 4 - 5;
    modifies
        rax; rbx; rcx; rdx; rsi; rdi; rbp; rsp; r8; r9; r10; r11; r12; r13; r14; r15;
        efl; mem;
    ensures
        finish_in == 0 ==>
          modp(h210) == poly1305_hash_blocks
```

```
void
poly1305_mac(uint8_t *tag, uint32_t len, uint8_t *msg, uint8_t
*key)
{
    uint64_t tmp [10] = { 0 };
    uint64_t *acc = tmp
    uint64_t *r = tmp + (uint32_t)5;
    uint8_t s[16] = { 0 };
    Crypto_Symmetric_Poly1305_poly1305_init(r, s, key);
    Crypto_Symmetric_Poly1305_poly1305_process(msg, len, acc,
r);
    Crypto_Symmetric_Poly1305_poly1305_finish(tag, acc, s);
}
```

Team Members



Secure components: Where we are today

Close to production quality

- HACL*: High-assurance Crypto Library
  - Verified implementations of crypto algorithms
  - Performance comparable to hand-written C
  - Functionally correct, cryptographically secure, side-channel resistant
  - in Firefox Quantum, Tezos Blockchain, mbedTLS, etc.
- VALE: Verified Assembly for Everest
  - Assembly level crypto, with performance comparable to OpenSSL assembly
- TLS Record Layer Protection:
  - Verified, efficient code compiled to C
  - Functionally correct, cryptographically secure, reasonably fast [used in Windows prototype]
- EverCrypt = HACL* + Vale
  - a complete cryptographic library with agile APIs, CPU auto-detection, C/ASM automatic switches, and re-written algorithms for maximum performance and deep integration
- EverParse
  - verified, efficient parsing and serialization for binary formats, including TLS handshake message format

Secure components: Where we are today

Close to production quality

- HACL*: High-assurance Crypto Library
  - Verified implementations of crypto algorithms
  - Performance comparable to hand-written C
  - Functionally correct, cryptographically secure, side-channel resistant
  - in Firefox Quantum, Tezos Blockchain, mbedTLS, etc.
- VALE: Verified Assembly for Everest
  - Assembly level crypto, with performance comparable to OpenSSL assembly
- TLS Record Layer Protection:
  - Verified, efficient code compiled to C
  - Functionally correct, cryptographically secure, reasonably fast [used in Windows prototype]

Talk by Jonathan Protzenko, October 14th, 10:30am

- EverCrypt = HACL* + Vale
  - a complete cryptographic library with agile APIs, CPU auto-detection, C/ASM automatic switches, and re-written algorithms for maximum performance and deep integration
- EverParse
  - verified, efficient parsing and serialization for binary formats, including TLS handshake message format

Secure components: Where we are today

Close to production quality

- HACL*: High-assurance Crypto Library
  - Verified implementations of crypto algorithms
  - Performance comparable to hand-written C
  - Functionally correct, cryptographically secure, side-channel resistant
  - in Firefox Quantum, Tezos Blockchain, mbedTLS, etc.
- VALE: Verified Assembly for Everest
  - Assembly level crypto, with performance comparable to OpenSSL assembly
- TLS Record Layer Protection:
  - Verified, efficient code compiled to C
  - Functionally correct, cryptographically secure, reasonably fast [used in Windows prototype]
- EverCrypt = HACL* + Vale
  - a complete cryptographic library with agile APIs, CPU auto-detection, C/ASM automatic switches, and re-written algorithms for maximum performance and deep integration

This talk

- EverParse
  - verified, efficient parsing and serialization for binary formats, including TLS handshake message format

Secure components: Where we are today

Research prototypes

- TLS 1.3 full protocol
  - 1 year of interop with
    other early implementations
  - Deployments within existing
    clients (IE, Curl) and servers (nginx)
  - Partial verification (in progress)
- QUIC library
  - A TLS handshake library suitable for use from QUIC
    -- used within **Windows**

Everest in Action, so far

Some production deployments of Everest Verified Cryptography

WinQUIC: Delivered Everest TLS 1.3 and crypto stack to Windows Networking, in the latest Windows

Mozilla NSS runs Everest verified crypto for several core algorithms

Everest verified crypto in the Linux kernel (soon) via WireGuard secure VPN

Goal: High-assurance security components

Showing why they are trustworthy, not just asking for trust.

Method: Computer-aided verification

We develop clean-slate, modular, verification-oriented models, specifications, and implementations.

We co-develop compilers & provers to automatically check that implementations meets their (simpler) functional and security specifications.

What do we verify?

Safety

Memory- and type-safety. Mitigates buffer overruns, dangling pointers, code injections.

Functional correctness

Our fast implementations behave precisely as our simpler specifications.

Secrecy

Access to secrets, including crypto keys and private app data is restricted according to design.

Cryptographic security

We bound the probability that an attacker may break any secrecy or integrity properties

Our specifications and implementations are written together, in one language (F*)
Drift between spec and implementation cannot happen.

Will Everest be perfectly secure? No.

Our models make assumptions, e.g.

- The private signing key must remain private and not used in other protocols
- We assume security for core crypto algorithms, based on hard problems.

Our models may not be complete

- Our detailed models are designed to exclude all known attacks, but may be blind to new classes of attack (hardware faults,…)

Our verification toolchain may be buggy

- Our TCB includes Z3, Kremlin, C compilers… Efforts to reduce it are under way.

Computer-aided verification also has advantages: once in place, proof verification is

- automated (but takes hours)
- compositional (we can re-use verified component as building blocks for others)
- maintainable (we can extend or modify our code, and re-check everything as part of CI).

Cryptographic protocol: TLS

Internet Standard

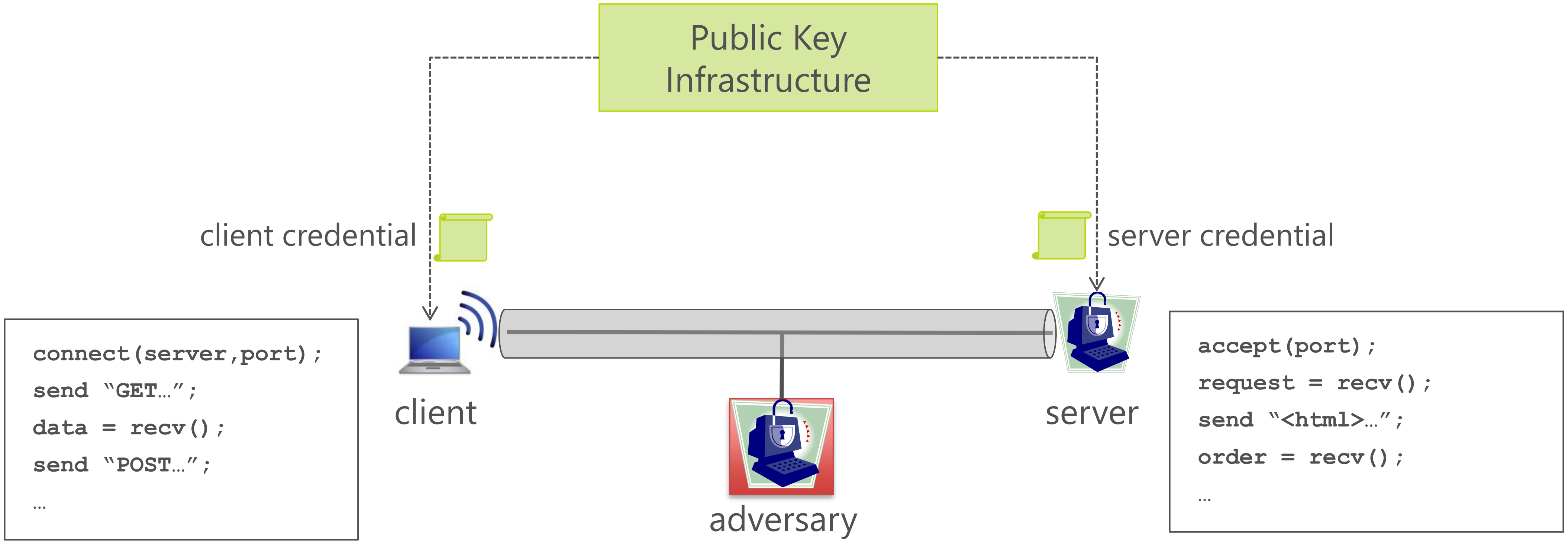
1994	Netscape's Secure Sockets Layer
1995	SSL3
1999	TLS 1.0 (≈SSL3)
2006	TLS 1.1
2008	TLS 1.2
2018?	TLS 1.3

Implementations:

OpenSSL SChannel NSS SecureTransport PolarSSL JSSE GnuTLS miTLS

- Large C++ codebase (400K LOC), many forks https://github.com/openssl/openssl
- Optimized cryptography for 50 platforms
- Terrible API
- Frequent critical patches https://openssl.org/news/vulnerabilities.html
- **Never secure so far**

TLS Verification Goal: Secure Channel

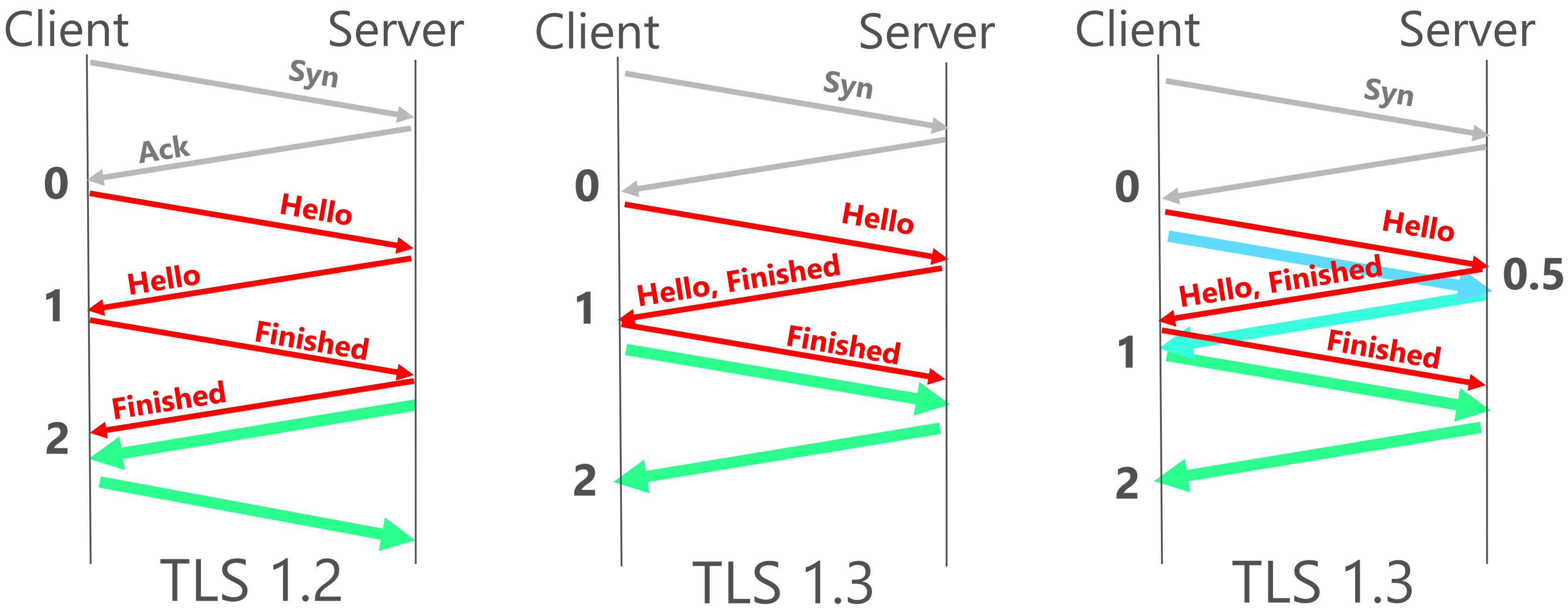


Top-level verification theorem: As long as the adversary does not control the long-term credentials of the client and server, it cannot

- Inject forged data into the stream (authenticity)
- Distinguish the data stream from random bytes (confidentiality)

Save roundtrips to lower latency

TLS over TCP



Two roundtrips before sending application data

One roundtrip before sending application data

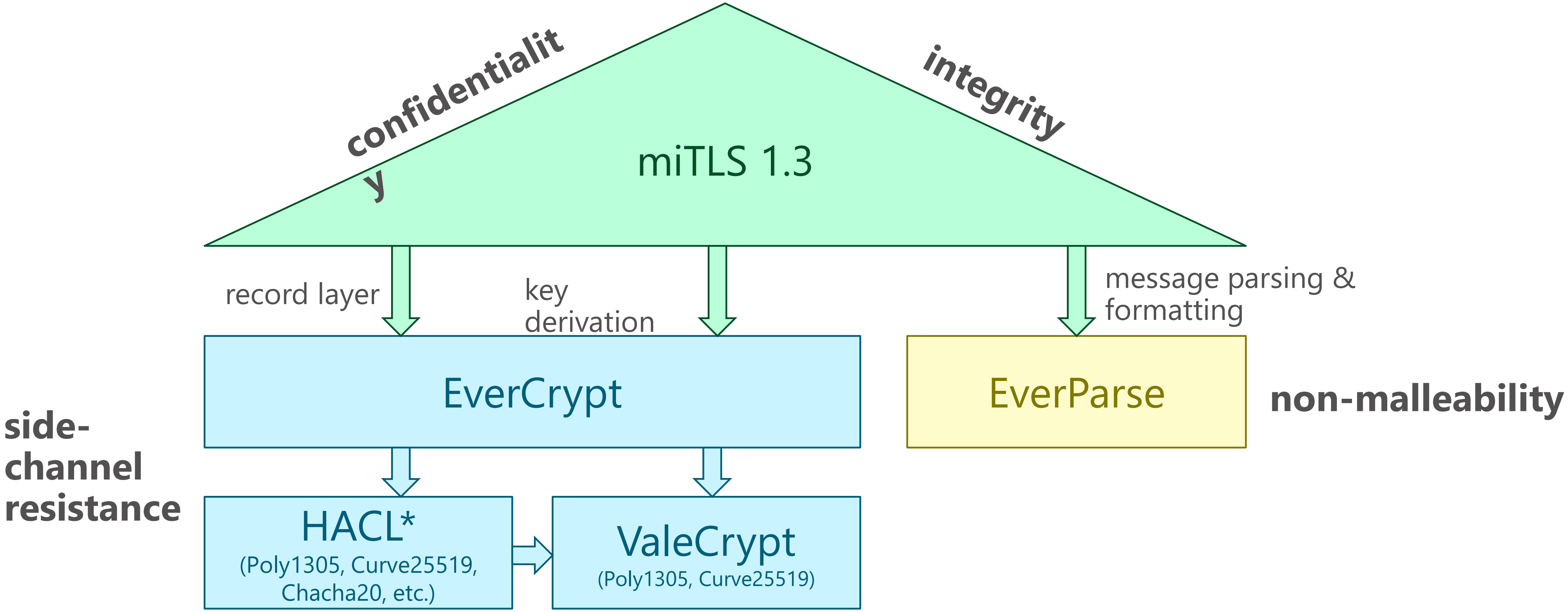
Zero roundtrip before sending application data

Latency matters

Amazon: 100ms ~ 1% rev.

Bing: 3.5ms ~ 1 dev / yr

An overview of the Everest verification scaffolding



+ memory safety, functional correctness

Is parser research done?

- “So 1980s?”
- No: parser bugs still in the news
  - “Cloudbleed” (2017)

“The leakage was the result of a bug in an HTML parser chain Cloudflare uses to modify webpages as they pass through the service's edge servers. [...]. When the parser was used in combination with three Cloudflare features [...] it caused Cloudflare edge servers to leak pseudo random memory contents into certain HTTP responses.”


ars TECHNICA
BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STO

BIZ & IT —

Serious Cloudflare bug exposed a *potpourri* of secret customer data

Service used by 5.5 million websites may have leaked passwords and authentication tokens.

DAN GOODIN - 2/23/2017, 5:35 PM

Goals

- Generate verified C parsers and serializers for data exchange formats
  - Everest miTLS, Bitcoin, ASN.1 PKCS1 signatures, etc.
- Zero-copy C implementations
  - Parsing: read values directly from input buffer, no copies
  - Serialization: minimize and track allocation of intermediate objects, no implicit GC
- Properties
  - Memory safety
  - Serialization and parsing are inverse of each other
  - Injectivity of parsing for message authentication (to avoid malleability attacks)
- Current status:
  - Paper accepted at USENIX Security 2019
  - miTLS handshake message types all covered, integration ongoing

Injective parsers for authenticated messages

- Cryptography (hashing, etc.) only authenticates bytes
- Example: Bitcoin
  - Up to 5B$ (2018 value) stolen from MtGox in 2014
  - parse(m) = t, hash(m) = TXID(t)
  - Forge a message m’ with the same TXID and replace it in the blockchain
- Other known attacks
  - PKCS1 certificate forgery (Bleichenbacher etc.)
  - ASN.1 signature encoding: BIP66 (2015)
  - ECDSA complement: BIP62

BIP: Bitcoin Improvement Proposal

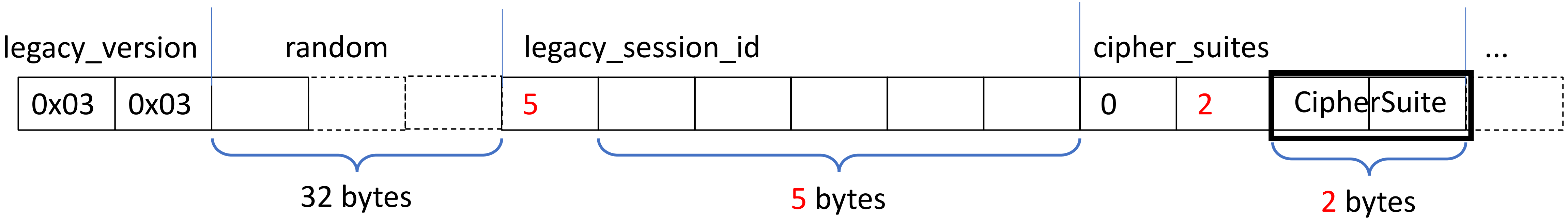
Malleability attack: PKCS#1 signature forgery (Bleichenbacher 1998, 2006)

- RSA message signing in principle
  - Public key (N large, e=3), unknown private key *d*
  - Hash the message, sign the hash (because usually N << |message|)
  - Signing: compute (h^d), so that (h^d)^e = h mod N

- Actual format of the (algo & hash) to be signed:
  - 00 01 FF FF … FF <hash algorithm OID> <hash>, **total length N**
  - Buggy implementations of signature verifiers do not check total length, just remove 00 01 FF FF … FF
  - Forgery: H = 00 01 FF <hash algorithm OID> <hash> <garbage>
  - Adjust garbage and number of FFs to compute S = an *approximate* cubic root of H so that H’ = S^3 = 00 01 FF <hash algorithm OID> <hash> <*garbage’*> works

TLS Parsing

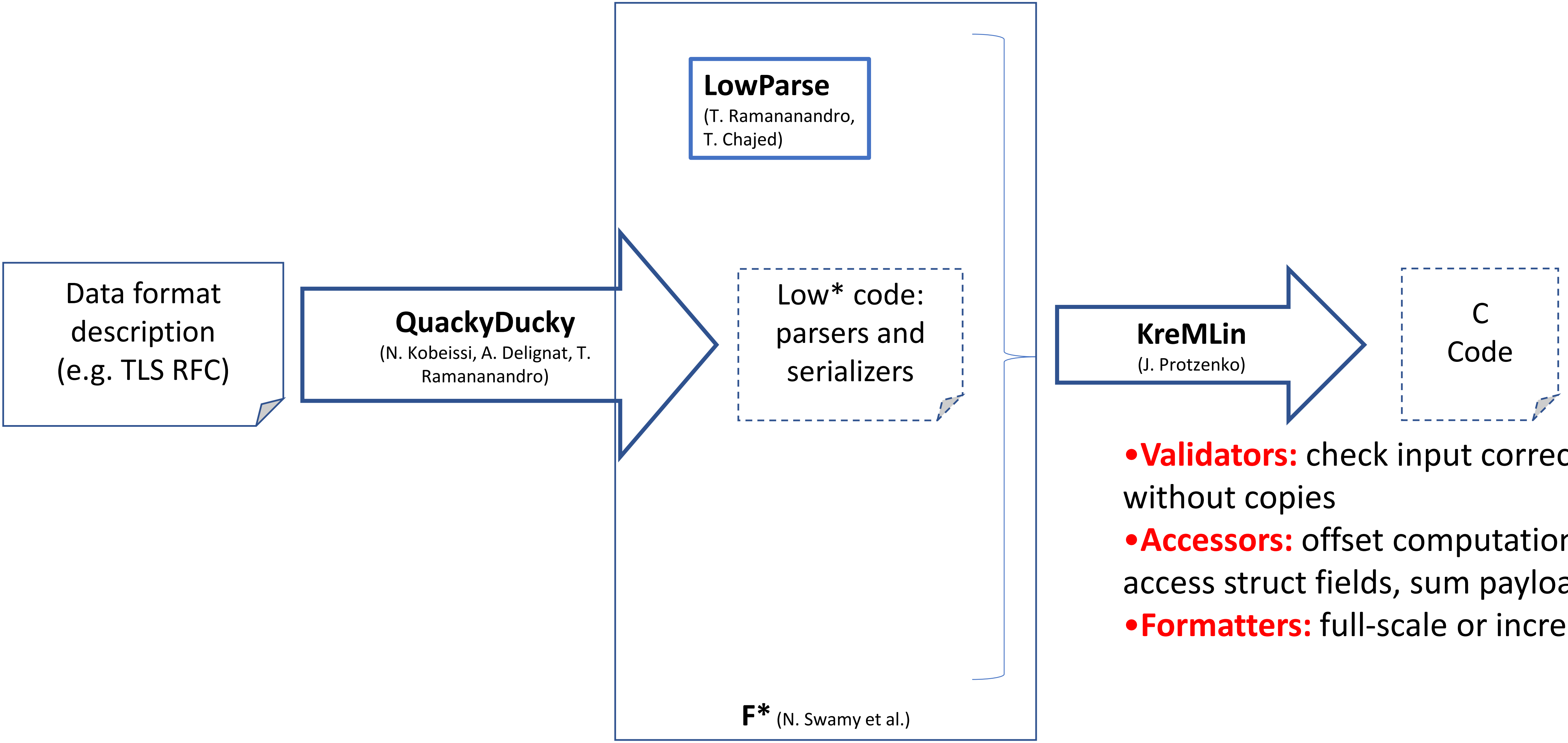
https://tools.ietf.org/html/rfc8446




```
uint16 ProtocolVersion; opaque Random[32]; uint8 CipherSuite[2];

struct {
    ProtocolVersion legacy_version = 0x0303;
    Random          random;
    opaque          legacy_session_id<0..32>;
    CipherSuite     cipher_suites<2..2^16-2>;
    opaque          legacy_compression_methods<1..2^8-1>;
    Extension       extensions<8..2^16-1>;
} ClientHello;
```

EverParse: QuackyDucky and LowParse



- **Validators:** check input correctness without copies
- **Accessors:** offset computations to access struct fields, sum payloads, etc.
- **Formatters:** full-scale or incremental

Methodology

- Specify parser and serializer **combinators** in pure F*
  - Prove injectivity and inverse properties
- Implement **combinators** in pure/effectful Low*
  - Prove functional correctness wrt. pure parser specification
  - Prove memory safety
- *Generate parser/serializer specifications and pure/effectful Low* implementations for TLS* ***using those combinators***
  - No manual proof required: correctness by virtue of typing
- Extract generated Low* to C using KreMLin
  - Inlining, loops instead of recursion, and F* tactics

- 2 person-year library and tool effort, mostly in proof engineering and modularization
- Specification adequacy by interop testing

What we generate

- High-level F* data type
- F* Parser and serializer specifications
- Functional parser and serializer implementations (with copies and conservative GC)
- Low*/C Low-level implementations:
  - Validators: check input correctness without copies
  - Accessors: offset computations to access struct fields, sum payloads, etc.
  - Validity lemmas/finalizers: derive struct validity from fields validity, etc.

Example: TLS ClientHello

```
type clientHello = {
  version : protocolVersion;
  random : random;
  session_id : sessionID;
  cipher_suites : clientHello_cipher_suites;
  compression_method : clientHello_compression_method;
  extensions : clientHelloExtensions;
}
```

```
struct {
    ProtocolVersion legacy_version = 0x0303;
    Random          random;
    opaque          legacy_session_id<0..32>;
    CipherSuite     cipher_suites<2..2^16-2>;
    opaque          legacy_compression_methods<1..2^8-1>;
    Extension       extensions<8..2^16-1>;
} ClientHello;
```

```
inline_for_extraction noextract let clientHello_parser_kind = LP.strong_parser_kind 43 131396 None

noextract val clientHello_parser: LP.parser clientHello_parser_kind clientHello

noextract val clientHello_serializer: LP.serializer clientHello_parser

val clientHello_validator: LL.validator clientHello_parser

inline_for_extraction val accessor_clientHello_session_id : LL.accessor clientHello_parser
sessionID_parser clens_clientHello_session_id

val clientHello_valid (h:HS.mem) (input:LL.slice) (pos0:U32.t) : Lemma
  (requires
    LL.valid protocolVersion_parser h input pos0 ∧ (
    let pos1 = LL.get_valid_pos protocolVersion_parser h input pos0 in
    LL.valid random_parser h input pos1 ∧ (
    let pos2 = LL.get_valid_pos random_parser h input pos1 in
    LL.valid sessionID_parser h input pos2 ∧ (
```

Parser and Serializer Specifications

```
type parser (t: Type) = (p: bytes → option (t × ℕ) {
  ∀ b1 b2: bytes .
  (Some? (p b1) ∧
  Some? (p b2) ∧ (
    let (Some (v1, len1)) = p b1 in
    let (Some (v2, len2)) = p b2 in
    v1 == v2
  )) ⟹  (
    let (Some (v1, len1)) = p b1 in
    let (Some (v2, len2)) = p b2 in
    (len1 <: ℕ) == (len2 <: ℕ) ∧
    Seq.slice b1 0 len1 == Seq.slice b2 0 len2
  )})

type serializer (#t: Type) (p: parser t)  = (f: t → bytes {
  ∀ (x: t) . p (f x) == Some (x, Seq.length (f x))
})
```

And more:

- Consumption bounds
- Strong prefix property
- Etc.

Controlled by metadata

Generated F* Parser specification

```
let clientHello'_parser : LP.parser clientHello' =
  protocolVersion_parser
  `LP.nondep_then` random_parser
  `LP.nondep_then` sessionID_parser
  `LP.nondep_then` clientHello_cipher_suites_parser
  `LP.nondep_then` clientHello_compression_method_parser
  `LP.nondep_then` clientHelloExtensions_parser

inline_for_extraction let synth_clientHello (x: clientHello') : clientHello =
  let (((((version, random), session_id), cipher_suites), compression_method), extensions) = x in
  {
    version = version;
    random = random;
    session_id = session_id;
    cipher_suites = cipher_suites;
    compression_method = compression_method;
    extensions = extensions;
  }

let clientHello_parser : LP.parser clientHello =
  clientHello'_parser `LP.parse_synth` synth_clientHello
```

Validators

```
type validator (#t: Type) (p: parser t) =
  (sl: slice) →
  (pos: U32.t) →
  HST.Stack U32.t
  (requires (λ h → live_slice h sl ∧ U32.v pos ≤ U32.v sl.len ∧ U32.v
sl.len ≤ U32.v validator_max_length))
  (ensures (λ h res h' →
    B.modifies B.loc_none h h' ∧ (
    if U32.v res ≤ U32.v validator_max_length
    then
      valid_pos p h sl pos res
    else
      (¬ (valid p h sl pos))
  )))
```

Generated Low* Validator implementation

```
inline_for_extraction let clientHello'_validator : LL.validator clientHello'_parser =
  protocolVersion_validator
  `LL.validate_nondep_then` random_validator
  `LL.validate_nondep_then` sessionID_validator
  `LL.validate_nondep_then` clientHello_cipher_suites_validator
  `LL.validate_nondep_then` clientHello_compression_method_validator
  `LL.validate_nondep_then` clientHelloExtensions_validator

let clientHello_validator : LL.validator clientHello_parser  =
  LL.validate_synth clientHello'_validator synth_clientHello ()
```

Low* code for validate_nondep_then

```
inline_for_extraction
let validate_nondep_then
  (#k1: parser_kind)
  (#t1: Type0)
  (#p1: parser k1 t1)
  (p1' : validator p1)
  (#k2: parser_kind)
  (#t2: Type0)
  (#p2: parser k2 t2)
  (p2' : validator p2)
: Tot (validator (nondep_then p1 p2))
= fun (input: slice) (pos: U32.t) ->
  let h = HST.get () in
  [@inline_let] let _ = valid_nondep_then h p1 p2 input pos in
  let pos1 = p1' input pos in
  if pos1 `U32.gt` validator_max_length
  then begin
    pos1
  end
  else
    [@inline_let] let _ = valid_facts p2 h input pos1 in
    p2' input pos1
```

High-level verification for low-level code

For code, the programmer:

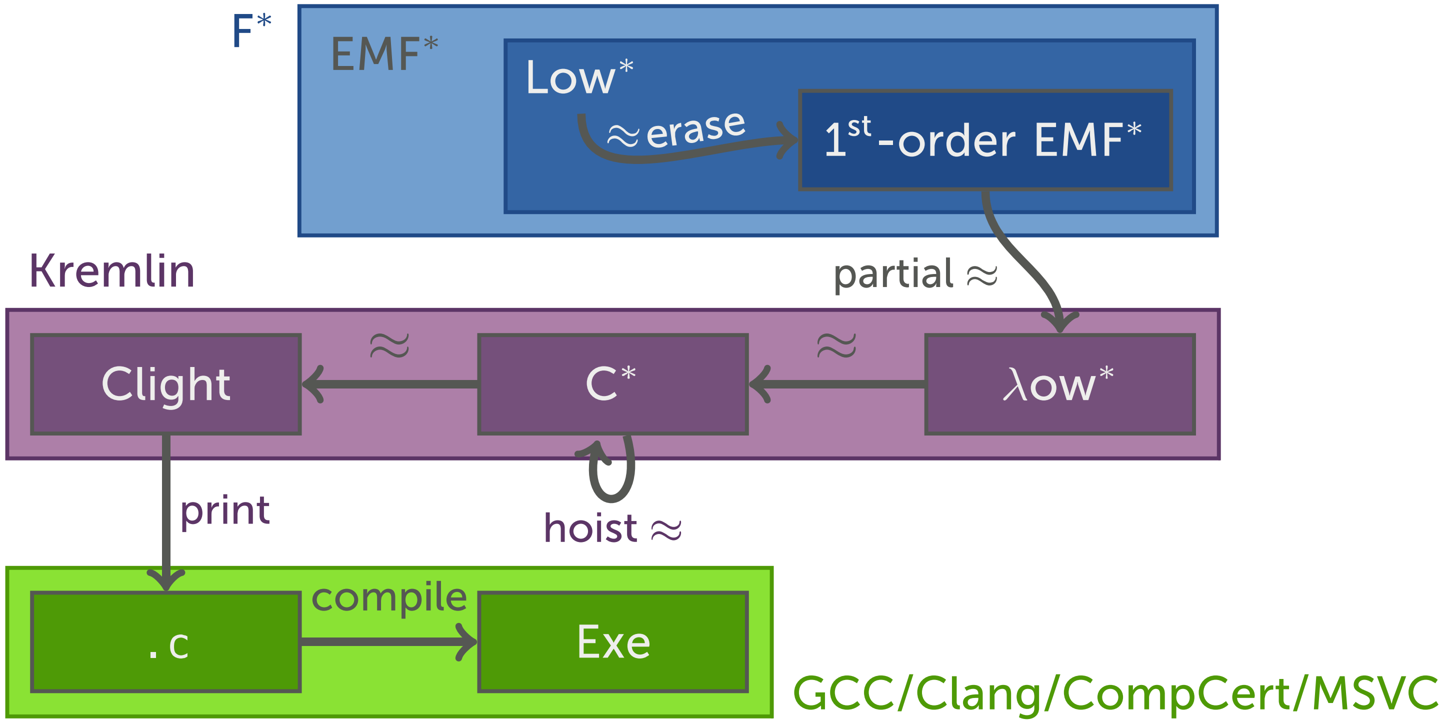
- opts in the Low* effect to model the C stack and heap;
- uses low-level libraries for arrays and structs;
- leverages combinator libraries to get C loops;
- meta-programs first-order code;
- relies on data types sparingly.

For proofs and specs, the programmer:

- can use all of F*,
- prove memory safety, correctness, crypto games, relying on
- erasure to yield a first-order program.

Motto: the code is low-level but the verification is not.

With a diagram



Disclaimer: these steps are supported by hand-written proofs.

Generated C code for Validator

```
uint32_t Parsers_ClientHello_clientHello_validator(LowParse_Low_Base_slice input, uint32_t pos)
{
  uint32_t pos10 = Parsers_ProtocolVersion_protocolVersion_validator(input, pos);
  uint32_t pos11;
  if (pos10 > (uint32_t)4294967279U)
    pos11 = pos10;
  else
    pos11 = Parsers_Random_random_validator(input, pos10);
  uint32_t pos12;
  if (pos11 > (uint32_t)4294967279U)
    pos12 = pos11;
  else
    pos12 = Parsers_SessionID_sessionID_validator(input, pos11);
  uint32_t pos13;
  if (pos12 > (uint32_t)4294967279U)
    pos13 = pos12;
  else
    pos13 = Parsers_ClientHello_cipher_suites_clientHello_cipher_suites_validator(input, pos12);
  uint32_t pos1;
  if (pos13 > (uint32_t)4294967279U)
    pos1 = pos13;
  else
    pos1 =
      Parsers_ClientHello_compression_method_clientHello_compression_method_validator(input, pos13);
  if (pos1 > (uint32_t)4294967279U)
    return pos1;
  else
    return Parsers_ClientHelloExtensions_clientHelloExtensions_validator(input, pos1);
}
```

C… e code for Valid…

Error code, or offset one past the end of valid repr.

Input buffer (uint8_t*) and its length

Offset within the input buffer from which to read

Error code test

```
uint32_t Parsers_ClientHello_clientHello_validator(LowParse_Low_Base_slice input, uint32_t pos)
{
  uint32_t pos10 = Parsers_ProtocolVersion_protocolVersion_validator(input, pos);
  uint32_t pos11;
  if (pos10 > (uint32_t)4294967279U)
    pos11 = pos10;
  else
    pos11 = Parsers_Random_random_validator(input, pos10);
  uint32_t pos12;
  if (pos11 > (uint32_t)4294967279U)
    pos12 = pos11;
  else
    pos12 = Parsers_SessionID_sessionID_validator(input, pos11);
  uint32_t pos13;
  if (pos12 > (uint32_t)4294967279U)
    pos13 = pos12;
  else
    pos13 = Parsers_ClientHello_cipher_suites_clientHello_cipher_suites_validator(input, pos12);
  uint32_t pos1;
  if (pos13 > (uint32_t)4294967279U)
    pos1 = pos13;
  else
    pos1 =
      Parsers_ClientHello_compression_method_clientHello_compression_method_validator(input, pos13);
  if (pos1 > (uint32_t)4294967279U)
    return pos1;
  else
    return Parsers_ClientHelloExtensions_clientHelloExtensions_validator(input, pos1);
}
```

Accessors

```
type accessor
  (#t1 #t2: Type) (p1: parser t1) (p2: parser t2)
  (cond: t1 -> Prop) (f: (x: t1 { cond x }) -> t2)
= (sl: slice) →
  (pos: U32.t) →
  HST.Stack U32.t
  (requires (λ h → valid p1 h sl pos ∧ cond (contents p1 h sl pos)))
  (ensures (λ h res h' →
    B.modifies B.loc_none h h' ∧
    valid p2 h sl res ∧
    contents p2 h sl res == f (contents p1 h sl res)
  ))
```

f: field destructor, or case destructor to the payload of a tagged union, etc. (guarded by cond)

Sample C code using validators and accessors

```
uint32_t count_cipherSuites_inplace(LowParse_Low_Base_slice input, uint32_t pos) {
  uint32_t pos_final = clientHello_validator(input,pos);
  if (4294967279 < pos_final)
    return 0;
  else {
    uint32_t pos_ciphersuites = accessor_clientHello_ciphersuites(input,pos);
    return clientHello_cipherSuites_count(input,pos_ciphersuites);
  }
}
```

Supported Constructs so far

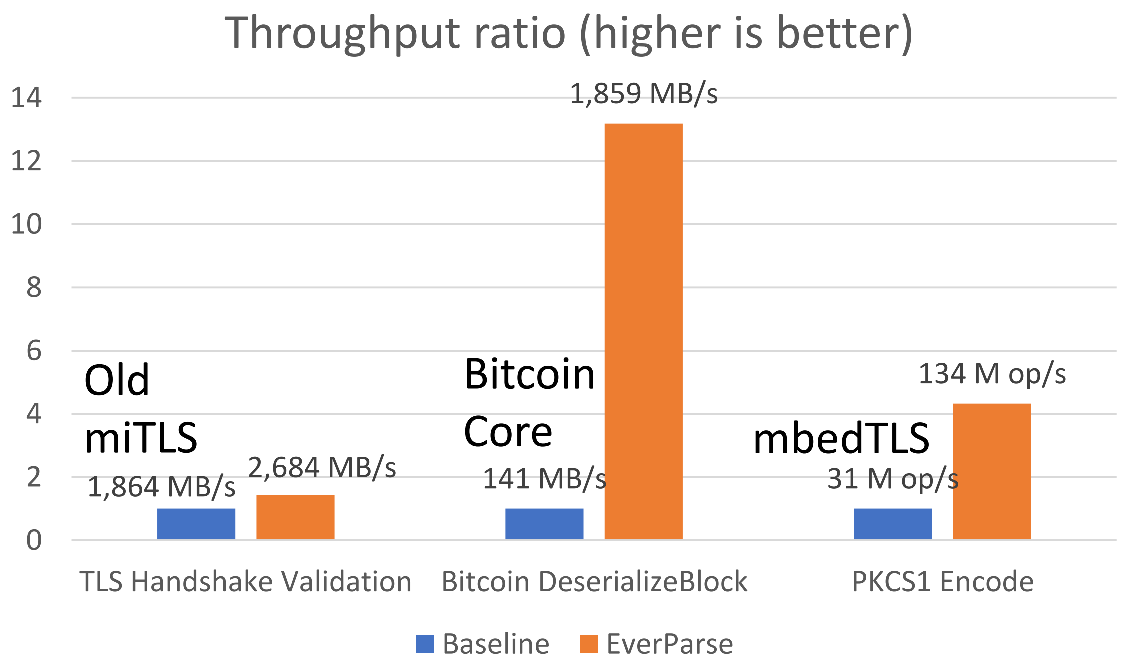
- Integers: parser uint8, parser uint16, parser uint32
  - Big-endian (TLS), little-endian (Bitcoin)
  - Some variable-length integers (Bitcoin, PKCS#1)
- Non-dependent pairs: parser t1 -> parser t2 -> parser (t1 * t2)
- Reinterpretations: parser t1 -> (t1 -> GTot t2) -> parser t2
- Refinements: parser t -> (f: (t -> GTot bool)) -> parser (x: t { f x })
- Enums
- Sum types: parsing dependent on an enum value ("tagged union")
  parser t1 -> ((x1: t1) -> parser (t2 x1)) -> parser (x1: t1 & t2 x1)
- Fixed-length bytes, data and arrays
- Variable-length bytes, data and lists, prefixed by their size in bytes (TLS) or element count (Bitcoin)
- Bitwise combinators in progress (QUIC)

Performance Results

	QD	F* LoC	Verify	Extract	C LoC	Obj.
TLS	1601	70k	46m	25m	190k	717KB
Bitcoin	31	2k	2m	2m	2k	8KB
PKCS1	117	5k	3m	3m	4k	26KB
LowParse		33k	4m	2m	0.2k	1KB

Takeaway:

- Scales to large data formats
- Code produced is fast



EverParse ~ Yacc for Data Exchange Formats

- Stop writing parsers and formatters by hand!

- Get fully-automatic efficient low-level parsers and formatters, with strong guarantees about their safety, correctness and security

- Ongoing applications
  - Ongoing integration into miTLS
  - Ongoing applications: QUIC, MS internal, etc.
  - Other formats? PDF (cf. DARPA SafeDocs)

- Paper: https://www.microsoft.com/en-us/research/publication/everparse/
- Code: https://github.com/project-everest/everparse

- Questions? taramana@microsoft.com