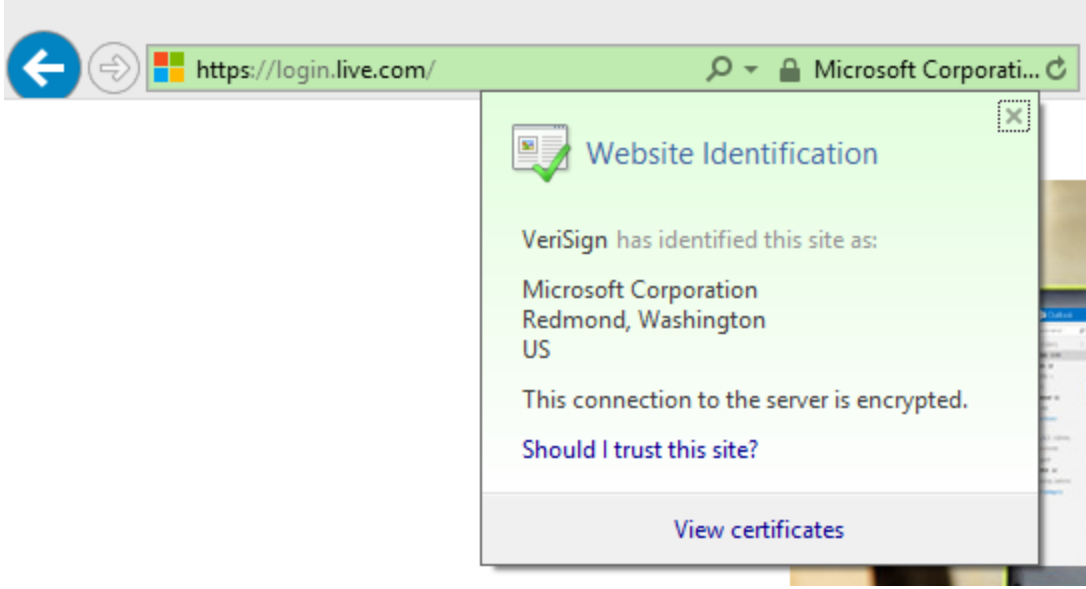
Microsoft Research

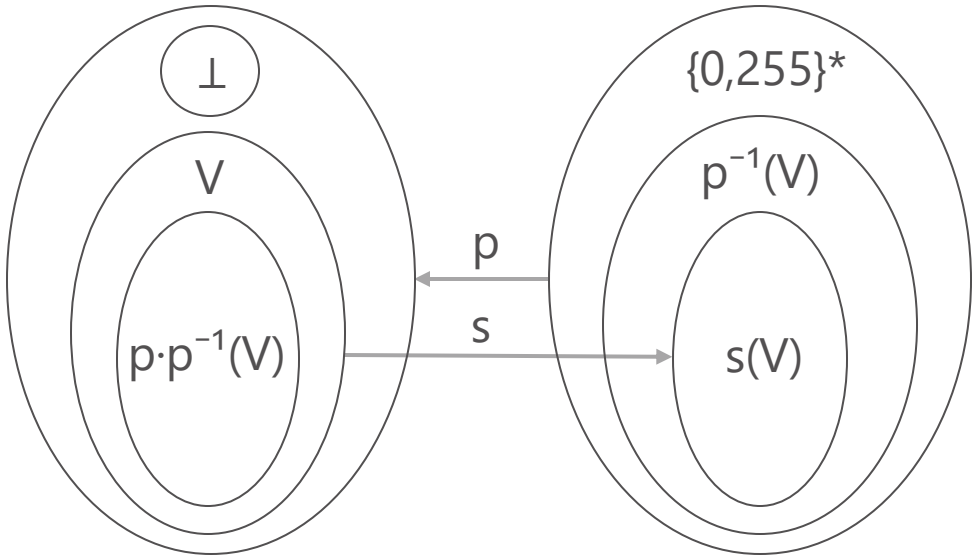


everparse

Verified Secure Zero-Copy Parsers for Authenticated Message Formats

Tahina Ramananandro,
Antoine Delignat-Lavaud,
Cédric Fournet,
Nikhil Swamy,
Tej Chajed,
Nadim Kobeissi &
Jonathan Protzenko

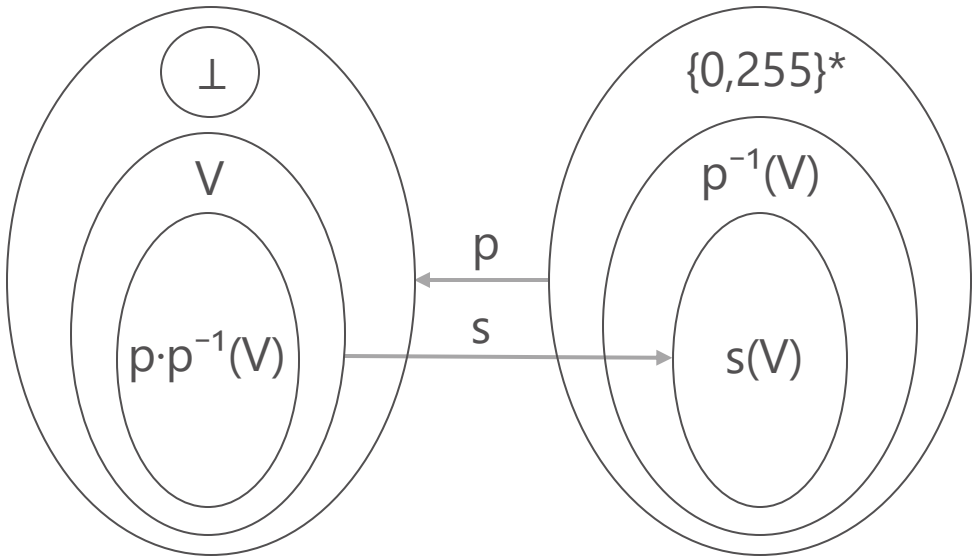
Verified Secure Zero-Copy Parsers for Authenticated Message Formats



- A serializer s maps a message m in V into a sequence of bytes
- A parser p maps a sequence of bytes to either a message in V, or to an error ⊥
- Applications rely on **parser correctness**:

$$\forall\, m \in V,\ p(s(m)) = m$$

Verified Secure Zero-Copy Parsers for Authenticated Message Formats



Different messages have different serializations

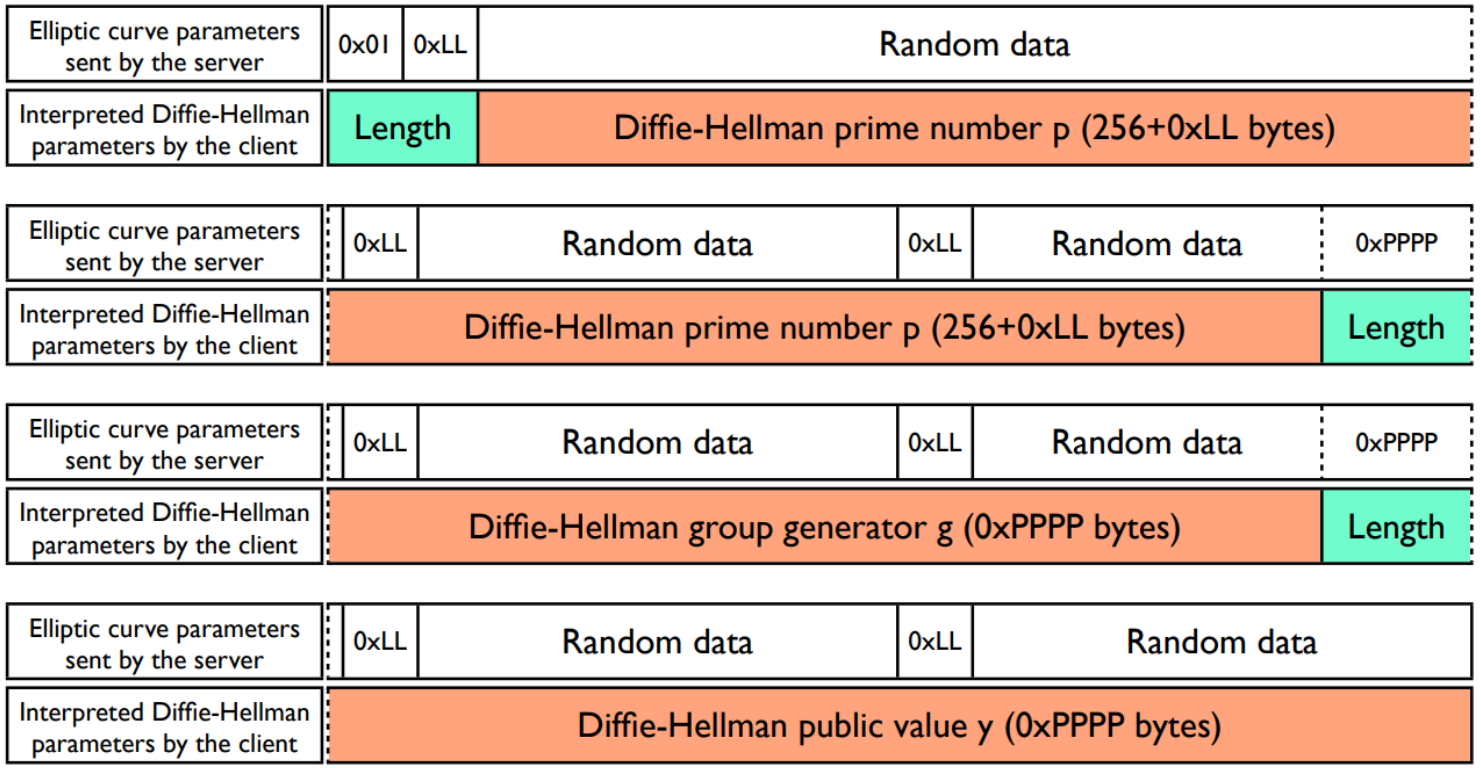
- Correct parsers need **injective serializers**:

  If s(x) = s(y), then p(s(x)) = p(s(y)),
  hence x = y

Some TLS Messages are Ambiguous

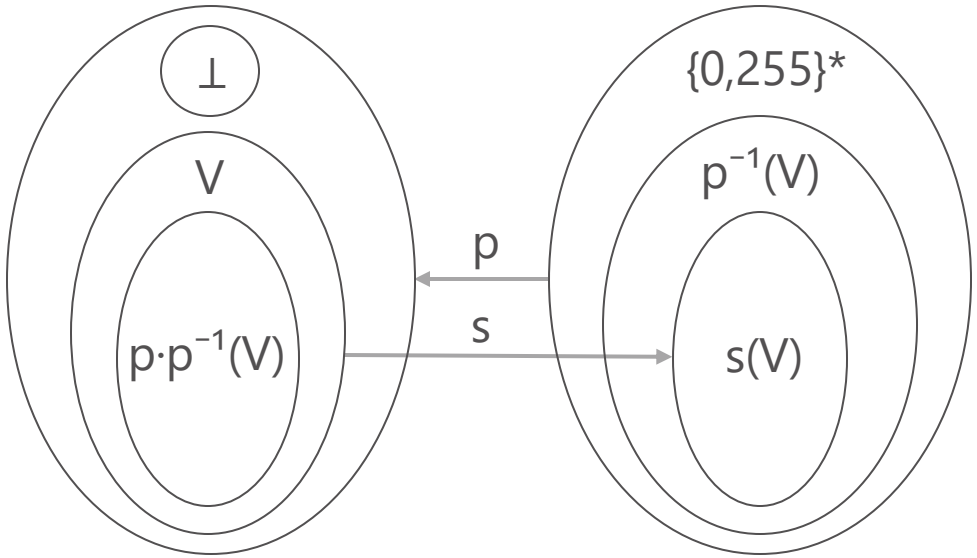
```
enum {dh_anon, dhe, ecdhe, rsa,(255)} KeyExchange;
struct {
  select (KeyExchange) {
    case dh_anon: DHAnonServerKeyExchange;
    case dhe: SignedDHKeyExchange;
    case ecdhe: SignedECDHKeyExchange;
    case rsa: Fail; /* Force error: no SKE in RSA */
  } key_exchange;
} ServerKeyExchange;
```

```
/* From RFC 8446, section 4.4.2 */
enum { X509(0), RawPublicKey(2), (255)} CertType;
opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
struct {
  select (certificate_type) {
    case RawPublicKey: ASN1_subjectPublicKeyInfo;
    case X509: ASN.1Cert; } cert;
  Extension extensions<0..2^16-1>;
} CertificateEntry;
struct {
  opaque certificate_request_context<0..2^8-1>;
  CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```



A cross-protocol attack on the TLS protocol, Mavrogiannopoulos et al.

Verified Secure Zero-Copy Parsers for **Authenticated Message Formats**



- Correct parsers may not be **exact**:
  $p(x) \neq \bot$ for some $x \notin s(V)$,

Difficult to detect, e.g. fuzzing

Bleichenbacher Signature Forgery

- PKCS#1 Signature
  - Hash message **6f9f544f3545f84977549d01efcf664cc4c1b603**
  - Format ASN.1 3021300906052**b0e03021a**050004146**f9f544f3545f84977549d01efcf664cc4c1b603**
  - Pad to public key size:

    0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff003021300906052**b**
    0e03021a05000414**6f9f544f3545f84977549d01efcf664cc4c1b603**

  - Modular exponentiation with private exponent d
  - Verification: un-pad, **parse**, check digest

DigestInfo ::= SEQUENCE {
SEQUENCE {
  hash_alg OID;
  parameters NULL;
}
digest OCTET STRING }

Bleichenbacher Low-Exponent Forgery

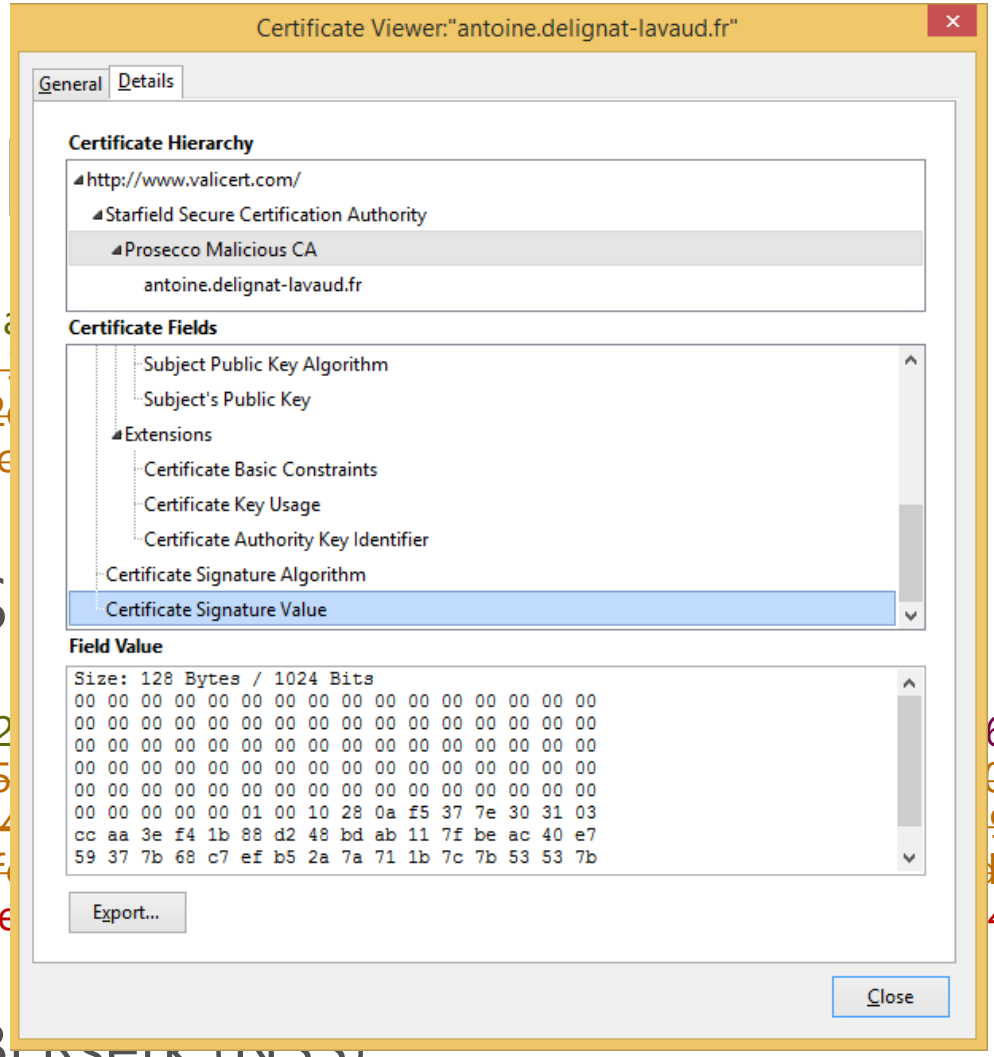
- Original Attac

0001ff00302130090605 2b0e03021a
d01efcf664cc4c1b603 7b031fced01
cc510abd12ca5cc0fceebb75912fc2
0d410c750da3cc510abd12ca5cc0fe

- Many variants

0001ff00302130090605 2b0e0302
d013437521bf3f36c44e0d410c75
c0fceebb75912fc2e38e953cea304
d410c750da3cc510abd12ca5cc0f
00146f9f544f3545f84977549d01e

```
unsigned length = 0;
for (i = 0; i < length_of_length; i++) {
  length <<= 8;
  length |= length[i]; }
```



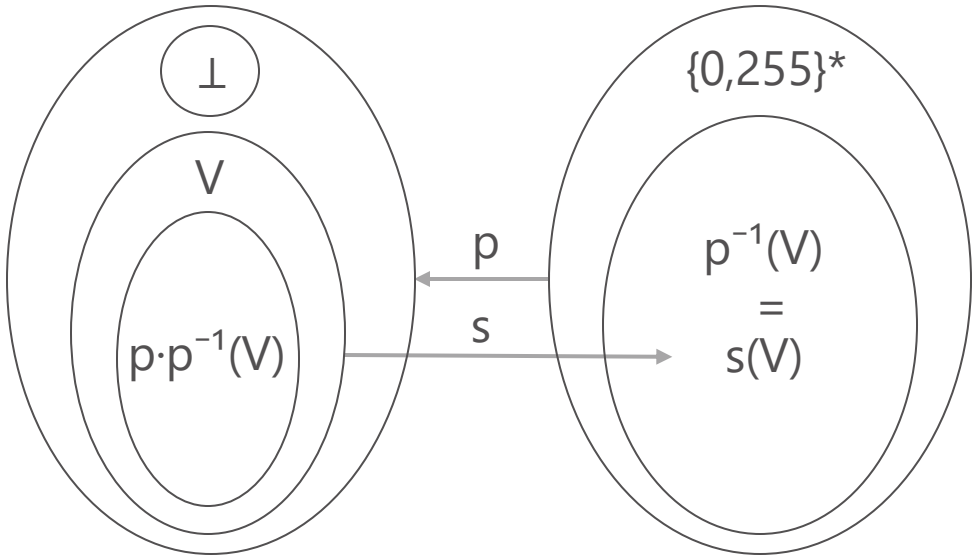
Hash_alg = 2b0e03021a
Digest =
6f9f544f3545f84977549d0
1efcf664cc4c1b603

6052b0e03021a05dc7b031fced01
0d410c750da3cc510abd12ca5cc0f
953cea30432e7521bf3f36c44e0d4
d12ca5cc0fceebb75912fc23cde143
4977549d01efcf664cc4c1b603

BERSERK (NSS)

Windows

Verified Secure Zero-Copy Parsers for **Authenticated Message Formats**

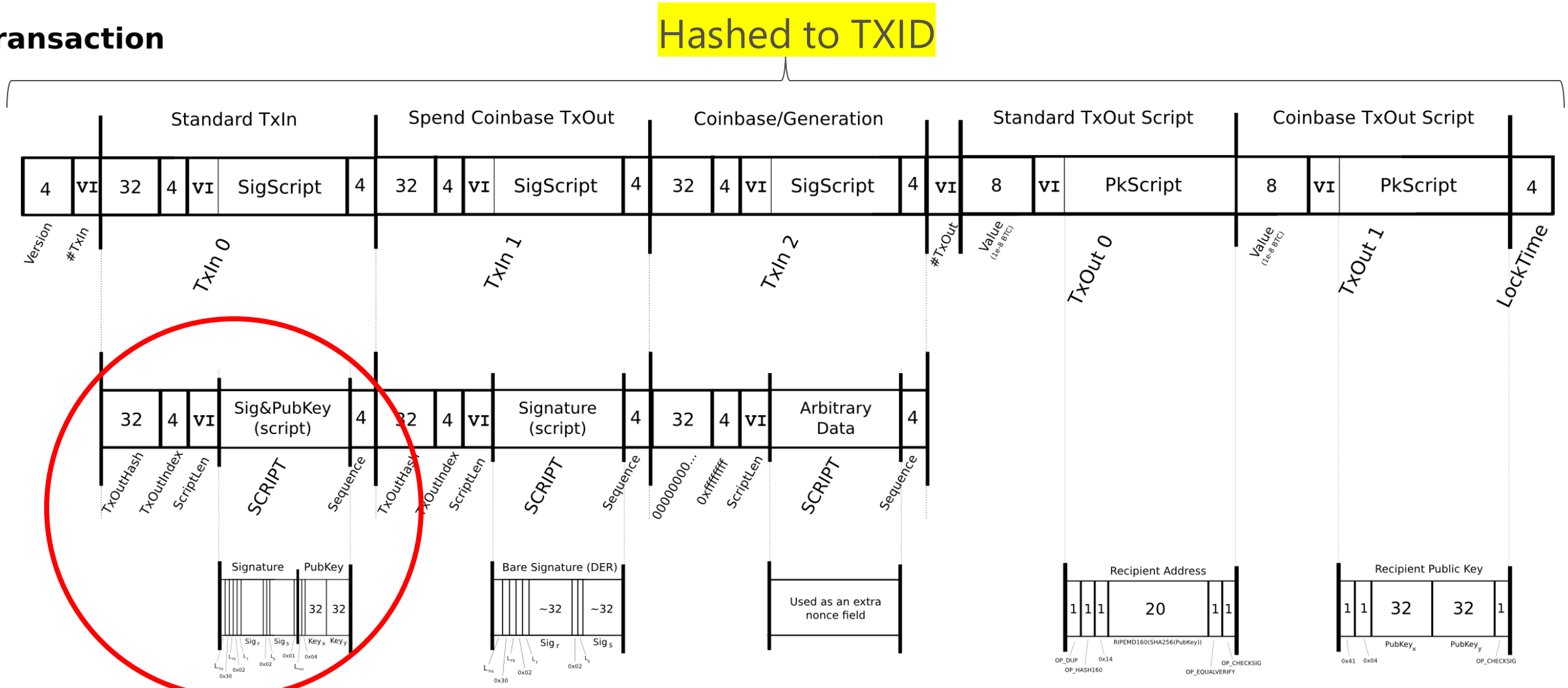


- A **secure** parser must be **exact**:
  $p^{-1}(V) = s(V)$
- A correct parser may be **malleable**:
  $p(x) = p(y)$ with $x \neq y$ and $p(x) \neq \bot$

Two representations of the same message

Bitcoin Transactions

Transaction

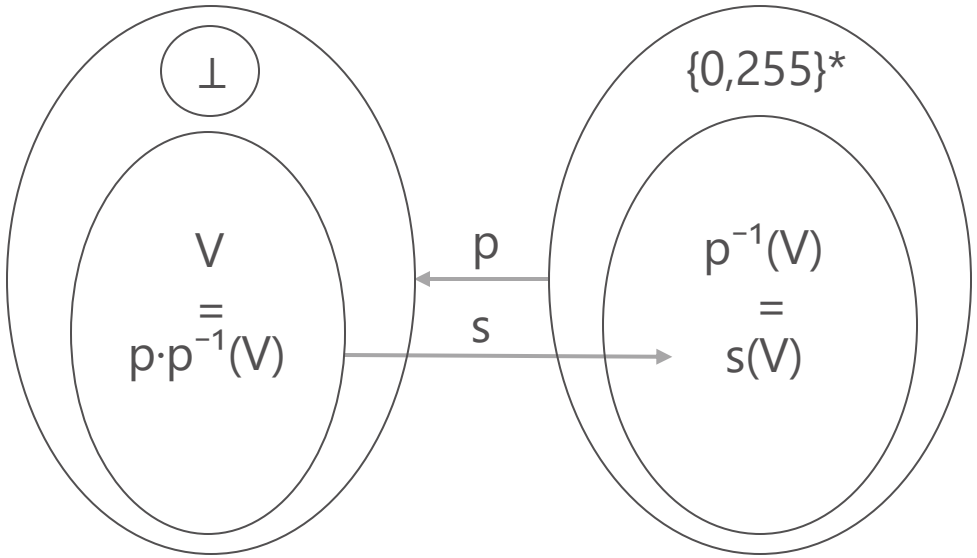


Scripts and DER encoding both use big-endian values, all other serializations use little-endian

Bitcoin Transaction Malleability

- Given a transaction t, it is possible to construct a **valid transaction t' such that h(t)≠h(t')**
- Most Bitcoin users identify transactions by TXID and don't expect that an adversary can change it.
- Forensic blockchain examination shows **thousands of transactions** confirmed under an altered TXID
- Difficult to fix: BIP62, BIP66, BIP141, BIP146, BIP147

Verified Secure Zero-Copy Parsers for **Authenticated Message Formats**



- A **secure** parser must be **exact**:

$$p^{-1}(V) = s(V)$$

- A **secure** parser must be **non-malleable**:

$$p(x) = p(y) \Rightarrow (x = y \vee p(x) = \bot)$$

Very few format description languages are secure by design, e.g. ASN.1 DER

Verified Secure **Zero-Copy** Parsers for Authenticated Message Formats

- Developers write custom formats and parsers for **performance reasons**
- Zero-copy: use the **serialized data representation in memory**
- This increases memory safety risks

Can we automatically generate parsers that are provably **correct, secure and safe**?

LowParse: Verified Combinators

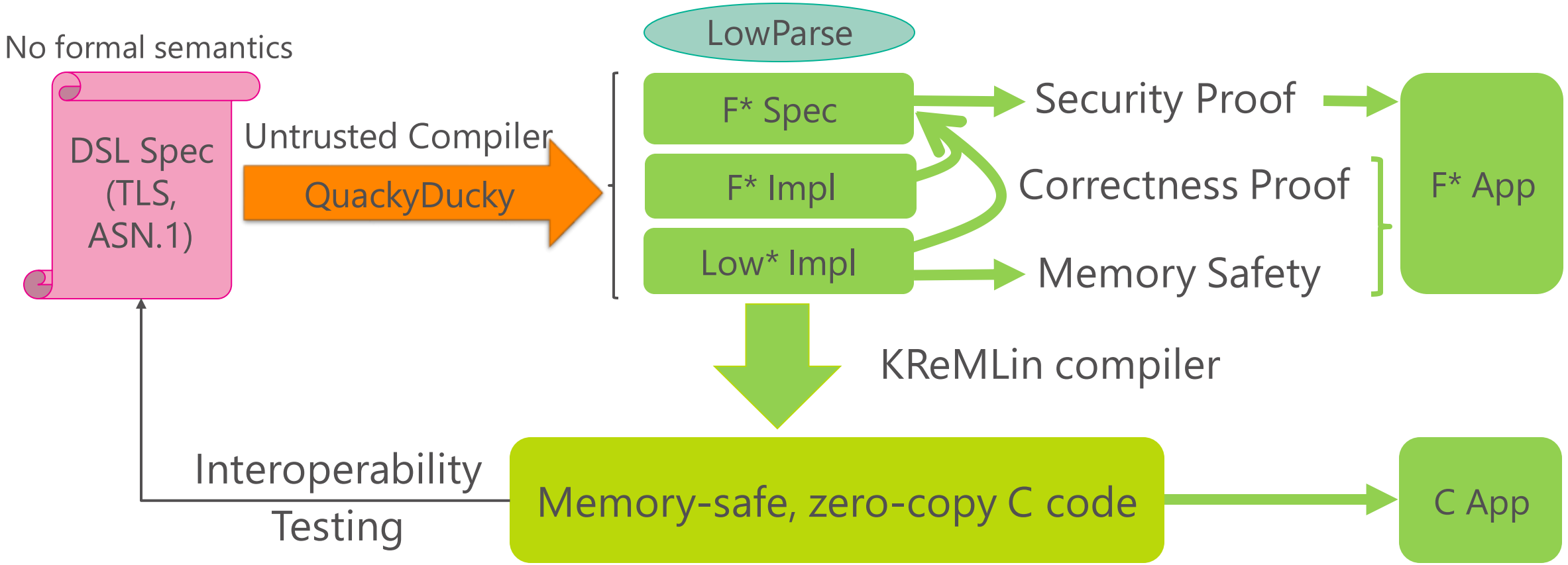
- We define combinators for correct parsers:

```
type parser t = b:bytes -> option (t * n:nat{n <= length b})
type serializer (p: parser t) = f:(t -> bytes) {
   forall x. p (f x) == Some (x, length (f x))
}
```

- With a well-known monadic structure:

```
val return: parser unit
val bind: parser t -> (t -> parser t') -> parser t'
val seq: parser t -> parser t' -> parser (t * t')
val map: f:(t -> t') -> parser t -> parser t'
```

EverParse Architecture



Supported Formats

QuackyDucky Syntax	Data Type	Parser Combinator
uint$N$, $N \in \{8, 16, 32, 64\}$	Unsigned integer within $0..2^N - 1$	parse_uN
t[N], $N \in \mathbb{N}$	Fixed-size array of ts of length N	plist[p] truncN
t<$M..N$>	List of ts, of variable length $M..N$	vldata(plist[p], M, N)
t{$M..N$}	List of ts of variable element count $M..N$	$(\text{parse\_u}k \text{ filter } (n \mapsto M \leq n \leq N)) \triangleright (n \mapsto p^n)$ where $k = 8 \times \log_{256} N$
struct{$t_1\ x_1; \ldots; t_n\ x_n;$}	Record with n fields named (x_i) of type (t_i)	$(p_1 \times \cdots \times p_n)$ synth $((v_1, \ldots, v_n) \mapsto \{x_1 = v_1; \ldots; x_n = v_n\})$
struct {…; uint$N$ x; t y[x]; …}	Variable-length field y prefixed by its length x	vldata$(p, 0, 256^{N/8} - 1)$
enum {$E_1(N_1)$, …, $E_n(N_n)$, (M)}	Constant integer enumeration (with maximal value $M = 2^N - 1$)	penum(parse_uN, $[(E_1, N_1); \ldots; (E_n, N_n)]$)
struct {t x; select(x) { case E_1: t_1; …; case E_n: t_n } y}	Tagged union (t must be an enum type)	$p \triangleright_f q$ where $f(E_i, x) = E_i$ and $q(E_i) = p_i$ synth $(y \mapsto (E_i, y))$

Low* Implementation (Validators)

```
let validator (p:parser t) =
  bs:array uint8 ->
  pos:u32 { pos <= length bs } ->
  ST u32
    (requires fun h0 -> h0 `contains` bs)
    (ensures fun h0 res h1 ->
       h0 == h1 /\
       (if res < ERROR_CODE then
          exists v. p (as_bytes bs h0 pos) = Some (v, res)
          (* parsing succeeds *)
        else p (as_bytes bs h0 pos) = None
          (* parsing fails *)))
```

Higher-order type, but gets specialized at extraction for every type t. Similar for `seq`

Example: Bitcoin Block Parsing

```
opaque sha256[32];
struct {
  sha256 prev_hash; uint32_le prev_idx;
  opaque scriptSig<0..10000 : bitcoin_varint>;
  uint32_le seq_no;
} txin;
struct {
  uint64_le value;
  opaque scriptPubKey<0..10000 : bitcoin_varint>;
} txout;
struct {
  uint32_le version;
  txin inputs{0..1000 : bitcoin_varint};
  txout outputs{0..11110 : bitcoin_varint};
  uint32_le lock_time;
} transaction;
struct {
  uint32_le version;
  sha256 prev_block; sha256 merkle_root;
  uint32_le timestamp;
  uint32_le bits; uint32_le nonce;
  transaction tx{0..2^16 : bitcoin_varint};
} block;
```

```
uint32_t Txin_txin_validator(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txin_txin_jumper(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txin_accessor_txin_prev_hash(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txin_accessor_txin_prev_idx(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txin_accessor_txin_scriptSig(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txin_accessor_txin_seq_no(LowParse_Low_Base_slice input, uint32_t pos);
```

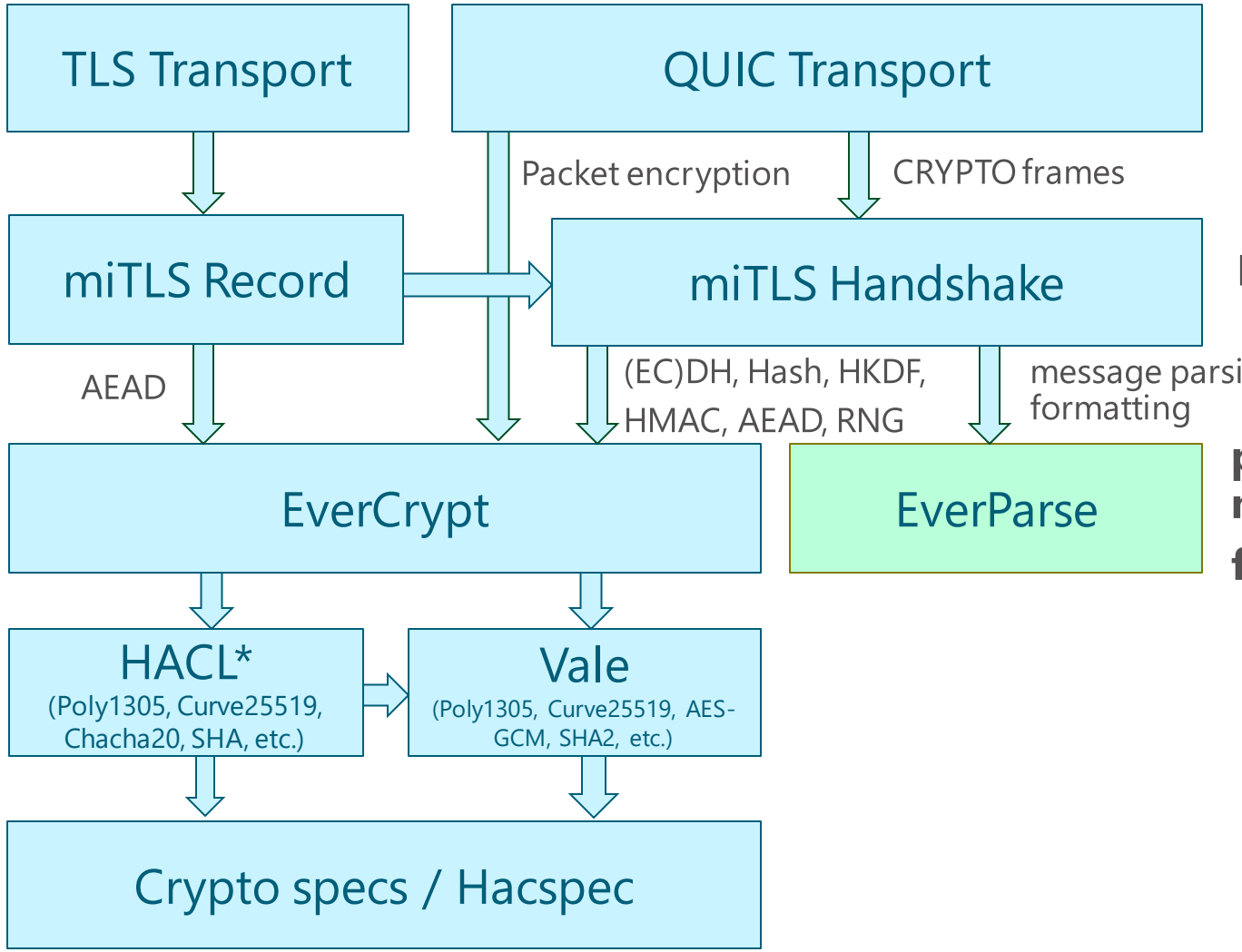
```
uint32_t Txout_txout_validator(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txout_txout_jumper(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txout_accessor_txout_value(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Txout_accessor_txout_scriptPubKey(LowParse_Low_Base_slice input, uint32_t pos);
```

```
uint32_t Transaction_transaction_validator(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Transaction_transaction_jumper(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Transaction_accessor_transaction_version(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Transaction_accessor_transaction_inputs(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Transaction_accessor_transaction_outputs(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Transaction_accessor_transaction_lock_time(LowParse_Low_Base_slice input, uint32_t pos);
```

```
uint32_t Block_block_validator(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_block_jumper(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_version(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_prev_block(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_merkle_root(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_timestamp(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_bits(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_nonce(LowParse_Low_Base_slice input, uint32_t pos);
uint32_t Block_accessor_block_tx(LowParse_Low_Base_slice input, uint32_t pos);
```

EverParse in Everest

Data stream/multi-stream security, …



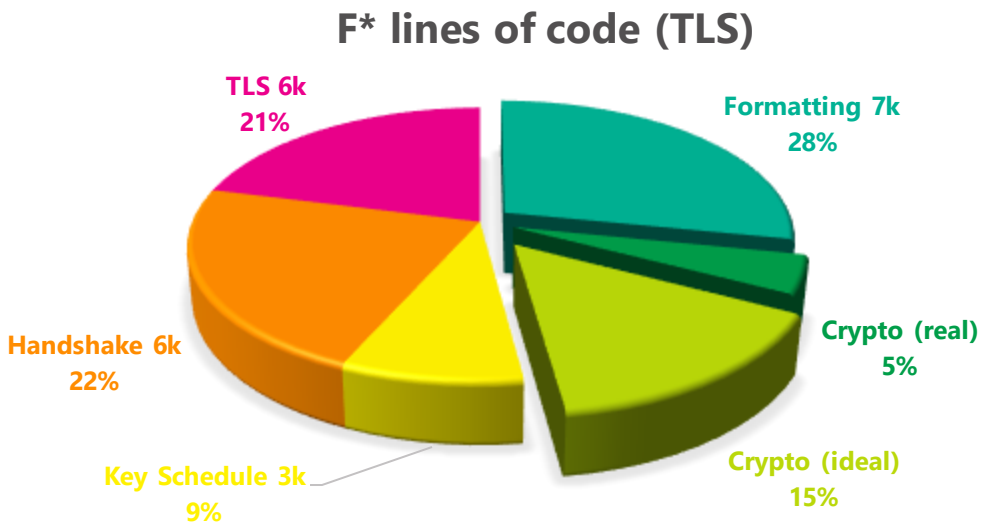
Key exchange security

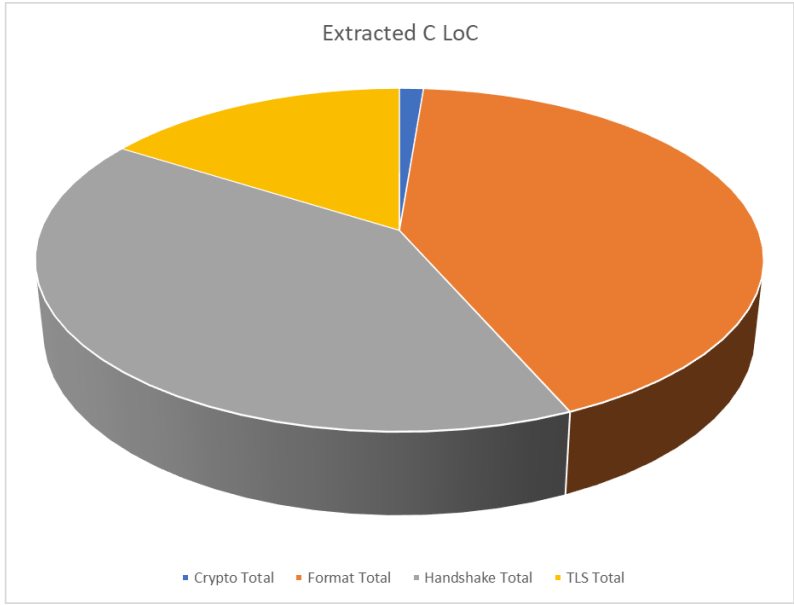
side-channel resistance, memory safety, functional correctness

parser security, memory safety, functional correctness

EverParse in Everest

- Message processing is a large portion of the miTLS codebase
- This code is repetitive and incredibly tedious to verify
- Inefficient heap allocations during parsing is a major obstacle to deployment





TLS Difficulties

```
struct {
  select (kex) {
    case dh_anon: DHAnonServerKeyExchange;
    case dhe: SignedDHKeyExchange;
    case ecdhe: SignedECDHKeyExchange;
    /* Force a parsing error if handshake tries to parse SKE in RSA kex */
    case rsa: Fail;
  } ske; /*key_exchange*/
} ServerKeyExchange;
```

TLS Difficulties

ServerKeyExchange parser requires an extra KeyExchangeAlg argument

```
struct {
  /*@implicit*/ KeyExchangeAlgorithm kex;
  select (kex) {
    case dh_anon: DHAnonServerKeyExchange;
    case dhe: SignedDHKeyExchange;
    case ecdhe: SignedECDHKeyExchange;
    /* Force a parsing error if handshake tries to parse SKE in RSA kex */
    case rsa: Fail;
  } ske; /*key_exchange*/
} ServerKeyExchange;
```

TLS Difficulties

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

padding_length MUST be at least 16, and equal to TLSPlaintext.length-payload_length-3

TLS Difficulties

```
struct {
  HeartbeatMessageType type;
  opaque payload<0..2^14-21>;
  opaque padding<16..2^14-3>;
} HeartbeatMessage;
```

TLS Difficulties

Dependency on a parent type

```
struct {
  select (Handshake.msg_type) {
    case client_hello:
      ProtocolVersion versions<2..254>;
    case server_hello: /* and HelloRetryRequest */
      ProtocolVersion selected_version;
  };
} SupportedVersions;
```

TLS Difficulties

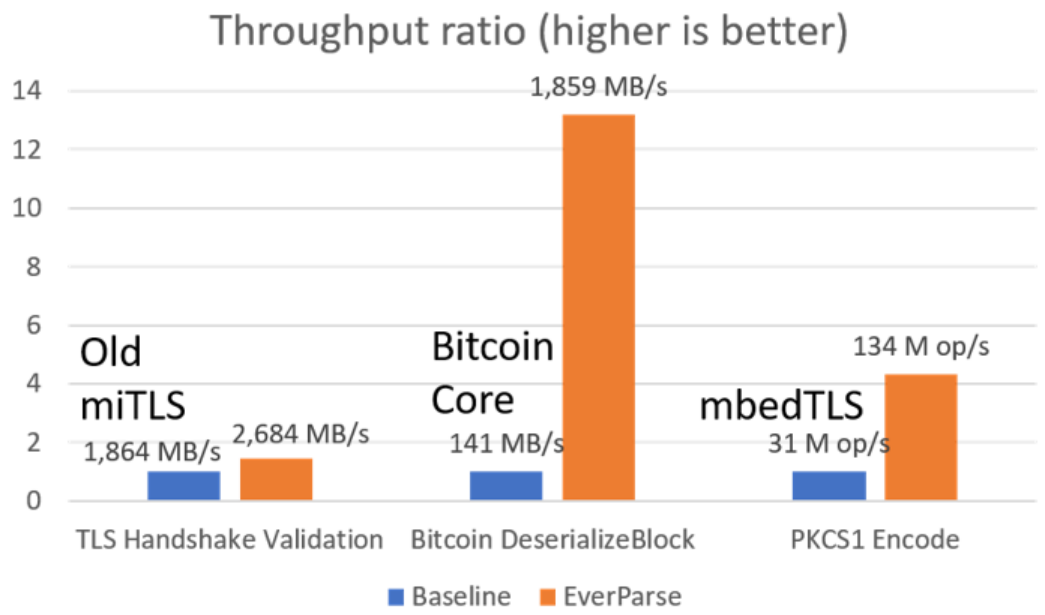
```
struct {
  ExtensionType type;
  uint16 len;
  select(type)
  {
    case server_name: ServerNameList;
    // ...
    case supported_versions:
      SupportedVersions;
    // ...
    default: opaque;
  } CHE[len]; /*extension_data*/
} ClientHelloExtension;
```

```
struct {
  ExtensionType type;
  uint16 len;
  select(type)
  {
    case server_name: Empty;
    // ...
    case supported_versions:
      ProtocolVersion;
    // ...
    default: opaque;
  } SHE[len]; /*extension_data*/
} ServerHelloExtension;
```

Evaluation

	Data Types	QD	F* LoC	Verify	Extract	C LoC	Obj.
TLS 1.2-1.3	315	1601	70k	46m	25m	190k	717KB
Bitcoin	6	31	2k	2m	2m	2k	8KB
PKCS1	19	117	5k	3m	3m	4k	26KB
LowParse			33k	4m	2m	0.2k	1KB

- We can express real world formats
- We scale to large and complex schemas
- We produce high-performance code



Proof Engineering Challenges

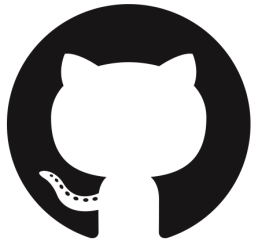
- Many proof conditions scale with the size of the type
- This can quickly overwhelm the SMT in F*
- We rely on new F* features to scale proofs:
  - Extensive use of proof by normalization
  - F* meta-programming via tactics, e.g. to generate implementations
    Meta-F: Proof Automation with SMT, Tactics, and Metaprograms, ESOP'19*

Main Takeaways

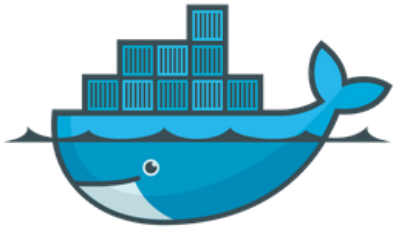
- The first parser generation tool to generate both **zero-copy** and **secure** parsers
- Automated proofs of **correctness, security and safety**
- Generated code is **high performance**
- Useful to analyse message format security

Microsoft Research

everparse



https://github.com/project-everest/everparse
https://github.com/project-everest/mitls-fstar
https://fstar-lang.org



https://hub.docker.com/r/projecteverest/quackyducky-linux

F* Tactics

- F* Tactics can operate on the reflected syntax of F* terms
- Tactics are defined as an F* effect that operates on proof terms.

```
type term_view =
  | Tv_BVar : v:dbvar → term_view
  | Tv_Var : v:name → term_view
  | Tv_FVar : v:qname → term_view
  | Tv_Abs : bv:binder → body:term → term_view
  | Tv_App : hd:term → a:(term * aqual)
                     → term_view

val inspect : term → Tac term_view
val pack : term_view → term
```

```
type error = exn * proofstate (* error and proofstate at the time of failure *)
type result a = | Success : a → proofstate → result a | Failed : error → result a
let tac a = proofstate → Div (result a)
let t_return #a (x:a) = λps → Success x ps
let t_bind #a #b (m:tac a) (f:a → tac b) : tac b = λps → ... (* omitted, yet simple *)
let get () : tac proofstate = λps → Success ps ps
let raise #a (e:exn) : tac a = λps → Failed (e, ps)
new_effect { TAC with repr = tac ; return = t_return ; bind = t_bind
                    ; get = get ; raise = raise }
```

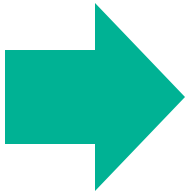
```
type proofstate
type goal
type term
type env

val goals_of : proofstate → list goal
val goal_env : goal → env
val goal_type : goal → term
val goal_solution : goal → term
```

Implementation Generation Tactic

```
module T = FStar.Tactics
let rec gen_parser32' (env: T.env) (t: T.term) (p: T.term) : T.Tac T.term =
  let (hd, tl) = app_head_tail p in
  if hd `T.term_eq` (`(parse_ret)) then T.mk_app (`(parse32_ret)) tl else
  if hd `T.term_eq` (`(parse_u8)) then (`(parse32_u8)) else
  if hd `T.term_eq` (`(and_then)) then match tl with
  | [(t, _); (p, _); (t', _); (p', _)] →
    begin match T.inspect p' with
    | T.Tv_Abs bx body →
      let p32 = gen_parser32' env k t p in
      let env' = T.push_binder env bx in
      let body' = gen_parser32' env' k' t' body in
      let p32' = T.pack (T.Tv_Abs bx body') in
      (`(parse32_and_then #(`#t) #(`#p) (`#p32) (`#t') (`#p') (`#p32')))
    | _ → ...
    end
```

```
let example : parser_spec FStar.Int16.t =
  and_then parse_u8
  (λ lo → if lo <^ 128uy then parse_ret (cast_u8_to_i16 lo)
    else parse_synth parse_u8 (λ hi → cast_u8_to_i16 (lo %^ 128uy)
                                    +^ cast_u8_to_i16 hi))
```



```
let gen_parser32 () : T.Tac unit =
  let (hd, tl) = app_head_tail (T.cur_goal ()) in
  if hd `T.term_eq` (`parser32)
  then match tl with
  | [(t, _); (p, _)] →
    let env = T.cur_env () in
    let p32 = gen_parser32' env t p in
    T.exact_guard p32;
  | _ → ...
```

```
parse32_and_then parse32_u8 (λ lo →
  parse32_ifthenelse (lo <^ 128uy) (λ _ →
    parse32_ret (cast_u8_to_i16 lo)) (λ _ →
      parse32_synth parse32_u8 (λ hi →
        cast_u8_to_i16 (lo %^ 128uy) +^
          cast_u8_to_i16 hi)))
```