everparse

Verified Parser Generation for Security Critical Applications

Nik Swamy

OPLSS 20201

https://project-everest.github.io/everparse/

Tahina Ramananandro, Aseem Rastogi, Tej Chajed, Antoine Delignat-Lavaud, Cedric Fournet, Nadim Kobeissi, Guido Martinez, Jonathan Protzenko, Irina Spiridonova

Incorrect handling of attacker-controlled inputs

➔ Leading cause of software security attacks

2020 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 | Analysis | Methodology | Scoring Metrics | On the Cusp | Limitations | Remapping

Introduction

The 2020 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses (CWE Top 25) is a demonstrative list of the most common and impact the previous two calendar years. These weaknesses are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a syst an application from working. The CWE Top 25 is a valuable community resource that can help developers, testers, and users — as well as project managers, security res provide insight into the most severe and current security weaknesses.

To create the 2020 list, the CWE Team leveraged Common Vulnerabilities and Exposures (CVE®) data found within the National Institute of Standards and Technology (N Database (NVD), as well as the Common Vulnerability Scoring System (CVSS) scores associated with each CVE. A formula was applied to the data to score each weakne and severity.

The CWE Top 25

Below is a brief listing of the weaknesses in the 2020 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50

Incorrect handling of attacker-controlled inputs

Dire in low-level code where input validation and parsing code is

Hand written in C/C++

- for performance
- for deployability (e.g, in kernel)
- for legacy

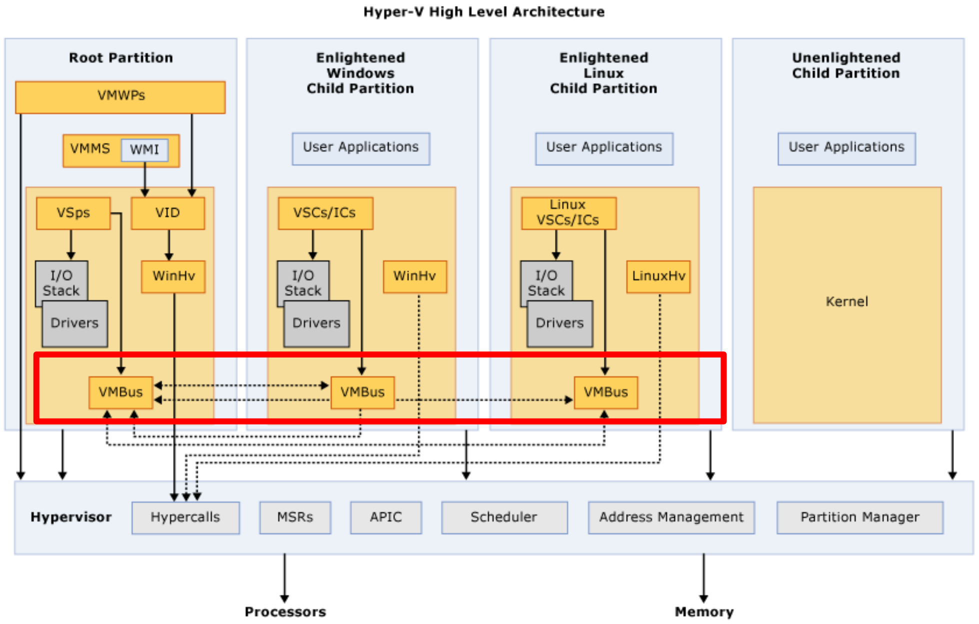
And errors are catastrophic due to a lack of memory safety

Incorrect handling of attacker-controlled inputs

In many guises, across the stack

Remote adversary

Also deep within critical systems when traversing trust boundaries



Incorrect handling of attacker-controlled inputs

In many guises, across the stack

Bad formats
U
Bad parser implementaions

1. Hand-rolled data-exchange formats, hand-rolled parsers
   - What could possibly go wrong?

2. Standardized formats, hand-rolled parsers
   - Windows 10 Bad Neighbor: TCP/IP ICMPv6 Router Advertisement

     Improper parsing of variable length inputs leading to remote code execution/BSOD
   - Heartbleed:

     Improper parsing of variable length input leading to information disclosure

3. Standardized formats: buggy formats, buggy parsers
   - E.g., Malleability: leading to crypto vulnerabilities
     - PKCS #1 signature forgery, Bitcoin transaction malleability

…m hand-rolled …ions

…neration for parsing and serializing

…e ric…

Preventative Measures

It is our policy that any time a security problem is found, we will not only fix the problem, but also implement new measures to prevent the class of problems from occurring again. To that end, here's what we're doing doing to avoid problems like these in the future:

1. A fuzz test of each pointer type has been added to the standard unit test suite.
2. We will additionally add fuzz testing with American Fuzzy Lop to our extended test suite.
3. In parallel, we will extend our use of template metaprogramming for compile-time unit analysis (kj::Quantity in kj/units.h) to also cover overflow detection (by tracking the maximum size of an integer value across arithmetic expressions and raising an error when it overflows). More on this below.
4. We will continue to require that all tests (including the new fuzz test) run cleanly under Valgrind before each release.
5. We will commission a professional security review before any 1.0 release. Until that time, we continue to recommend against using Cap'n Proto to interpret data from potentially-malicious sources.

I am pleased to report that measures 1, 2, and 3 all detected both integer overflow/underflow problems, and AFL additionally detected the CPU amplification problem.

CAP'N PROTO

cerealization protocol

infinitely faster!

News

Get Email Updates

Follow on Twitter

Security Advisory -- And how to catch integer overflows with template metaprogramming

Integer overflow bugs

As the installation page has always stated, I do not yet recommend using Cap'n Proto's C++ library for handling possibly-malicious input, and will not recommend it until it undergoes a formal security review. That said, security is obviously a high priority for the project. The security of Cap'n

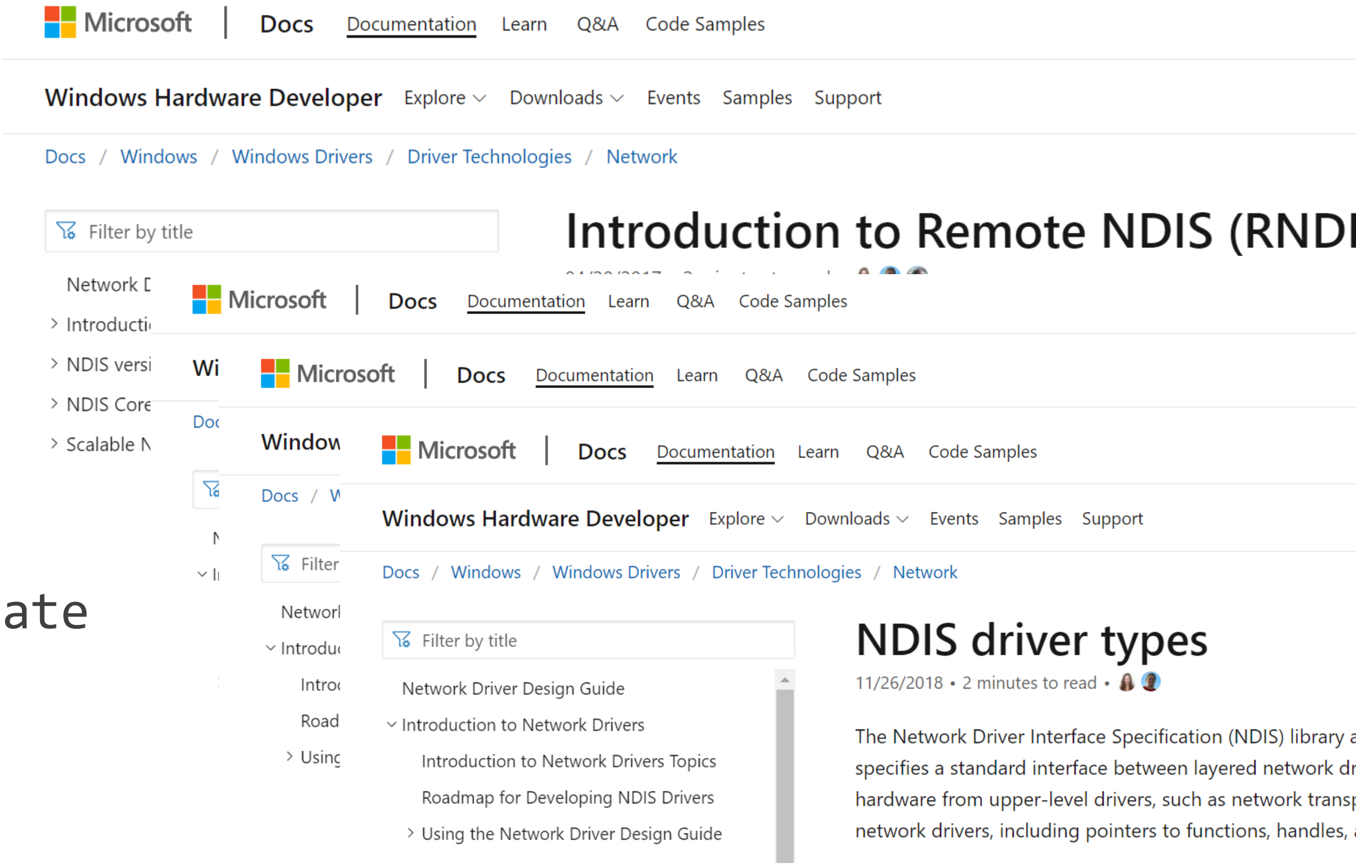
Plus, many legacy formats remain

Designed for

- Compactness
- ABI compatibility
- mmap'able

Serialization: `memcpy`

Parsing: `reinterpret_cast<T> . validate`



Standardized formats have their challenges too

Wire formats prescribed by RFCs in a semi-formal notation

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

Or in other notations like ASN.1

```
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;
```

Are the formats well-designed?

- E.g., non-malleable?

Are their parsers and serializers correctly implemented?

everpʌrse

A Mathematically Proven Low-level Parser Generator

For a variety of formats, ranging from mmap'able binary wire formats to semi-formal RFC specs

Our goal
- Abolish writing low-level binary format parsers by hand
- Instead, specify formats in a high-level declarative notation
- Auto-generate performant low-level code to parse binary messages
- Integrate seamlessly with existing codebases in a variety of languages (C, C++, Rust, …)

With formal proofs that:
- Formats enjoy various good properties, e.g., non-malleability
- Generated code is
  - Memory safe (no access out of bounds, no use after free etc.)
  - Arithmetically safe (no overflow/underflow)
  - Functionally correct (that it parses exactly those messages that conform to the high-level spec)
  - Free from double-fetches, so safe against time-of-check/time-of-use bugs

https://project-everest.github.io/everparse/

Starting from a high-level language of message formats

EverParse auto-generates parsing code that is

- Safe
- Correct
- Fast (zero-copy)

Correctness:

```
parse (serialize msg) = msg
valid msg ==> serialize (parse msg) = msg
```

Performa

ASN.

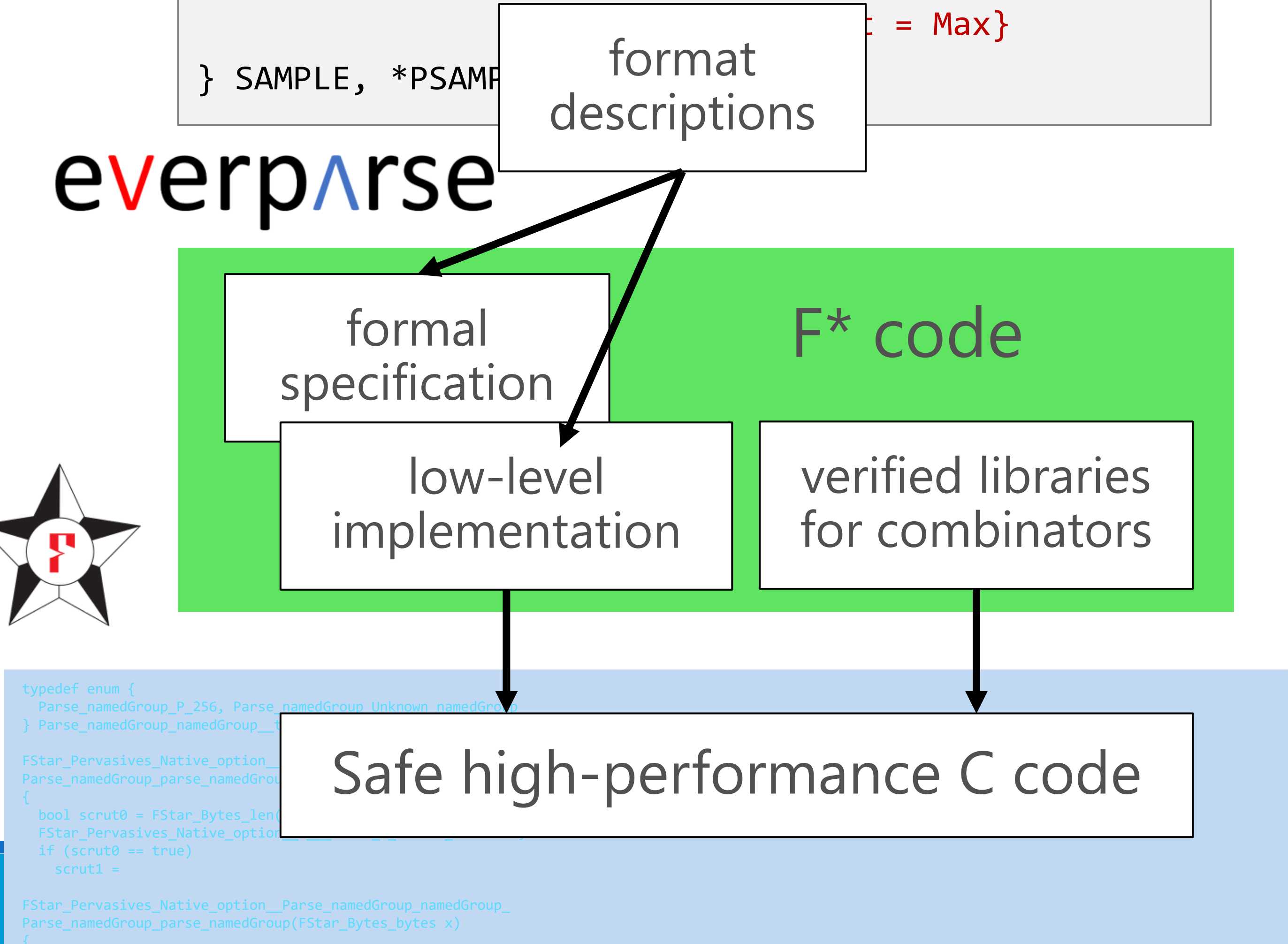
EverParse

LS

A type-theory-based proof assistant and programming language
https://fstar-lang.org

Hello.3d:

```
typedef struct _Sample(mutable PUINT32 out) {
  UINT32      MajorVersion { MajorVersion = 1 };
  UINT32      MinorVersion { MinorVersion = 0 };
  UINT32      Min;
  UINT32      Max { Min <= Max }
                                   = Max}
} SAMPLE, *PSAMP
```



everparse

Hardening critical applications in C/C++

Since spring 2020

Every network packet passing through Microsoft Hyper-V is validated by EverParse

Hyper-V: Core isolation technology of the Microsoft Azure cloud

Multiple layers of headers, many verified already, further layers in progress

Custom binary wire formats designed to also be ABI-compatible and mmap’able

Verified parsers and serializers for non-malleable wire formats in verified F* applications

TLS 1.3 record and handshake messages

QUIC record layer messages

DICE Secure Measured Boot for IoT devices, ASN.1

Bitcoin transaction log validation

All producing verified high-performance C code

Hardening critical applications in C/C++

A diversity of existing wire formats

- Designed for efficiency, compactness and ABI compatibility
- So that parsing and serialization can be done by memcpy/reinterpret_cast

We designed a new specification language to capture a wide variety of these formats

Produce functionally correct, memory safe, double-fetch free validators of these formats in C

And interpose our validators at the attack surface to ensure that ill-formed data doesn't reach the rest of the system

Dependent Data Descriptions in 3D: A source language of message formats

Augmenting C data types with constraints, variable-length structures, and actions

Refinement types for data validity constraints

```
typedef struct _SAMPLE {
  UINT32      MajorVersion { MajorVersion = 1 };
  UINT32      MinorVersion { MinorVersion = 0 };
  UINT32      Min;
  UINT32      Max { Min <= Max }
} SAMPLE;
```

Augmenting C data types with constraints, variable-length structures, and actions

Contextually tagged unions with casetype

```
typedef union _MessageUnion {
    Init  init;
    Query query;
    Halt  halt;
} MessageUnion;
```

```
typedef struct _Message {
    UINT32 tag;
    MessageUnion message;
} Message;
```

Augmenting C data types with constraints, variable-length structures, and actions

Contextually tagged unions with casetype

```
casetype _MessageUnion(UINT32 tag) {
  switch(tag) {
  case INIT_MSG:
    Init  init;
  case QUERY_MSG:
    Query query;
  case HALT_MSG:
    Halt  halt;
  }
} MessageUnion;
```

```
typedef struct _Message {
   UINT32 tag;
   MessageUnion(tag) message;
} Message;
```

Structures with variable-length fields

```
typedef struct _VLDATA {
  UINT32      ByteLength;
  SAMPLE      Samples[:byte-size ByteLength]
} VLDATA;
```

Augmenting C data types with constraints, variable-length structures, and actions

Structures with variable-length fields

```
typedef struct _VLDATA(UINT32 TotalMessageLength) {
  UINT32      ByteLength;
  UINT32      Offset
  { is_range_okay(TotalMessageLength, Offset, ByteLength) &&
    Offset >= sizeof(this) };
  UINT8       Padding[:byte-size Offset – sizeof(this)]
  SAMPLE      Samples[:byte-size ByteLength]
} VLDATA;
```

Imperative actions for selective parsing and further validation

```
typedef struct _Sample(mutable PUINT32 out) {
  UINT32      MajorVersion { MajorVersion = 1 };
  UINT32      MinorVersion { MinorVersion = 0 };
  UINT32      Min;
  UINT32      Max { Min <= Max }
                  {:on-success *out = Max}
} SAMPLE;
```

Augmenting C data types with constraints, variable-length structures, and actions

Imperative actions
for selective parsing and further validation

```
<---------------- SizeOfAs  ------------->|<---- sizeof(B_ENTRY) * (i_0 + ... + i_n) --->
.-------------------------------------------------------------------------------------.
|   A_ENTRY { i_0 } | ... | A_ENTRY { i_n } |    B_ENTRY      |    ...    |      B_ENTRY    |
.-------------------------------------------------------------------------------------.
```

```
typedef struct _A(mutable UINT32* accum) {
  …
  UINT32      NumBEntries
  {:on-success  accum += NumBEntries }
} A;
```

```
typedef struct _B(mutable UINT32* expectedB)
{  …  {:on-success  expectedB--} } B;
```

```
typedef struct _C(UINT32 TotalLength,
                  mutable UINT32* accum) {
  UINT32      SizeOfAs;
  A(accum)    As[:byte-size SizeOfAs];
  B(accum)    Bs[:byte-size TotalLength –SizeOfAs – 4]
  {:on-success  return (*accum == 0) }
} C;
```

Generated C code, after verification

- C code aims to be human-readable, human patchable
- Propagates comments from source spec
- Generates predictable descriptive names

Theorems:

- CheckPacket returns true if and only if the bytes in *base contains a valid representation of the format specification for a Packet
- CheckPacket reads no byte of *base more than once
- Mutates at most the out parameters, dataOffset … perPacketInfoLength in a type-correct manner

Insert a call to CheckPacket on attack surface

```
BOOLEAN
CheckPacket(
    uint32_t ___PacketLength,
    uint32_t ___HeaderLength,
    uint32_t *dataOffset,
    uint32_t *dataLength,
    uint32_t *perPacketInfoOffset,
    uint32_t *perPacketInfoLength,
    uint8_t *base,
    uint32_t len);
```

Experience, spec archaeology, …

- Developed specifications for 4 core message formats for various virtualized device
  - Ultra-high value scenarios: Security bugs here are catastrophic for the entire cloud
  - Layered protocols, with incremental parsing
  - Many more to come

- Totaling around 6000 lines of specification in 3d
  - Automatically generating ~30KLOC of verified C code
  - Working on a clang-based frontend to better integrate 3d specs with C headers

- Highly performance sensitive in certain scenarios
  - Target: Less than 2% measured performance overhead
  - Result: Overhead is unmeasurable
    - In some cases, our code is more efficient, since we can aggressively avoid copies that were previously incurred due to defenses against double fetches

- Main challenge: Discovering a specification
  - Proprietary specs, intentionally divergent from official standards from which they were derived
  - Backward compatibility & complex testing matrices

Using EverParse with Verified F* Applications

TLS (miTLS)

- Verified TLS secure channel with formal security model (IEEE S&P 2017)
- Memory-safe, functionally correct, secure
- Handshake verification in progress
- USENIX Security 2019: TLS handshake message formats

QUIC (EverQUIC)

- Verified QUIC record layer with formal security model
- Memory-safe, functionally correct, side-channel resistant, secure
- IEEE S&P 2021 (to appear): QUIC packet format

DICE/RIoT (DICE*)

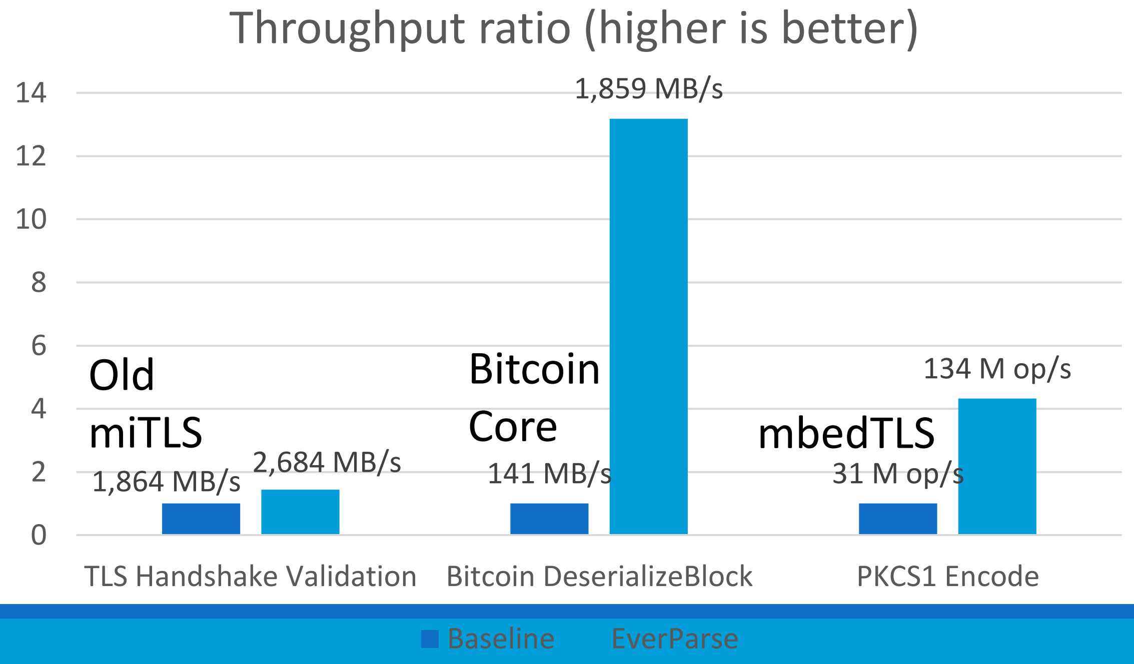
- Verified measured boot for embedded devices (secured boot with measurements)
- Memory-safe, functionally correct, side-channel resistant
- Submitted: ASN.1 X.509 certificates

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



Parser and Serializer Specifications

```
let parser (t: Type) = (p: bytes → option (t × ℕ) {
  ∀ (b1 b2: bytes) . match p b1, p b2 with
  | Some (x1, len1), Some (x2, len2) ->
    x1 = x2 ==> slice b1 0 len1 = slice b2 0 len2
  | _ -> True
})
```

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

With parser refinement:

- Injectivity

And more:

- Consumption bounds
- Strong prefix property
- Etc.

Controlled by metadata

Validators

```
type validator (#t: Type) (p: parser t) =
  (b: bytes) →
  (res: bool {
    res = true ⇔ Some? (p b)
  })
```

Not compositional!

Validators

```
type validator (#t: Type) (p: parser t) =
  (b: bytes) →
  (res: option nat {
    match p b with
    | None → res = None
    | Some (_, consumed) → res = Some consumed
  })
```

What are bytes in C?

Validators

```
type validator (#t: Type) (p: parser t) =
  (b: buffer UInt8.t) →
  (len: UInt32.t { len = length b }) →
  ST (option UInt32.t)
  (requires λ mem → live mem b)
  (ensures λ mem res mem' →
    modifies loc_none mem mem' /\
    match p (as_seq mem b), res with
    | None, None → True
    | Some (_, consumed), Some consumed' → consumed' = consumed
    | _ → False
  )
```

Validators

```
type validator (#t: Type) (p: parser t) =
  (b: buffer UInt8.t) →
  (len: UInt32.t { len = length b }) →
  ST (option UInt32.t)
  (requires λ mem → live mem b)
  (ensures λ mem res mem' →
    modifies loc_none mem mem' /\
    match p (as_seq mem b), res with
    | None, None → True
    | Some (_, consumed), Some consumed' → consume
    | _ → False
  )
```

Precondition

Postcondition

Validators

```
type validator (#t: Type) (p: parser
  (b: buffer UInt8.t) →
  (len: UInt32.t { len = length b })
  ST (option UInt32.t)
  (requires λ mem → live mem b)
  (ensures λ mem res mem' →
    modifies loc_none mem mem' /\
    match p (as_seq mem b), res with
    | None, None → True
    | Some (_, consumed), Some consumed' →
    | _ → False
  )
```

F* abstraction for uint8_t *

Memory safety: b is not a dangling pointer

Proof model of the buffer contents

Memory footprint: what is modified, deallocated, etc.

Example F* Parser specification

```
type employee = {
  name : employee_name;
  salary : UInt16.t;
}

type employe

let employee'_pa
  parse_pair employee_name_parser uint16_parser

let rewrite_employee (x: employee') : employee =
  let (name, salary)
  { name =

let employee_pa
  parse_rewrite employee'_parser rewrite_employee
```

```
struct {
    employee_name name;
    uint16 salary;
} Employee;
```

parse_pair: parser t1 → parser t2 → parser (t1 * t2)

parse_rewrite: parser t1 → (t1 → t2) → parser t2

Generated F* Validator implementation

```
let employee'_validator : validator employee'_parser =
  validate_pair employee_name_validator uint16_validator

let employee_validator : validator employee_parser  =
  validate_rewrite employee'_validator rewrite_employee
```

```
type employee' = (employee_name × UInt16.t)

let employee'_parser : parser employee' =
  parse_pair employee_name_parser uint16_parser

let rewrite_employee (x: employee') : employee =
  let (name, salary) = x in
  { name = name; salary = salary; }

let employee_parser : parser employee =
  parse_rewrite employee'_parser rewrite_employee
```

Generated C code for Validator

```
type slice = {
  base: buffer UInt8.t;
  len: UInt32.t { len ≤ length b /\ len ≤ max_length };
}

let employee'_validator : validator employee_parser =
  validate_pair employee_name_validator uint16_validator

let employee_validator : validator clientHello_parser  =
  validate_rewrite employee'_validator rewrite_employee
```

```
typedef struct {
  uint8_t *base;
  uint32_t len;
} LowParse_Slice_slice;

uint32_t Employee_employee_validator(LowParse_Slice_slice input, uint32_t pos)
{
  uint32_t pos1 = Employee_name_employee_name_validator(i
  if (pos1 > LOWPARSE_LOW_BASE_VALIDATOR_MAX_LENGTH)
    return pos1;
  else if (input.len - pos1 < (uint32_t)2U)
    return LOWPARSE_LOW_BASE_VALIDATOR_ERROR_NOT_ENOUGH_DATA;
  else
    t32_t)2U;
```

uint16_validator inlined

validate_pair and
validate_rewrite inlined

Bang for the buck: Focus on parsing, protect the attack surface