LangSec @ IEEE S&P 2023
Keynote
May 25th, 2023

eVerpArse

Secure Binary Data Parsers for Everyone

Tahina Ramananandro
Research in Software Engineering
Microsoft Research Redmond

Microsoft

RiSE

everest
Verified End-to-End Secure Transport

EverParse: A Wide Collaboration

Microsoft Research

- Barry Bond
- Antoine Delignat-Lavaud
- Cédric Fournet
- Guido Martinez
- Jonathan Protzenko
- Tahina Ramananandro
- Aseem Rastogi
- Irina Spiridonova
- Nikhil Swamy

Microsoft Product Groups

- Justin Campbell
- Omar Cardona
- Mike Ebersol
- Arti Gupta
- Dmitry Malloy
- A. Malysh
- Michael Tang
- Khoa To
- Juan Vazquez Pongiluppi

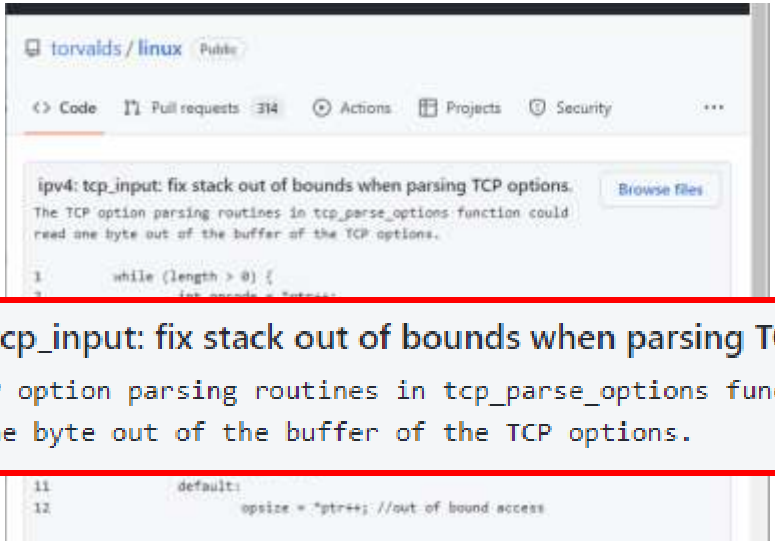
External Contributors and Former Interns

- Tej Chajed
- Nadim Kobeissi
- Haobin Ni

Secure Parsing is Critical

- Improper input validation = MITRE 2020 Top #3, 2021/22 Top #4 most dangerous CVE software weakness

- Still a thing today in widely-used >30-year-old formats
  - Linux TCP parsing bug fix as late as 2019
  - Windows 10 Bad Neighbor (ICMPv6, 2020)



ipv4: tcp_input: fix stack out of bounds when parsing TCP options.

```
The TCP option parsing routines in tcp_parse_options function could
read one byte out of the buffer of the TCP options.
```

A remote code execution vulnerability exists when the Windows TCP/IP stack improperly handles ICMPv6 Router Advertisement packets. An attacker who successfully exploited this vulnerability could gain the ability to execute code on the target server or client.

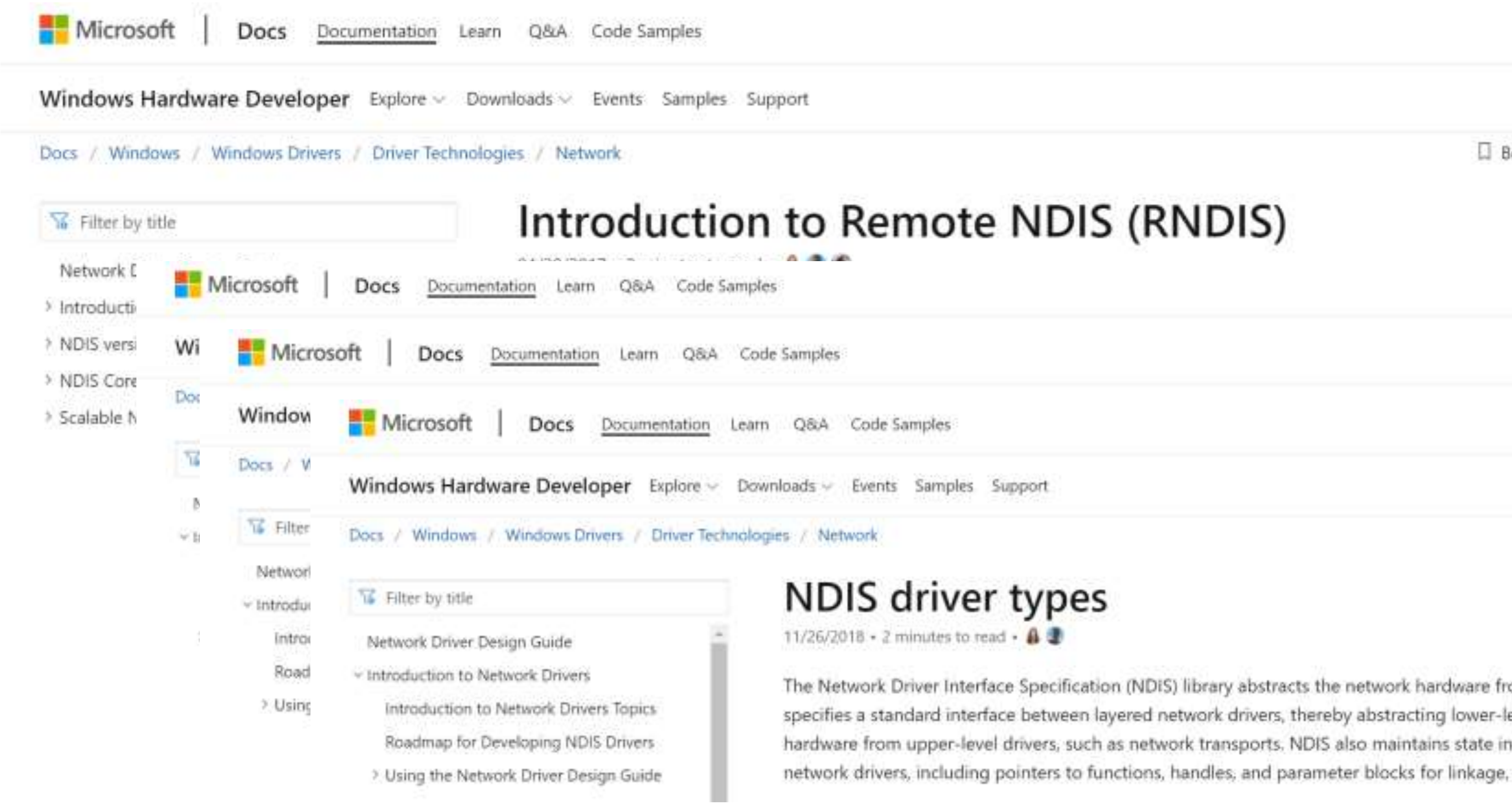
To exploit this vulnerability, an attacker would have to send specially crafted ICMPv6 Router Advertisement packets to a remote Windows computer.

The update addresses the vulnerability by correcting how the Windows TCP/IP stack handles ICMPv6 Router Advertisement packets.

Many legacy formats and parsers remain

- Designed for:
  - Compactness
  - ABI compatibility
  - Mmap'able

- Serialization:
  - memcpy
- Parsing:
  - reinterpret_cast<T> . validate



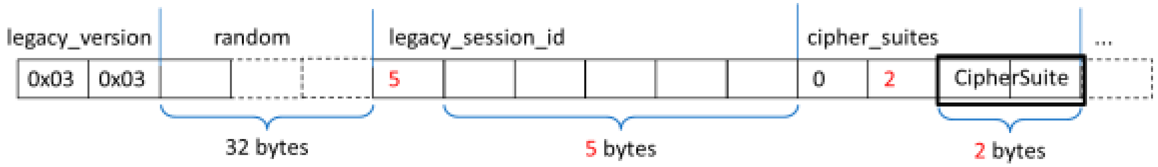
Standardized formats have their challenges too

- Wire formats prescribed by RFCs
  - In a semi-formal notation
  - Or in other notations like ASN.1
- Are the formats well-designed?
- Are their parsers and serializers correctly implemented?

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

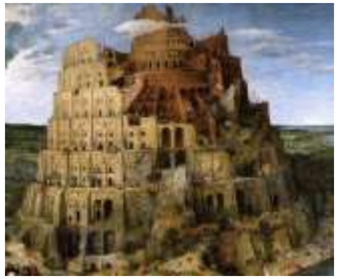
```
uint16 ProtocolVersion; opaque Random[32]; uint8 CipherSuite[2]

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;
```



Handwritten parsing still around

- Handwritten C/C++ code
  - Performance, deployability (e.g. OS kernel), legacy
- Bratus et al. (Usenix Mag. 2017), LangSec:
  - "Roll your own crypto" considered harmful
  - "Roll your own parsers" also should be



- Ongoing push for automatically generated parsers
  - ProtocolBuffers, FlatBuffers, Cap'n Proto,...
  - But those libraries choose the data formats
  - **What about formats dictated by external constraints?** (TCP, ICMP...)

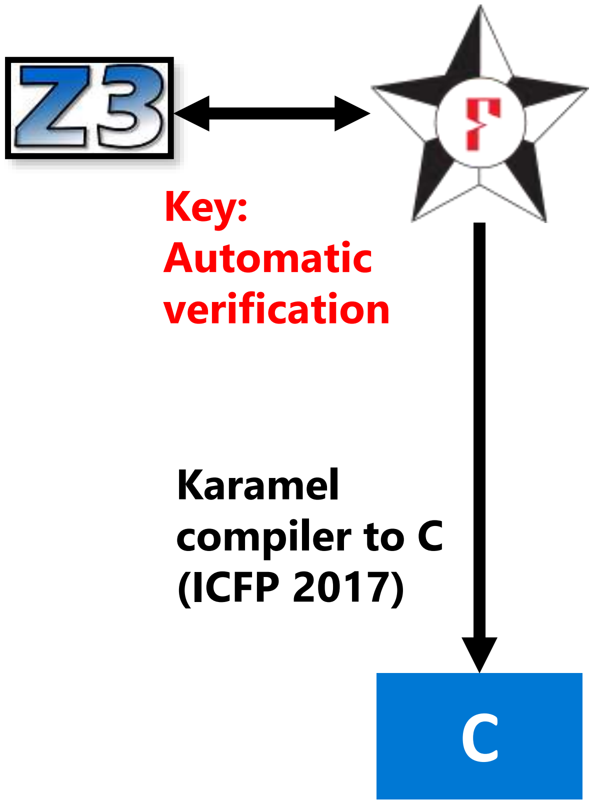
What we advocate: The EverParse Manifesto

- Identify families of security-critical data formats, and design **format specification languages** to support them
- Implement and use provably correct low-level **code generators** (producing C, Wasm, etc.) to produce format manipulation code: parsing, serializing, reading…
- **Integrate within larger applications:** both legacy/unverified, and fresh/verified

What we advocate: The EverParse Manifesto

- Identify families of security-critical data formats, and design **format specification languages** to support them
- Implement and use provably correct low-level **code generators** (producing C, Wasm, etc.) to produce format manipulation code: parsing, serializing, reading…
- **Integrate within larger applications:** both legacy/unverified, and fresh/verified
  - In this talk, focus on Microsoft Hyper-V network virtualization
  - But there are many others: ELF for eBPF, ASN.1, CBOR for DICE, TLS, QUIC, etc.

Our Vehicle: F*, a proof-oriented language for verified low-level programming



F* implementation and specification

```
let multiply_by_9 (a:uint32) : Pure uint32
  (requires 9 * a <= MAX_UINT_32)
  (ensures λ result -> result == 9 * a)
  =
  let b = a << 3ul in
  a + b
```

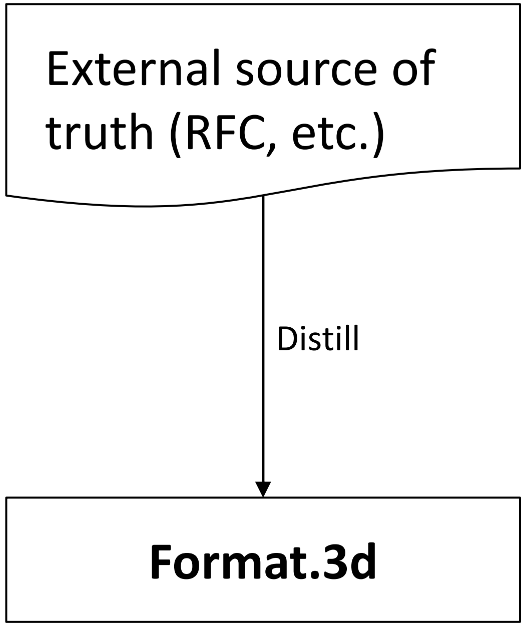
Efficient C implementation

Verification imposes no runtime performance overhead

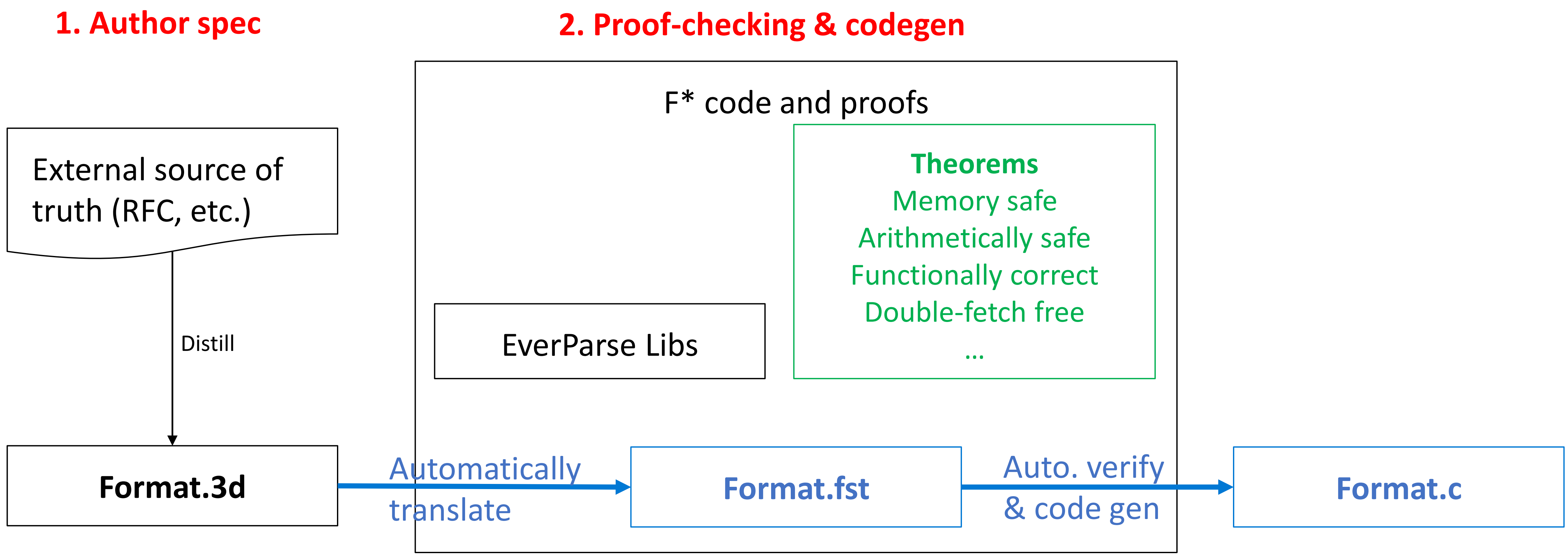
```
uint32_t multiply_by_9(uint32_t a)
{
  uint32_t b = a << (uint32_t)3;
  return a + b;
}
```

Step 1: Figure out the data format specification

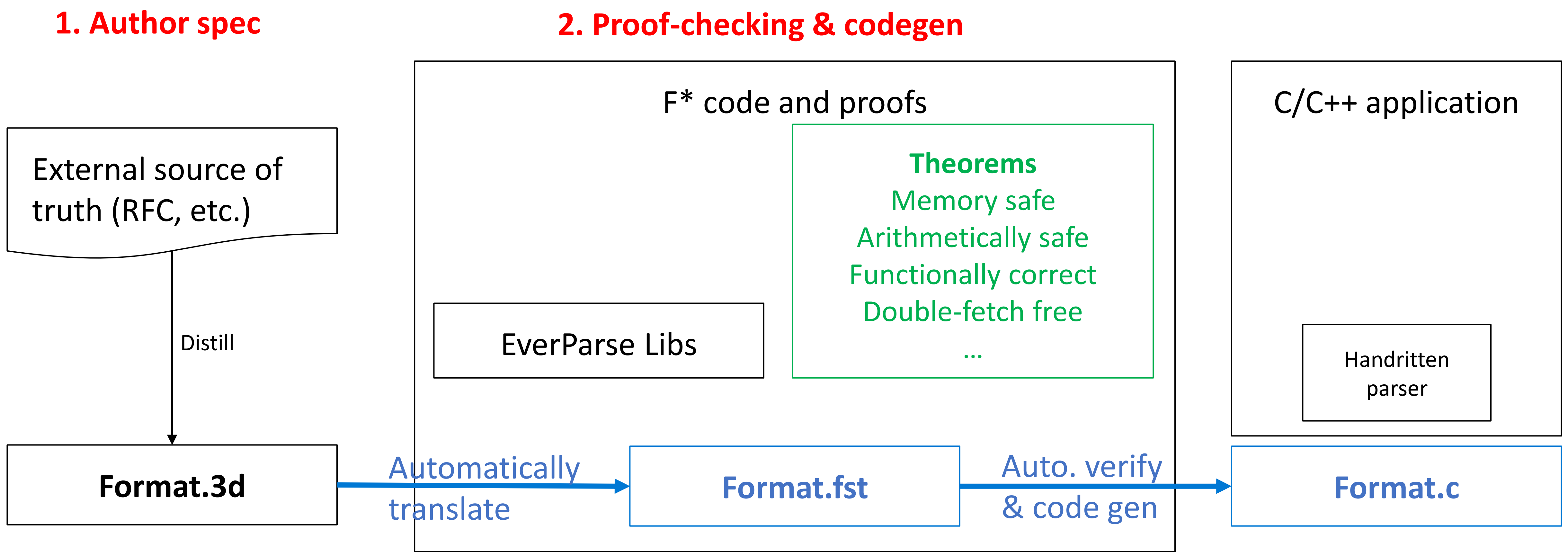
1. Author spec



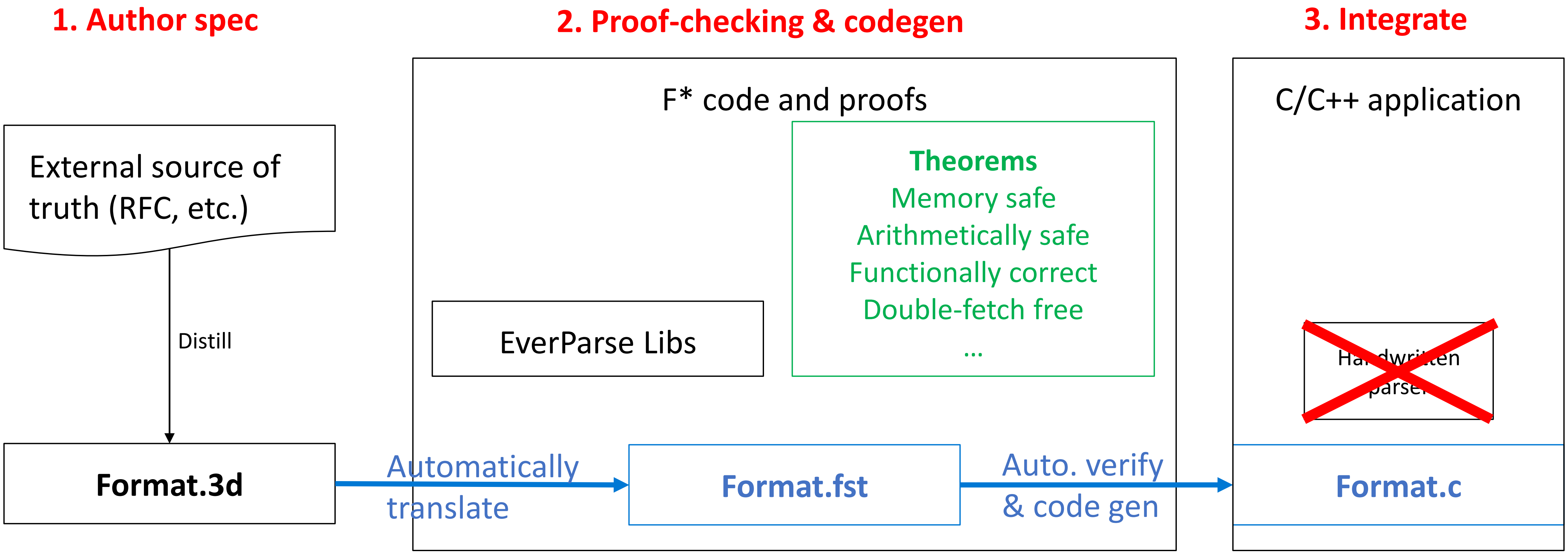
Step 2: Generate and Verify Code



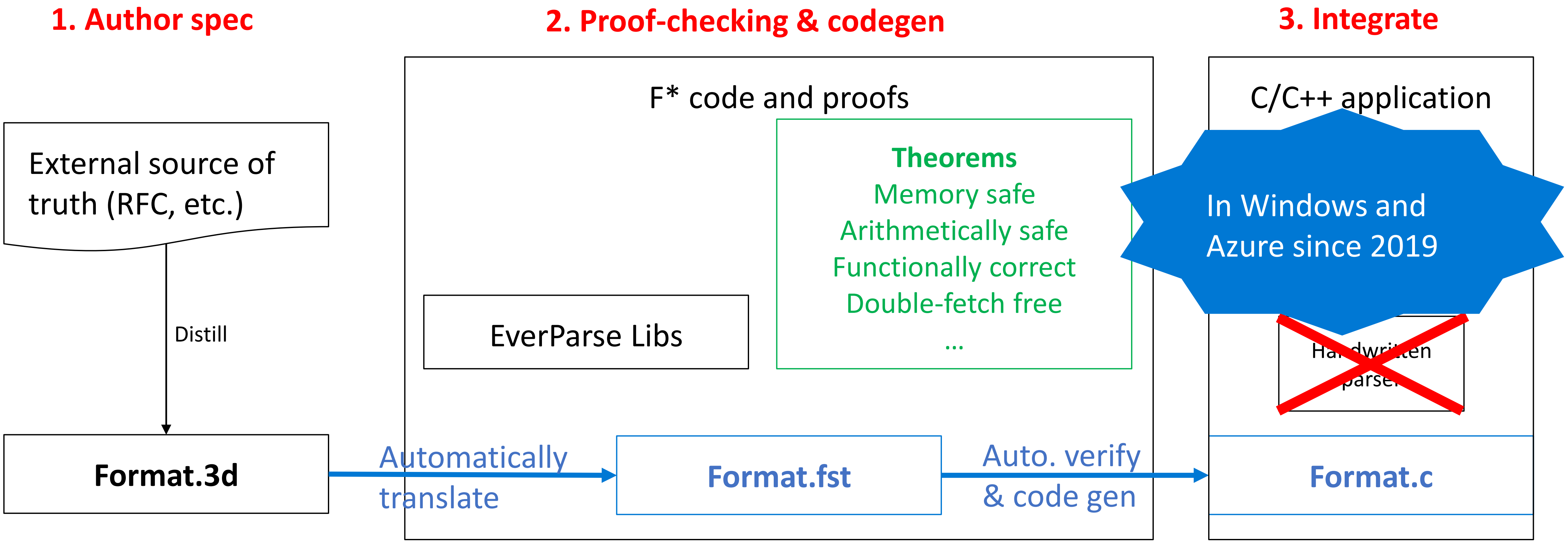
Step 3: Deploy and Maintain the Code



Step 3: Deploy and Maintain the Code



Step 3: Deploy and Maintain the Code



EverParse Guarantees

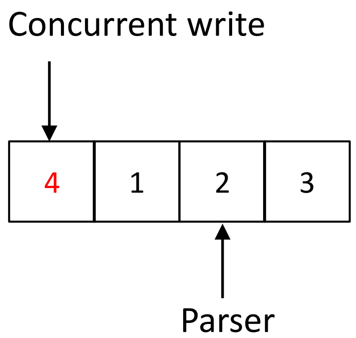
- Memory safety: no buffer overrun
- Arithmetic safety: no integer overflow

```
uint32_t fld_offset = input[current];
uint32_t fld = input[current+offset];
```

Missing checks for integer/buffer overflows

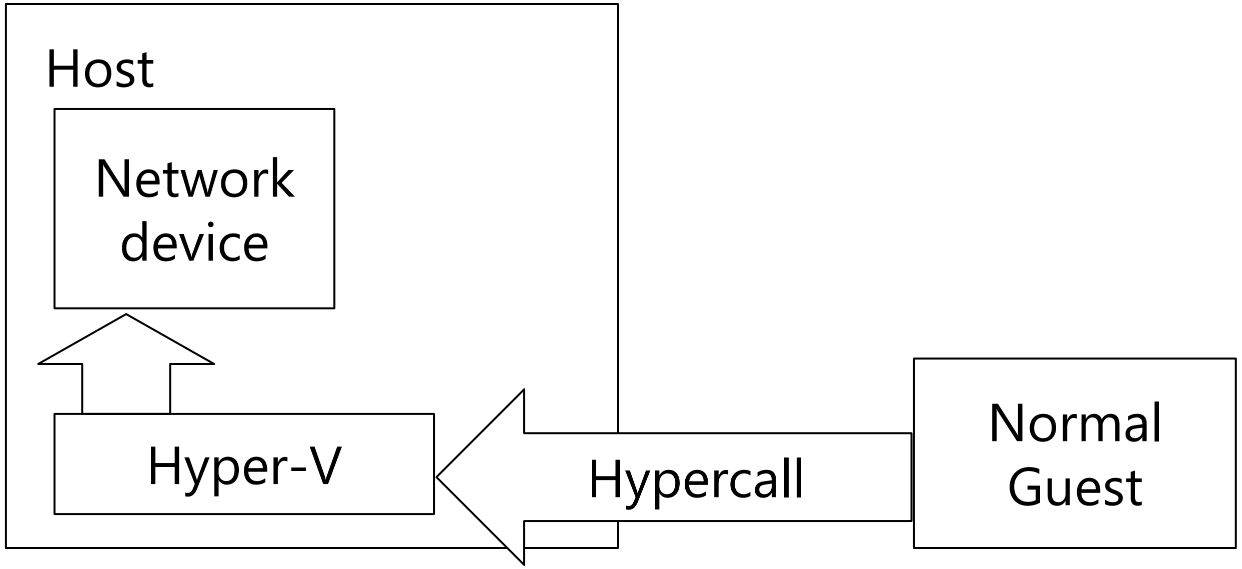
- Functional correctness:
  - All ill-formed packets are rejected
  - Every valid packet is accepted

- Double-fetch freedom: no “time-of-check to time-of-use” bugs
  - No exclusive read access to the input buffer
  - Solution: Read each byte at most once
  - Validation on a “logical snapshot” of the input data



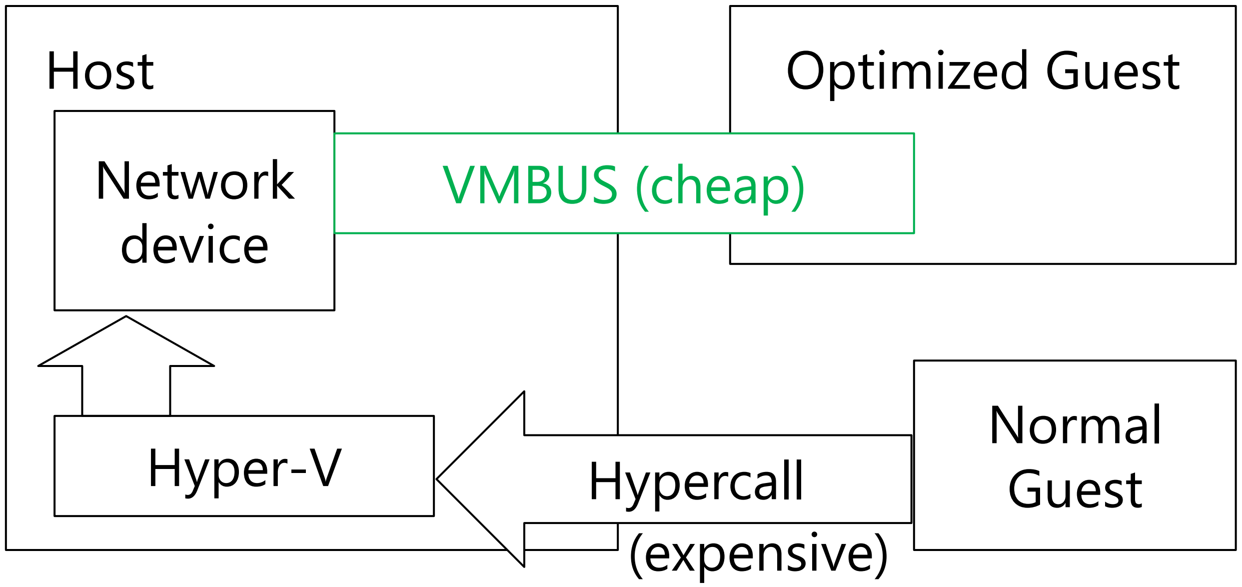
Microsoft Hyper-V Network Virtualization

- Hyper-V: Hypervisor for Windows 10, 11, and all Azure Cloud
- vSwitch: Dispatches network packets from/to guests



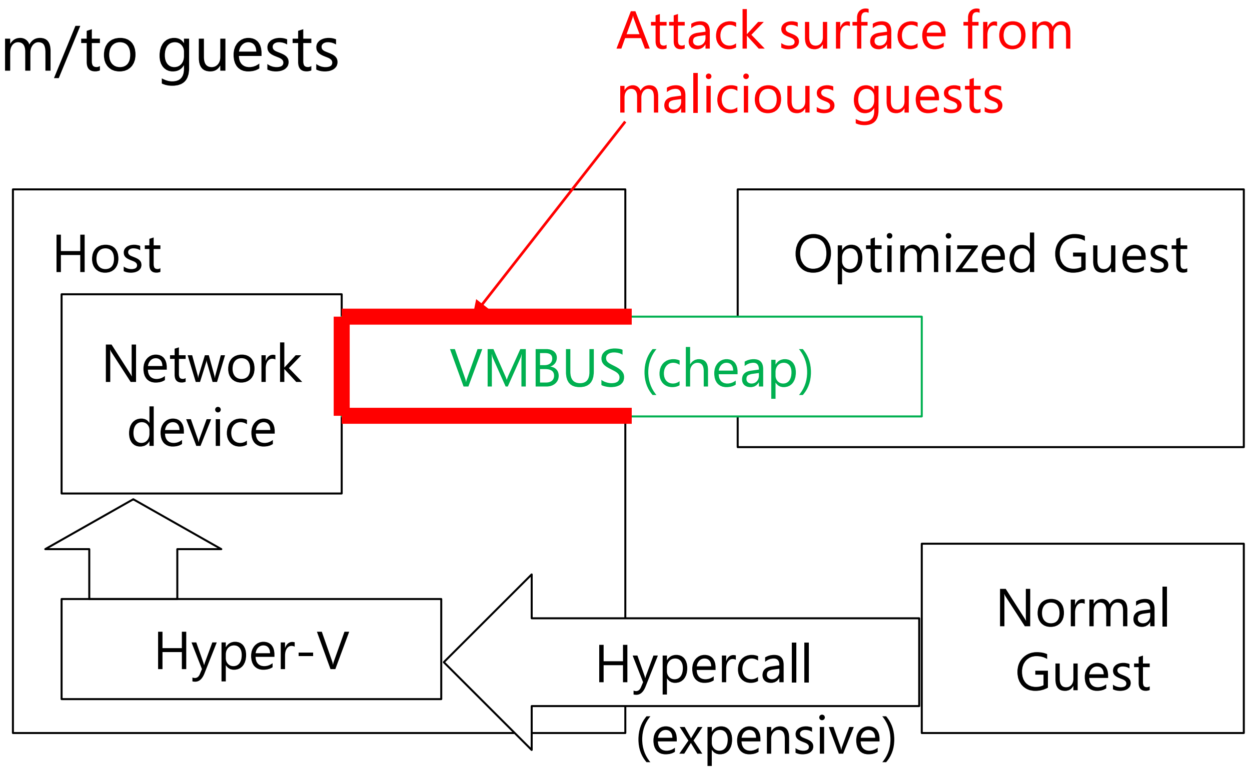
Microsoft Hyper-V Network Virtualization

- Hyper-V: Hypervisor for Windows 10, 11, and all Azure Cloud
- vSwitch: Dispatches network packets from/to guests
- Some guest-side optimizations to give some direct hardware access (VMBUS), bypassing a hypercall



Microsoft Hyper-V Network Virtualization

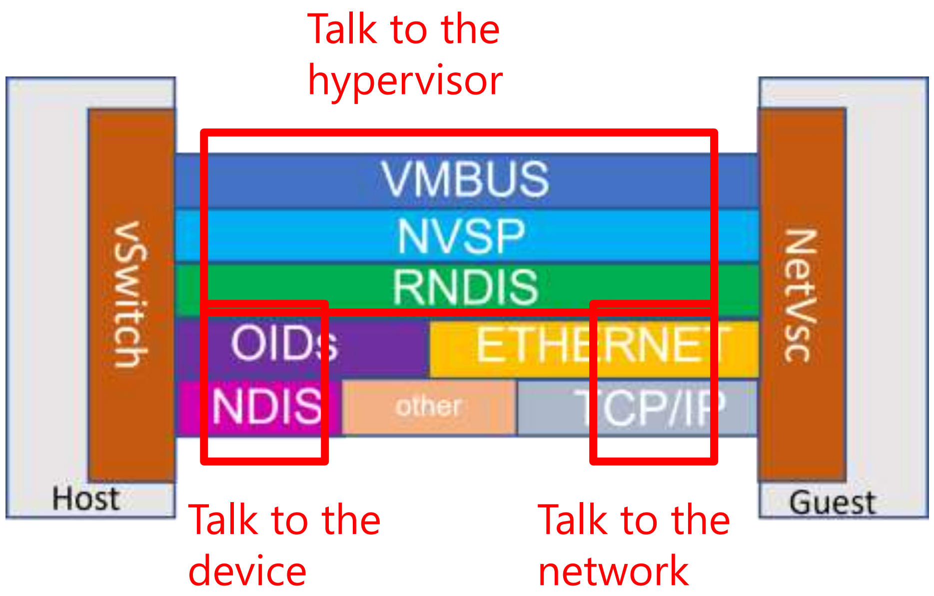
- Hyper-V: Hypervisor for Windows 10, 11, and all Azure Cloud
- vSwitch: Dispatches network packets from/to guests
- Some guest-side optimizations to give some direct hardware access (VMBUS), bypassing a hypercall
- Need to protect against attacks from network or malicious guests crafting ill-formed packets to break isolation / gain host access



Hyper-V vSwitch with EverParse

- **Now in Windows 10, 11, and Azure Cloud**: Every network packet passing through Hyper-V is validated by EverParse formally verified code

- NVSP, RNDIS, OIDs and NDIS
  - Some of which are proprietary
  - Other formats (TCP, etc.) in progress
- 6K lines of specification
  - 137 structs, 22 unions, 30 enum types
- Verified in 82 s
- Generated 30K C code



A multi-year (since summer 2019), multi-org effort

Research Team

Product Team

Testing Team

Security Team

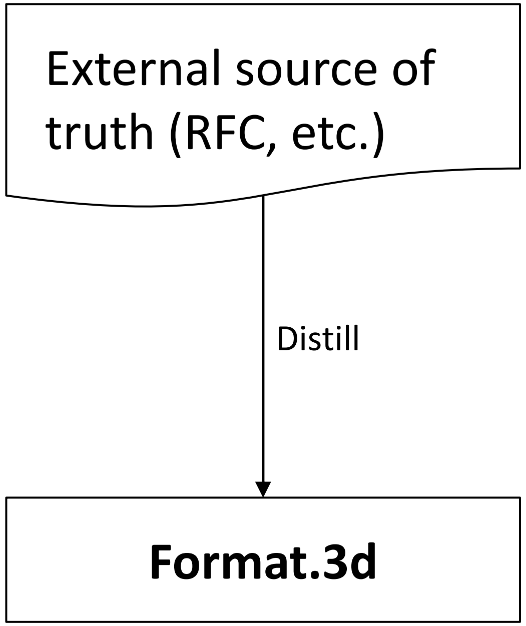
Step 0: Gather Requirements

Research Team Product Team

- Parsing actions
- Double-fetch freedom
- <2% perf overhead
- Generated C code quality (guidelines, etc.)

Step 1: Figure out the data format specification

1. Author spec



Step 1: Figure out the data format specification

Research Team Product Team Testing Team

- Some protocols have no pre-existing specs
- Some proprietary extensions
- Backward compatibility
- Complex testing matrices

- We introduce 3D, a specification language for Dependent Data Descriptions
  - Syntax close to C data types to flatten learning curve

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

```
typedef struct _TCP_HEADER
{
  …
  UINT16 CWR:1;  UINT16 ECE:1;  UINT16 URG:1;  UINT16 ACK:1;
  UINT16 PSH:1;  UINT16 RST:1;  UINT16 SYN:1;  UINT16 FIN:1;  …
  URGENT_PTR UrgentPointer;


  OPTION          Options   [];
  UINT8           Data      [];
} TCP_HEADER;
```

```
typedef union _OPTION_PAYLOAD {



       all_zeros EndOfList;


       unit Noop;
     …
} OPTION_PAYLOAD;

typedef struct _OPTION {
    UINT8 OptionKind;
    OPTION_PAYLOAD
          OptionPayload;
} OPTION;
```

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

Augmenting C data types with value constraints,

```
typedef struct _TCP_HEADER
{
  …
  UINT16 CWR:1;  UINT16 ECE:1;  UINT16 URG:1;  UINT16 ACK:1;
  UINT16 PSH:1;  UINT16 RST:1;  UINT16 SYN:1;  UINT16 FIN:1;  …
  URGENT_PTR UrgentPointer  {UrgentPointer == 0 || URG == 1 } ;


  OPTION          Options   [];
  UINT8           Data      [];
} TCP_HEADER;
```

```
typedef union _OPTION_PAYLOAD {


    all_zeros EndOfList;


    unit Noop;
   …
} OPTION_PAYLOAD;

typedef struct _OPTION {
   UINT8 OptionKind;
   OPTION_PAYLOAD
         OptionPayload;
} OPTION;
```

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

Augmenting C data types with value constraints, variable-length structures

```
typedef struct _TCP_HEADER(UINT32 SegmentLength)
{
  …
  UINT16 CWR:1;  UINT16 ECE:1;  UINT16 URG:1;  UINT16 ACK:1;
  UINT16 PSH:1;  UINT16 RST:1;  UINT16 SYN:1;  UINT16 FIN:1;  …
  URGENT_PTR UrgentPointer  {UrgentPointer == 0 || URG == 1 } ;


  OPTION          Options   [:byte-size (DataOffset * 4) - sizeof(this)];
  UINT8           Data      [SegmentLength - (DataOffset * 4)];
} TCP_HEADER;
```

```
typedef union _OPTION_PAYLOAD {



       all_zeros EndOfList;


       unit Noop;
     …
} OPTION_PAYLOAD;

typedef struct _OPTION {
    UINT8 OptionKind;
    OPTION_PAYLOAD
          OptionPayload;
} OPTION;
```

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

Augmenting C data types with value constraints, variable-length structures, value-dependent unions

```
typedef struct _TCP_HEADER(UINT32 SegmentLength)
{
  …
  UINT16 CWR:1;  UINT16 ECE:1;  UINT16 URG:1;  UINT16 ACK:1;
  UINT16 PSH:1;  UINT16 RST:1;  UINT16 SYN:1;  UINT16 FIN:1;  …
  URGENT_PTR UrgentPointer  {UrgentPointer == 0 || URG == 1 } ;


  OPTION(SYN==1)  Options   [:byte-size (DataOffset * 4) - sizeof(this)];
  UINT8           Data      [SegmentLength - (DataOffset * 4)];
} TCP_HEADER;
```

```
casetype _OPTION_PAYLOAD
  (UINT8 OptionKind, Bool MaxSegSizeAllowed) {
  switch(OptionKind)  {
    case OPTION_KIND_END_OF_OPTION_LIST:
      all_zeros EndOfList;
    case OPTION_KIND_NO_OPERATION:
      unit Noop;
    …
}} OPTION_PAYLOAD;

typedef struct _OPTION(Bool MaxSegSize) {
    UINT8 OptionKind;
    OPTION_PAYLOAD(OptionKind, MaxSegSize)
          OptionPayload;
} OPTION;
```

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

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

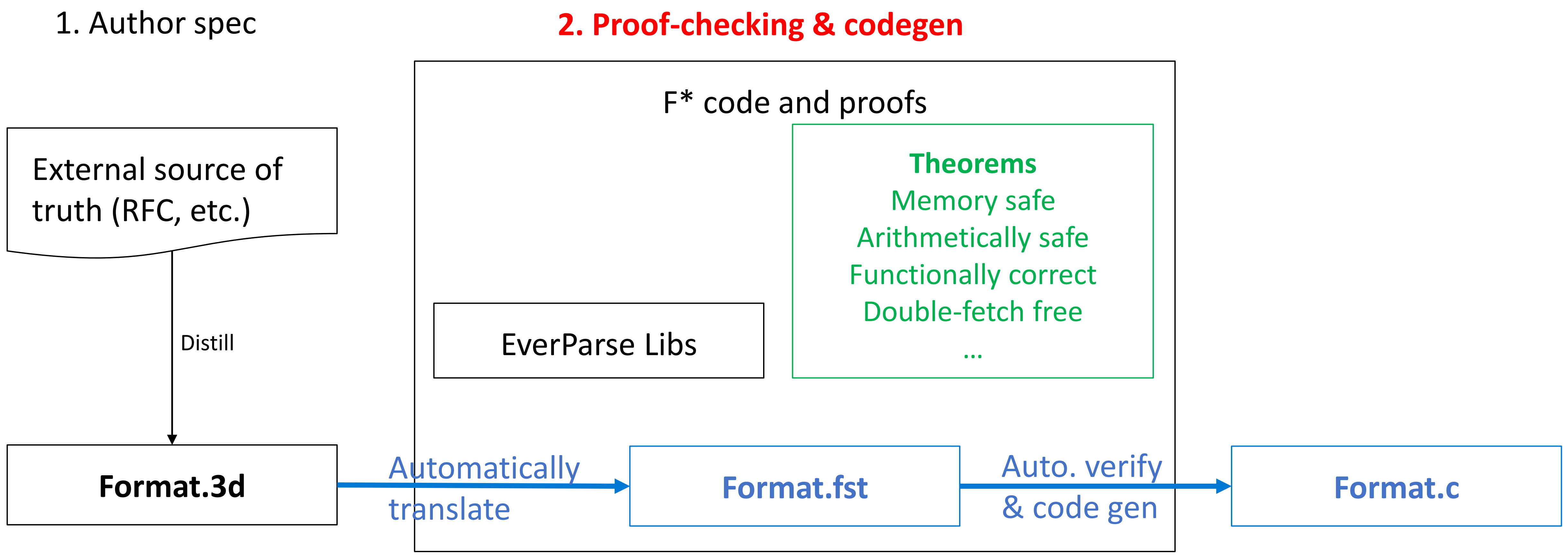
```
typedef struct _TCP_HEADER(UINT32 SegmentLength, mutable URGENT_PTR *Dst)
{
  …
  UINT16 CWR:1;  UINT16 ECE:1;  UINT16 URG:1;  UINT16 ACK:1;
  UINT16 PSH:1;  UINT16 RST:1;  UINT16 SYN:1;  UINT16 FIN:1;  …
  URGENT_PTR UrgentPointer  {UrgentPointer == 0 || URG == 1 }
                            {:on-success *Dst = UrgentPointer; };

  OPTION(SYN==1)  Options   [:byte-size (DataOffset * 4) - sizeof(this)];
  UINT8           Data      [SegmentLength - (DataOffset * 4)];
} TCP_HEADER;
```

```
casetype _OPTION_PAYLOAD
  (UINT8 OptionKind, Bool MaxSegSizeAllowed) {
  switch(OptionKind)  {
    case OPTION_KIND_END_OF_OPTION_LIST:
      all_zeros EndOfList;
    case OPTION_KIND_NO_OPERATION:
      unit Noop;
    …
}} OPTION_PAYLOAD;

typedef struct _OPTION(Bool MaxSegSize) {
    UINT8 OptionKind;
    OPTION_PAYLOAD(OptionKind, MaxSegSize)
          OptionPayload;
} OPTION;
```

Step 2: Generate and Verify Code



Step 2: Generate and Verify Code

- 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);
```

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

```
let op = fst:UINT32 & snd:UINT32{fst <= snd}
```

Generated F* data type

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

Generated F* data type

```
let op = fst:UINT32 & snd:UINT32{fst <= snd}
```

Generated parser specification in F*

```
let parser t = b:bytes -> option (x:t & nat) { ... and some conditions here ... }
let parse_op : parser op =
  p_dep_pair p_u32 (fun fst ->
  p_refine p_u32 (fun snd ->
  if fst <= snd then p_ret (fst, snd) else fail))
```

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

Generated F* data type

```
let op = fst:UINT32 & snd:UINT32{fst <= snd}
```

```
let parser t = b:bytes -> option (x:t & nat) { ... and some conditions here ... }
let parse_op : parser op =
  p_dep_pair p_u32 (fun fst ->
  p_refine p_u32 (f
  if fst <= snd then p_                 ) else fail))
```

Generated parser specification in F*

Combinators

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

Generated F* data type

```
let op = fst:UINT32 & snd:UINT32{fst <= snd}
```

```
let parser t = b:bytes -> option (x:t & nat) { ... and some conditions here ... }
let parse_op : parser op =
  p_dep_pair p_u32 (fun fst ->
  p_refine p_u32 (f
  if fst <= snd then p_               ) else fail))
```

Generated parser specification in F*

Combinators

```
let validator (p:parser t) = input:array U8.t -> pos:U32.t -> len:U32.t -> ST errcode
  (requires (* some preconditions, liveness of input etc. *))
  (ensures (* the validator refines the parser spec p *))
let validate_op : validator parse_op =
  v_dep_pair v_u32 (fun fst ->
  v_refine v_u32 (fun snd ->
  if fst <= snd then true else false))
```

Generated validator implementation in Low*, a subset of F* to model C

EverParse at work

User-provided data format description in 3D

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;
```

Generated F* data type

```
let op = fst:UINT32 & snd:UINT32{fst <= snd}
```

Generated parser specification in F*

```
let parser t = b:bytes -> option (x:t & nat) { ... and some conditions here ... }
let parse_op : parser op =
  p_dep_pair p_u32 (fun fst ->
  p_refine p_u32 (f
  if fst <= snd then p_                ) else fail))
```

Combinators

```
let validator (p:parser t) = input:array U8.t -> pos:U32.t -> len:U32.t -> ST errcode
  (requires (* some preconditions, liveness of input etc. *))
  (ensures (* the validator refines the parser spec p *))
let validate_op : validator parse_op =
  v_dep_pair v_
  v_refine v_u32 (f
  if fst <= snd th
```

Combinators implemented in Low*
unfolded by F*/KaRaMeL
to produce C code

plementation in
odel C

EverParse at work

```
typedef struct s_OP { UINT32 fst; UINT32 snd { fst <= snd }; } OP;

let op = fst:UINT32 & snd:UINT32{fst <= snd}

let parser t = b:bytes -> option (x:t & nat) { ... and some con     ons here ... }
let parse_op : parser op =
  p_dep_pair p_u32 (fun fst ->
  p_refine p_u32 (fun snd ->
  if fst <= snd then p_ret (fst, snd)

let validator (p:parser t) = in                                       errcode
  (requires (* some pre
  (ensures (* the validato
let validate_op : validator pa
  v_dep_pair v_u32 (fun fs
  v_refine v_u32 (fun snd ->
  if fst <= snd then true else fals
```

3 times the same code!

Hyper-V validator verification and extraction time: 5 min (6K specs → 30K C code)

EverParse: leverage partial evaluation

- We can actually compute the denotational semantics of every 3D specification as an F* value of this type:

```
type dtyp = { t:Type; p:parser t; v:validator p }
```

The EverParse3D strategy:

1. Introduce a 3D AST in F*
2. Interpret this 3D AST 3 times: once for t, once for p, once for v, with interpreters proven **correct once and for all**
3. **Partially evaluate** the validator interpreter, down to validator combinators (aka 1st Futamura projection)
4. Unfold the validator combinators to produce C code as before

With that, Hyper-V parser code generation and verification time drops from 5 min to 1 min 20 s

Generated C code

```c
#include "EverParse.h"

uint64_t
TestValidateSOp(
  uint8_t *Ctxt,
  uint8_t *Input,
  uint64_t InputLength,
  uint64_t StartPosition
)
{
  /* Checking that we have enough space for a UINT32, i.e., 4 bytes */
  BOOLEAN hasBytes0 = (uint64_t)4U <= (InputLength - StartPosition);
  uint64_t positionAfterfst;
  if (hasBytes0)
  {
    positionAfterfst = StartPosition + (uint64_t)4U;
  }
  else
  {
    positionAfterfst =
      EverParseSetValidatorErrorPos(EVERPARSE_VALIDATOR_ERROR_NOT_ENOUGH_DATA,
        StartPosition);
  }
  if (EverParseIsError(positionAfterfst))
  {
    return positionAfterfst;
  }
  uint32_t fst = Load32Le(Input + (uint32_t)StartPosition);
 /* Validating field snd */
  /* Checking that we have enough space for a UINT32, i.e., 4 bytes */
  BOOLEAN hasBytes = (uint64_t)4U <= (InputLength - positionAfterfst);
  uint64_t positionAftersnd_refinement;
  if (hasBytes)
  {
    positionAftersnd_refinement = positionAfterfst + (uint64_t)4U;
  }
  else
  {
    positionAftersnd_refinement =
      EverParseSetValidatorErrorPos(EVERPARSE_VALIDATOR_ERROR_NOT_ENOUGH_DATA,
        positionAfterfst);
  }
  uint64_t positionAfterSOp0;
  if (EverParseIsError(positionAftersnd_refinement))
  {
    positionAfterSOp0 = positionAftersnd_refinement;
  }
  else
  {
    /* reading field_value */
    uint32_t snd_refinement = Load32Le(Input + (uint32_t)positionAfterfst);
    /* start: checking constraint */
    BOOLEAN snd_refinementConstraintIsOk = fst <= snd_refinement;
    /* end: checking constraint */
    positionAfterSOp0 =
      EverParseCheckConstraintOk(snd_refinementConstraintIsOk,
        positionAftersnd_refinement);
  }
  return positionAfterSOp0;
}
```

Generated C code

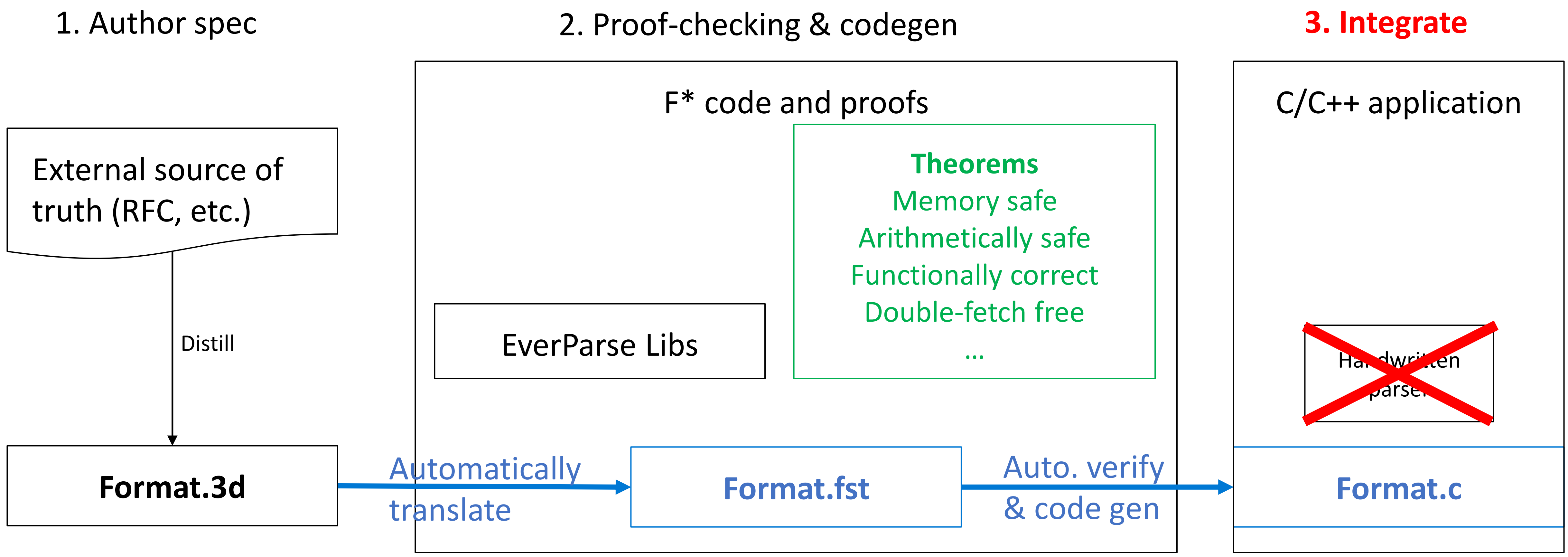
```
#include "EverParse.h"

uint64_t
TestValidateSOp(
  uint8_t *Ctxt,
  uint8_t *Input,
  uint64_t InputLength,
  uint64_t StartPosition
)
{
  /* Checking that we have enough s
  BOOLEAN hasBytes0 = (uint64_t)4
  uint64_t positionAfterfst;
  if (hasBytes0)
  {
    positionAfterfst = StartPosition +
  }
  else
  {
    positionAfterfst =
      EverParseSetValidatorErrorPos(E
        StartPosition);
  }
  if (EverParseIsError(positionAfterfst))
  {
    return positionAfterfst;
  }
  uint32_t fst = Load32Le(Input + (uint32_t)StartPosition);
```

- Combinators unfolded
- Generated code is auditable
  - Readable variable names
  - Comments

```
/* Validating field snd */
 /* Checking that we have enough space for a UINT32, i.e., 4 bytes */
 BOOLEAN hasBytes = (uint64_t)4U <= (InputLength - positionAfterfst);
 uint64_t positionAftersnd_refinement;
 if (hasBytes)
 {
   positionAftersnd_refinement = positionAfterfst + (uint64_t)4U;
 }
 else
 {
ERROR_NOT_ENOUGH_DATA,
sitionAfterfst);
   BOOLEAN snd_refinementConstraintIsOk = fst <= snd_refinement;
   /* end: checking constraint */
   positionAfterSOp0 =
     EverParseCheckConstraintOk(snd_refinementConstraintIsOk,
       positionAftersnd_refinement);
 }
 return positionAfterSOp0;
}
```

Step 3: Deploy and Maintain the Code



Step 3: Deploy and Maintain the Code

Research Team Product Team

Productivity Improvements

- EverParse now part of the Windows build environment (incl. Z3, F*, Karamel)
- Critical to meet product deadlines:
  - saves code writing cost
  - more focused security reviews

*"EverParse was **instrumental** in our ability to implement the product. The scope and complexity of protocols/messages involved could not have been addressed manually to meet the project's timelines."*

– Omar Cardona, engineering lead of the Network Virtualization team

Performance

Generated code is fast…

- Our code passed internal performance regression testing, imposing **less than 2% cycles/byte** overhead
- In some cases, our code is more efficient by virtue of eliminating unneeded copies

… thanks to careful design

- Validators operate in-place
- Validators only read data at most once: client code no longer needs to copy data before validating it
- Layered specifications + one single pass = fail early

Security Evaluation

Security Team

- Spec audited, security team wrote unit tests
- vSwitch code fuzzers stopped finding bugs:
  - Malformed packets properly rejected by our parsers
  - Helped refocus fuzzers to functionality fuzzing

Active Maintenance

+ Other teams (servicing, etc.)

Product Team Testing Team Security Team

Up and running for 3 years already:

- Product teams change the specs as they integrate new features
- Backport to older product versions
- Generated C code checked in the product repo to aid other teams' understanding

Using EverParse with Verified F* Applications

Using EverParse with Verified F* Applications

TLS

- Verified TLS secure channel with formal security model
- Handshake message formats
- Verified non-malleable (unique binary representation, for signature-based authentication)
- USENIX Security 2019

QUIC

- Verified QUIC record layer with formal security model
- Parsing and serialization proven constant-time for side-channel resistance
- IEEE S&P 2021

ASN.1, DICE

- Verified measured boot for embedded devices (secured boot with measurements)
- Right-to-left serialization for length-prefixed data
- ASN.1 X.509 certificate subset
- DICE: USENIX Security 2021
- Full ASN.1 spec language: ACM CPP 2023
- CBOR in progress

ASN.1 DER

- ASN.1 is a standardized data format description language since 1984
  - Used for security-critical data in communication
  - e.g., X.509 standard for public-key certificates

- DER (Distinguished Encoding Rules) is a set of translation rules between high-level values and binary representations

ASN.1 Declaration

```
let fooQuotaion ::= SEQUENCE {
  serialNumber   INTEGER,
  quote          IA5String
}
```

A concrete value

```
let foo ::= {
  serialNumber = 42,
  quote = "To be or not to be"
}
```

DER Encoding

```
30 17 // SEQUENCE, 23 bytes
   02 01 2A   // INTEGER
   16 12 546F206265206F72…
        // ASCII STRING
```

- DER is designed to be *unambiguous* and *non-malleable* for security
  - Unambiguous: any binary string represents a single value or none
  - Cf. Bleichenbacher PKCS1 signature forgery,
    Bitcoin consensus bug avoided by adoption of DER in Bitcoin Improvement Proposal 66

The ASN.1* Recipe (ACM CPP 2023)

Domain-specific data format language + Verified compiler = Provably-correct parsers

Users required for defining new data formats

The compiler is implemented and verified once for all

F*

ASN1*

```
let fooQuotation : declaration = asn1_sequence [
     "serialNumber" *^ (PLAIN ^: asn1_integer);
     "quote"        *^ (PLAIN ^: asn1_ia5string)
]
```

ASN1*_as_parser

```
let asn1_as_parser (d : declaration)
: asn1_strong_parser (asn1_as_type d) =
  match d with
  | ILC id c → parse_ILC id (content_as_parser c)
  ...
```

everparse

Extract → OCaml Parser → Empirical Evaluation

ASN.1*

- The first formalization of ASN.1 DER
  - **Main theorem:** all well-formed ASN.1 declarations induce non-malleable parsers
- An experimental evaluation of the extracted parsers on practical formats
  - EFF Corpus of 11M X.509 v3 certificates and Certificate Revocation Lists (CRL)
- A significant effort in understanding 400-page ITU spec and distilling it as ~2000 lines of F* (8 person-months)
  - Generates an OCaml reference implementation
  - Low-level C code for serialization available for an X.509 subset used in DICE*, coverage extension in progress

Discussion:
Challenges and Opportunities

More data formats

- Support for recursive formats beyond lists
  - CBOR in progress
  - Stack consumption considerations
- Offsets within data
  - "Table of contents"
  - PDF, VHDX
- Text
  - Growing literature for verified parsing of context-free languages
    (e.g. CoStar, Lasser et al. PLDI 2021)
- …

- Currently supported:
  - Various integer formats
  - Byte size-prefixed data/lists
  - Element count-prefixed lists
  - Zero-terminated strings
  - Unions with read or computed tag
  - Bit fields, with dependencies
  - …

Beyond parsing: towards safe data processing

- Many apps are structured as single-pass transformation on input data to output responses
  - Examples: stream analytics, certificate issuance, network firewall, etc.

- Generalize EverParse to a verified domain-specific language for these applications
  - In-place access and transformations over binary data
  - Cf. Nail (Bangert et al. OSDI 2014), unverified
  - Ongoing experiments: verified iterator pattern on a subset of 3D, with “fold” operations on lists

Specification Discovery

- Even if EverParse automates verification and code generation, significant effort is still necessary to come up with the specification
  - Starting from existing code
  - And/or from hundreds of pages of informal specifications

- Can this effort also be automated? With AI?

Here's a hex dump from WireShark for a TCP packet. Can you parse its fields? 0000 d0 3c 1f 6a 97 2b c8 34 8e 44 1f ac 08 00 45 00 0010 00 3c 1e cc 40 00 80 06 00 00 0a 11 70 42 0a 11 0020 74 b1 d7 dd 1e 00 6b 40 b9 a6 00 00 00 00 a0 02 0030 fe 88 f9 43 00 00 02 04 05 b4 01 03 03 08 04 02 0040 08 0a 89 b1 e8 c9 00 00 00 00

Searching for: **parse TCP packet hex dump**

Generating answers for you...

Sure, I can try to parse the hex dump for you. Here is a possible breakdown of the fields:

- The first 14 bytes are the Ethernet header [1] [2]. They contain:
  - The destination MAC address: d0:3c:1f:6a:97:2b
  - The source MAC address: c8:34:8e:44:1f:ac
  - The EtherType field: 08 00 (indicating IPv4)
- The next 20 bytes are the IPv4 header [1] [2]. They contain:
  - The version field: 4 (indicating IPv4)
  - The header length field: 5 (indicating 20 bytes)
  - The differentiated services field (DSCP and ECN): 00
  - The total length field: 00 3c (indicating 60 bytes)
  - The identification field: 1e cc
  - The flags and fragment offset fields: 40 (indicating no fragmentation) and
    - Flags : Don’t Fragment bit set
    - Fragment offset : zero respectively.

Yes, I know about TCP

can you write down a collection of C structures to describe this packet? Please also describe any length relationships among the fields of the structs

Searching for: **C structures for TCP packet**

Generating answers for you…

Sure, I can try to write down a collection of C structures to describe this packet. Here is a possible example:

```
// Ethernet header structure
struct ethhdr {
    unsigned char h_dest[6]; // destination MAC address
    unsigned char h_source[6]; // source MAC address
    unsigned short h_proto; // EtherType field
};

// IPv4 header structure
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8 ihl:4,
         version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8 version:4,
         ihl:4;
#else
#error "Please fix <asm/byteorder.h>"
#endif
    __u8 tos; // differentiated services field
    __be16 tot_len; // total length field
    __be16 id; // identification field
```

The length relationships among the fields of the structs are as follows:

- The Ethernet header is always 14 bytes long.
- The IPv4 header length is variable, depending on the value of the ihl field. The minimum value is 5, which means 20 bytes. The maximum value is 15, which means 60 bytes. The actual length is ihl * 4 bytes.
- The TCP header length is also variable, depending on the value of the doff field. The minimum value is 5, which means 20 bytes. The maximum value is 15, which means 60 bytes. The actual length is doff * 4 bytes.
- The TCP options are optional and variable in length, depending on the kind and len fields. Some options have a fixed length of one byte (kind only), some have a fixed length of two bytes (kind and len), and some have a variable length (kind, len and data). The total length of all options must be a multiple of four bytes.

I hope this helps you understand how to describe this packet using C structures.

Are you familar with the RNDIS protocol format? It is described on msdn. Can you show me a sample RNDIS header?

Searching for: **RNDIS protocol format msdn**

Generating answers for you...

Yes, I am familiar with the RNDIS protocol format. It is a **Microsoft proprietary protocol** used mostly on top of USB [1]. It provides a **virtual Ethernet link** to most versions of the Windows, Linux, and FreeBSD operating systems [1].

A sample RNDIS header is shown below:

```
typedef struct _RNDIS_MESSAGE {
  union {
    struct {
      UINT32 MessageType;
      UINT32 MessageLength;
      union {
        RNDIS_INITIALIZE_REQUEST InitializeRequest;
        RNDIS_INITIALIZE_COMPLETE InitializeComplete;
        RNDIS_HALT_REQUEST HaltRequest;
        RNDIS_QUERY_REQUEST QueryRequest;
        RNDIS_QUERY_COMPLETE QueryComplete;
        RNDIS_SET_REQUEST SetRequest;
        RNDIS_SET_COMPLETE SetComplete;
        RNDIS_RESET_REQUEST ResetRequest;
        RNDIS_RESET_COMPLETE ResetComplete;
        RNDIS_INDICATE_STATUS IndicateStatus;
        RNDIS_KEEPALIVE_REQUEST KeepaliveRequest;
        RNDIS_KEEPALIVE_COMPLETE KeepaliveComplete;
      } RequestMessage;
    } NdisMessageType;
    struct {
      UINT32 MessageType;
      UINT32 MessageLength;
      UINT32 DataOffset;
      UINT32 DataLength;
      UINT32 OOBDataOffset;
      UINT32 OOBDataLength;
      UINT32 NumOOBDataElements;
      UINT32 PerPacketInfoOffset;
      UINT32 PerPacketInfoLength;
    } PacketMessageType;
  } u ;
} *PRNDIS_MESSAGE ;
```

This header defines a **bus-independent message set** and a description of how this message set operates over the USB bus [2]. The `MessageType` field indicates the type of message, such as `REMOTE_NDIS_PACKET_MSG` for data packets or `REMOTE_NDIS_INITIALIZE_MSG` for initialization messages [3]. The `MessageLength` field indicates the total length of the message in bytes [3]. The other fields vary depending on the message type and contain additional information or data for the message [3].

Learn more: 1. en.wikipedia.org 2. learn.microsoft.com 3. download.microsoft.com +2 more 1 of 15

Can you give me a sample _RNDIS_MESSAGE header in hex?

Sure. Here is a sample _RNDIS_MESSAGE header in hex for a data packet:

```
01 00 00 00 // MessageType = REMOTE_NDIS_PACKET_MSG
28 00 00 00 // MessageLength = 40 bytes
1C 00 00 00 // DataOffset = 28 bytes from the start of the message
0C 00 00 00 // DataLength = 12 bytes
00 00 00 00 // OOBDataOffset = not used
00 00 00 00 // OOBDataLength = not used
00 00 00 00 // NumOOBDataElements = not used
1C 01 C0 DE // PerPacketInfoOffset = not used
AD BE EF CA // PerPacketInfoLength = not used
```

The data packet itself would follow this header and contain the Eth received.

Yes, I also know about less-common formats such as RNDIS

This is a hex dump of what I think is an RNDIS packet. Can you parse its fields fo
00 32 00 00 00 1C 00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 1C 0 CA

Sure. Here is the parsing of the hex dump:

```
04 00 00 00 // MessageType = REMOTE_NDIS_RESET_MSG
32 00 00 00 // MessageLength = 50 bytes
1C 00 00 00 // DataOffset = not used
0C 00 00 00 // DataLength = not used
00 00 00 00 // OOBDataOffset = not used
00 00 00 00 // OOBDataLength = not used
00 00 00 00 // NumOOBDataElements = not used
1C 01 C0 DE // PerPacketInfoOffset = not used
AD BE EF CA // PerPacketInfoLength = not used
```

This is a reset message that requests the Remote NDIS device to perform a soft reset and reinitialize itself. The message length is incorrect, it should be **24 bytes** instead of **50 bytes**. The rest of the fields are not used for this message type and should be set to zero.

3 of 15

Do you know EverParse and its specification language 3D? Can you write a specifiation for a TCP header in 3D?

Searching for: **EverParse 3D**

Generating answers for you...

Yes, I know EverParse and its specification language 3D. EverParse is a tool that lets you write a data format description and automatically produces efficient data validation code formally verified for memory-safety, functional correctness and security [1] [2]. 3D is a frontend for EverParse that enables specifying data formats in a style resembling type definitions in the C programming language [3].

A TCP header can be specified in 3D as follows:

```
// A TCP header consists of 20 bytes of fixed fields followed by optional fields
type tcp_header = {
  // The source port number (16 bits)
  src_port: uint16;
  // The destination port number (16 bits)
  dst_port: uint16;
  // The sequence number (32 bits)
  seq_num: uint32;
  // The acknowledgment number (32 bits)
  // The urgent pointer (16 bits)
  urgent_ptr: uint16;
} where {
   offset >= sizeof(tcp_header) / sizeof(uint32);
   /* ensures that the data offset is at least equal to
      the size of the fixed fields divided by four */
};
```

Learn more: 1. project-everest.github.io 2. microsoft.com +3 more 1 of 15

Yes, I also know about EverParse and 3D

I have a protocol with three different kinds of messages: INIT, START, and STOP. The INIT message has no payload. The START and STOP messages take an uint32 as a payload. Can you describe a message format in 3D for my protocol?

Sure, I can try to describe a message format in 3D for your protocol. One possible way to do it is as follows:

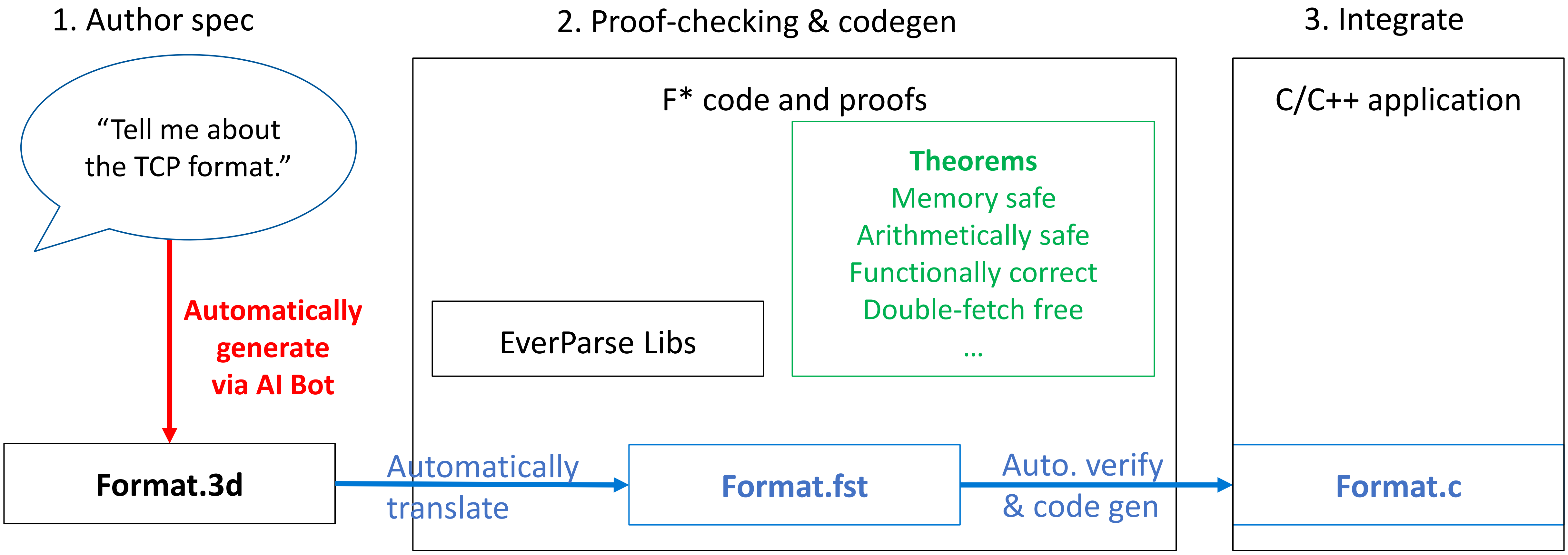
```
// A message consists of a header and an optional payload
type message = {
  // The header consists of a tag and a length
  header: {
    // The tag is an enum that indicates the type of the message
    tag: enum {
      INIT = 0;
      START = 1;
      STOP = 2;
    };
    // The length is an uint8 that indicates the size of the payload in bytes
    length: uint8;
  };
  // The payload is either empty or an uint32 depending on the tag
  payload: switch (header.tag) {
    case INIT:
      unit; // empty payload
    case START:
      uint32; // start value
    case STOP:
      uint32; // stop value
  };
} where {
   header.length == sizeof(payload);
   /* ensures that the length field matches the actual size of the payload */
};
```

Yes, I can generate some 3D spec for your custom format

EverParse + AI?

“Tell me about
the TCP format.”

EverParse + AI?



EverParse + AI?

1. Author spec

"Tell me ab
the TCP form

Form

Integrate

ication

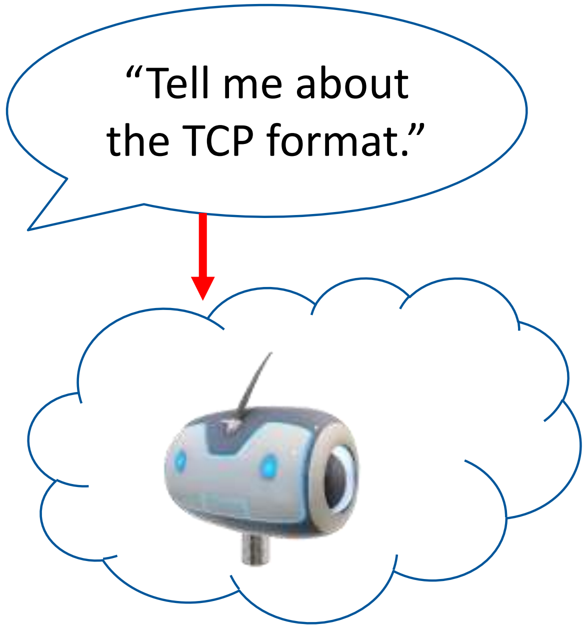
Format.c

ALAS, THIS IS A PIPE DREAM

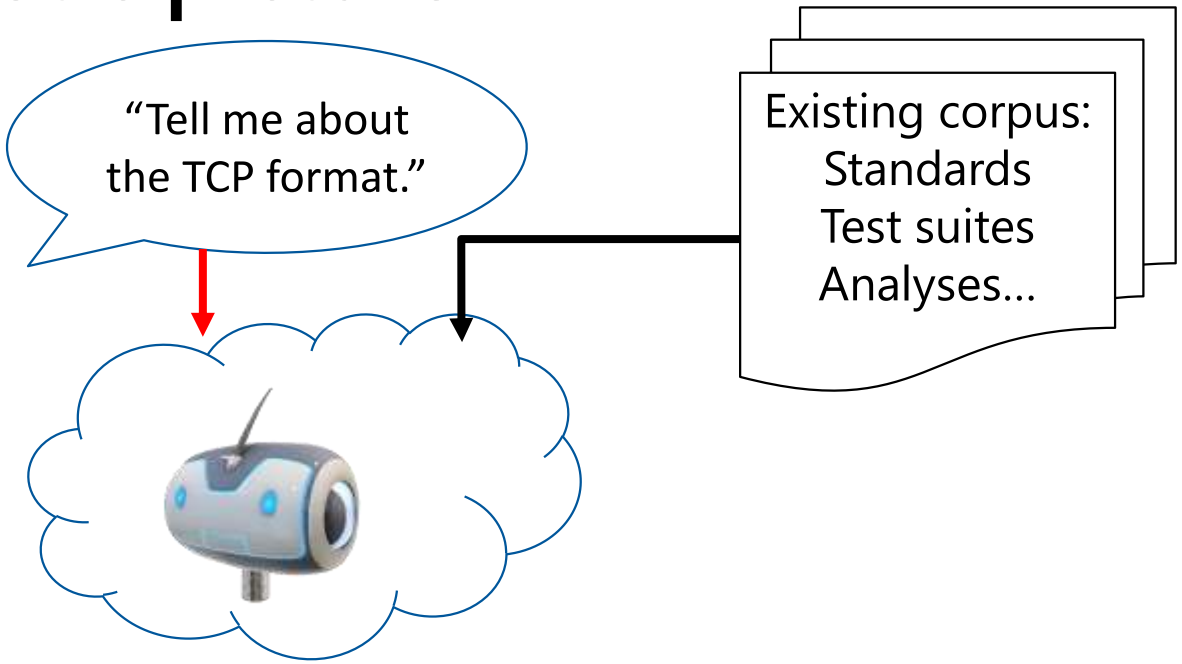
The AI Bot cannot tell whether the spec is valid

(do we really want to trust it?)

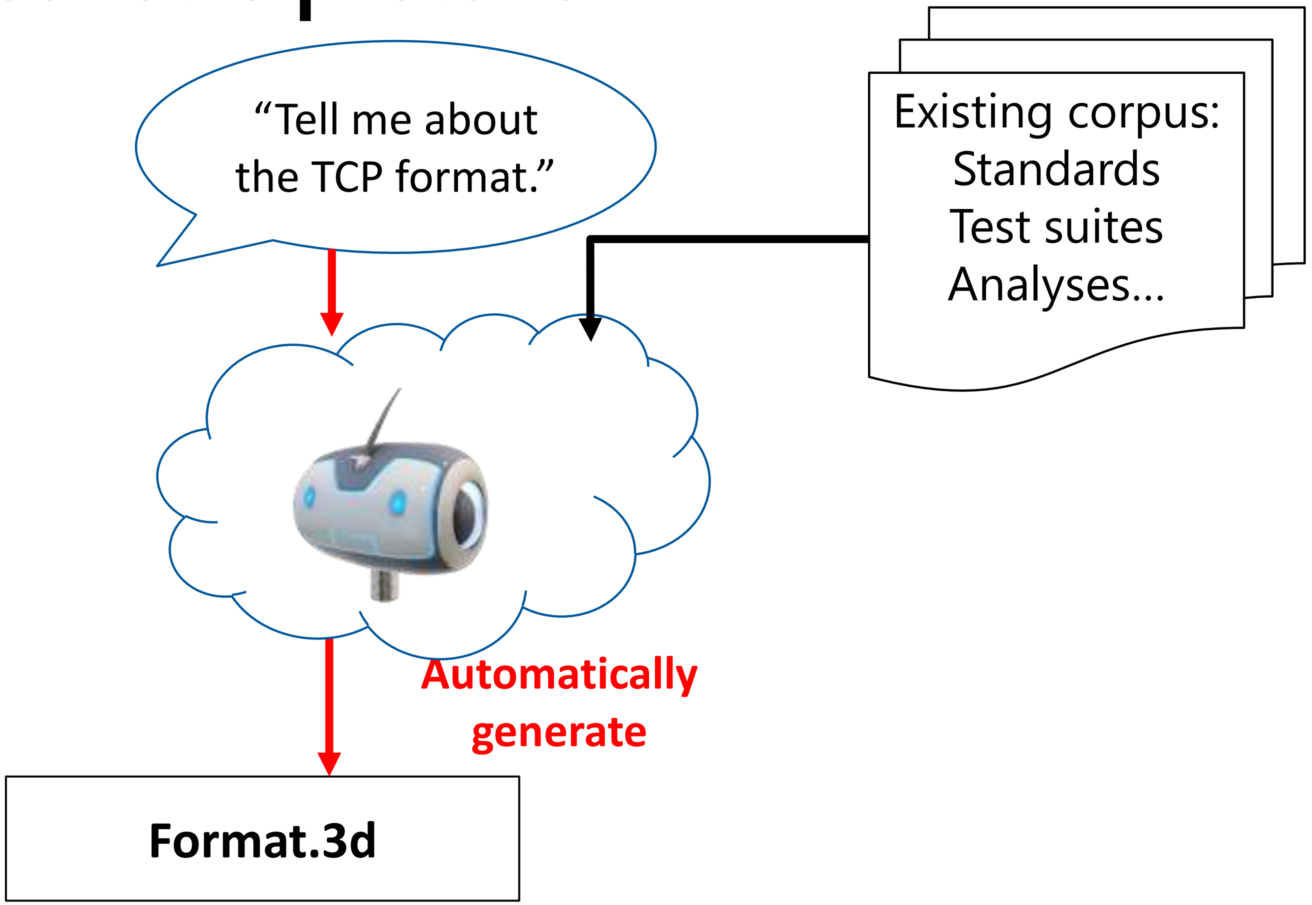
EverParse + AI: A more realistic picture



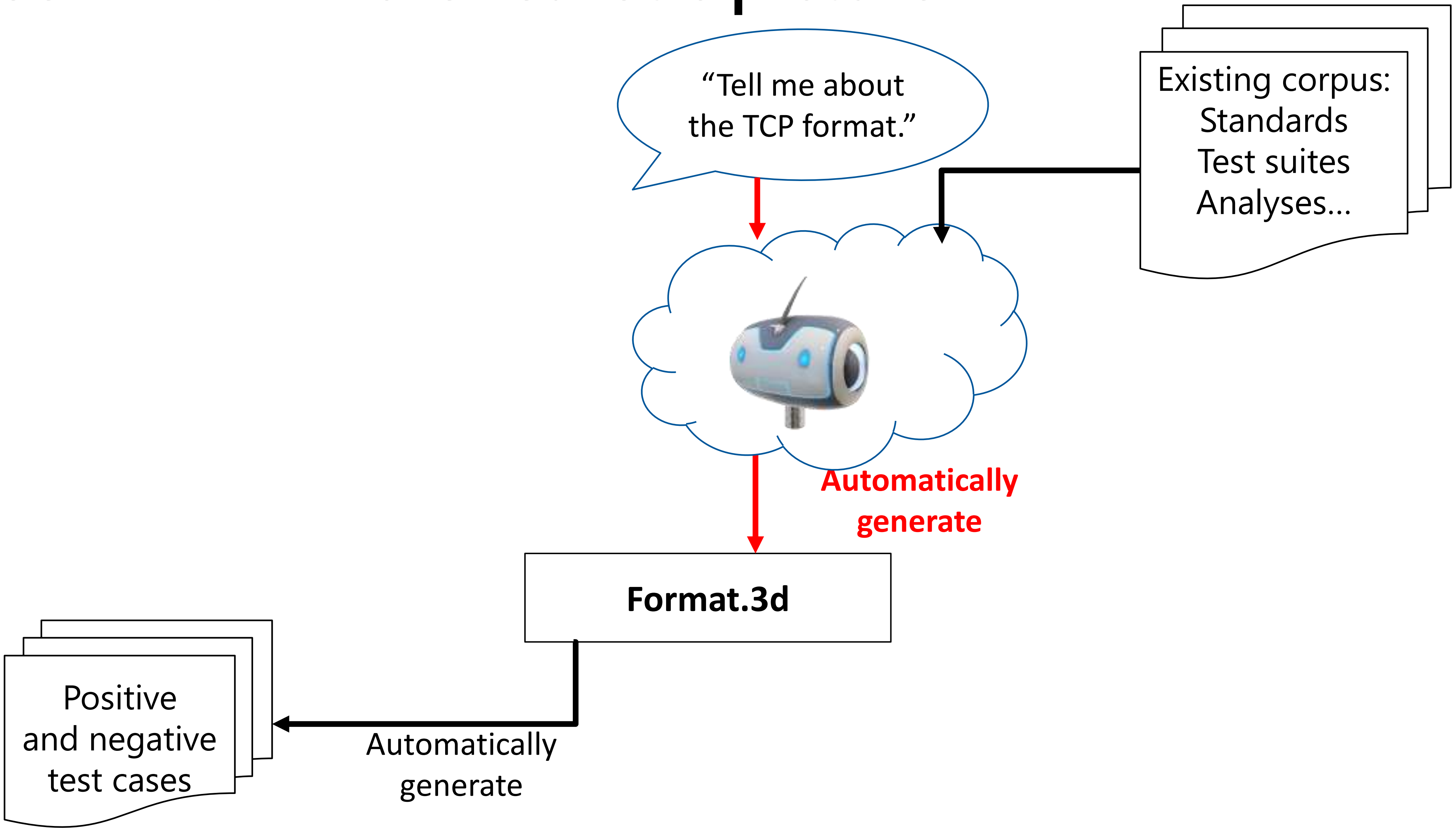
EverParse + AI: A more realistic picture



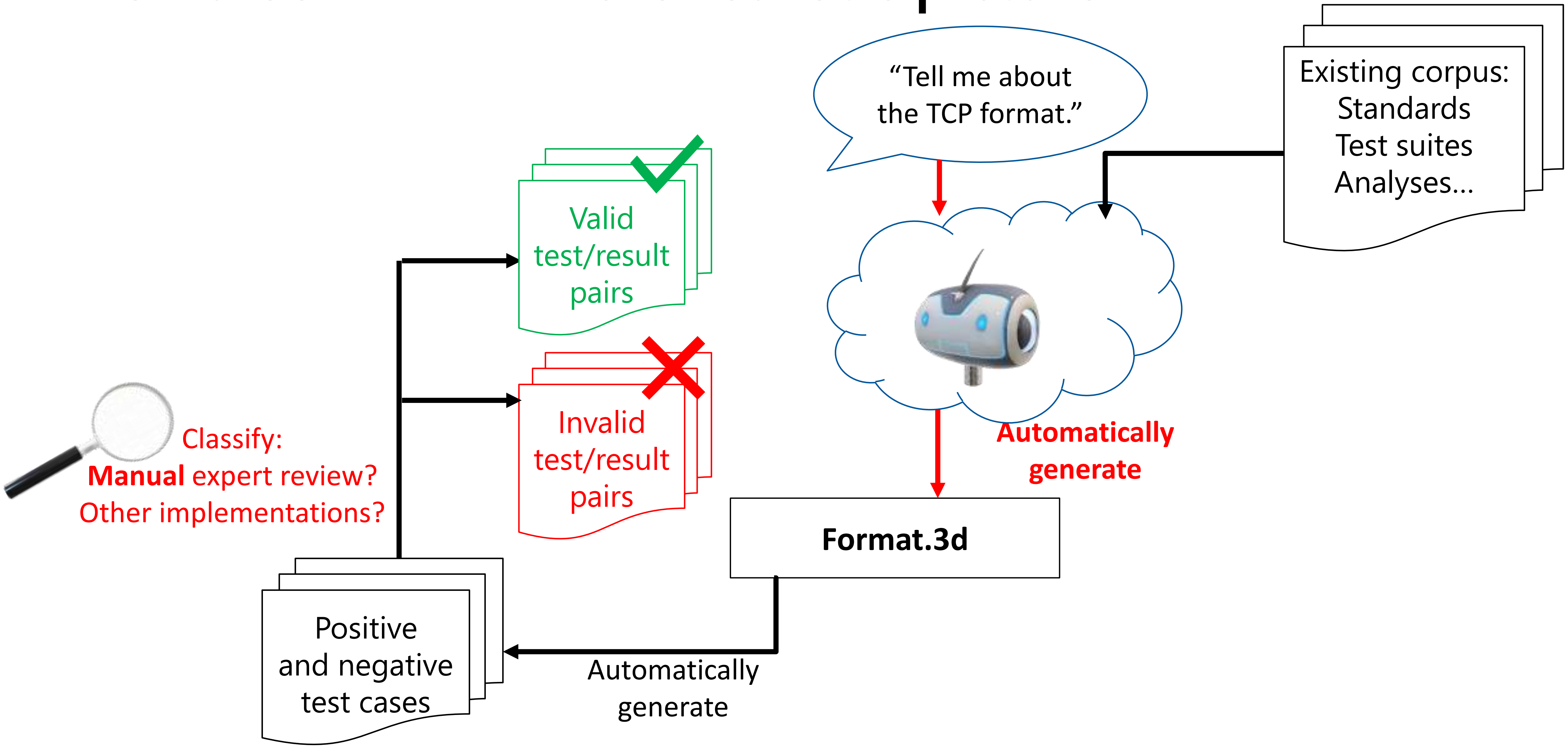
EverParse + AI: A more realistic picture



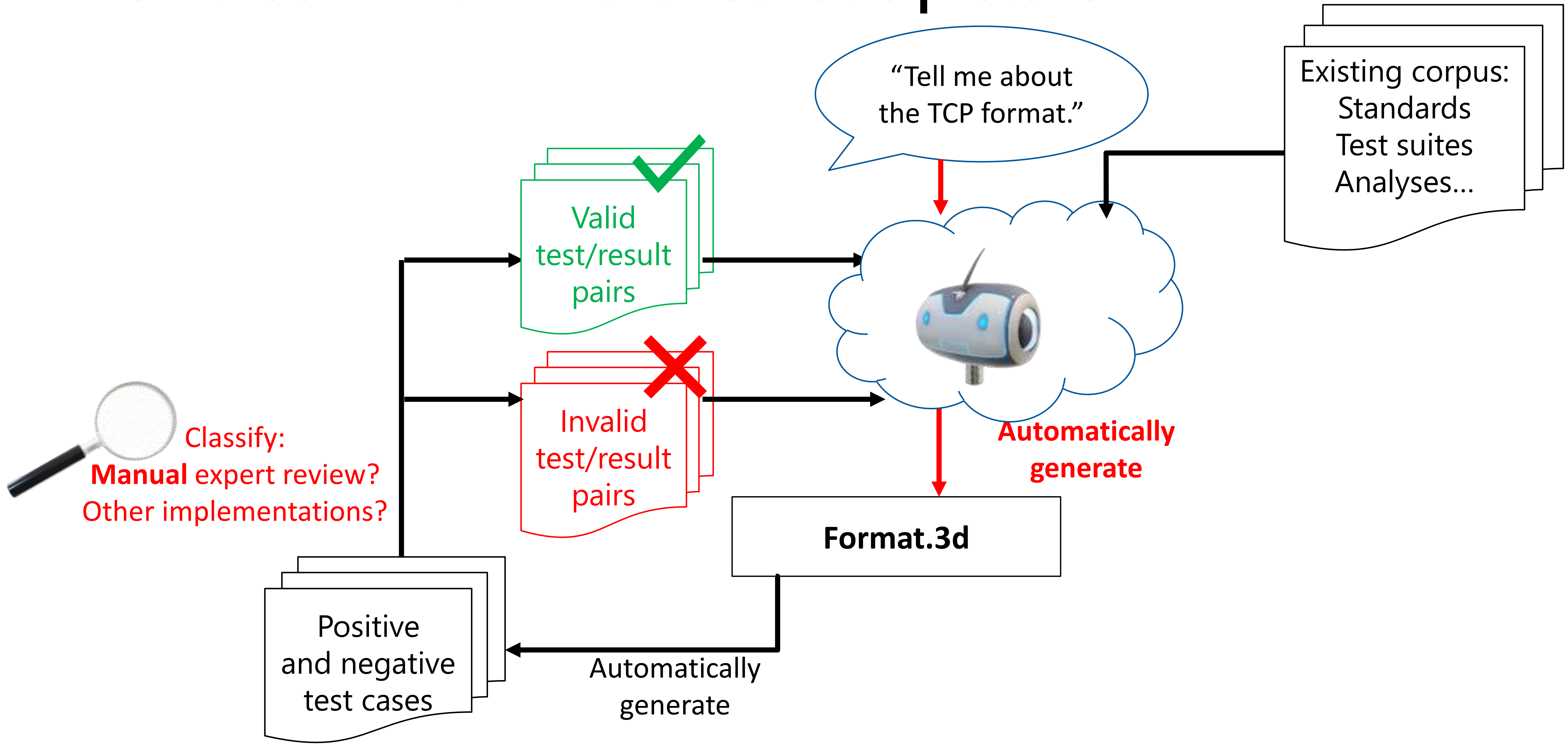
EverParse + AI: A more realistic picture



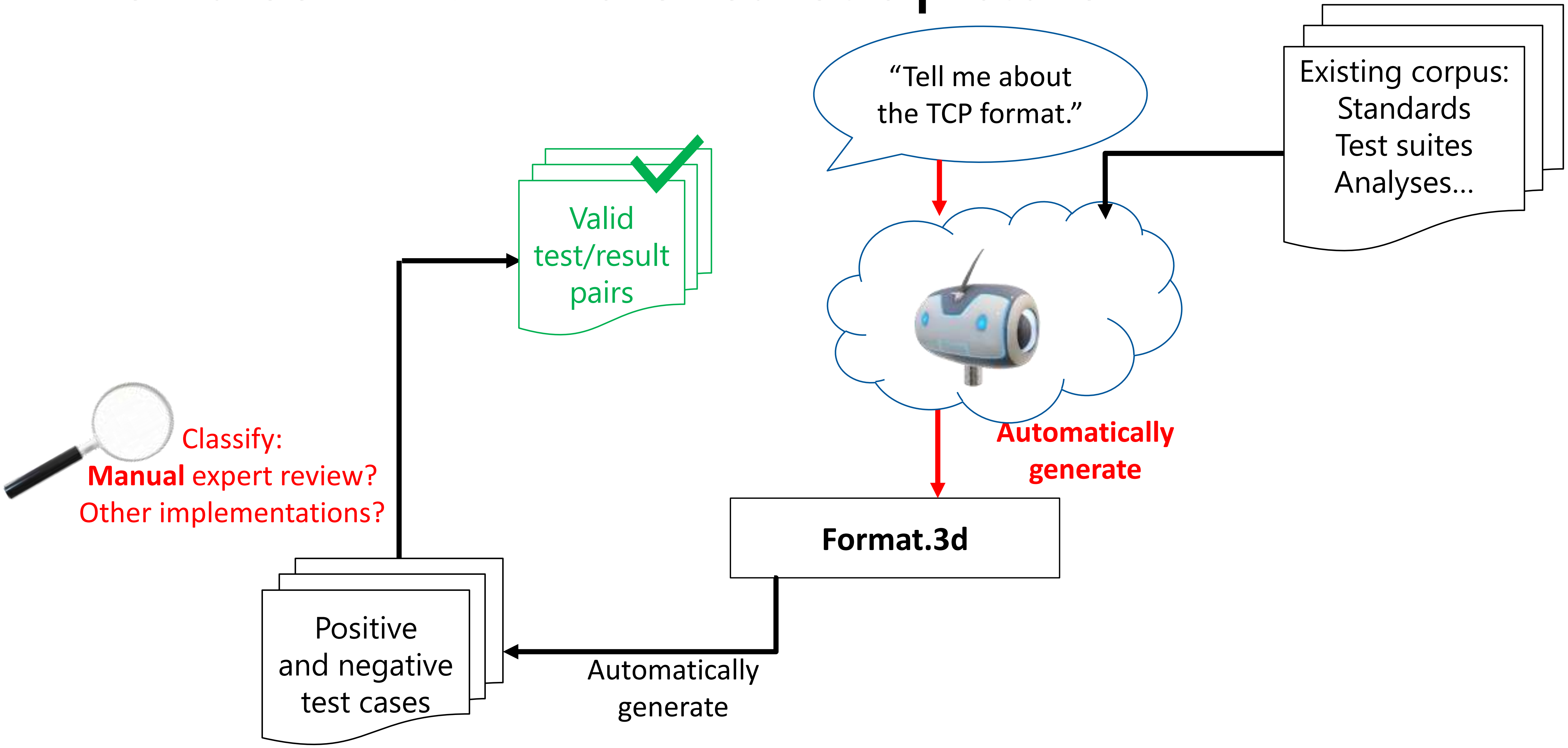
EverParse + AI: A more realistic picture



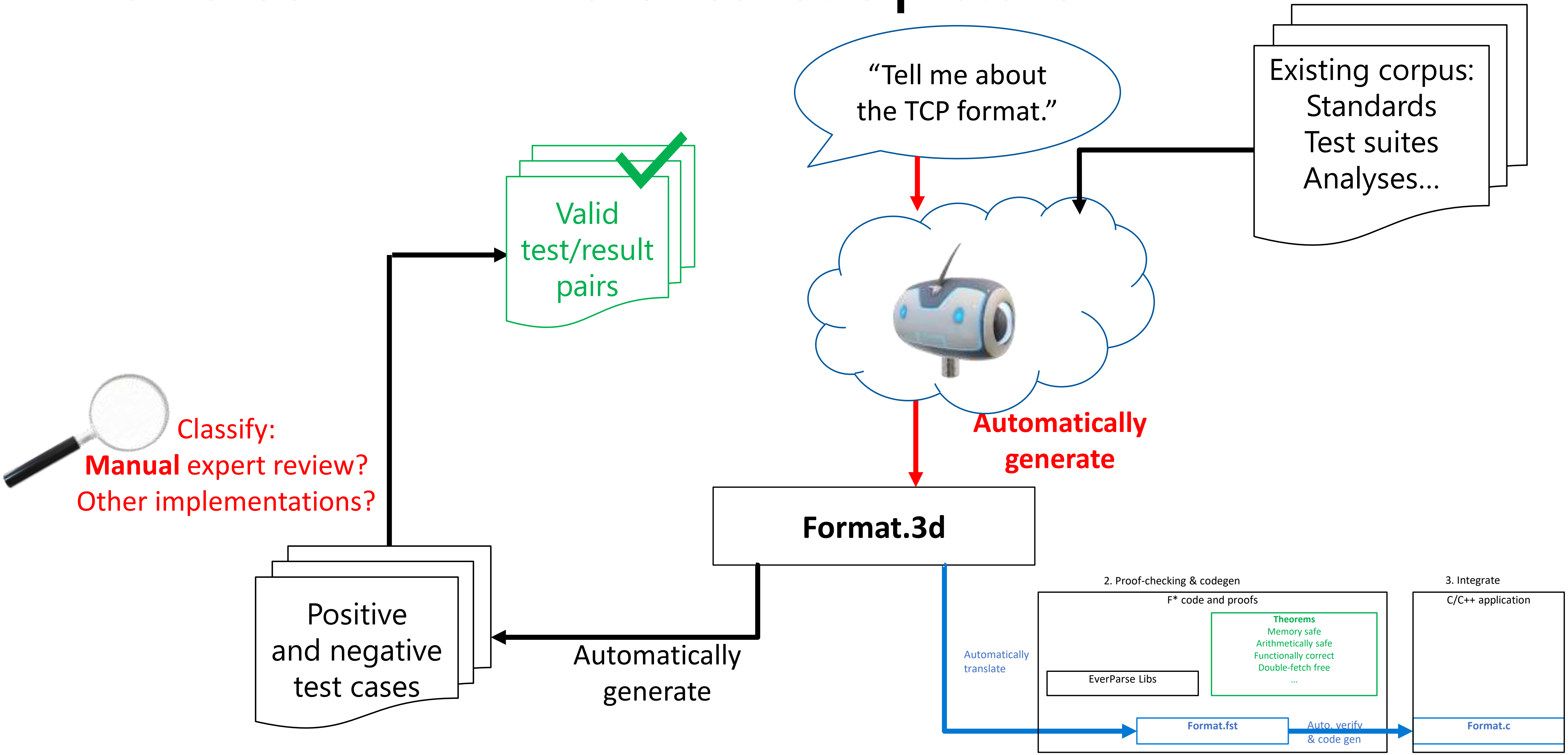
EverParse + AI: A more realistic picture



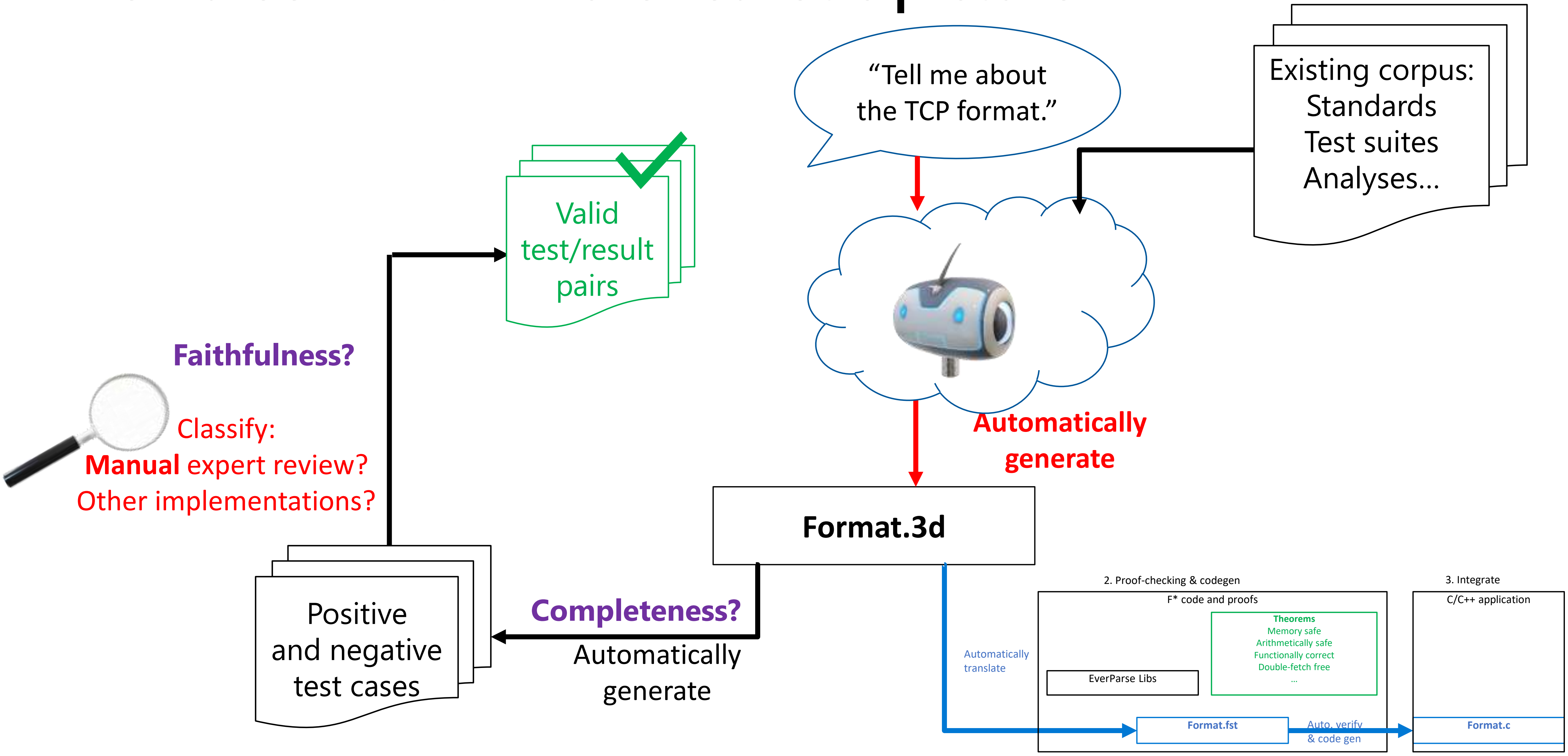
EverParse + AI: A more realistic picture



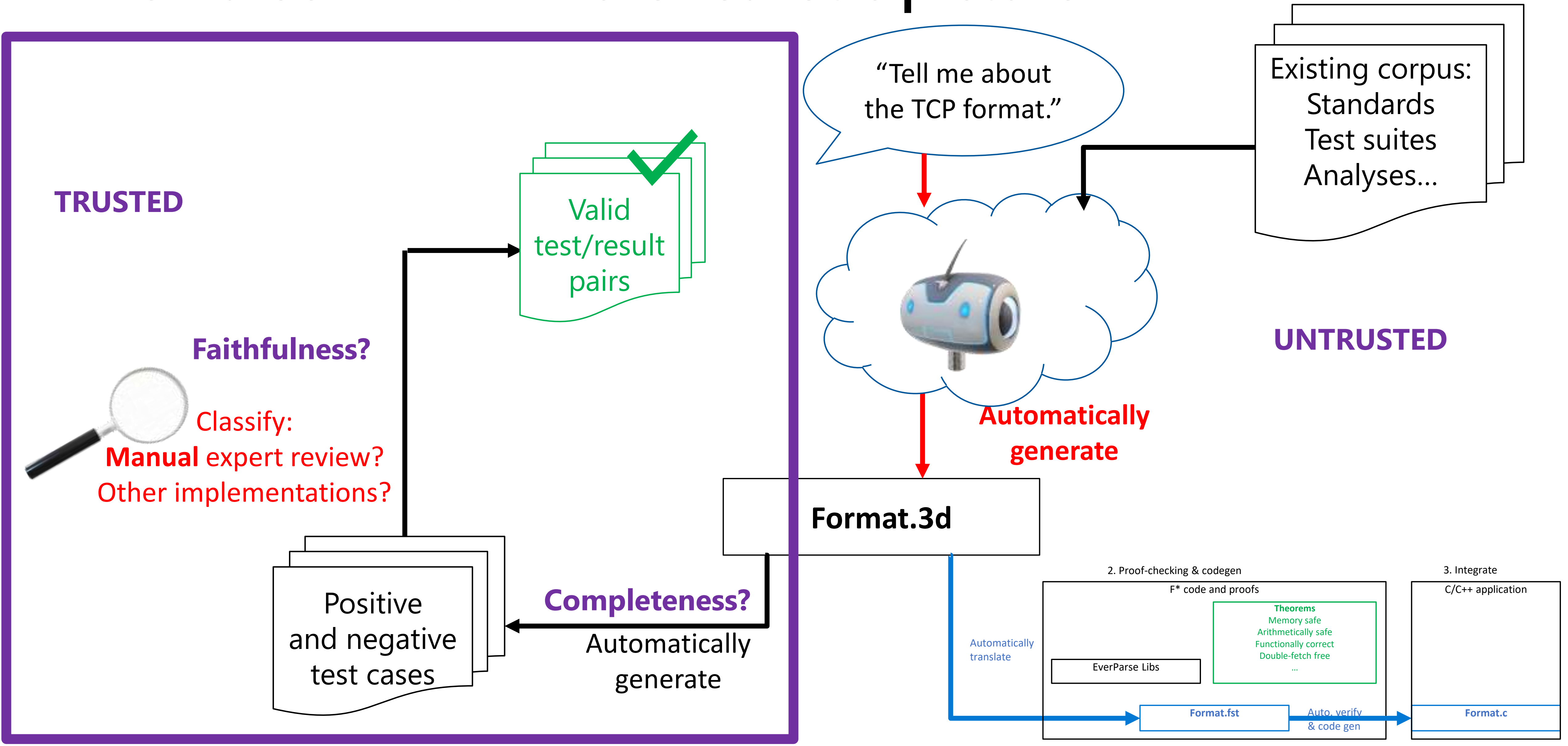
EverParse + AI: A more realistic picture



EverParse + AI: A more realistic picture



EverParse + AI: A more realistic picture



EverParse Takeaway

- A sweet spot for formal verification
  - Strong guarantees of memory safety, functional correctness and security
  - Provably correct by construction: Zero user proof effort
  - Handwritten parsers are a thing of the past (even more so with AI)
  - High return on investment wrt. attack surface
- A dedicated frontend for each family of formats
  - 3D, ASN.1, TLS RFC
  - Other formats in progress (CBOR, ...)
- Project page and manual: https://project-everest.github.io/everparse/
  - Open-source (Apache 2 license)
  - Binary releases for Linux and Windows
  - For more info: {taramana,nswamy,aseemr}@microsoft.com