IETF 123 UFMRG
July 24th, 2025

eVerpArse

Verified Secure Binary Data Parsing for All

Tahina Ramananandro
taramana@microsoft.com
RiSE – Research in Software Engineering
Microsoft Research, Redmond WA, USA

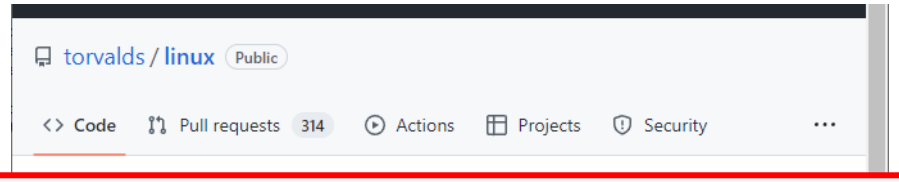
Microsoft

RiSE

everest
Verified End-to-End Secure Transport

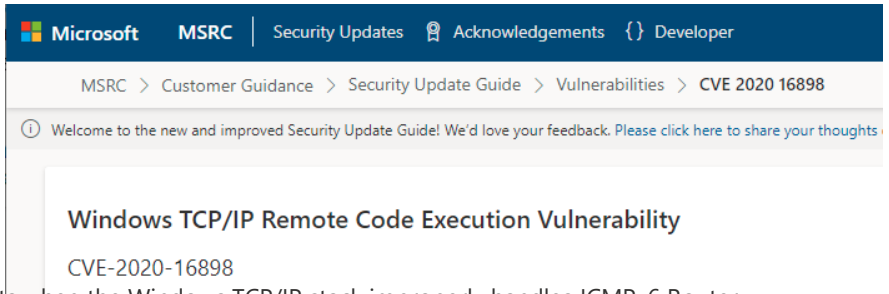
Secure Binary Data Parsing is Critical

- Parsing and input validation failures: A major root cause of software security vulnerabilities
- 80%, according to DARPA, MITRE
- Mostly due to handwritten parsing code
  - Especially disastrous in memory unsafe languages
  - Better in Rust, but still open to runtime panics and functional bugs
- Writing functionally correct parsers is hard
  - Endianness, data dependencies, size constraints, etc.



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.

What we advocate: The EverParse Manifesto

- Identify families of security-critical data formats, and design **format specification languages** to support them
- For a given family, implement a once-and-for-all provably correct low-level **code generator** (producing C, Rust, Wasm, etc.) to produce format manipulation code: parsing, serializing, reading…
- Write data format specifications and run the code generator, obtaining guarantees with **zero user proof effort**
- **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/CDDL, TLS, QUIC, etc.

EverParse: A Wide Collaboration

- Microsoft Research

- Barry Bond
- Yi Cai
- Antoine Delignat-Lavaud
- Gabriel Ebner
- Sarah Fakhoury
- Cédric Fournet
- Guido Martinez
- Jonathan Protzenko
- Tahina Ramananandro
- Aseem Rastogi
- Irina Spiridonova
- Nikhil Swamy

- Microsoft Product Groups

- Justin Campbell
- Omar Cardona
- Dragana Damjanovic
- Mike Ebersol
- Arti Gupta
- Alan Jowett
- Dmitry Malloy
- A. Malysh
- Kam Reypour
- Michael Tang
- Khoa To
- Juan Vazquez Pongiluppi

- External Contributors and Former Interns

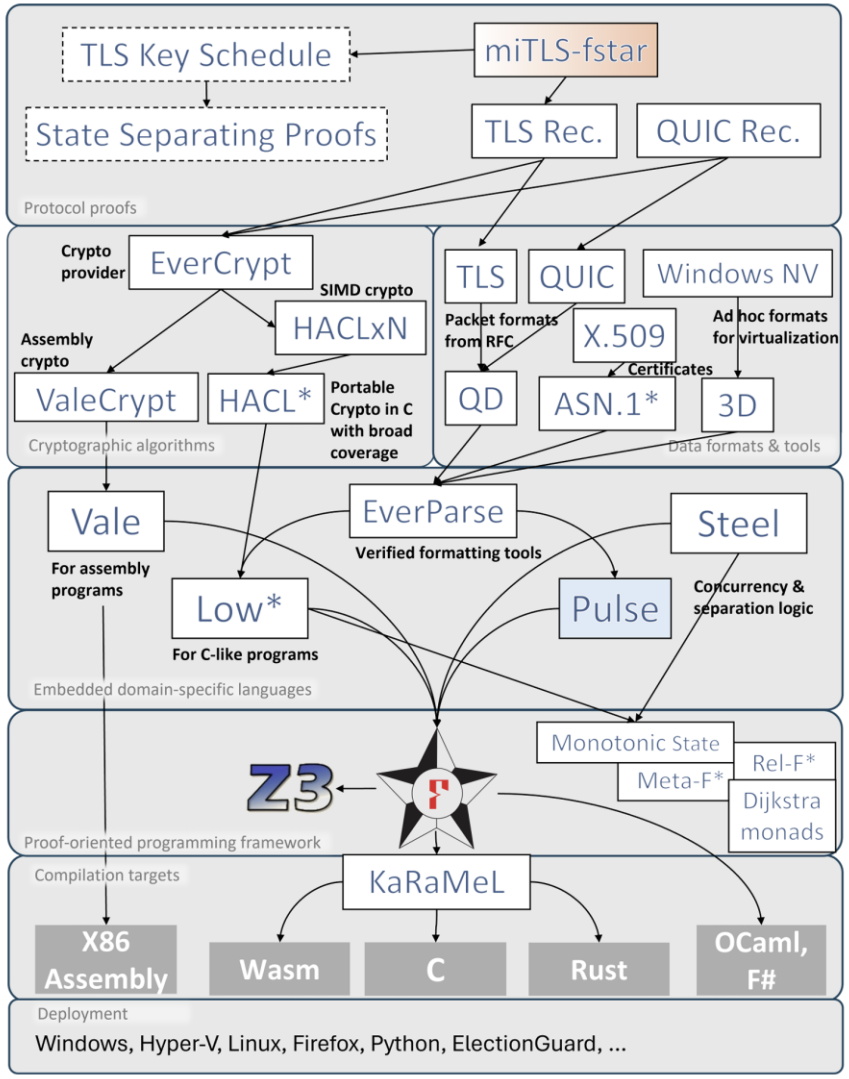
- Tej Chajed
- Nadim Kobeissi
- Haobin Ni
- Mingwei Zheng

Proof-Oriented Programming At Scale

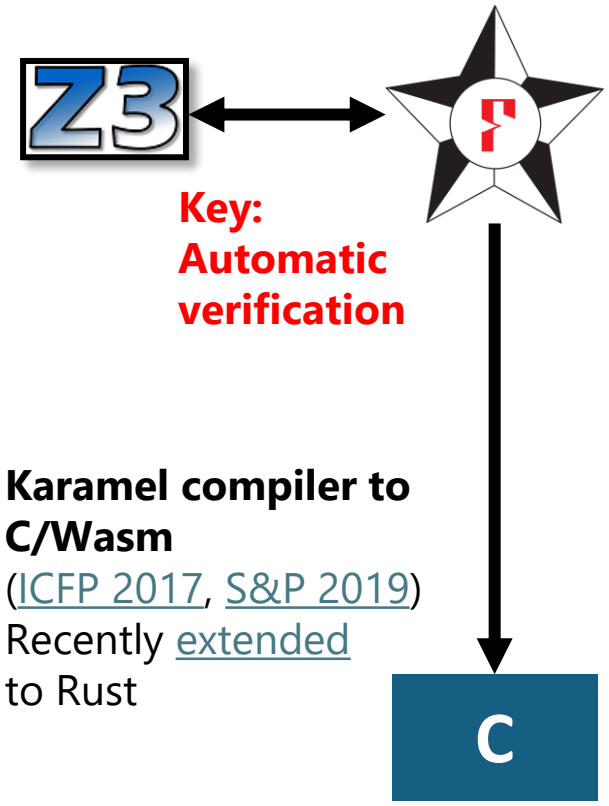
Project Everest (2016-2021)

- ~1 million lines of F* code
- Cryptography, verified parsers, …
- Deployed in Linux, Firefox, Python, Windows, Azure, …
- https://project-everest.github.io/

Everest offspring (including EverParse) grown up as active standalone projects



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



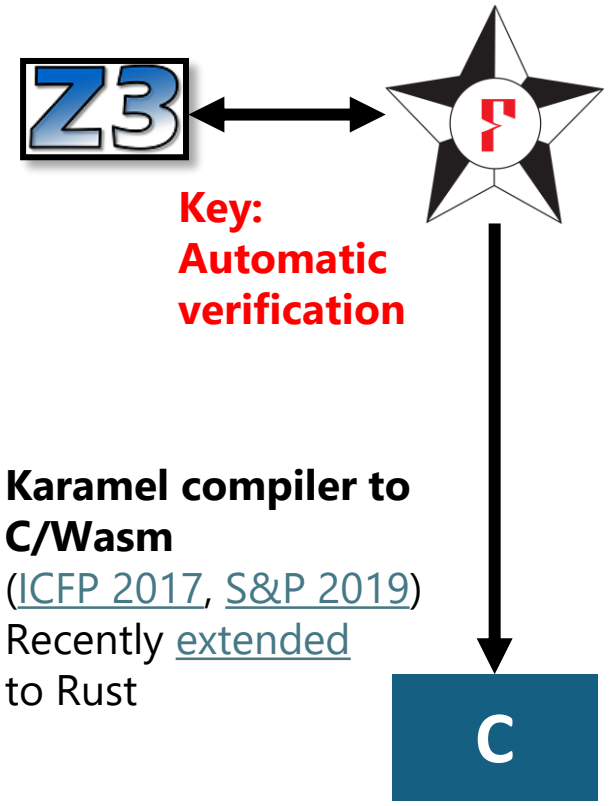
F* implementation and specification

```
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/Wasm/Rust 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;
}
```

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



Karamel compiler to C/Wasm
(ICFP 2017, S&P 2019)
Recently extended to Rust

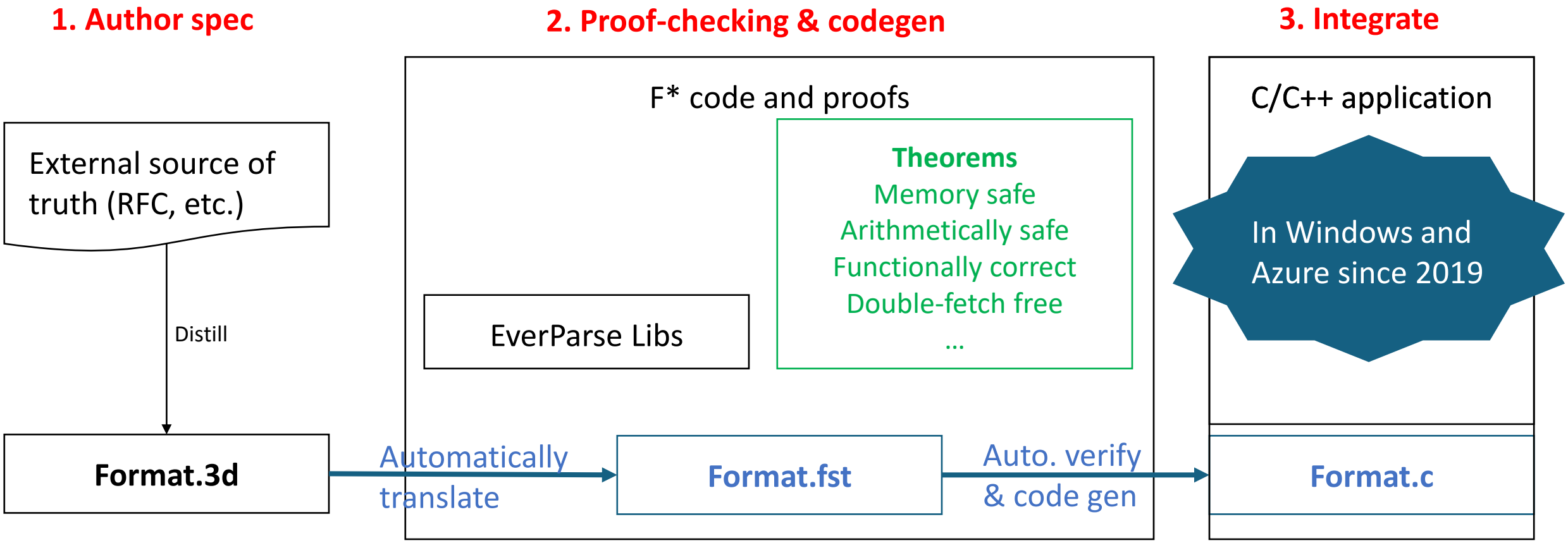
Implementation in the Pulse DSL embedded in F* **and specification** using separation logic (PLDI 2025)

```
fn inc (r: ref uint32_t)
  (requires r |-> 'vr * pure (vr + 2 <= MAX_UINT_32))
  (ensures r |-> ('vr + 2))
{
  let v = !r;
  r := v + 2
}
```

Efficient C/Wasm/Rust implementation
Verification imposes no runtime performance overhead

```
void inc(uint32_t *r)
{
  uint32_t v = *r;
  *r = v + 2;
}
```

The EverParse Manifesto, applied to Hyper-V



EverParse Formal Guarantees

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

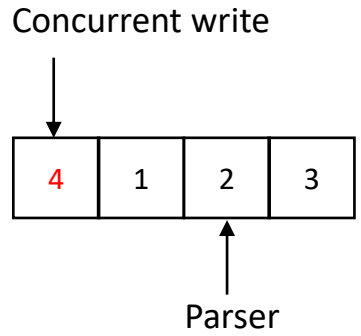
```
uint32_t fld_offset = input[current];
uint32_t fld = input[current+offset];
```

Missing checks for integer/buffer overflows

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

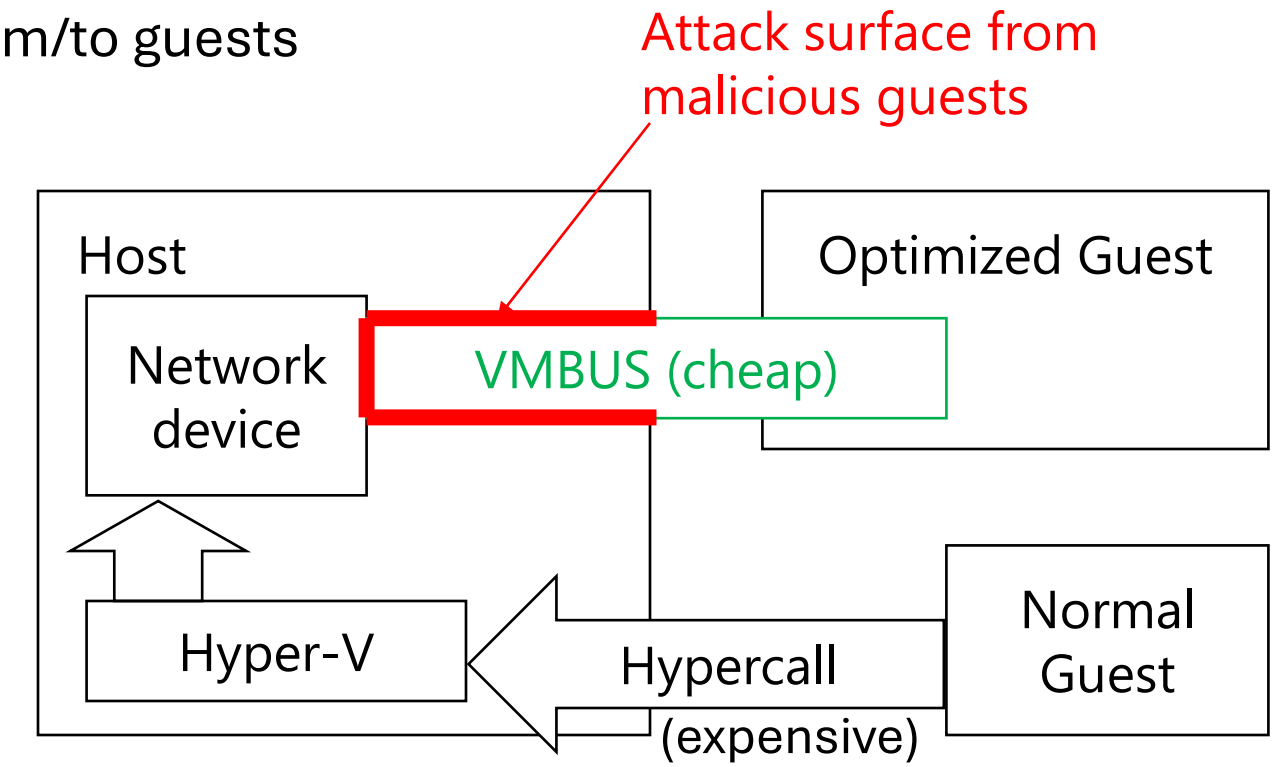
- Non-ambiguity, non-malleability
  - TLS, ASN.1 DER, CBOR §4.2, CDDL, …
  - Non-ambiguity: serialization parses back to the same value
  - Non-malleability: unique binary representation for hash-based authentication

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



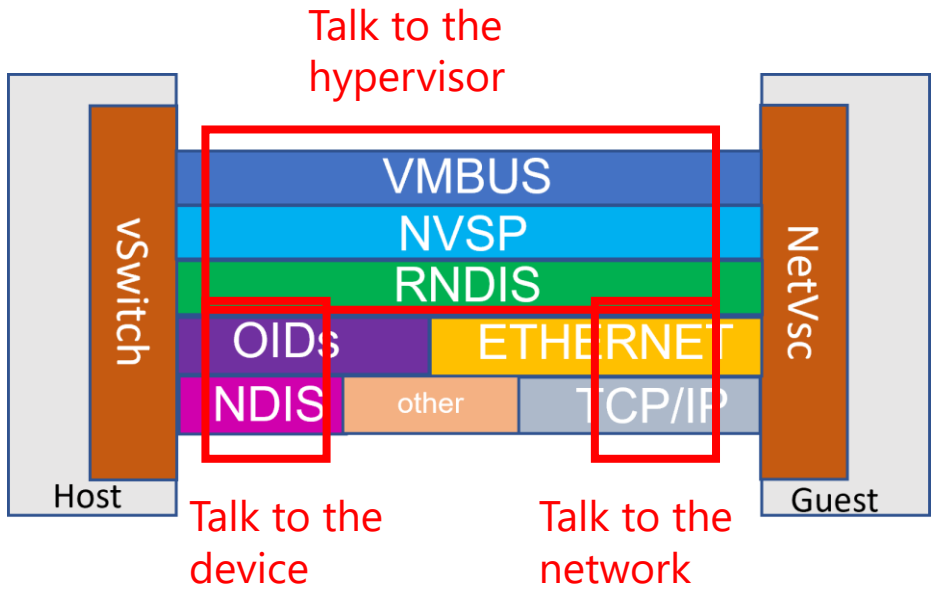
Microsoft Hyper-V Network Virtualization

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

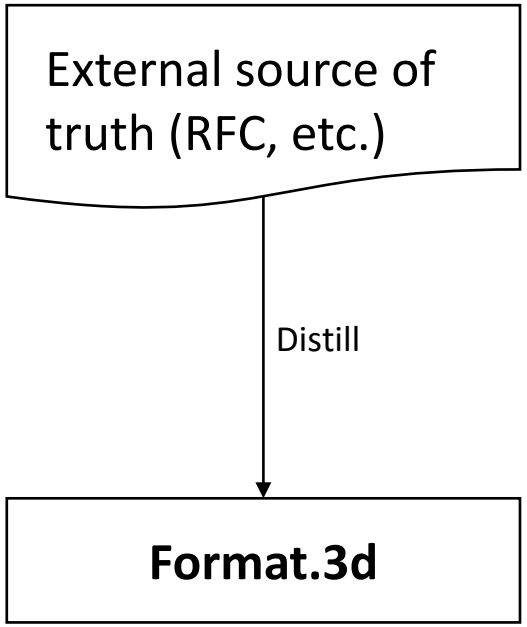


Hyper-V vSwitch with EverParse

- **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
- PLDI 2022; MSR Blog Post



Step 1: Figure out the data format spec



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;
```

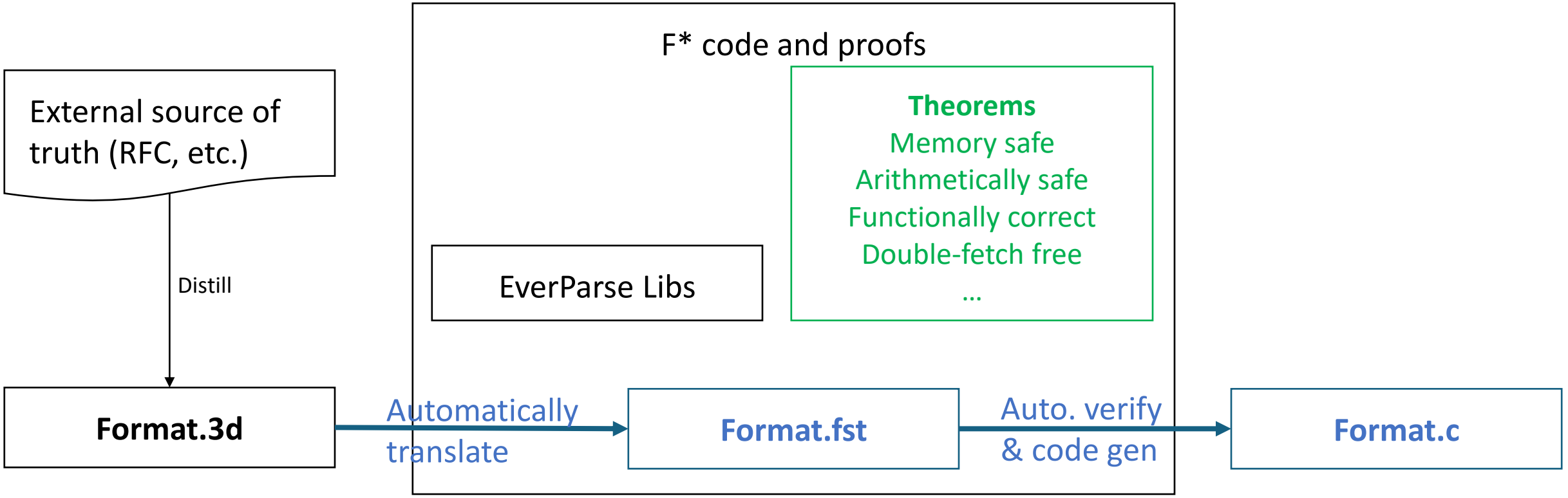
```
casetype _OPTION_PAYLOAD
  (UINT8 OptionKind, Bool MaxSegSizeAllowed) {
  switch(OptionKind)  {
     case OPTION_KIND_END_OF_OPTION_LIST:
       all_zeros EndOfList;
     case OPTION_KIND_NO_OPERATION:
       unit Noop;
     …
}} OPTION_PAYLOAD;

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

Step 2: Generate and Verify Code

1. Author spec

2. Proof-checking & codegen



Generated C code

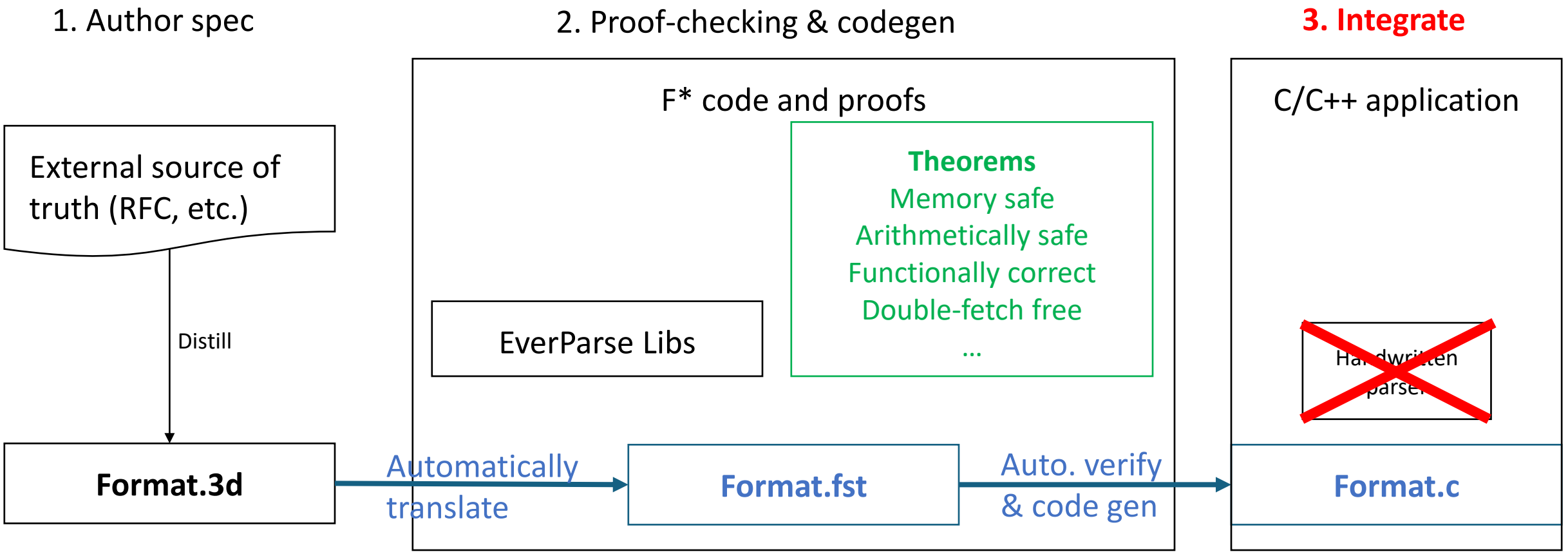
```
#include "EverParse.h"

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

- Generated C 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
                                                    DATOR_ERROR_NOT_ENOUGH_DATA,
 /* 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;
}
```

Step 3: Deploy and Maintain the Code



Step 3: Deploy and Maintain the Code

- Performance bar met: <2% overhead
  - Sometimes faster than original code thanks to double-fetch freedom
- Productivity improvement critical to meet product deadline
  - Saves code writing cost
  - More focused security reviews
- Product teams autonomously update spec and run EverParse

- Since then, EverParse has propagated
  - 3 other areas of Windows Networking
  - eBPF for Windows
  - …

*"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

Using EverParse with Verified F* Applications

TLS

- Verified TLS secure channel with formal security model
- Verified serialization and **zero-copy** parsing of handshake message formats
- Verified non-malleable (unique binary representation, for signature-based authentication)
- USENIX Security 2019

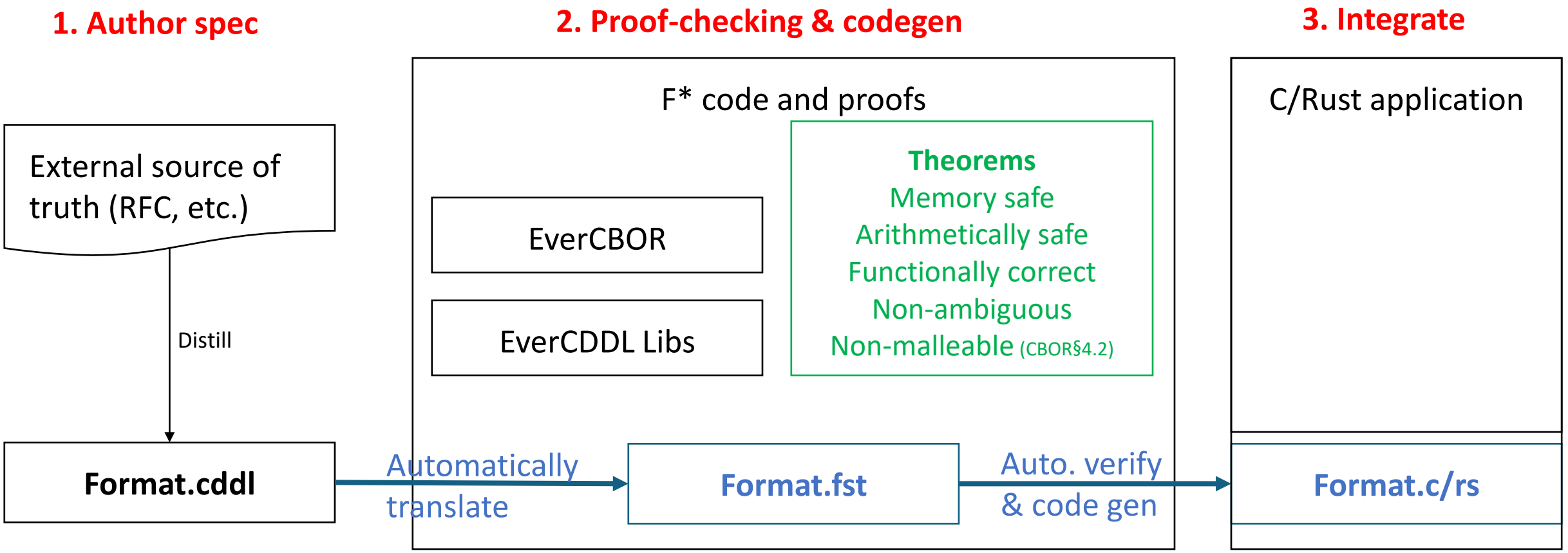
QUIC

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

ASN.1, CBOR/CDDL, DICE/DPE

- 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, CDDL:** ACM CCS 2025

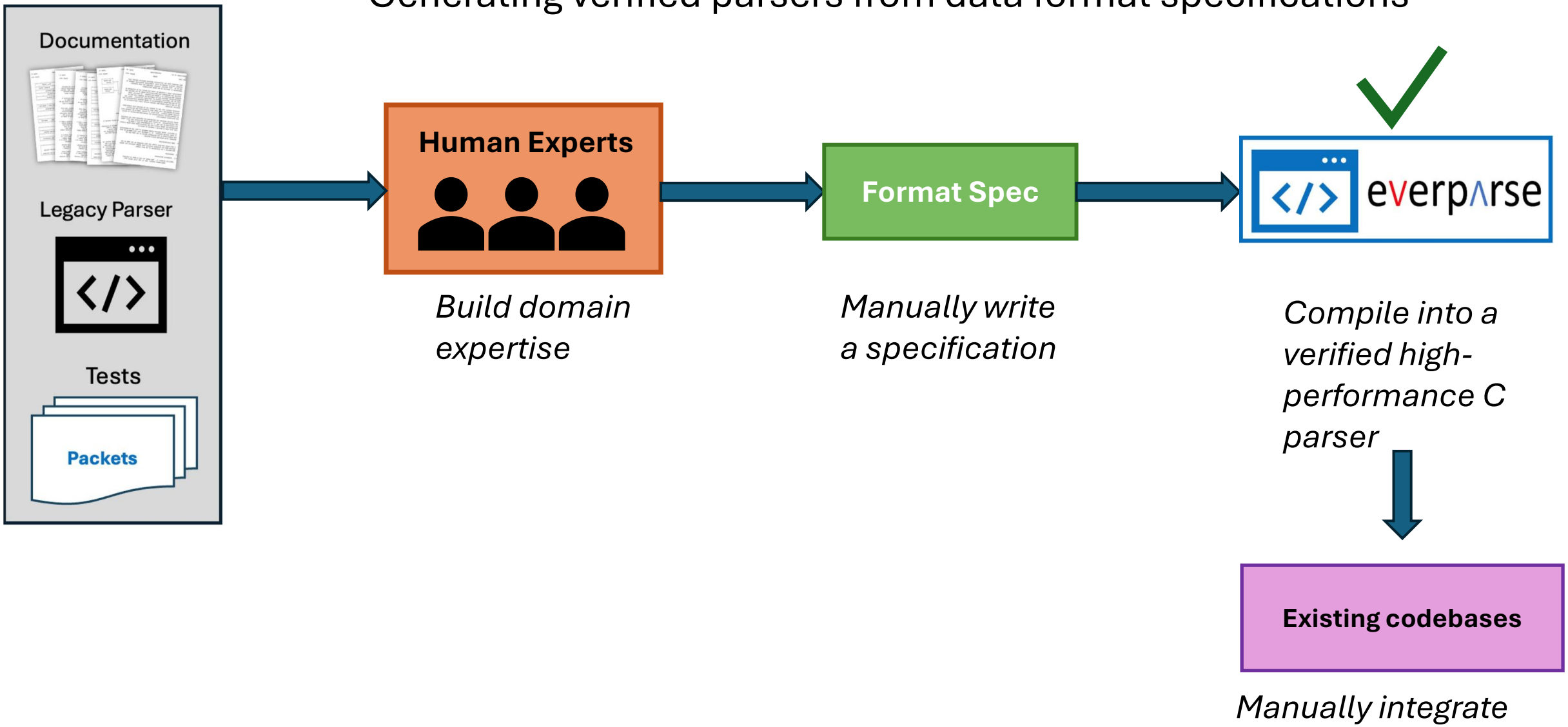
The EverParse Manifesto, applied to CDDL



ACM CCS 2025 (accepted for publication, to appear)

EverParse

Generating verified parsers from data format specifications



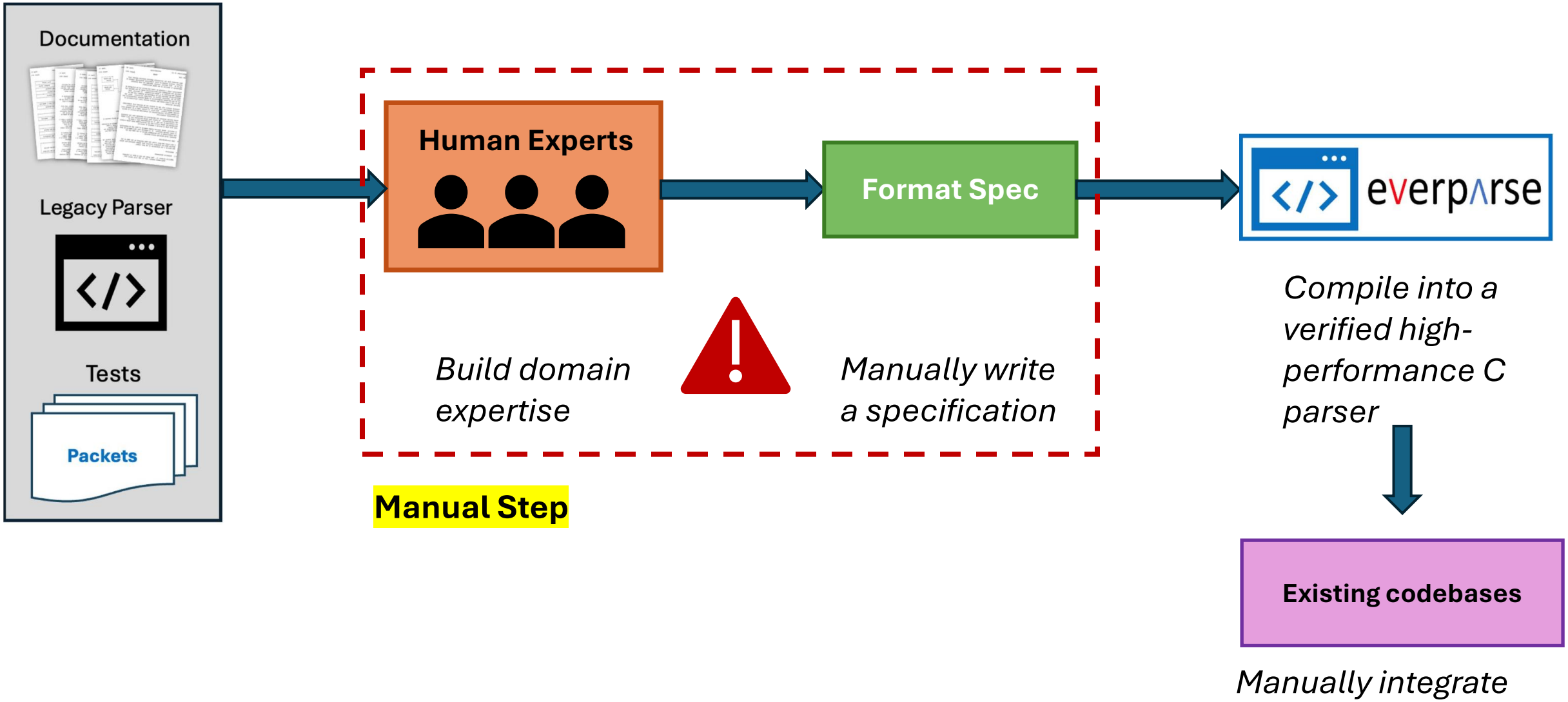
Build domain expertise

Manually write a specification

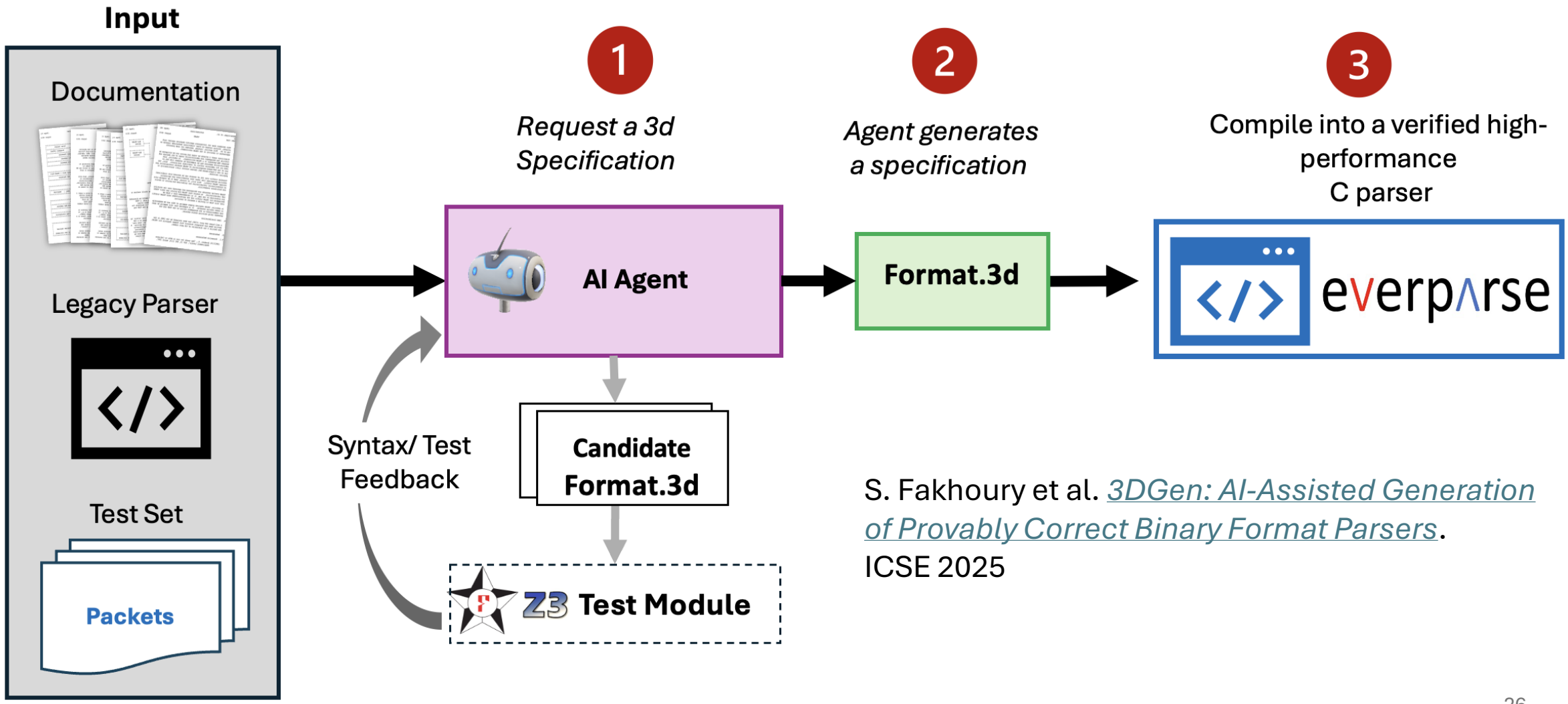
Compile into a verified high-performance C parser

Manually integrate

Challenge: Automating spec writing



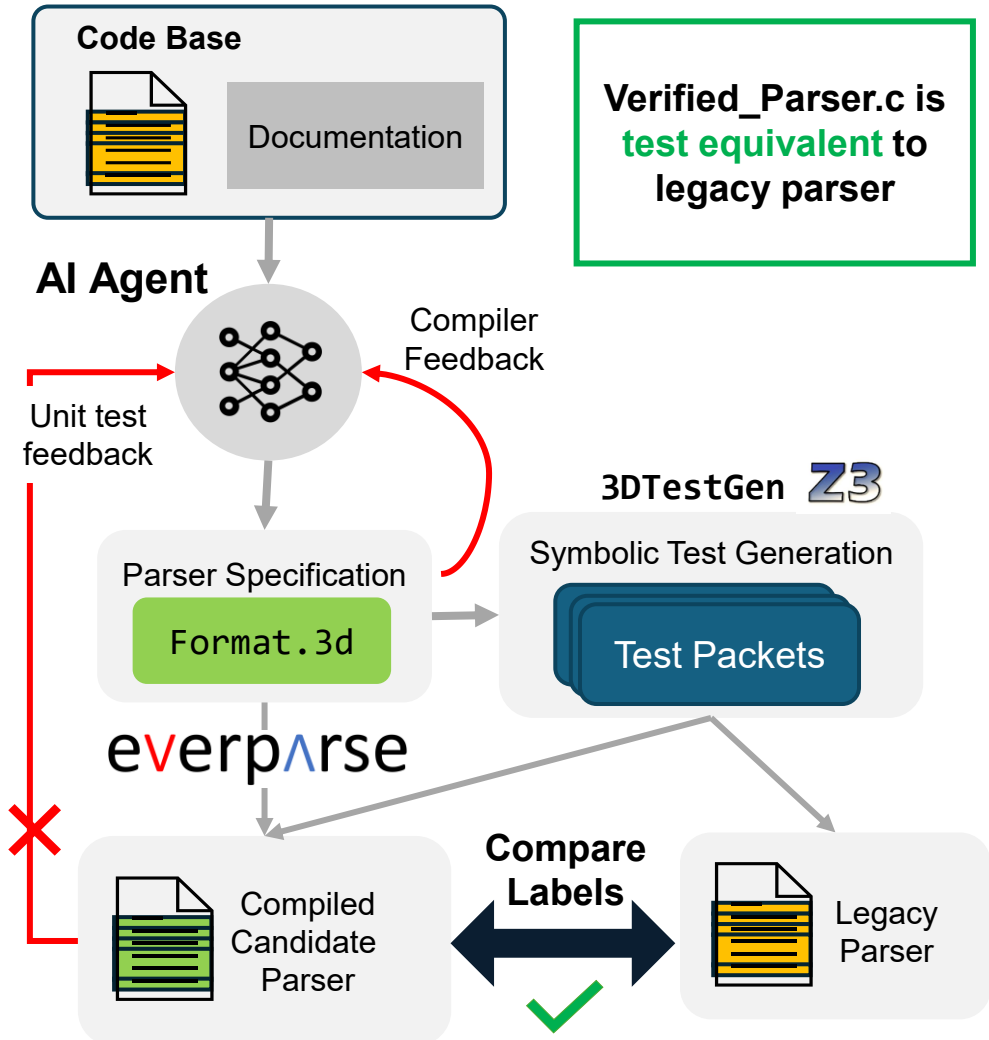
3DGen
AI-Assisted Generation of Data Format Specifications



S. Fakhoury et al. *3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers*. ICSE 2025

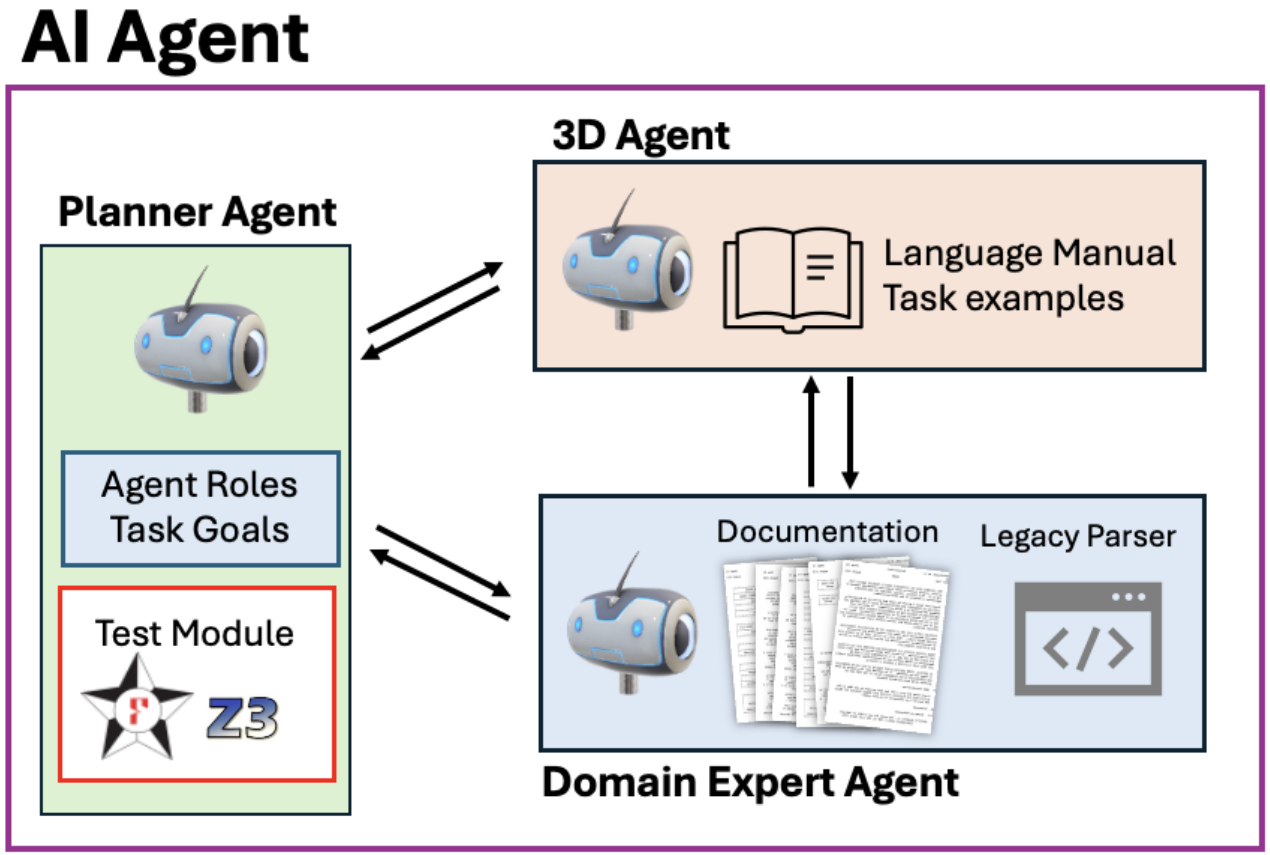
3DGen Workflow

- ✓ Workflow constrains what AI Agents can produce
  - ✓ Provides memory safety guarantees for free with 3D
- ✓ Output backed by symbolic tools
  - Compilers, test generators, proof checkers
- ✓ Tools provide useful + concrete feedback, both for humans & Agents



Agent Implementation

- Three Agent personas collaborating:
  - **Planner:** dictates roles, orchestrates conversation
  - **Domain Expert Agent**: Extracts constraints from NL or Code, provides feedback about generated specification
  - **3D Agent:** Extracts 3D specifications
- Implemented with AutoGen [3]
  - Composable
    Retrieval Augmented (RAG) agents
- No fine-tuning, easy migration to GPT-X!
  - Gpt-4-32k model



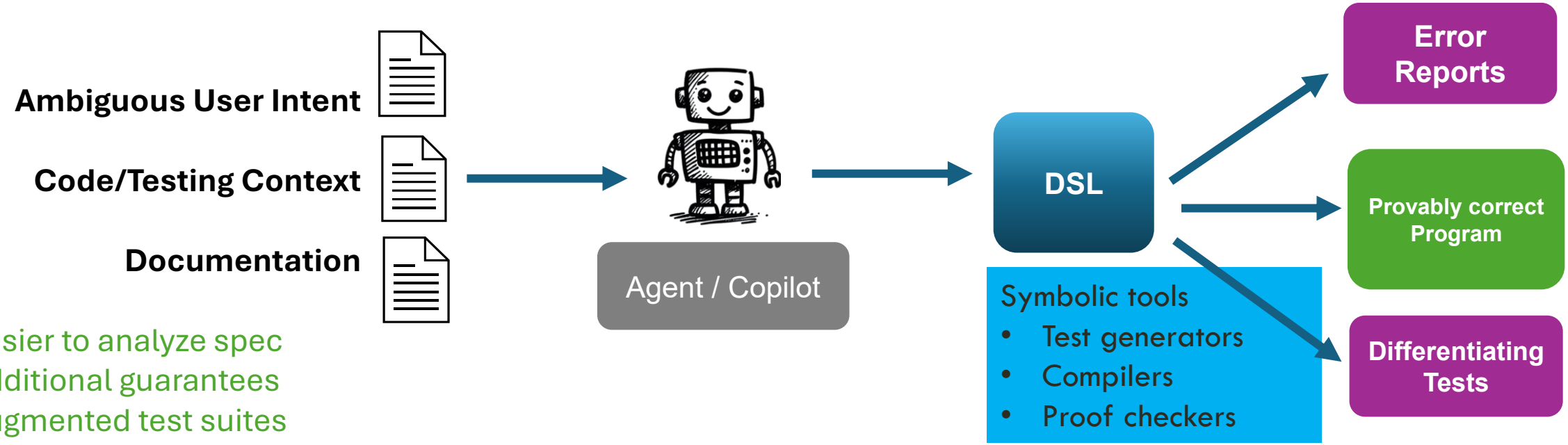
[3] Wu, Qingyun, et al. "AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework." *arXiv preprint arXiv:2308.08155* (2023)

3DGen Results on 20 Network Protocol RFCs

#	Protocol	RFC	Length (Pages)	Description
1	UDP*	768	3	User Datagram Protocol
2	ICMPv4 *	792	21	Internet Control Message Protocol
3	VXLAN*	7348	22	Virtual eXtensible Local Area Network
4	IPV6*	2460	39 (24)	Internet Protocol version 6
5	IPV4*	791	45 (12)	Internet Protocol version 4
6	TCP*	793	85 (10)	Transmission Control Protocol
7	Ethernet*	7348	22	Ethernet II Frames in VXLAN
8	GRE	2784	9	Generic Routing Encapsulation
9	IGMPv2	2236	24	Internet Group Managment Protocol
10	DHCP	2131	45 (4)	Dynamic Host Configuration Protocol
11	DCCP	4340	129 (14)	Datagram Congestion Control Protocol
12	ARP	826	10	Address Resolution Protocol
13	NTP	5905	110 (4)	Network Time Protocol
14	NBNS	1002	84 (6)	NetBIOS Name Service
15	NSH	8300	40 (8)	Network Service Header
16	TFTP	1350	11	Trivial File Transfer Protocol
17	RTP	3550	104 (3)	Transport Protocol for Real-Time Applications
18	PPP	1661	52 (11)	Point-to-Point Protocol
19	TPKT	2126	25	ISO Transport Service on top of TCP
20	OSPF	5340	94 (13)	Internet Official Protocol Standards

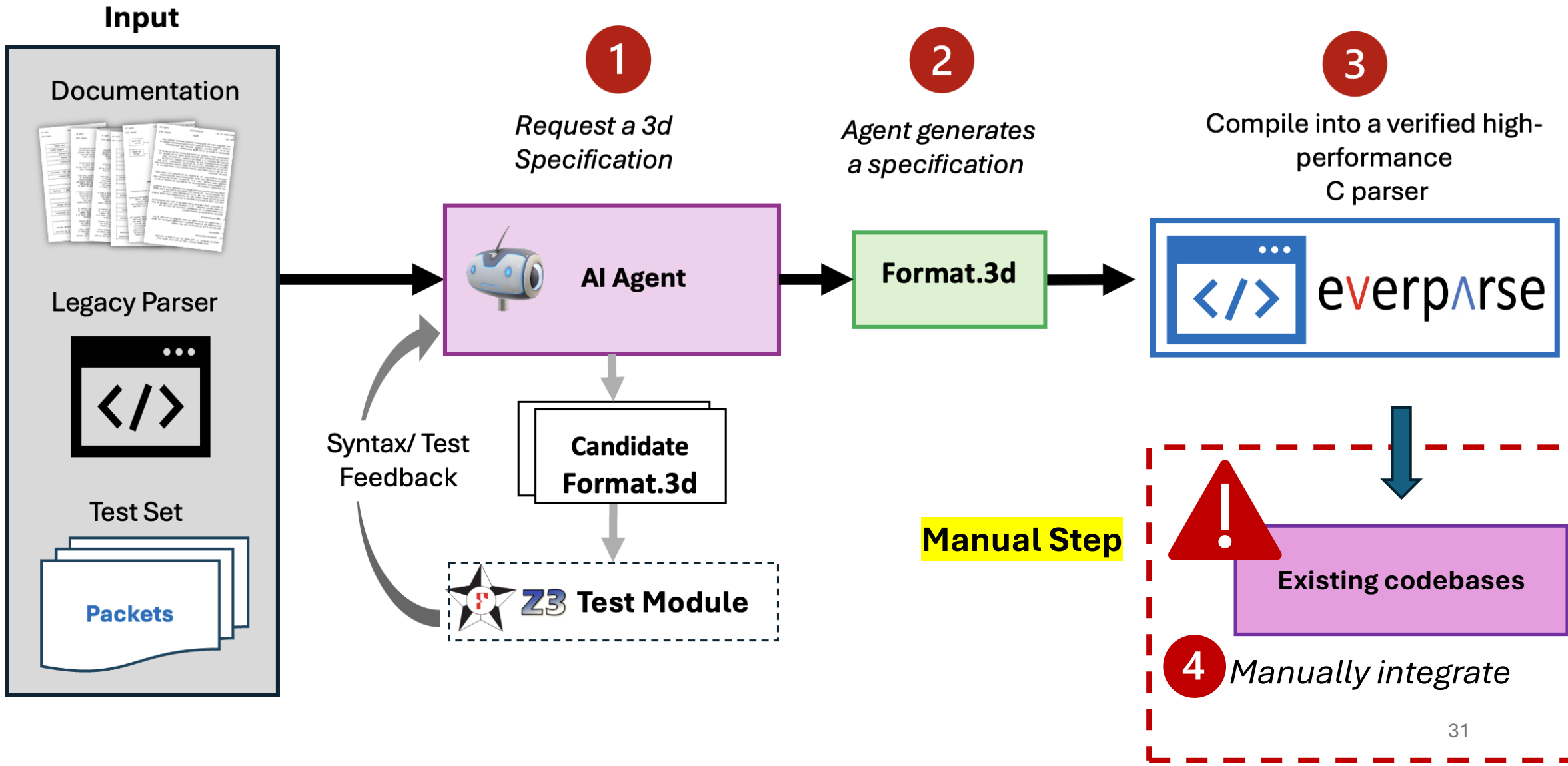
Constraining AI Output and Lowering Barrier of Use

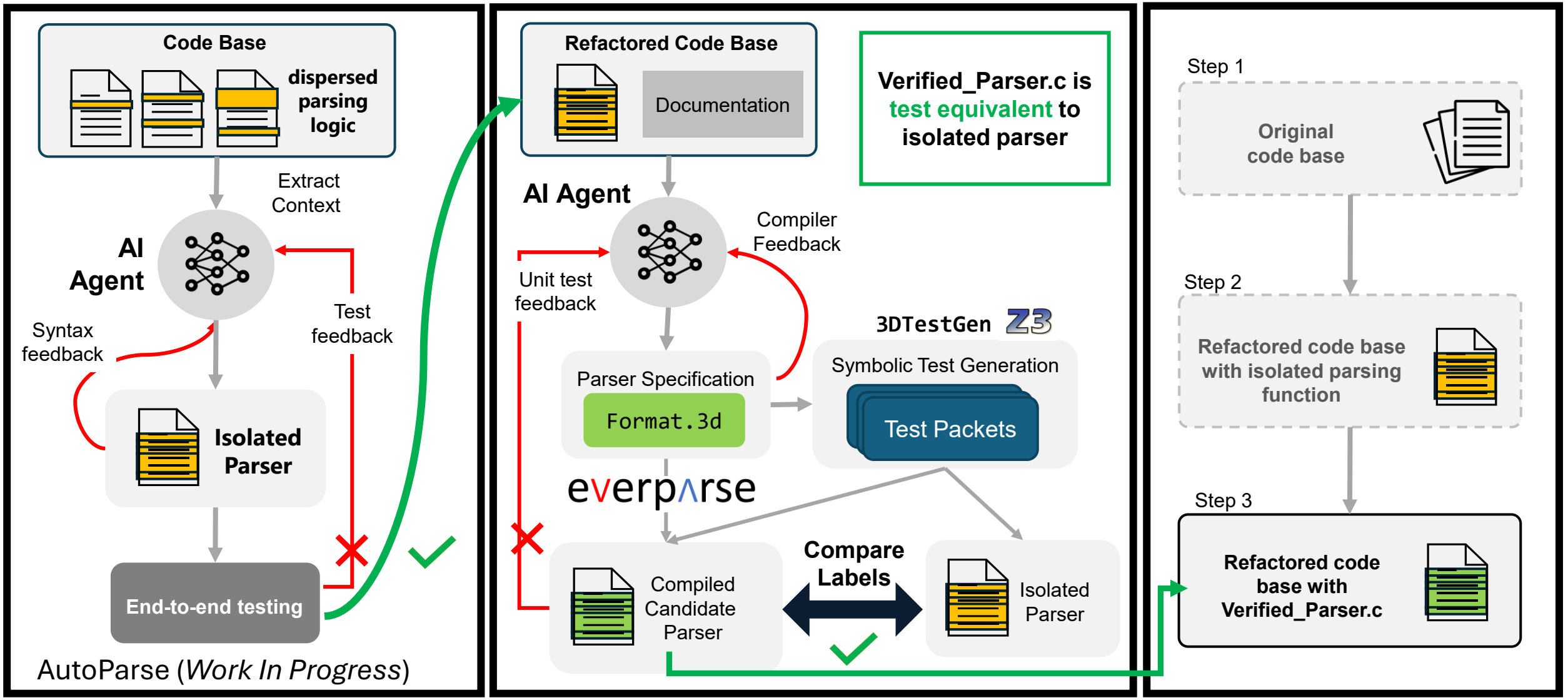
- Restrict code generation space to provide domain-specific guarantees
- AI agents can be easily taught to code in restrictive DSLs that provide users with safety & correctness, optimizations, etc.
- Reduce (but do not eliminate) validation burden on the user compared to unconstrained AI use alone



+ Easier to analyze spec
+ Additional guarantees
+ Augmented test suites
- Spec review still necessary
- Possible loss of trust assumed in the writing process

Challenge: Automating integration





Step 1: Refactor to create isolated parsing function

Step 2: Specification inference and testing

Step 3: Replace isolated parser with verified code

Call to Action

- For IETF spec writers
  - Consider providing a formal spec of your message formats in RFCs amenable to automatic processing
  - Using languages like 3D, CDDL, etc., that are accompanied by verified code generators

- For implementors: Postel's principle should be revisited in a security setting
  - Be conservative in what you send: OK
  - But: Be paranoid about what you receive
  - How to interpret RFC 1122 Section 1.2.2?

  - Using our code generators for parsing & serialization provides a robust first layer of security
    - But, also eases interop: guaranteed to parse only all well formed messages

Secure Parsing at MSR RiSE

- **AutoParse**
- **CBOR, CDDL, COSE** [CCS 2025]
- **3DGen** [ICSE 2025]
- **3D, Hyper-V** [PLDI 2022, MSR Blog 2021]
- ASN.1 DER [CPP 2023]
- QUIC [S&P 2021]
- TLS 1.3 [USENIX Security 2019]
- Beyond EverParse: RE#, efficient regex parsing with derivatives, deployed in .NET [POPL 2025]
- …

EverParse is open-source: http://project-everest.github.io/everparse

For more info: taramana@microsoft.com