Hardening Attack Surfaces with Formally Proven Binary Format Parsers

Nikhil Swamy
Microsoft Research

Tahina Ramananandro
Microsoft Research

Aseem Rastogi
Microsoft Research

Irina Spiridonova
Microsoft Research

Haobin Ni*
Cornell University

Dmitry Malloy
Microsoft

Juan Vazquez
Microsoft

Michael Tang
Microsoft

Omar Cardona
Microsoft

Arti Gupta
Microsoft

Abstract

With an eye toward performance, interoperability, or legacy concerns, low-level system software often must parse binary encoded data formats. Few tools are available for this task, especially since the formats involve a mixture of arithmetic and data dependence, beyond what can be handled by typical parser generators. As such, parsers are written by hand in languages like C, with inevitable errors leading to security vulnerabilities.

Addressing this need, we present EVERPARSE3D, a parser generator for binary message formats that yields performant C code backed by fully automated formal proofs of memory safety, arithmetic safety, functional correctness, and even double-fetch freedom to prevent certain kinds of time-of-check/time-of-use errors. This allows systems developers to specify their message formats declaratively and to integrate correct-by-construction C code into their applications, eliminating several classes of bugs.

EVERPARSE3D has been in use in the Windows kernel for the past year. Applied primarily to the Hyper-V network virtualization stack, the formats of nearly 100 different messages spanning four protocols have been specified in EVERPARSE3D and the resulting formally proven parsers have replaced prior handwritten code. We report on our experience in detail.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Security and privacy** → **Virtualization and security**.

Keywords: Network formats, Parser generators, Formal proofs

ACM Reference Format:
N. Swamy, T. Ramananandro, A. Rastogi, et al.. 2022. Hardening Attack Surfaces with Formally Proven Binary Format Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523708

*Work done during an internship at Microsoft Research

1 Introduction

As recently as in 2019, `tcp_input.c`, a file in the Linux kernel responsible for parsing TCP headers (and present in the kernel for 20 years or more) was patched to add a bounds check when parsing TCP options—without the check, it could have been possible to trigger an out-of-bounds access on the stack (Young-X 2019). Errors like this, a form of incorrect input validation, remain depressingly common in software today. Indeed, input validation failures are one of the leading causes of software security vulnerabilities, listed at #3 on Mitre's Top 25 Most Dangerous Software Weaknesses list, and implicated in several others (Mitre Corp 2020). The problem is especially serious in low-level software, such as operating system kernels or hypervisors, where a single missing check can compromise the entire software stack. Using a memory-safe language would help, but memory safety alone does not ensure that inputs are parsed correctly.

One compelling perspective on the problem is presented by Bratus et al. (2017), who argue that the root cause of such input validation failures is due to handwritten parsers improperly recognizing the formal language of a program's input. Bratus et al. point out that while there is widespread acceptance that "rolling your own crypto" is a bad idea, "rolling your own parser" can be just as bad and arguably worse, since it can be easier to exploit. They argue instead for tools to automate parser generation for a variety of data formats.

There is reason to be hopeful that the software community is beginning to embrace parser generation for a variety of protocol formats. Tools and libraries like Protocol Buffers, FlatBuffers, and even JSON offer parsing and serialization tools for exchanging structured data. However, these libraries choose the wire format, limiting their usability in scenarios where wire formats are dictated by external concerns—one cannot use, say, Protocol Buffers to parse TCP headers.

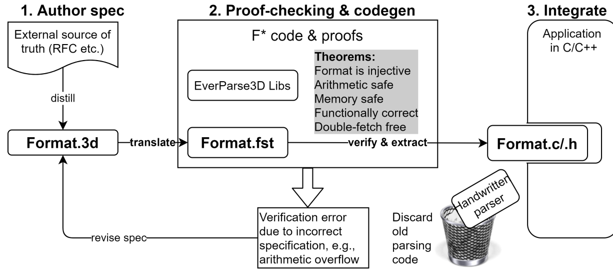


Figure 1. The EVERPARSE3D workflow

1.1 Low-level Binary Format Parsing, with Proofs

The format of a TCP header is specified in RFC 793 (Postel 1981) in English prose and several block diagrams. The format is designed to be efficiently "parseable" in a language like C—so much so, that most implementations work by defining a C data type to represent a TCP header (tcphdr in Linux), potentially relying on compiler pragmas to ensure that the compiler's layout of the type and the wire format coincide, and then simply cast an array of bytes holding a packet to the type and then proceed to validate and read its contents. For example, here's a fragment from Linux's tcp_parse_options, where skb is a buffer holding the packet to be processed.

```
const struct tcphdr *th = //cast untrusted bytes to tcphdr
    (struct tcphdr*)(skb⟶ head + skb⟶ transport_header);
int length = (th⟶ doff * 4) − sizeof(struct tcphdr);
ptr = (const unsigned char *)(th + 1);
while (length > 0) { /*roughly*/ check(*ptr); ptr++; length--; }
```

This kind of code, particularly when the input data is untrusted, is dangerous, since it involves a combination of pointer arithmetic to process variable-length data and nontrivial case analysis, as formats are often data dependent—the value of one field determines the set of legal values of some of the fields that follow.

Rather than writing this kind of code by hand, we offer a new tool, EVERPARSE3D, a parser generator that produces formally verified C code from a high-level description of a binary data format. Figure 1 outlines our proposed methodology, in three steps.

Step 1: Specification. Based on some external source of truth (ranging from published RFCs to legacy code), a programmer authors a data format specification in a language called 3D (standing for "Dependent Data Descriptions"). 3D offers a syntax similar to C's language of type definitions, including enumerations, structures, and unions, extended with dependent refinement types, powerful enough to specify many complex formats used in practice.

Step 2: Verified Code Generation. EVERPARSE3D compiles the user's 3D specification to a type description in F★, a proof-oriented programming language (Swamy et al. 2016). The type description is checked for well-formedness, e.g., to ensure that it does not use any unsafe arithmetic. From a well-formed type description, our verified libraries generate C code to validate byte streams against their specified format. The C code is proven to be safe, functionally correct, and free from double-fetches, important in concurrent settings to protect against certain classes of time-of-check/time-of-use attacks (Wang et al. 2017).

Step 3: Integration. Finally, one integrates the generated C code within a larger application. Any prior handwritten parsing code can be discarded, and the formally verified parser can be used in its place. The rest of the application can now rely on the guarantee that only inputs valid according to the specification are accepted and can work over a parsed representation as opposed to the raw bytes.

1.2 Contributions

Ramananandro et al. (2019) introduce EverParse, consisting of a library of parser and serializer combinators called LowParse coupled with a frontend to generate parsers from IETF RFCs, notably those used in the TLS protocol standard (Rescorla 2018). Their work targeted the use of parsers from within a verified F★ application, rather than our goal of using formally verified parsers within larger, unverified applications in C or C++. Our contributions evolve Ramananandro et al.'s EverParse in service of this new goal.

The design and mechanized formalization of 3D. We formalize 3D by giving it three related semantic denotations within F★: first, a *type* denotation, representing a 3D program as an F★ type; second, a *parser* denotation, describing the wire format as a pure function; and, third, a *validator* denotation, an imperative program that validates a stream of bytes while running user-provided parsing actions. Our main theorem, mechanically checked in F★, relates the three denotations, proving that the validator is a refinement of the parser, which, in turn, is a parser for values of the type denotation.

From semantics to a certified compiler. We turn our denotational semantics into a compiler by exploiting the first Futamura (1971) projection on dependently typed F★ programs. Specifically, for a given 3D program, by partially evaluating the validator denotation, we produce imperative F★ code extractable to C.

A new library of parser combinators. Underpinning our semantics is a new library of parser and parsing action combinators, enhancing Ramananandro et al.'s LowParse library in several ways, including the following highlights:

- Parsing with arbitrary data dependences on values that fit in a machine word, important for describing ad hoc tagged formats used in practice.
- Validators integrated with imperative parsing actions, allowing both the construction of parsed structures

from binary input formats as well as error-reporting callbacks for diagnostics. Although our specifications do not capture their functional correctness, actions are proven to be memory safe, to respect our double-fetch freedom guarantees, and to satisfy other auxiliary properties, e.g., constraints on their read and write footprints.

- Parsing from non-contiguous or streaming data sources, with on-demand fetching of data, important for use in scatter/gather-IO scenarios or when parsing large inputs that don't fit in memory.
- Compositional proofs that all our combinators are free from double-fetches.

Evaluating EverParse3D in Windows. EverParse3D has been in use in the Windows kernel for the past year. Applied primarily to the Hyper-V network virtualization stack, the formats of nearly 100 different messages spanning four proprietary protocols have been specified in 3D and resulting formally proven parsers have replaced prior handwritten code. Our work has contributed to virtualization-based security in Windows 11 and other releases.

Software artifacts. The application of our toolchain to Windows includes proprietary source code and is not publicly available. However, the 3D toolchain, including the EverParse libraries, the F* programming language, Z3 SMT solver, and the KaRaMeL C code generator are all open source and developed publicly on GitHub. Documentation, code samples, and links to our latest tool releases are available from https://project-everest.github.io/everparse.

2 A Tour of 3D

A 3D program is a sequence of type definitions. Unlike C, where type definitions do not produce any code, in 3D a type definition for T yields (in its simplest form) code with the following signature:

```
BOOLEAN CheckT(uint8_t *base, uint32_t len);
```

This is the type of a C procedure, CheckT, which, when given a pointer base to an array of bytes of length at least len, checks that the contents of base correctly represents a value described by the binary format specified by T. For example, if we have enum T { A=0, B=3, C=4 }, then CheckT simply checks that base contains at least four bytes (the default size of an enum is four bytes), and that it contains a little-endian representation of either 0, 3, or 4. A client program can use CheckT to validate the contents of base, particularly when it is untrustworthy, before accessing it.

Of course, a client program will want to do more with base than simply validate its contents. In general, one may want to parse the raw bytes of base into some more structured form. Or, in case parsing fails, one may want to recover a precise reason for the failure. Or, it may be that the raw bytes of a message are scattered in memory, rather than being stored contiguously in the base array. EverParse3D supports these features and more. For now, it will be helpful to just keep in mind that when defining a type T in 3D, one is implicitly defining a C procedure to check that a sequence of raw bytes accurately represents a T.

In designing 3D, our main goals were *expressiveness* and *explicitness*, and a notation easy to grasp for C programmers.

For expressiveness, we aimed to capture the data formats used in practice, such as in the Windows kernel. This demanded a rich language of data types—pleasingly, a fairly canonical core language of zero-order dependent types sufficed, as summarized by the type algebra below:

$$t ::= b \mid x{:}t\{e\} \mid x{:}t_0 \,\&\, t_1 \mid \text{if } e \text{ then } t_0 \text{ else } t_1$$

where b ranges over a rich collection of base types; $x{:}t\{e\}$ is a refinement type inhabited by $x{:}t$ such that e evaluates to true; $x{:}t_0\& \ t_1$ is the type of dependent pairs, where the type of the second field t_1 depends on the value x of the first field; and if e then t_0 else t_1 is the type defined by case analysis on e.

Although this algebra is sufficient for encoding arbitrary products and sum types with data dependence, it lacks recursive types and so a means to directly express unbounded data. However, as we will see, the base types b include several forms of variable-length collections, which have sufficed so far. Additionally, base types include $\perp$, the type with no inhabitants, whose validator fails immediately; the unit type of size 0, whose validator always succeeds; UINT8, the type of a single byte; and little- and big-endian versions of 2, 4, and 8-byte unsigned integers.

The goal of explicitness pervades our design in many respects, as we will see. Most significantly, our validators have no implicit allocations and do not parse wire formats into some default canonical representation. Instead, 3D allows decorating types with imperative actions, which gives explicit control over which parts of a message are read and into which structures.

In what follows, we present an overview of 3D using simple examples in its C like concrete syntax. We conclude the section with a specification of the TCP header format, which brings together several features of the language.

2.1 Structures, Dependency, Refinements

The type Pair below defines a structure with two fields.

```
typedef struct _Pair { UINT32 fst; UINT32 snd } Pair;
```

This defines a binary format of 8 bytes, with four bytes to represent each little-endian UINT32. Unlike in C, the layout and alignment of fields in 3D in a struct is explicit. So, by default, the type ByteInt below is represented in 5 bytes, with no alignment padding.

```
typedef struct _ByteInt { UINT8 fst; UINT32 snd } ByteInt;
```

3D offers an attribute on a type definition to trigger the insertion of alignment padding between the fields of a struct so as to match the C ABI. However, for the purposes of this

paper, we ignore this option and require the layout of a type to be entirely explicit. EverParse3D also emits static assertions in the generated C code to check that the user-specified layout of a type and a C compiler's view are compatible.

3D allows dependence among the fields of a structure, e.g., an OrderedPair's fst field is less than or equal to its snd field.

```
typedef struct _OrderedPair {
  UINT32 fst;
  UINT32 snd { fst <= snd };
} OrderedPair;
```

The refinement associated with a type (here fst ≤ snd) can be any boolean-valued expression built from the names in scope and a small but expressive language of pure operators (integer comparisons, arithmetic etc.) and conditional expressions. Enumerated types in 3D are just syntactic sugar for integer refinement types.

2.2 Value-parameterized Types

Type definitions in 3D can be parameterized by values. For example, PairDiff(n) is the type of a pair of UINT32, whose snd component is at least n greater than fst.

```
typedef struct _PairDiff (UINT32 n) {
  UINT32 fst;
  UINT32 snd { fst <= snd && snd − fst >= n };
} PairDiff;
```

Refinement expressions are checked for arithmetic safety, ensuring the absence of overflow and underflow errors. In the refinement above, the conjunction operator && is left-biased, and the check fst ≤ snd ensures that the subtraction following it, snd − fst, does not underflow. Without the fst ≤ snd check, F⋆'s would reject the program due to a potential underflow.

Parameterized types can be instantiated and used within other types. For example, the type Triple below defines a dependent pair of a bound and a PairDiff(bound).

```
typedef struct _Triple {
   UINT32 bound;
   PairDiff(bound) pair;
} Triple;
```

3D does not yet offer type-parameterized types, since our primary goal has been to represent types in C, rather than templatized types in a language like C++. However, we expect our compilation technique to extend to type-parameterized types in the future, which would help promote more modular specifications.

2.3 Casetype: Contextually Discriminated Unions

When using untagged unions in C, determining which case of the union is active is left implicit in the code. In 3D, the case of a union is made explicit using a type defined by case analysis. The type ABCUnion(tag) is a union of three cases: when tag=A, it is a UINT8; when tag=B it is a UINT16; and when tag=C it is a PairDiff(17). As with struct fields, by default, 3D does not insert any padding—so, unlike a C union, the length of ABCUnion(tag) is variable, depending on the value of the tag itself.

```
casetype _ABCUnion (ABC tag) {
   switch (tag) {
   case A: UINT8 a;
   case B: UINT16 b;
   case C: PairDiff(17) c;
}} ABCUnion;
```

Case types are often used within some larger type, where some information in the context determines the value of the tag. For example, in TaggedUnion below, the value of the first field determines the case of the third payload field.

```
typedef struct _TaggedUnion {
   ABC tag;
   UINT32 otherStuff;
   ABCUnion(tag) payload;
} TaggedUnion;
```

2.4 Variable-Length Data

To represent variable-length data in C, programmers resort to various ad hoc conventions. For instance, to package a variable-length array with its length, a common style in C is to use a zero-length array as the last field of a struct, e.g., `struct { UINT32 len; UINT64 array[0] }`. 3D instead offers several forms of variable-length array and string types (in addition to the variable-length casetype covered earlier).

The type VLA pairs a len field with a TaggedUnion array whose length in bytes (*not* the element count) is exactly len.

```
typedef struct _VLA {
   UINT32 len;
   TaggedUnion array[:byte-size len];
} VLA;
```

Another form of variable-length data is the zero-terminated string, where T f[:zeroterm−byte−size−at−most n], is the type of a zero-terminated string of T (with a well-defined zero element) consuming no more than n bytes. We introduce other forms of variable-length types supported by 3D when presenting more examples.

2.5 Parsing Actions

So far, all our examples produce validators that simply check whether a given sequence of bytes is a valid representation of a given data format. However, 3D also provides a safe way of accessing the fields of a structure by associating imperative parsing actions with parts of a format specification.

In the example below, the type VLA1 takes an *out-parameter*, marked with the mutable qualifier. The field another is decorated with an *action*, a piece of imperative code that is executed by the generated validator immediately after the associated field has been validated.

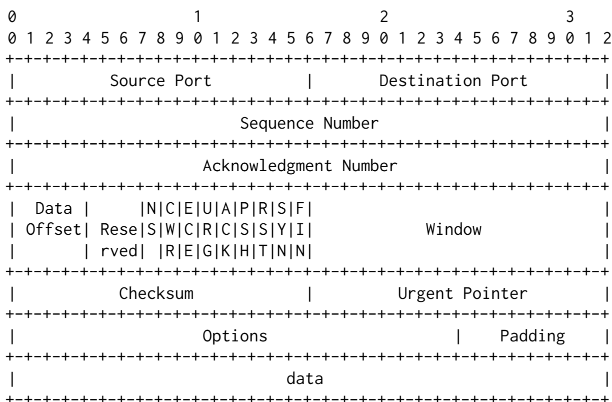
```
typedef struct _VLA1(mutable UINT64* a) {
  UINT32 len;
  TaggedUnion array[:byte-size len];
  UINT64 another {:act *a = another };
} VLA1;
```

In this case, the validator for VLA1(a) reads the first len field; then validates the variable-length TaggedUnion array; then validates the UINT64 another field—if and when this last step succeeds, the action is executed, assigning the value of another to the out-parameter a. Our verifier ensures that actions are safe to execute. As such, by using an action to read the value of another into a, the client program is saved the trouble and risk of accessing the another field by reading from a computed offset (base + 4 + len) from the base pointer.

More complex parsing actions allow, in general, to invoke, in a type-safe way, user-provided callbacks. In our experience, when interfacing with a C application, one typically uses a mixture of actions to parse variable-length data that cannot be represented in C types, while fixed-length parts of a format, after validation, are simply cast to a C type and read using C's native support for structured field access.

2.6 Putting it together: Parsing a TCP Header

The format of a TCP segment header is specified in RFC 793 with several revisions since the original 1981 version.



The picture above is shows the bit-level layout of the header, with data dependences among the fields—the (variable) length of options is constrained by `Data Offset`, since the `data` field must start `Data Offset * 4` bytes from the start of the header. Note the picture is inaccurate since the Options field is actually variable length, padded out to consume a multiple of 4 bytes. Off-the-shelf tools like Protocol Buffers cannot express ad hoc binary formats like this.

Also of interest for our purposes is the structure into which the `Options` array is parsed. We use a struct named OptionsRecd, similar to tcp_options_received defined in the Linux (2021a) TCP implementation. 3D allows specifying structures that are used in parsing actions, while marking them with the output keyword to indicate that no validation code should be generated for such a struct. OptionsRecd contains aggregated fields for all the TCP options.

```
output typedef struct _OptionsRecd {
  UINT32 RCV_TSVAL; /*Timestamp TCP option related fields*/
  UINT32 RCV_TSECR;
  UINT16 SAW_TSTAMP : 1; ... /*Fields related to other options*/
} OptionsRecd;
```

With the type of our "parse tree" set, we can define a parser for TCP headers as shown below.

```
typedef struct _TCP_HEADER(UINT32 SegmentLength,
                           mutable OptionsRecd* opts,
                           mutable PUINT8* data) {
  PORT SourcePort; ... /*Other fields are elided*/
  UINT16 DataOffset:4
  { 20 <= DataOffset * 4 && DataOffset * 4 <= SegmentLength };
  ...
  OPTION(opts) Options[:byte-size (DataOffset * 4) − 20];
  UINT8 Data[:byte-size SegmentLength − (DataOffset * 4)]
      {:act *data = field_ptr }
} TCP_HEADER;
```

A parser for TCP_HEADER(len, opts, data) checks that the input buffer contains a TCP segment header that fits in exactly len bytes. It parses Options into the OptionsRecd struct, and stores a pointer to the Data field in data.

The DataOffset field consumes 4 bits of a 16 bit field, since its value is in units of 32 bit words, we multiply by 4 for the byte offset of Data. The refinement checks that the byte offset of Data is at least 20 (the size of all the static fields just prior to the Options field), and that it fits within SegmentLength.

The Options field is a variable-length array of OPTION(opts) consuming all the bytes remaining following the Urgent pointer until the beginning of the Data field—as we will see, our specification includes the Padding field.

For the variable-length Data field, after validating that the input is large enough to hold it, we take a pointer to it (using the field_ptr primitive action) and store it in the out parameter data. The client may then call into another validator specific to the data, copying or transforming it in a single pass into another structure.

Each element of the Options array is a tagged union, with an OptionKind field describing the payload that follows.

```
typedef struct _OPTION(mutable OptionsRecd* opts) {
  UINT8 OptionKind;
  OPTION_PAYLOAD(OptionKind, opts) PL; } OPTION;
```

The payload itself is a casetype, branching on OptionKind. In case we have an OPTION_KIND_END_OF_LIST, the specification dictates that all the bytes that follow are zero, including the padding if any, until the start of the Data field. We use 3D's all_zeros type, a variable-length type that accepts strings of zeroes up to the length of the enclosing type:

```
casetype _OPTION_PAYLOAD(UINT8 OptionKind,
                         mutable OptionsRecd* opts) {
  switch(OptionKind) { ...
    case KIND_END_OF_OPTION_LIST: all_zeros EndOfList;
    case KIND_TIMESTAMP: TS_PAYLOAD(opts) Timestamp; }
} OPTION_PAYLOAD;
```

We have kind-specific formats in the other cases, e.g., TS_PAYLOAD for the timestamp TCP option:

```
typedef struct _TS_PAYLOAD(mutable OptionsRecd* opts) {
  UINT8 Length { Length == 10 }; UINT32 Tsval;
  UINT32 Tsecr {:act opts->SAW_TSTAMP = 1;
                     opts->RCV_TSVAL = Tsval;
                     opts->RCV_TSECR = Tsecr;}
} TS_PAYLOAD
```

The interesting part is that when we succeed in parsing the Tsecr field of the TS_PAYLOAD, the action updates the corresponding fields in opts. This is similar to the handling of the timestamp TCP option in Linux (2021b) except in 3D, the options parsing code is written in a declarative style, free of any user-written pointer arithmetic. Other options are parsed in a similar manner—our online code repository shows the complete 3D TCP spec.

The generated C code for CheckTcpHeader requires base to at least contain len bytes, and checks that it contains a valid TCP_HEADER stored in SegmentLength bytes and, if so, the return value is TRUE, the options are written into opts, and *data points to the start of the Data field in base.

```
BOOLEAN CheckTcpHeader(uint32_t SegmentLength,
                       OptionsRecd *opts, uint8_t **data,
                       uint8_t *base, uint32_t len);
```

In the next section, we describe our formalization and implementation of 3D and its toolchain in F★, with proofs that the generated validators are safe and behave according to their formal specification.

3 Implementing EverParse3D

3D is formalized and implemented as an embedded domain-specific language (DSL) within F★, a programming language and proof assistant based on a dependent type theory (like Coq, Agda, or Lean). F★ also offers an effect system, extensible with user-defined effects, and makes use of SMT solving to automate some proofs. We use F★'s syntax in this section, and provide a brief primer below.

F★'s toolchain includes support for several other embedded DSLs, notably Low* (Protzenko et al. 2017), an imperative language with a Hoare logic for functional correctness proofs. Low* programs can be extracted from F★ to C, and a metatheorem establishes that extracted C programs simulate the F★ program. However, the tool implementing this extraction pipeline (named KaRaMeL), although modeled after this metatheory, is not formally verified and is part of our trusted computing base, which also includes the F★ typechecker and the Z3 SMT solver. The 3D parser and frontend that desugars the C-like concrete syntax into F★'s formal notation is also trusted to accurately reflect the user's intention. Modulo these trust assumptions, we offer mechanically checked theorems attesting to the soundness of EverParse3D.

Syntax: Binders, lambda, arrows, computation types. F★ syntax is roughly modeled on OCaml (val, let, match etc.) with differences to account for the additional typing features. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1) \ldots (b_n) \rightarrow t$ introduces a lambda abstraction, whereas $b_1 \rightarrow \ldots \rightarrow b_n \rightarrow c$ is the shape of a curried function type. Refinement types are written b{t}, e.g., x:int{x≥ 0} is the type of non-negative integers (i.e., nat). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. The c to the right of an arrow is a *computation type*. An example of a computation type is Tot bool, the type of total computations returning a boolean. By default, function arrows have Tot co-domains, so, rather than decorating the right-hand side of every arrow with a Tot, the type of, say, the pure append function on vectors can be written #a:Type → #m:nat → #n:nat → vec a m → vec a n→ vec a (m + n), with the two explicit arguments and the return type depending on the three implicit arguments marked with '#'. We often omit implicit binders and treat all unbound names as implicitly bound at the top, e.g., vec a m → vec a n → vec a (m + n)

3.1 Parsers, Validators, and Readers

The C code emitted by our toolchain is a binary format validator. As such, our semantics revolves around two central notions: a specificational *parser* describing the format and an imperative *validator* whose type relates it to the parser.

Core parsers. Drawing on the work of Ramananandro et al. (2019), a core_parser k t is a function f which when applied to b:bytes can either fail (returning None) or succeed and return Some (v:t, n:nat), where n describes the number of bytes of b that were consumed. Additionally, f is required to be *injective*, meaning that f uniquely determines the value v that can be represented by the bytes b, a useful property that ensures that the formats defined by parsers do not admit security bugs that arise due to parsing ambiguities.

```
let core_parser (k:parser_kind) (t:Type) =
   f: ((b:bytes) → option (t & n:nat{ n ≤ length b }))
     { injective f ∧ has_kind f k }
```

Parsers and their kinds. Further, a notion of *parser kind* provides metadata about the parser, placing, for instance, lower and upper bounds on the number of bytes consumed by the parser. The full details of parser kinds are described by Ramananandro et al., however, as we will see in §3.2,

parser kinds are essential to ensure that 3D programs are well defined. For our purposes, it suffices to work with an abstraction of parser kinds called pk nz wk, where nz:bool records whether or not a parser consumes at least some non-zero bytes, and wk:weak_kind, describes whether (1) a parser consumes all the bytes given to it (wk = ConsumesAll); (2) whether it consumes a prefix of the bytes and is insensitive to the remaining bytes (wk = StrongPrefix); or (3) if not much else is known about it (wk = Unknown). A small algebra of kinds allows them to compose sequentially with and_then, and to be partially ordered according to glb, or greatest lower bound. The type of parsers we use in the remainder of this section is parser k t, core parsers for type t indexed by pk nz wk kinds.

```
type weak_kind = | ConsumesAll | StrongPrefix | Unknown
let pk (nz:bool) (wk:weak_kind) = k:parser_kind
    { (nz ⟹ consumes_non_zero_bytes k) ∧
      respects_weak_kind k wk }
let parser (k:pk nz wk) (t:Type) = core_parser k t
```

Validators. A parser is a pure function operating over a functional representation of the input bytes. While suitable for a specification, executing it would incur many implicit allocations, making it unsuitable for integration in a C application. Instead, we use a *validator*, an imperative procedure that refines a given parser, without any implicit allocation. Our validators may also run a user's explicit parsing actions. The type of a validator, validate_with_action p i l ar, is shown in Figure 2. This type captures the main guarantees provided by our system and appears in the main theorem of §3.3, so we describe it in detail.

An inhabitant v:validate_with_action p i l ar is an imperative procedure in Low*, where p is a specificational parser describing the functional behavior of v; i is a memory invariant of all the imperative parsing actions that v may execute; l describes the set of memory locations that v may modify, i.e., the out parameters; and ar, a boolean standing for "allowed to read", signifying whether a continuation can safely read the input byte stream without incurring a double fetch.

Input streams: The first argument of a validator is st, an instance of a typeclass of input streams, encapsulating various forms of data sources on which a validator can be run. The next argument, pos, is the current position in the stream, from which v is to start validating the format specified by p. The simplest instance of an input_stream_t is an array of bytes, but our framework can be instantiated for use with arbitrary streams, e.g., to validate huge formats that don't fit in memory, or to validate messages that are scattered in memory. Our input streams are designed with a permission model that allows us to prove that validators are double-fetch free. In particular, reading a byte from the stream advances it and makes it provably impossible to read that byte again. One can also check if a stream contains some number of bytes, without advancing it.

```
let validate_with_action (p:parser k t) (i:inv) (l:loc) (ar:bool) =
 st:input_steam_t → pos: pos_t → Stack uint64
 (requires λh →
   disjoint l (fp st) ∧ sti st h ∧ i (fp st) h ∧ pos==read_len st h)
 (ensures λh res h' →
   sti st h' ∧ i (fp st) h' ∧ modifies (l ∪ perm_fp st) h h' ∧
   let s, s' = remaining st h, remaining st h' in
   s' `is_suffix_of` s ∧
   if is_success res then
     if ar then res ≥ pos ∧ s' == s ∧ valid_pos p s s'
     else valid_consumed p s s' ∧ res == read_len st h'
   else not (is_action_failure res) ⟹ not (valid p s))
```

Figure 2. The type of a validator (simplified)

Effect and result type: The effect of running v st pos is described by Stack uint64 (requires pre) (ensures post), a computation type. The effect label Stack proves that v does not allocate on the heap (meeting our requirement of no implicit allocations). The return type is uint64, a 64-bit unsigned integer describing the position in the stream reached after running the validator. We reserve a small number of bits in the result type to hold error codes, in case the validator fails. The pre is a precondition, a predicate on the input state h; the post is postcondition, a predicate relating the input state h, the result res, and the output state h'.

Precondition: The precondition requires the set of mutable locations l to be disjoint from (fp st), the footprint of the input stream; the stream's internal invariant (sti st) is expected to hold; the invariant i of the actions are expected to hold, and can relate the actions to the footprint of the stream; and pos is expected to be current position of the bytes in the stream that have been read so far.

Postcondition: The postcondition restores the invariants sti and i and the modifies clause states that v mutates at most the locations in l and the read permissions of the prefix of the stream (not the stream itself). The variables s and s' represent a logical view of the entire suffix of the stream in the initial and final states. If the result res is successful, then we have two cases. If the ar flag is set, then v has validated that the res-length prefix of s is well-formatted according to the specification p, *without* consuming any input bytes (e.g., it must have been possible to validate p by just checking the stream's capacity). If ar is not set, then s has been validated up to res and those bytes have been consumed. If the result res is not successful, and the error code indicates that no action failed, then the input buffer is ill-formed with respect to p, i.e., v's success and failure behaviors are characterized by p, except for additional failures that can be raised by :check-actions. The behavior of actions is underspecified—we only prove that validators maintain action invariants and mutate at most the out parameters. In §4.3, we consider adding functional correctness specifications for actions.

Readers. After running a v: validator p i l ar, if ar = true, then one can run a *reader*, a procedure of type reader (p:parser k t), which reads a x:t out of a stream known to be valid with respect to p. While, in principle, it is possible to define readers for all types, reading a complex value involves struct-passing, which is not always efficiently handled by C compilers. As such, we generally restrict ourselves to *leaf readers*, readers for word-sized values, like the various machine integer types, so complex values are read a word at a time. When validating a field, if the continuation depends on the value of that field (e.g., because it appears in a refinement, type parameter, or an action), we immediately read the value on to the stack while validating it. As such, in a single pass through the input stream, we read and validate all the fields that we need without incurring double fetches.

The types of parsers, validators, and readers set our goal posts. We aim to give a semantics such that every well-typed 3D program can be interpreted as an inhabitant of validate_with_action—if so, then this interpretation yields a verified imperative procedure in Low*, extractable to efficient C code, to correctly decide if an input C array of bytes is well-formatted according to its 3D specification, while running the user's chosen parsing actions.

Error handling. What we have described is a simplification of the type of validators. In reality, validators take two additional arguments, an application context ctxt and an error-handling callback. When a parsing error is found, we call the error handler, passing it the ctxt, together with the type at which the failure occurred, the field within that type, and a reason for the error. The handler can process and record this as needed in the ctxt. We then then propagate the error code to the caller. As we pop the parsing stack, we call any error handlers encountered, thereby allowing applications to reconstruct the full stack trace in case an error.

3.2 A Type System for 3D

Figure 3 defines a typed abstract syntax for 3D in F⋆— we show 6 representative constructors in the language, eliding 11 other constructs that are similar in spirit. An ad hoc front end for 3D translates the concrete C-like syntax used in §2 into an element of the type typ. The representation of typ uses a variety of techniques, including deep and shallow embeddings, and other folklore techniques that Chlipala (2021) recently referred to as *mixed* embeddings. The indexing structure of typ defines a type system for 3D, which abstracts a program typ k i l ar by its four indices—k, its kind; i and l, a memory invariant and footprint of its actions; and ar, a flag indicating whether or not the format described by the program has a reader. The rules of composition of a 3D program restrict and combine these indices in various ways to ensure that every inhabitant of typ can be given a semantics. As we will see, every term of type typ k i l ar enjoys a threefold denotational semantics: as a type t; a p:parser k t for that type; and a validator of type validate_with_action p i l ar.

Pre-denoted types. The first constructor of typ is T_shallow, which allows for any inhabitant of the F⋆ type dtyp k i l ar to be lifted to a typ

```
type dtyp k i l b = { t:Type; p:parser k t;
                      r:option (reader p) { b ⟹ r≠None};
                      v:validate_with_action p i l b }
```

A dtyp is a *shallow* embedding of an existing F⋆ program of a type suitable for a 3D program, packaging a type t; a parser p for that type; an optional reader for that parser; and a validator refining the parser. T_shallow allows us to introduce primitive types into the 3D language just by defining a suitable dtyp for them, e.g., dtyp_u32 : dtyp u32_kind ⊤{} true is a primitive for parsing a UINT32. Further, every type definition provided by the user also introduces a dtyp instance, allowing subsequent type definitions to refer to prior ones. This is important since, although typ can be used compositionally, if one were to simply use typ literals everwhere, the denotation of the resulting type as a validator would fully inline the validators of all the types it mentions, leading to a code blowup. With T_shallow we avoid these blowups and ensure that the procedural structure of our generated code matches the type definition structure of the source specification, an important criterion for code acceptance.

Pairs. The 3D type definition struct { UINT32 f; UINT32 g} is desugared to T_pair (T_shallow dtyp_u32) (T_shallow dtyp_u32). The indexing structure of typ allows any two types to be paired; sequentially composing their kinds, conjoining their invariants and footprints; and marking the type as unreadable, since we intentionally restrict readers to word-sized values.

Casetypes. T_if_else is used to represent a casetype. Although at the surface language we support n-ary case analysis with switch, this is desugared to nested conditionals in the front end. In T_if_else b t1 t2 the conditional expression b is shallow, i.e., any F⋆ boolean expression. T_if_else weakens the kinds of the branches to their greatest lower bound. For example, switch (t) { case V_1: t_1; case V_2: t_2} is desugared to T_if_else (t=V_1) [[t_1]] (T_if_else (t=V_2) [[t_2]] typ_⊥), where [[t]] is the desugaring of t and typ_⊥ is the always-failing parser for the empty type.

Refinements. T_refine d r refines the shallow type d with the refinement predicate r. The type d must support a reader, since validating the refinement involves reading a value v and checking if r v holds. The refinement r is a shallow F⋆ boolean function, defined over the type that d parses. For example, UINT32 f { f < 17 && f + g < 42 } is represented as T_refine dtyp_u32 (λf → f < 17 && f + g < 42). By representing refinements shallowly, we can easily use F⋆'s SMT-assisted dependent type checker to check that refinement expressions

```
type typ : pk nz wk → inv → eloc → bool → Type =
| T_shallow: dtyp k i l b → typ k i l b
| T_pair:
  t1:typ k1 i1 l1 b1 → t2:typ k2 i2 l2 b2 →
  typ (and_then k1 k2) (i1 ∧ i2) (l1 ∪ l2) false
| T_if_else:
  b:bool → t1:typ k1 i1 l1 b1 → t2:typ k2 i2 l2 b2 →
  typ (glb k1 k2) (i1 ∧ i2) (l1 ∪ l2) false
| T_refine:
  t1:dtyp k1 i1 l1 true → refine:(dtyp_type t1 → bool) →
  typ (filter k1) i1 l1 false
| T_dep_pair_with_refinement_and_action:
  base:dtyp k1 true i1 l1 →
  refine:(dtyp_type base → bool) →
  k:((x:dtyp_type base { refine x }) → typ k2 i2 l2 b2) →
  act:((x:dtyp_type base { refine x }) → action i3 l3 b3 bool) →
  typ (and_then (filter k1) k2) (i1 ∧ i2 ∧ i3) (l1 ∪ l2 ∪ l3) false
| T_byte_size:
  n:U32.t → t:typ k i l b → typ kind_nlist i l false
```

Figure 3. A typed abstract syntax for 3D programs (partial)

are safe, e.g., that there is no arithmetic overflow. In contrast, had we embedded refinements deeply, we would have to build a safety checker for them from scratch.

Dependent pairs and actions. The most general form of sequential composition of types in 3D is represented by T_dep_pair_with_refinement_and_action. This allows composing a parser for base (equipped with a reader) with a refinement refine; a continuation k returning a typ but whose binder is a shallow term corresponding to the refined interpretation of base; and a dependent action act, where action is a small mixed datatype representing the monadic sub-language of 3D parsing actions, shown below.

```
type action : inv → eloc → Type → Type =
| Deref: x:pointer a → action (live x) {} a
| Assign: x:pointer a → rhs:a → action (live x) {x} unit
...
| Bind: head:atomic i0 l0 t0 → k:(t0 → action i1 l1 t1) →
        action (i0 ∧ i1) (l0 ∪ l1) t1
| Cond: hd:bool → then_:(_:unit{hd} → action i0 l0 t) →
        else_:(_:unit{not hd} → action i1 l1 t) →
        action (i0 ∧ i1) (l0 ∪ l1) t1
```

Actions include primitives like Deref and Assign to read or write shallowly embedded pointer values, with indexes describing the pointers they expect to be live, the pointers they may mutate, and their return type. Actions can be composed in sequence (with Bind) or conditionally using Cond—in Cond, each branch is typeable in a context assuming the branch condition or its negation, as appropriate.

Variable-length data. Finally, we show T_byte_size n t, which represents the surface syntax t f [:byte−size n], i.e., an array of t whose length in bytes is n.

3.3 Three Related Denotations

Our main theorem has the following type, stating that every well-typed 3D program t:typ k i l b has an interpretation as a validator. The type of as_validator t states that it refines as_parser t, the parser interpretation of t; which in turn references as_type t, the type interpretation.

```
val as_validator(t:typ k i l b):validate_with_action (as_parser t) i l b
val as_parser(t:typ k i l b):parser k (as_type t)
val as_type(t:typ k i l b):Type
```

With the carefully chosen structure of typ, defining these three denotations is relatively straightforward dependently typed programming, with proofs automated using F⋆'s SMT-assisted typechecker. We show a few cases, where parse_pair, validate_pair etc. are combinators from the LowParse3D library, proven correct once and for all.

```
let rec as_type t = match t with ...
  | T_shallow d → dtyp_type d
  | T_pair t1 t2 → as_type t1 & as_type t2
  | T_refine t f → x:as_type t { f x }
let rec as_parser t = match t with ...
  | T_shallow d → dtyp_parser d
  | T_pair t1 t2 → parse_pair (as_parser t1) (as_parser t2)
  | T_refine t f → parse_filter (as_parser t) f
let rec as_validator t = match t with ...
  | T_shallow d → dtyp_validator d
  | T_pair t1 t2 → validate_pair (as_validator t1) (as_validator t2)
  | T_refine t f → validate_filter (as_validator t) f
```

Given a 3D program t=T_Pair typ_u32 typ_u32, to run the validator on some input stream st, one could simply run as_validator t base 0: this would work, but it would be slow, since we would, in effect, interleave the interpretation of t with the actual work of validating the contents of base. Further, we actually want executable C code from our toolchain, not a slow interpreter—the partial evaluation capabilities of a dependent type checker can help.

F⋆'s type checker, like many other dependent type checkers, already has a facility to symbolically reduce F⋆ terms on an abstract machine. For example, to decide if a type nlist a ((λx → (x + 1) + y) 1) is equivalent to nlist a (2 + y), F⋆ reduces both types as much as it can and then compares them for syntactic equality. This machinery is the basis for "type-level computation", a distinctive feature of dependent types, and as our small example illustrates, it can be used to even partially evaluate terms that contain variables, though things have to be carefully arranged to ensure terms reduce as much as one expects, e.g., ((λx → (x + 1) + y) 1) reduces to 2 + y, whereas ((λx → x + (1 + y)) 1) would only reduce to 1 + (1 + y),

Module	.3d LOC	.c/.h LOC	Time (s)
NVBase	106	549/138	7.0
NvspFormats	947	4195/90	12.8
RndisBase	102	226/121	4.6
RndisHost	776	3157/200	12.7
RndisGuest	1157	5612/165	14.6
NetVscOIDs	553	2594/90	11.4
NDIS	1385	6060/253	17.2
VSwitch total	5026	22393/1057	82.1
Ethernet	143	521/48	5.3
TCP	279	1689/61	11.1
UDP	27	150/38	4.8
ICMP	190	2147/122	9.3
IPV4	78	556/61	7.4
IPV6	78	354/40	6.5
VXLAN	24	221/38	4.9

Figure 4. Using EVERPARSE3D on various protocol formats

the latter only being provably equal to 2 + y (e.g., relying on the SMT solver).

Exploiting the first Futamura (1971) projection, given a concrete program t, we partially evaluate as_validator t by reducing it on F*'s normalization-by-evaluation abstract machine. Having eliminated all the interpreter overhead (after careful tuning of our definitions to make sure that everything is arranged to reduce as much as we want), we first get a fully applied term built from LOWPARSE3D combinators only, e.g., validate_pair validate_u32 validate_u32, which, after some more partial evaluation on F*'s normalizer, produces fully specialized first-order code that can be extracted by F* to C. Once extracted to C, validating a pair looks like:

```
uint64_t positionAfterFst = ValidateU32(Input, StartPosition);
if (IsError(positionAfterFst)) { return positionAfterFst; }
return ValidateU32(Input, positionAfterFst);
```

Starting from our C-like 3D front end, we use SMT-assisted refinement typechecking; followed by dependently typed generic programming and a denotational semantics involving several layers of indexed-monadic semantics of parsers, validators, and actions; then partial evaluation; finally yielding idiomatic, high-performance, provably correct and safe C code.

4 Hardening Windows Virtual Switch

Figure 4 summarizes a quantitative evaluation of EVERPARSE3D at work, including execution times of the tool running on an Intel Core-i7 laptop. We have used it to specify hundreds of message types spanning 11 networking protocols. Our main experience has been its use in Windows Virtual Switch, where we have deployed in a production setting more than 23,000 lines of verified C code produced by our toolchain.

Virtual Switch is a component in the Windows kernel which provides para-virtualized access to the network to enlightened virtual machine (VM) guests. The architecture of Virtual Switch is depicted in Figure 5. A component called vSwitch runs in the privileged root partition of the host and dispatches network packets to and from NetVsc, a component running on the guest, and the host's network interface card. Since vSwitch runs in the host's kernel, it cannot trust NetVsc to send it well-formed messages. Conversely, in some confidential computing scenarios (Microsoft Corp 2021; Russinovich 2021) NetVsc does not trust vSwitch either and must properly validate the messages it receives.

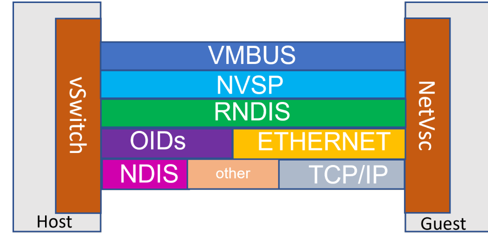


Figure 5. Layering of Virtual Switch protocols

The messages exchanged between vSwitch and NetVsc are structured into several layered protocols. At the base layer, a Hyper-V interface known as the VMBUS serves as the basic transport for packets. A Virtual Switch packet on the VMBUS begins with an NVSP (Network Virtualization Service Protocol) header. Some NVSP messages encapsulate RNDIS (Remote Network Driver Interface Specification) messages, which in turn contain either an Ethernet frame, or one of several OIDs (Operation Identifiers). Each OID itself carries a payload, which can include messages offloaded to a protocol called NDIS (Network Driver Interface Specification).

So far, we have focused on specifying formats for NVSP, RNDIS, OIDs, and NDIS in 3D, using 137 structs, 22 casetypes, and 30 enum type definitions. While describing those message formats required careful specification engineering and discovery (some of these protocols involve proprietary formats with a long history of evolution), because of our formal guarantees of correctness and safety, the generated C code can be deployed with confidence. To be clear, verified parsing alone does not provide end-to-end formal assurances about the entire Virtual Switch; however, it does address a common source of vulnerabilities on its main attack surface. Based on our experience, we are currently working on expanding our coverage of verified parsers in the networking stack by specifying and integrating verified parsers for Ethernet and the TCP/IP protocol suite—Figure 4 also includes information about our specification of these protocols, though these are not yet released in production code, we are currently working on their integration

Performance evaluation. Of course, Virtual Switch is also a performance-critical component. A major concern was whether adding additional input validation checks would reduce its performance. As such, we spent a substantial amount

of effort on optimization, though sharing detailed performance profiles on our production code is infeasible. However, since EverParse3D parsers operate in place and, because of their double-fetch freedom, guarantee to never read a memory location more than once, they are inherently fast. Further, we designed our specifications and input validation strategy in a layered manner, staying faithful to the layered protocol structure and incrementally parsing each layer rather than incurring the upfront cost of validating a packet in its entirety before processing. For acceptance for deployment, our verified parsers were required to introduce no functionality regressions and incur no more than a 2% cycles-per-byte performance overhead bar. Detailed internal measurements comparing the performance of the old and new versions of the code confirmed that we met both these targets. In some configurations, our verified parsers were found to be marginally faster than the prior handwritten code, since our code is systematically designed to be double-fetch free hence avoiding some copies that the prior code incurred. In summary, our parsers met internal "no-performance-regression" goals which allowed the feature to go into production.

Security evaluation. The re-engineering of parsers in Hyper-V's Network Virtualization stack using EverParse3D was motivated primarily as a way to improve security. As with performance and functionality testing, our code was also subject to security review and testing, and was deployed only after passing this scrutiny. The full details of our security reviews and tests are also hard for us to disclose, however we summarize our process and findings below.

Security code reviews and tests were performed by a separate team, independent from both the EverParse authors and the Network Virtualization team. In addition to code reviews, the security evaluation included a study of historic and open security bugs that our work was aimed at addressing and closed or would have closed.

Security testing included fuzzing efforts, which did not uncover any bugs in our parsing code. An interesting anecdote is that once EverParse3D's parsers were integrated into Virtual Switch, several fuzzers stopped working effectively, since their fuzzed input would always be rejected by our parsers, preventing the exploration of deeper code paths. We have subsequently been working with the fuzzing teams to use our formal specifications to help design these fuzzers, ensuring that the fuzzers only produce well-formed inputs. This is an interesting synergy between formal verification & specification projects and other bug-finding efforts.

Additionally, we have also fuzzed the code produced by EverParse3D toolchain by running Sage (Godefroid et al. 2012) (an SMT-assisted whitebox fuzzer) on our code for several days without uncovering any bugs.

Productivity and maintenance. Aside from the technical improvements to security, the use of EverParse3D also increased the productivity of the team, with the engineering lead of the Network Virtualization team commenting that EverParse3D enabled them to meet project timelines that could not otherwise have been met. Additionally, while maintaining our code in the Windows code base over the past 18 months, EverParse3D 's formal proofs of correctness and security have provided confidence that code changes don't introduce unnecessary risk. For instance, once, when doing a large refactoring of 3D specifications, we proved in $F^\star$ that no semantic changes were inadvertently introduced, by relating the initial and refactored specifications semantically. EverParse3D and its underlying toolchain, including $F^\star$ and Z3, are also integrated with the build environment of Windows, so that all developers can easily generate code from 3D specifications as part of their regular builds.

Virtualized networking hardened with EverParse3D is available broadly in Windows today, including in the recent Windows 11 release. In this section, we highlight several interesting elements of our work from each of the four protocols that we have addressed so far.

4.1 NVSP

The host vSwitch component handles 13 different kinds of NVSP messages, where each message is packaged with a MessageType tag, as specified below.

```
struct NVSP_HOST_MESSAGE {
  UINT32 MessageType;
  NVSP_HOST_MESSAGES(MessageType) Message; }
```

The message type NvspMessage1TypeSendRNDISPacket indicates an enclosed RNDIS message payload, including an offset into the packet buffer at which the RNDIS message begins. All the remaining message types involve fixed-size structures of various kinds. As such, on the host side, when receiving an untrusted NVSP message, our parsers check that it is a valid NVSP_HOST_MESSAGE, and then the rest of vSwitch's code proceeds as usual: it inspects that MessageType tag and based on it casts the Message field to the expected case of NVSP_HOST_MESSAGES and proceeds.

On the guest side NetVsc component, things are a little more subtle. The handling of incoming NVSP messages in NetVsc was scattered in several parts of the code. We refactored the code to isolate two points at which NetVsc handles two flavors of incoming NVSP messages: five data messages and seven completion messages. Each message is tagged with a MesssageType, as in NVSP_HOST_MESSAGE shown above.

One of the data messages encapsulates an RNDIS message. Of the remaining four, one of them is particularly interesting since its payload includes S_I_TAB, a variable-length structure depicted by the diagram below, where the MessageType field is held in the enclosing tagged union structure.

```
                            .-------------.
                            |             v
.-------------.-------.--------.---------.---------.
| MessageType | Count | Offset | padding | Table   |
.-------------.-------.- ------.---------.---------.
```

This is easily specified in 3D as follows, where the caller provides a MaxSize parameter to bound the entire size of the message, and an out parameter in which to receive a pointer to the beginning of the Table field. The predicate is_range_okay(size, offset, extent) is a function in 3D's library which checks that extent ≤ size && offset ≤ (size − extent), ensuring here that Offset points to the start of a Table field large enough to hold an array of Count-numbered UINT32 elements—it turns out that in NetVsc, Count is expected to be a constant. Additionally, Offset is expected to be at least MIN_OFFSET = 3 ∗ sizeof(UINT32), since it must point after the first three fields in the diagram shown above.

```
struct _S_I_TAB(UINT32 MaxSize, mutable PUINT8 *out) {
  UINT32 Count { Count == /* Some constant */ };
  UINT32 Offset {
     is_range_okay(MaxSize, Offset, sizeof(UINT32) * Count) &&
     Offset >= MIN_OFFSET };
  UINT8 padding[:byte-size Offset − MIN_OFFSET];
  UINT32 Table[:byte-size Count * sizeof(UINT32)]
              {:act *out = field_ptr }
}
```

4.2 RNDIS

The host vSwitch handles 9 types of RNDIS messages, while the guest handles 11, each with several sub-cases. RNDIS messages include both control-path and data-path messages, the handling of the latter being particularly performance sensitive. RNDIS packets on the data path may reside in memory buffers that are shared between the host and guest, to avoid the overhead of copying memory. However, validating and reading packets in shared memory is delicate, since, for example, an adversarial guest can change the contents of the packet while it is being validated at the host. To protect against this, an important discipline is to validate and read the packet in a single pass, never fetching any byte of the packet more than once. Adhering to this discipline ensures that the host observes a single logical snapshot of the packet even in the presence of concurrent mutations by the guest, inasmuch as what the host sees after a concurrent mutation, the untrusted guest could just as well have put in the packet to begin with. Relying on EverParse3D's double-fetch freedom, we write parsers with actions that validate and copy the input in a single pass into the local address space, and further processing of the packet can continue without needing to worry about concurrent mutations. We focus on the handling of data path messages received at the host—the specification on the guest side is analogous, though differs in many details.

The main part of our 3D specification that handles data-path packets is for a type called PPI_ARRAY, an array that holds several structures, each of which must be validated and copied into local memory owned by the host. The caller passes in the total length of the enclosing packet, the expected size of the array, a bound on that size (asserted by the where constraint, checked at runtime), and 12 out parameters into which various parts of each PPI struct in the payload array are to be copied.

```
struct PPI_ARRAY(UINT32 PacketLength,
                 UINT32 Expected, UINT32 Max,
                 mutable T1 *out1,..., mutable T12 *out12)
    where (Expected <= Max) {
  PPI(PacketLength, out1, ..., out12) payload[:byte-size Expected];
}
```

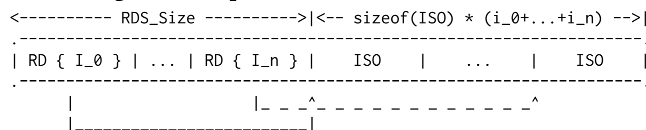
Each PPI is an encoding of a variable-length tagged union, where the Size field records the length of the PPI_UNION and the Type encodes its case. The format was designed originally to allow for some padding between the end of the PPIOffset field and the start of the PPI_UNION. However, since this is a performance-critical data packet, it was decided that padding was wasteful and should never be present, hence the PPIOffset should always be 12 = 3 ∗ sizeof(UINT32).

```
struct PPI(UINT32 PacketLength, mutable T1 *out1, ...) {
  UINT32 Size;
  UINT32 Type : 31;
  UINT32 IsTypeInternal : 1;
  UINT32 PPIOffset { Size >= PPIOffset && PPIOffset == 12 };
  PPI_UNION(Type, PacketLength, out1, ..., out12)
    payload [:byte-size-single-element-array Size − PPIOffset ]
}
```

Finally, PPI_UNION contains 12 cases, each populating one of the out-parameters using an action.

4.3 OIDs and NDIS Offloads

The payload of some RNDIS messages contains an OID (Operation Identifier) and potentially some operands. Our validation of OIDs in NetVsc handles 56 different cases. The operands for some of these cases are structures from the NDIS protocol. One of the NDIS messages is particularly interesting and is depicted below.



We have two adjacent arrays, one for RD structures and another for ISO structures. The size of the RD array is known by the context to be RDS_Size and the total size of the buffer is also known. However, each RD entry contains a field I which describes the number of ISO entries associated with it, so the number of ISO entries is the sum of all the I fields in the RD array. Further, each RD entry contains an Offset field that describes the offset into the ISO array at which its associated ISO entries begin. Of course, one could design a simpler layout of this structure by interleaving the RD and ISO arrays, but this layout is dictated by the NDIS standard.

Using 3D actions and their support for mutable state, we were able to write a specification to check the layout of this structure in a single pass. The type below takes two mutable parameters RDPrefix, for the size of all the RD entries parsed so far; and N_ISO, which holds the sum of their RD.I fields, both initialized to 0 using an action on the start field.

```
struct _RD_ISO_ARRAY(UINT32 RDS_Size, UINT32 TotalSize,
                     mutable UINT32* RDPrefix,
                     mutable UINT32* N_ISO)
where (RDS_Size <= TotalSize) {
    unit start {:act *RdPrefix = 0; *N_ISO=0; }
    RD(RD_Size, RDPrefix, N_ISO) rds[:byte-size RDS_Size];
    ISO(N_ISO) isos[:byte-size TotalSize − RDS_Size]
    unit finish {:check return (*N_ISO == 0) }}
```

When parsing each RD (using the specification below), we increment (guarded by overflow checks) the size of the prefix of RD entries parsed so far and sum the I field into the N_ISO accumulator. We also check that the Offset field points to the expected offset in the ISO array, skipping past n_iso entries—the :check action allows an action to return a boolean to signal whether or not parsing should continue.

```
struct RD (UINT32 RDS_Size, mutable UINT32* RDPrefix,
           mutable UINT32* N_ISO) {
  NDIS_OBJECT_HEADER ... Header; UINT32 I;
  UINT32 Offset {:check
   var prefix = *RDPrefix; var n_iso = *N_ISO;
   if ( /* overflow checks */ ) {
      *RDPrefix = prefix + sizeof(RD); *N_ISO = n_iso + I;
      return Offset == RDS_Size − prefix + n_iso * sizeof(ISO);
   } else { return false; } }}
```

Finally, when parsing the ISO array, we decrement the N_ISO accumulator after parsing each entry and when we reach the finish field of the _RD_ISO_ARRAY, we check that no ISO entries remain to be parsed.

```
struct ISO (UINT32* N_ISO) {
  NDIS_OBJECT_HEADER ... Header; ... /* other fields */
  UINT32 ISO_ID {:check
    var n = *N_ISO;
    if(n > 0) { *N_ISO = n − 1; return true; }
    else { return false; }}}
```

This was the one message in our entire Virtual Switch specification that relied on imperative code to validate invariants of the data format. In all other cases, actions were only used to build (partial) parsed structures of the input data. The ability to use actions to check intricate invariants such as the one here speaks to their expressiveness. However, such uses also make it harder to reason about the correctness of our specification. For the future, we are considering adding support for a small program logic to reason about the behavior of imperative actions—this would allow us to formally prove that our checks above correctly validate a more abstract format description.

5 Related Work & Conclusions

Formally verified parsers and parser generators have received a lot of attention in recent years. In summary, while there have been many tools and frameworks for verified parsers, EverParse3D appears to be unique in producing verified C code, while also handling a language of formats that includes arbitrary refinement constraints, dependent pairs, and untagged unions. The deployment of EverParse3D in widely used commercial software also appears to be a first for verified parser generators.

Blaudeau and Shankar (2020) build a verified packrat parser for parsing expression grammars (PEGs) (Ford 2002, 2004) in the PVS proof assistant (Shankar 1996). Lasser et al. (2019) build a verified implementation of an LL(1) parser generator and Lasser et al. (2021) verified an implementation of the ALL(*) parsing algorithm, both in the Coq proof assistant (The Coq development team 1989). These lines of work yield executable functional implementations for structured grammars similar in expressiveness to context-free grammars, and usable for, say, parsing programming language syntax. In supporting arbitrary forms of refinement constraints, including arithmetic, while lacking recursion, 3D's expressiveness is incomparable to context-free grammars. However, being based on monadic parser combinators, EverParse allows expressing context-sensitive grammars. Recursion would certainly be useful to support some kinds of grammars, e.g., in hierarchical document formats. However, the unbounded structures that we have seen so far in network packet formats (ranging from the examples shown in this paper to message formats in higher level protocols like TLS) have not required recursion; instead, arrays of various flavors have sufficed. That said, in specific networking scenarios like protocol encapsulation, including features like IP-in-IP, recursion would be useful, and we are considering ways to add it to EverParse. Some of the challenges include finding a good way to generically define inductively defined trees for the representation type for a specificational recursion parser combinator. Further, the use of recursion in the Network Virtualization stack is generally discouraged, particularly in parsing, since this can result in adversarially controllable stack depths, so ensuring that source grammars never result in non-tail recursion is likely to remain an important restriction.

More closely related is the Parsley project (Mundkur et al. 2020), which extends PEGs with constraints and actions, with verification-oriented tooling based on PVS, while also targeting binary formats used in network protocols and other systems applications. Enhancing EverParse3D to support more of what Parsley offers in terms of expressiveness of the grammar would be an interesting direction of future work. In principle, being based on LowParse's monadic parser combinators, EverParse3D should be amenable to such an extension, while offering high performance C code.

Also targeting protocol formats, Ye and Delaware (2019) use a Coq-based framework called Narcissus (Delaware et al. 2019) to develop purely functional parsers for Protocol Buffers, a popular data exchange format. As mentioned in the introduction, while the adoption of formats like Protocol Buffers is a welcome trend away from hand-rolled formats and parsers, it does not cater to the needs of high performance or legacy applications where the wire format is already fixed.

The PADS project (Fisher and Walker 2011) developed several tools to work with ad hoc data formats. Although, to our knowledge, a verified implementation of PADS was never built, its formalization has many similarities to our work here. In particular, the PADS authors developed a new dependently typed calculus DDC^{α} in which to formalize PADS, giving it three related denotational semantics in this calculus, including a type denotation, a parser denotation, and a metadata denotation. The type and parser denotations are similar in spirit to our denotational semantics, though we formalize 3D mechanically within an existing dependent type theory ($F^{\star}$). Unlike PADS, our main denotation, the validator denotation, provides a C-level semantics for our language. PADS' metadata denotation gives a semantics to parsing errors and statistics that the tool tracks. Although 3D has no direct analog, our validator denotation includes a semantics for parsing actions and error handlers, which can be used to build application-specific metadata generators. Further, our use of partial evaluation to extract a compiler from our interpreter-based semantics, seems to be a novel application of Futamura projections, at least in the context of parser generators.

Also related is Bangert and Zeldovich's (2015) Nail parser and formatter generator. Analogous to 3D's format and output type specifications, Nail supports defining both a protocol grammar and its internal object model. However, by excluding arbitrary semantic actions, Nail is able to support the generation of both parsers and formatters providing transformations in both directions from the binary format to the object model. The EverParse libraries underlying 3D also support formatting, with proofs that formatting and parsing are mutually inverse on valid data, however these formatters are not leveraged by 3D. We are keen to explore building on ideas from Nail to build formally proven parsers *and formatters* from a single source specification.

Conclusions. Given the scourge of software security bugs with root causes in parsing failures, tools that replace handwritten, buggy parsers with trustworthy counterparts can make a significant impact on software security and correctness. EverParse3D's push-button approach to generating verified C code from high-level format specifications is an appealing point in the design space. Its expressiveness coupled with explicit control offered by parsing actions, while yielding performant and double-fetch free C code, allowed the system to meet the functionality, security, and performance requirements of the Windows kernel. EverParse3D is open source and available on GitHub.

Acknowledgments

We thank Justin Campbell, Jeffrey Tippet, Praveen Balasubramaniam, Randy Miller, Saruhan Karademir, Maxime Villard, Lander Brandt, and many others for discussions and feedback that influenced the design, implementation, and adoption of our system. We are also grateful to Jonathan Protzenko for supporting the KaRaMeL tool, to Guido Martínez for work on compiler plugins, to all the members of Project Everest for many useful discussions, to the anonymous reviewers, and to Tej Chajed, the shepherd of this paper.

References

Julian Bangert and Nickolai Zeldovich. 2015. Nail: A Practical Tool for Parsing and Generating Data Formats. *login Usenix Mag.* 40, 1 (2015). https://www.usenix.org/publications/login/feb15/bangert

Clement Blaudeau and Natarajan Shankar. 2020. A Verified Packrat Parser Interpreter for Parsing Expression Grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 3–17. https://doi.org/10.1145/3372885.3373836

Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon Momot, Meredith L. Patterson, and Anna Shubina. 2017. Curing the Vulnerable Parser: Design Patterns for Secure Input Handling. *login Usenix Mag.* 42, 1 (2017). https://www.usenix.org/publications/login/spring2017/bratus

Adam Chlipala. 2021. Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 94 (Aug. 2021), 28 pages. https://doi.org/10.1145/3473599

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* 3, ICFP (2019), 82:1–82:29. https://doi.org/10.1145/3341686

Kathleen Fisher and David Walker. 2011. The PADS project: an overview. In *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, Tova Milo (Ed.). ACM, 11–17. https://doi.org/10.1145/1938551.1938556

Bryan Ford. 2002. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 36–47. https://doi.org/10.1145/581478.581483

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 111–122. https://doi.org/10.1145/964001.964011

Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 2, 5 (1971), 45–50. https://ci.nii.ac.jp/naid/10000032872/en/

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2019. A Verified LL(1) Parser Generator. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach

(Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:18. https://doi.org/10.4230/LIPIcs.ITP.2019.24

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: a verified ALL(*) parser. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 420–434. https://doi.org/10.1145/3453483.3454053

Linux. 2021a. tcp.h. https://elixir.bootlin.com/linux/v5.15.1/source/include/linux/tcp.h#L81

Linux. 2021b. tcp_input.c. https://elixir.bootlin.com/linux/v5.15.1/source/net/ipv4/tcp_input.c#L4001

Microsoft Corp. 2021. Quickstart: Create Intel SGX VM in the Azure portal. https://docs.microsoft.com/en-us/azure/confidential-computing/quick-create-portal

Mitre Corp. 2020. Common Weakness Enumeration. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. 2020. Research Report: The Parsley Data Format Definition Language. In *2020 IEEE Security and Privacy Workshops (SPW)*. 300–307. https://doi.org/10.1109/SPW50608.2020.00064

Jon Postel. 1981. Transmission Control Protocol. https://datatracker.ietf.org/doc/html/rfc793

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. https://doi.org/10.1145/3110261

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(USENIX Security 2019)*. USENIX Association, USA, 1465–1482.

E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446. https://tools.ietf.org/html/rfc8446

Mark Russinovich. 2021. Azure and AMD announce landmark in confidential computing evolution. https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-confidential-computing/

Natarajan Shankar. 1996. PVS: Combining Specification, Proof Checking, and Model Checking. In *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1166)*, Mandayam K. Srivas and Albert John Camilleri (Eds.). Springer, 257–264. https://doi.org/10.1007/BFb0031813

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

The Coq development team. 1989. *The Coq proof assistant.* http://coq.inria.fr

Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1–16. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei

Qianchuan Ye and Benjamin Delaware. 2019. A Verified Protocol Buffer Compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/3293880.3294105

Young-X. 2019. Git commit. https://github.com/torvalds/linux/commit/9609dad263f8bea347f41fddca29353dbf8a7d37