binpac: A yacc for Writing Application Protocol Parsers

Ruoming Pang*
Google, Inc.
New York, NY, USA
rpang@google.com

Vern Paxson
International Computer Science Institute and
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
vern@icir.org

Robin Sommer
International Computer Science Institute
Berkeley, CA, USA
robin@icir.org

Larry Peterson
Princeton University
Princeton, NJ, USA
llp@cs.princeton.edu

ABSTRACT

A key step in the semantic analysis of network traffic is to parse the traffic stream according to the high-level protocols it contains. This process transforms raw bytes into structured, typed, and semantically meaningful data fields that provide a high-level representation of the traffic. However, constructing protocol parsers by hand is a tedious and error-prone affair due to the complexity and sheer number of application protocols.

This paper presents `binpac`, a declarative language and compiler designed to simplify the task of constructing robust and efficient semantic analyzers for complex network protocols. We discuss the design of the `binpac` language and a range of issues in generating efficient parsers from high-level specifications. We have used `binpac` to build several protocol parsers for the "`Bro`" network intrusion detection system, replacing some of its existing analyzers (handcrafted in C++), and supplementing its operation with analyzers for new protocols. We can then use `Bro`'s powerful scripting language to express application-level analysis of network traffic in high-level terms that are both concise and expressive. `binpac` is now part of the open-source `Bro` distribution.

Categories and Subject Descriptors: C.2.2 [Network Protocols]: Applications

General Terms: Languages

Keywords: Parser Generator, Protocol

1. INTRODUCTION

Many network measurement studies involve analyzing network traffic in application-layer terms. For example, when studying Web traffic [2, 13] one often must parse HTTP headers to extract information about message length, content type, and caching behavior. Similarly, studies of Email traffic [21, 46], peer-to-peer applications [37], online gaming, and Internet attacks [29] require understanding application-level traffic semantics. However, it is tedious, error-prone, and sometimes prohibitively time-consuming to build application-level analysis tools from scratch, due to the complexity of dealing with low-level traffic data.

We can significantly simplify the process if we can leverage a common platform for various kinds of application-level traffic analysis. A key element of such a platform is *application-protocol parsers* that translate packet streams into high-level representations of the traffic, on top of which we can then use measurement scripts that manipulate semantically meaningful data elements such as HTTP content types or Email senders/recipients, instead of raw IP packets. Application-protocol parsers are also useful beyond network measurements—they form important components of network monitoring tools (e.g., tcpdump [19], Ethereal [12], NetDude [24, 23]), real-time network intrusion detection systems (e.g., Snort [35, 36] and `Bro` [31]), smart firewalls, and application layer proxies.

Building application-protocol parsers might appear straightforward at first glance, given a specification of the corresponding protocol. In practice, however, writing an efficient and robust parser is surprisingly difficult for a number of reasons. First, many of today's protocols are complex. For example, when analyzing the predominant HTTP protocol [14], one has to deal with pipelined requests, chunked data transfers, and MIME multipart bodies. The NetWare Core Protocol [33]—a common protocol used for remote file access—has about 400 individual request types, each with a distinct syntax. Second, even for simpler protocols, it is tedious and error-prone to manually write code to parse their structure: the code must handle thousands of connections in real-time to cope with the traffic in large networks, and protocol specifications are seldom comprehensive (i.e., they often ignore corner-cases, which a parser nevertheless must handle robustly as they *do* occur in real-world traffic). In potentially adversarial environments, an attacker may even deliberately craft ambiguous or non-conforming traffic [34, 17]. Furthermore, several severe vulnerabilities have been discovered in existing protocol parsers ([41, 42, 43, 44])—including one which enabled a worm to propagate through 12,000 deployments of a security product worldwide in tens of minutes [38, 25]—which demonstrates how difficult it is to comprehensively accommodate non-conforming input with hand-written code.

Given the care that writing a good protocol analyzer requires, it is unfortunate that existing analyzers are generally not reusable,

*The work was mostly done while the author was with Princeton University.

IMC'06, October 25–27, 2006, Rio de Janeiro, Brazil.

because their operation is usually tightly coupled with their specific application environments. For instance, the two major open-source network intrusion detection systems (NIDSs), Snort [36] and `Bro` [31], both provide their own HTTP analyzers, each exhibiting different features and shortcomings. Ethereal contains a huge collection of protocol parsers, but it is very difficult to reuse them for, e.g., `Bro` due to their quite different interfaces and data structures. Even inside a single software product, low-level code is generally inlined rather than factored into modules. For example, the Ethereal version 0.99 source code contains more than 8,000 instances of incrementing or decrementing by a hard-coded numeric constant, the vast majority of which are adjusting a pointer or a length while stepping through a buffer. Any instance of an incorrect constant can of course result in an incorrect parsing of a protocol, but is not detectable at compile-time since using the wrong numeric constant still type-checks.

We believe that the major reason for all of these difficulties is a significant lack of abstraction. In the programming language community, no one writes parsers manually. Instead, there are tools like `yacc` [20] and ANTLR [30] that support *declarative* programming: one expresses the syntax of the language of interest in a high-level meta-grammar, along with associated semantics. The parser generator then translates the specification into low-level code automatically. In this work, we propose to use similar abstractions for application-layer network protocols. By doing so, users building analyzers can concentrate on high-level protocol semantics, while at the same time achieving correctness, robustness, efficiency, and reusability.

However, we observe that existing parser-generation tools are not suitable for parsing network protocols. Common idioms of network protocols, such as data fields preceded by their actual length (sometimes not adjacent), cannot be easily expressed as a context-free grammar. Furthermore, when analyzing protocols, we often need to correlate across the two directions of a single connection; sometimes even syntax depends on the semantics of the byte stream in the other direction. Finally, parsers generated by these tools process input in a "pull" mode and thus cannot concurrently parse multiple, incomplete input streams.

To improve this situation we designed and implemented `binpac`—a declarative language and its compiler—to simplify the task of building protocol analyzers. Users specify parsers by defining message formats, dependencies between message fields, and additional computations to perform (e.g., printing ASCII records or triggering further analysis) upon parsing different message elements. The compiler translates the declarations into parsers in C++. `binpac` takes care of all the common and tedious (and thus error-prone) low-level tasks, such as byte-order handling, application-layer fragment reassembly, incremental input, boundary checking, and support for debugging. `binpac` also facilitates protocol parser *reuse* by supporting separation of different components of analyzers. One can readily plug in or remove one part of a protocol analyzer without modifying others. Such separation allows analysis-independent protocol specifications to be reused by different analysis tasks, and simplifies the task of protocol extension (for example, adding or removing NFS to the RPC parser).

Our goal is to ensure that the generated parsers are as efficient as carefully hand-written ones, so that they can handle large traffic volumes. Our main strategy is to leverage *data dependency analysis*—to tailor the generated parser to the analysis requirements at *compilation time*. For example, `binpac` identifies appropriate units for buffering of incomplete input based on the data layout specified by the user.

To demonstrate the power of our approach, we have used `binpac` to build several protocol parsers for the `Bro` NIDS, including HTTP, DNS, CIFS/SMB, DCE/RPC, NCP, and Sun RPC. (We emphasize that `binpac` is not however tied in any significant way to `Bro`.) Having written many protocol analyzers manually in the past, our experience is that `binpac` greatly eases the process. In future work we envision further using these `binpac` specifications to compile analyzers to alternative execution models: in particular, directly to custom hardware, without any intermediate C++ code, as sketched in [32].

The rest of this paper is organized as follows. We begin with related work in Section 2. In Section 3 we discuss specific characteristics of application protocols compared to languages targeted by traditional parser-generators. Section 4 describes the `binpac` language for specifying protocols and how the user associates semantic protocol analysis along with the description. Section 5 discusses the process of generating a parser from a `binpac` specification, including handling incremental input as well as performing robust error detection and recovery. Section 6 presents our experiences with using `binpac` to develop protocol parsers for the `Bro` NIDS, and we compare their performance with that of manually written ones. We summarize and present future directions in Section 7.

2. RELATED WORK

Considerable previous work has addressed facets of describing data and protocol layouts using declarative languages. First, there are various Interface Description Languages for describing the service interface for specific protocols. For instance, the External Data Representation Standard (XDR) [40] defines the way to describe procedure parameters and return values for the Remote Procedure Call (RPC) protocol [39]. The XDR compiler generates the underlying code to marshal/unmarshal data to/from raw bytes. Targeting a wider range of protocols, ASN.1 [3] is a language for describing the abstract syntax of communication protocols, including a set of encoding schemes. Unlike `binpac`, these languages dictate the underlying data representation or focus on a specific type of protocol, while `binpac` tries to describe the data layout of a wide range of existing (thus, already designed) protocols that span a variety of formats and styles.

Augmented BNF (ABNF) [9] is used in many protocol standards to specify the protocol syntax. However, the goal of ABNF is to provide a concise, yet incomplete, way to define a protocol, rather than for complete protocol specification from which one can generate a parser. In addition, ABNF targets ASCII protocols.

People have also designed languages for writing network protocol implementations, including both protocol parsing and processing logic. Abbott et al. [1] proposed a language for designing and implementing new network protocols. Prolac [22] is a language for writing modular implementations of networking protocols such as TCP. Biagioni et al. [5] experimented with implementing a TCP/IP network stack in ML. These efforts differ from `binpac` in that the goal is to build end-system implementations, instead of analyzers, of protocols. They also target protocols at the network and transport layers, rather than the wide range of application protocols.

There is also a rich body of work, e.g. [18, 6], in formal verification of design of protocols and, more generally, asynchronous process systems. These verification frameworks focus on abstract protocol behavior, instead of details of protocol syntax.

More related to `binpac`, there are efforts in the abstract description of *existing* protocol syntax. McCann and Chandra introduced PACKETTYPES [26], a language that helps programmers to handle binary data structures in network packets as if they were C types. Borisov et al. designed and implemented GAPA [7], a frame-

work for application protocol analyzers. Its protocol specification language, GAPAL, is based on (augmented) BNF and supports both ASCII and binary protocols. A protocol specification in GAPAL includes protocol syntax as well as analysis state and logic. While GAPA and `binpac` both target application-level traffic analysis in general, they are designed with different sets of goals and therefore take quite different approaches. GAPA targets traffic analysis at individual end hosts, and uses an interpreted, type-safe language. The `binpac` compiler, on the other hand, generates C++ parsers intended to process traffic of much higher volume at network gateways. Second, GAPA is a self-contained system that handles both protocol parsing and traffic analysis. `binpac`, on the other hand, focuses on parser generation and is designed as a building block for the construction of parsers that can be used by separate traffic analysis systems such as `Bro`.

Beyond network protocols, there are a number of languages for describing data formats in general. DATASCRIPT [4] is a scripting language with support for describing and parsing binary data. Developed more recently, PADS is a language for describing ad hoc data formats [15]. PADS's approach to data layout description is similar to that of `binpac` in a number of ways, such as the use of parameterized types. On the other hand, it is designed for a more general purpose than parsing network protocols, so it lacks abstractions and features particular to processing communication traffic, and the generated parsers cannot handle many input streams simultaneously. Related to PADS, Fisher et al. [16] described a calculus for reasoning about properties and features of data description languages in general. The calculus is used to discover subtle bugs, to prove the type correctness of PADS, and to guide the design of language features.

Hand-written application-layer protocol parsers are an important part of many network analysis tools. Packet monitors such as tcpdump [19], Ethereal [12], and DSniff [11] display protocol information. NetDude [24, 23] provides both visualization and editing of packet traces. NIDS such as Snort [36], `Bro` [31], and Network Flight Recorder [28] analyze protocol communications to detect malicious behavior. Protocol parsers are also components of smart firewalls and application-layer proxies.

3. CHARACTERISTICS OF APPLICATION PROTOCOLS

In this section we examine characteristics of network protocols which differ significantly from the sorts of languages targeted by traditional parser-generators. We discuss them in terms of syntax and grammar, input model, and robustness.

3.1 Syntax and Grammar Issues

In terms of syntax and grammar, application-layer protocols can be broadly categorized into two classes: *binary* protocols and human-readable *ASCII* protocols. The messages of binary protocols, like DNS and CIFS, consist of a (not necessarily fixed) number of data *fields*. These fields directly map to a set of basic data *types* such as integers and strings. Clear-text ASCII protocols, on the other hand, typically restrict their payload to a human-readable request/reply structure, using only printable ASCII-characters. Many of these protocols, such as HTTP and SMTP, are primarily line-based, i.e., requests/replies are separated by carriage-return/line-feed (CR/LF) tuples, and their syntax is usually specified with grammar production rules in protocol standards.

While these two types of protocols appear to exhibit quite distinct language characteristics, we in fact find enough underlying commonality between binary and ASCII protocols that we can treat both styles in a uniform fashion within declarative `binpac` specifications, as we will develop below. On the other hand, there are some critical differences between the grammars of network protocols (binary as well as ASCII) and those of programming languages:

Variable-length arrays. A common pattern in protocol syntax is to use one field to indicate the length of a later array. Such a length field often delimits the length of a subsequent (not necessarily contiguous) byte sequence, e.g., the HTTP "Content-Length" field, but can also indicate the number of complex elements, such as in the case of DNS [27] question and answer arrays. A conceptual variant of variable-length arrays is *padding*, i.e., filling a field with additional bytes to reach a specific length.

As long as the length-field has constant width, it is theoretically possible to describe arrays and padding with a context-free grammar. However, doing so is cumbersome and leads to complex grammars.

Selecting among grammar production rules. Both binary and ASCII protocols often use one or multiple data fields to select the interpretation of a subsequent element from a range of options. For example, DNS uses a type field to differentiate between various kinds of "resource records". HTTP uses multiple header fields to determine whether the message body is a consecutive byte sequence, a sequence of byte chunks, or multipart entities. Sometimes the selector even comes from the opposite flow of the connection, e.g., the syntax of a SUN/RPC reply depends on program and procedure fields in the corresponding RPC call [39]. In general such a selector can be easily expressed in a grammar by *parameterizing* non-terminal symbols—a very limited form of context-sensitive grammar which we describe later in Section 4.1.2, However it is very hard to specify a selector with a *context-free* grammar.

Encoding. In binary protocols, record fields directly correspond to values. Therefore it is crucial to consider the correct byte-encoding when parsing fields. For example, integers are often either encoded in big-endian or little-endian byte order. Similarly, string characters may be given in a (single-byte) ASCII encoding or in a (two-byte) Unicode representation. To complicate the problem, the byte order need not be fixed for a given protocol. For example, there is a field in DCE/RPC [10] header which explicitly indicates the byte order in which subsequent integers are encoded. In CIFS [8], a similar field gives the character encoding for strings (which in fact does not apply to all strings: certain ones are always in ASCII; similarly in CIFS, not all integers use the same byte-order). Handling data encoding is a tedious and error-prone task when writing a parser manually, and it is hardly expressible by means of an LALR grammar.

3.2 Concurrent Input

A fundamental difference between a protocol parser and a `yacc`-style parser is their input model. A protocol-parser has to parse many connections simultaneously and, within each connection, the two flows on opposite directions, *in parallel*. For example, in persistent HTTP connections each request needs to be associated with the correct reply. Similarly, the syntax of a SUN/RPC reply depends on program and procedure fields in the corresponding RPC call [39].

Parsers generated by `yacc/lex` process input in a "pull" fashion. That is, when input is incomplete, the parser blocks, waiting for further input. Thus, a thread can handle only one input stream at a time. To handle flows simultaneously without spawning a thread

```
class HTTP_Conn : public binpac::ConnectionAnalyzer
{
public:
  // Virtual functions defined in ConnectionAnalyzer.
  virtual void NewData(bool is_orig,
                       const u_char *begin,
                       const u_char *end);
  virtual void NewGap(bool is_orig, int gap_length);
  virtual void FlowEOF(bool is_orig);
};
```

Figure 1: The interface of a `binpac`-generated parser.

for each one, the parsers must instead process input *incrementally* as it comes in, partially scanning and parsing incomplete input and resuming where the analysis left off when next invoked.

3.3 Robustness

Parsing errors are inevitable when processing network traffic. Errors can be caused by irregularity in real-world traffic data (protocol deviations, corrupted contents) as well as by incomplete input due to packet drops when capturing network traffic, asymmetric routing (so that only one direction of a connection is captured), routing changes, or "cold start" (a connection was already underway when the monitor begins operation). Unlike compilers, protocol parsers cannot simply complain and stop processing, but must robustly detect and recover from errors. This is particularly important if we consider the presence of adversaries: an attacker might specially craft traffic to lead a protocol parser into an error condition.

4. THE LANGUAGE

In the previous section we examined the grammatical characteristics of network protocols. This section describes the design of the `binpac` language and its compiler, which are specifically tailored to address these properties.

Here we assume that a parser generated from the `binpac` language receives data from a lower-level protocol (such as TCP) analyzer, with the interface outlined in Figure 1. While one can also use the `binpac` language to build parsers for TCP/IP packets, a detailed description of a TCP analyzer—how to manage states of TCP connections and invoke corresponding application analyzers—is beyond scope of this paper. Instead we just assume existence of lower-level analyzers and focus on *application level* parsers in following discussion.

We begin with a description of `binpac`'s *data model* in Section 4.1, corresponding to production rules in BNF grammars. In Section 4.2 we discuss state-holding, in Section 4.3 how to add custom computation, and finally in Section 4.4 the "separation of concerns" to provide reusability.

Throughout the discussion we will refer to the examples in Figures 2 and 3, which show specifications of HTTP and DNS parsers in `binpac`, respectively. We use them to illustrate features of the `binpac` language. Note that the HTTP parser shown in Figure 2 is *complete* by itself (though simplified from the fully-featured one we built for `Bro`, and evaluate below), except for the MIME-decoding of HTTP bodies and escape sequences for URIs.[1] The former takes significant additional work to add; the latter can be incorporated easily by processing the raw, extracted URI with an additional function call. Due to space limitations, we only show an excerpt of the DNS parser, though this includes the technically most difficult element of parsing the protocol, namely compression-by-indirection of domain names.

[1]The string comparisons and has_prefix() in Figure 2 are in fact case-insensitive, but here simplified for presentation.

In the language, text between %.*{ and %} (for example, %header{ and %}) embeds C/C++ code. `binpac` keywords reflecting optional attributes start with "&" (e.g., `&oneline`). Keywords starting with "$", such as `$context` and `$element`, are macros instantiated during parsing. In the examples, we highlight `binpac` keywords (except for elementary types, introduced below) using bold slant fonts. Table 4 summarizes the `binpac` language constructs.

4.1 Data Model

`binpac`'s data model provides integral and composite types which allow us to describe basic patterns in protocol data layout, parameterized types to pass information between grammar elements, and derivative data fields to store intermediate computation results. We discuss these in turn.

4.1.1 Integral and Composite Types

A `binpac` *type* describes both the data layout of a consecutive segment of bytes and the resulting data structure after parsing. Type `empty` represents zero-length input. Elementary types `int8`, `int16`, `int32` represent 8-, 16-, and 32-bit integers, respectively, and so do their unsigned counterparts, `uint{8,16,32}`. As the specification of `HTTP_ReplyLine` shows (Figure 2), a string type can be represented with a constant string (line 80), a regular expression (line 81), or a generic `bytestring` either of a specific length (with `&length`, line 101) or running till the end of data (with `&restofdata`, line 92).

Elementary integer and string types map naturally to their counterparts in C++ (in the case of string, we define a simple C++ class to denote the begin and end of the string). This is how the results are stored and accessed, with one exception. We allow a string to be "*chunked*" to handle potentially very long byte sequences, such as HTTP bodies, with a `&chunked` attribute (Figure 2, line 100). A chunked string is not buffered. Rather, with the `&processchunk` attribute one may define computation on each chunk to process the byte sequence in a streaming fashion. For instance, to compute a MD5 checksum for every HTTP body we may add a `&processchunk` as follows (assuming `compute_md5` maintains intermediate results across chunks):

```
http_body: bytestring &chunked,
      &length = $context.flow.content_length(),
      &processchunk($context.flow.compute_md5($chunk));
```

External C++ types, including `bool`, `int`, and user-defined ones (declared with `extern type`), can be used in computation, e.g., as types of parameters, but cannot appear as types of data fields in protocol messages.

Users can define composite types: (1) `record`, a sequential collection of fields of different types; (2) `case`, a union (in the C-language sense) of different types; and (3) `array`, a sequence of single-type elements. `binpac` generates a C++ class for each user-defined type, with data fields mapped as class members, and a parse function to process a segment of bytes to extract various data fields according to the layout specification.

As we compare `record` and `case` types with context-free grammar production rules, we can see a clear correspondence between them: a concatenation of symbols maps to a `record` type and multiple production rules of a symbol map to a `case` type. But there is a difference in the latter mapping. The `case` type corresponds to a set of production rules with *zero look-ahead*. Instead, a production rule is selected based on an explicit indexing expression computed from other data fields or type parameters (Figure 2, line

Language Construct	Brief Explanation	Section	Example
`%header{ ... %}`	Copy the C++ code to the generated header file		Fig. 2, #15
`%code{ ... %}`	Copy C++ code to the generated source file		
`%member{ ... %}`	C++ declarations of private class members of connection or flow	§4.2	Figure 5
`analyzer ... withcontext`	Declare the beginning of a parser module and the members of `$context`	§4.2.2	Fig. 2, #1
`connection`	Define a connection object	§4.2.1	Fig. 2, #37
`upflow/downflow`	Declare flow names for two flows of the connection	§4.2.1	Fig. 2, #38
`flow`	Define a flow object	§4.2.1	Fig. 2, #40
`datagram = ... withcontext`	Declare the datagram flow unit type	§4.2.1	Fig. 3, #64
`flowunit = ... withcontext`	Declare the byte-stream flow unit type	§4.2.1	Fig. 2, #41
`enum`	Define a "enum" type		Fig. 2, #5
`type ... =`	Define a `binpac` type	§4.1.1	Fig. 2, #11
`record`	Record type	§4.1.1	Fig. 2, #49
`case ... of`	Case type—representing an alternation among case field types	§4.1.1	Fig. 2, #45
`default`	The default case	§4.1.1	Fig. 2, #103
⟨type⟩`[]`	Array type	§4.1.1	Fig. 2, #87
`RE/.../`	A string matching the given regular expression	§4.1.1	Fig. 2, #11
`bytestring`	An arbitrary-content byte string	§4.1.1	Fig. 2, #73
`extern type`	Declare an external type	§4.1.1	Fig. 2, #13
`function`	Define a function	§4.2	Fig. 3, #67
`refine typeattr`	Add a type attribute to the `binpac` type	§4.4	Fig. 6
⟨type⟩ `withinput` ⟨input⟩	Parse ⟨type⟩ on the given ⟨input⟩ instead of the default input	§4.1.4	Fig. 3, #59
`&byteorder`	Define the byte order of the type and all enclosed types (unless otherwise specified)	§4.1.3	Fig. 3, #7
`&check`	Check a predicate condition and raise an exception if the condition evaluates to false	§5.2.1	Fig. 3, #34
`&chunked`	Do not buffer contents of the bytestring, instead, deliver each chunk as `$chunk` to `&processchunk` (if any is specified)	§4.1.1	Fig. 2, #100
`&exportsourcedata`	Makes the source data for the type visible through a member variable `sourcedata`	§4.1.4	Fig. 3, #7
`&if`	Evaluate a field only if the condition is true		Fig. 3, #16
`&length = ...`	Length of source data should be ...	§4.1.1	Fig. 2, #101
`&let`	Define derivative types	§4.1.4	Fig. 2, #63
`&oneline`	Length of source data is one line	§5.1	Fig. 2, #63
`&processchunk`	Computation for each `$chunk` of bytestring defined with `&chunked`	§4.1.1	
`&requires`	Introduce artificial data dependency		
`&restofdata`	Length of source data is till the end of input	§4.1.1	Fig. 2, #73
`&transient`	Do not create a copy of the bytestring	§6	
`&until`	End of an array if condition (on `$element` or `$input`) is satisfied	§4.1.1	Fig. 2, #87

Table 1: Summary of `binpac` language constructs.

99). This allows production rule selection to be based on external information, and is in spirit similar to "predicated parsing" introduced in ANTLR [30]. On the other hand, the zero-look-ahead restriction simplifies parser construction, but at the same time poses little limitation on the range of protocols that can be specified in `binpac`. We believe that it is by design of protocols (rather than by coincidence) that there are few syntax patterns that require look-ahead. Since protocol data is generated and processed by programs, it is usually organized in a way that simplifies the (traditionally hand-written) implementation.

Although an array can be defined with recursive production rules, we find it a common enough idiom in protocol syntax that it justifies a separate abstraction. In `binpac`, the length of an array can be specified with an expression containing references to other data fields, as in the definition of `DNS_message` (Figure 3, lines 3-6). An array can also be defined without a length, but with some "terminate condition" that indicates the end of array. Such a condition is specified through the `&until` attribute with a conditional expression. The expression can be computed from the input data to each element (`$input`), as in `HTTP_Headers` (Figure 2, line 87), or from a parsed element (`$element`), as in `HTTP_Chunks` (line 106).

4.1.2 Type Parameters

As the examples of HTTP and DNS parsers show, *type parameters* (e.g., in type `HTTP_Body`, Figure 2, line 98) allow one to pass information between types without resorting to keeping external state. This is a powerful feature that can significantly simplify syntax specification.

```
type NDR_Format = record {
    # Note, field names taken from DCE/RPC spec.
    intchar   : uint8;
    floatspec : uint8;
    reserved  : padding[2];
} &let {
    ndr_byteorder = (intchar & 0xf0) ?
              littleendian : bigendian;
};

type DCE_RPC_Message = record {
    # Raise an exception if RPC version != 5
    rpc_vers       : uint8 &check(rpc_vers == 5);
    rpc_vers_minor : uint8;
    PTYPE          : uint8;
    pfc_flags      : uint8;
    # 'drep'--data representation
    packed_drep    : NDR_Format;
    ...
} &byteorder = packed_drep.ndr_byteorder;
```

Figure 4: Specifying dynamic byte order with `&byteorder`.

4.1.3 Byte Orders

For use with binary protocols, `binpac` allows the user to specify the byte order using a `&byteorder` attribute. Figure 4 shows the specification of dynamic byte-order in DCE/RPC, where at the bottom the user specifies that the byte-order is taken from the `ndr_byteorder` field that is defined earlier.[2]

In most cases we also want to propagate the byte-order specification along the type hierarchy to the other types. Conceptually we can pass byte order between types as a parameter (see Section 4.1.2), but in practice the byte order parameter is required

[2]We discuss the definition of "derivative fields" such as `ndr_byteorder` in Section 4.1.4.

```
1  analyzer HTTP withcontext {  # members of $context
2      connection: HTTP_Conn;
3      flow:       HTTP_Flow;
4  };
5  enum DeliveryMode {
6      UNKNOWN_DELIVERY_MODE,
7      CONTENT_LENGTH,
8      CHUNKED,
9  };
10 # Regular expression patterns
11 type HTTP_TOKEN = RE/[^()<>@,;:\\"\/\[\]?={} \t]+/;
12 type HTTP_WS    = RE/[ \t]*/;
13 extern type BroConn;
14 extern type HTTP_HeaderInfo;
15 %header{
16     // Between %.*{ and %} is embedded C++ header/code
17     class HTTP_HeaderInfo {
18     public:
19         HTTP_HeaderInfo(HTTP_Headers *headers) {
20           delivery_mode = UNKNOWN_DELIVERY_MODE;
21           for ( int i = 0; i < headers->length(); ++i ) {
22             HTTP_Header *h = (*headers)[i];
23             if ( h->name() == "CONTENT-LENGTH" ) {
24               delivery_mode = CONTENT_LENGTH;
25               content_length = to_int(h->value());
26             } else if ( h->name() == "TRANSFER-ENCODING"
27                    && has_prefix(h->value(), "CHUNKED") ) {
28               delivery_mode = CHUNKED;
29             }
30           }
31         }
32         DeliveryMode delivery_mode;
33         int content_length;
34     };
35 %}
36 # Connection and flow
37 connection HTTP_Conn(bro_conn: BroConn) {
38     upflow = HTTP_Flow(true);  downflow = HTTP_Flow(false);
39 };
40 flow HTTP_Flow(is_orig: bool) {
41     flowunit = HTTP_PDU(is_orig)
42                    withcontext(connection, this);
43 };
44 # Types
45 type HTTP_PDU(is_orig: bool) = case is_orig of {
46     true  -> request: HTTP_Request;
47     false -> reply:   HTTP_Reply;
48 };
49 type HTTP_Request = record {
50     request:    HTTP_RequestLine;
51     msg:        HTTP_Message;
52 };
53 type HTTP_Reply = record {
54     reply:      HTTP_ReplyLine;
55     msg:        HTTP_Message;
56 };
```

```
57 type HTTP_RequestLine = record {
58     method:     HTTP_TOKEN;
59     :           HTTP_WS;    # an anonymous field has no name
60     uri:        RE/[[:alnum:][:punct:]]+/;
61     :           HTTP_WS;
62     version:    HTTP_Version;
63 } &oneline, &let {
64     bro_gen_req: bool = bro_event_http_request(
65         $context.connection.bro_conn,
66         method, uri, version.vers_str);
67 };
68 type HTTP_ReplyLine = record {
69     version:    HTTP_Version;
70     :           HTTP_WS;
71     status:     RE/[0-9]\{3\}/;
72     :           HTTP_WS;
73     reason:     bytestring &restofdata;
74 } &oneline, &let {
75     bro_gen_resp: bool = bro_event_http_reply(
76         $context.connection.bro_conn,
77         version.vers_str, to_int(status), reason);
78 };
79 type HTTP_Version = record {
80     :           "HTTP/";
81     vers_str:   RE/[0-9]+\.[0-9]+/;
82 };
83 type HTTP_Message = record {
84     headers:    HTTP_Headers;
85     body:       HTTP_Body(HTTP_HeaderInfo(headers));
86 };
87 type HTTP_Headers = HTTP_Header[] &until($input.length() == 0);
88 type HTTP_Header = record {
89     name:       HTTP_TOKEN;
90     :           ":";
91     :           HTTP_WS;
92     value:      bytestring &restofdata;
93 } &oneline, &let {
94     bro_gen_hdr: bool = bro_event_http_header(
95         $context.connection.bro_conn,
96         $context.flow.is_orig, name, value);
97 };
98 type HTTP_Body(hdrinfo: HTTP_HeaderInfo) =
99           case hdrinfo.delivery_mode of {
100    CONTENT_LENGTH -> body: bytestring &chunked,
101                               &length = hdrinfo.content_length;
102    CHUNKED        -> chunks: HTTP_Chunks;
103    default        -> other: HTTP_UnknownBody;
104 };
105 type HTTP_Chunks = record {
106     chunks:     HTTP_Chunk[] &until($element.chunk_length == 0);
107     headers:    HTTP_Headers;
108 };
109 type HTTP_Chunk = record {
110     len_line:   bytestring &oneline;
111     data:       bytestring &chunked, &length = chunk_length;
112     opt_crlf:   case chunk_length of {
113         0       -> none: empty;
114         default -> crlf: bytestring &oneline;
115     };
116 } &let {
117     chunk_length: int = to_int(len_line, 16);  # in hexadecimal
118 };
```

Figure 2: A HTTP parser in `binpac` with Bro event generation, complete except for MIME and escape-sequence processing.

universally for binary protocols. Adding a parameter to each type would be tedious and clutter the specification. To solve this problem, we designate "byteorder" as an *implicit type parameter* that is always passed to referenced types unless it is redefined at the referenced type. The `binpac` compiler traverses the type reference graph to find out which types require byte-order specification and adds byte order parameters to their parse functions.

We have not yet added support for ASCII vs. Unicode to `binpac`, though conceptually it will be similar to the support for byte-order.

4.1.4 Derivative Fields

Sometimes it is useful to add user-defined *derivative fields* to a type definition to keep intermediate computation results (see the definition of `HTTP_Chunk.chunk_length` in Figure 2, line 117), or to further process parsing results (`DNS_label` in Figure 3, lines 58-60). Derivative fields are specified within `&let {...}` attributes.

A derivative field may take one of two forms. First, a derivative field can be defined with an expression, in the form of "`<id> = <expression>`", as in the HTTP example.

Second, it can be evaluated by mapping a type onto a piece of computed input, in the form of "`<id>: <type> withinput <input expression>`". Here `<input expression>` evaluates to a sequence of bytes, which are passed to the parse function of `<type>` as input data. Such `withinput` fields allow us to extend parsing beyond consecutive and non-overlapping pieces of original input data. For instance, the computed input data might be (1) a reassembly of fragments (e.g. a fragmented DCE/RPC message body), (2) a Base64-decoded Email body, or (3) a DNS name pointer (as defined in Section 4.1.4 in [27]), as shown in Figure 3, lines 55-60. In the DNS example, a DNS label can be a sequence of bytes or a "name pointer" pointing to a DNS name at some specific offset of the message's source data. In the latter case, we define a `withinput` field to redirect the input to the pointed location when parsing the DNS name (and add an attribute

```
1  type DNS_message = record {
2     header:     DNS_header;
3     question:   DNS_question(this)[header.qdcount];
4     answer:     DNS_rr(this)[header.ancount];
5     authority:  DNS_rr(this)[header.nscount];
6     additional: DNS_rr(this)[header.arcount];
7  } &byteorder = bigendian, &exportsourcedata;
8  type DNS_header = record { ... };
9  type DNS_question(msg: DNS_message) = record {
10    qname: DNS_name(msg);  qtype: uint16;  qclass: uint16;
11 } &let {
12    # Generate Bro event dns_request if a query
13    bro_gen_request: bool = bro_event_dns_request(
14            $context.connection.bro_conn,
15            msg.header, qname, qtype, qclass)
16        &if (msg.header.qr == 0); # if a request
17 };
18 type DNS_rr(msg: DNS_message) = record {
19    rr_name:    DNS_name(msg);
20    rr_type:    uint16;   rr_class:   uint16;
21    rr_ttl:     uint32;   rr_rdlen:   uint16;
22    rr_rdata:   DNS_rdata(msg, rr_type, rr_class)
23                &length = rr_rdlen;
24 } &let {
25    bro_gen_A_reply: bool  = bro_event_dns_A_reply(
26            $context.connection.bro_conn,
27            msg.header, this, rr_rdata.type_a)
28        &if (rr_type == 1);
29    bro_gen_NS_reply: bool = bro_event_dns_NS_reply(...);
30        &if (rr_type == 2);
31 };
32 type DNS_rdata(msg: DNS_message, rr_type: uint16,
33        rr_class: uint16) = case rr_type of {
34    1   -> type_a:   uint32 &check(rr_class == CLASS_IN);
35    2   -> type_ns:  DNS_name(msg);
36    # Omitted: TYPE_PTR, TYPE_MX, ...
37    default -> unknown:  bytestring &restofdata;
38 };
```

```
39 # A DNS name is a sequence of DNS labels
40 type DNS_name(msg: DNS_message) = record {
41    labels:     DNS_label(msg)[] &until($element.last);
42 };
43
44 # A label contains a byte string or a name pointer
45 type DNS_label(msg: DNS_message) = record {
46    length:     uint8;
47    data:       case label_type of {
48        0 ->    label:  bytestring &length = length;
49        3 ->    ptr_lo: uint8;   # the lower 8-bit of offset
50    };
51 } &let {
52    label_type: uint8   = length >> 6;
53    last: bool          = (length == 0) || (label_type == 3);
54
55    # If the label is a pointer ...
56    ptr_offset: uint16 = (length & 0x3f) << 8 + ptr_lo
57        &if(label_type == 3);
58    ptr: DNS_name(msg)
59        withinput msgdata(msg.sourcedata, ptr_offset)
60        &if(label_type == 3);
61 };
62
63 flow DNS_Flow {
64   datagram = DNS_message withcontext (connection, this);
65
66   # Returns the byte segment starting at <offset> of <msgdata>
67   function msgdata(msgdata: const_bytestring,
68                    offset: int): const_bytestring
69      %{
70      // Omitted: DNS pointer loop detection
71      if ( offset < 0 || offset >= msgdata.length() )
72        return const_bytestring(0, 0);
73      return const_bytestring(msgdata.begin() + offset,
74                              msgdata.end());
75      %}
76 };
```

Figure 3: An (abridged) DNS parser in `binpac`.

`&exportsourcedata` to the `DNS_message` to make the input visible as variable `sourcedata`).

The derivative members are evaluated once during parsing and can be accessed in the same way as record or case fields in the generated C++ class. The order that derivative fields, along with non-derivative ones, are evaluated depends on only the data dependency among fields; the order is undefined for fields that do not depend on each other. (Note, this lack of ordering is deliberate, as it keeps the door open for future parallelization.) On the other hand, `binpac` provides attributes for users to introduce artificial dependency edges between fields, in case the user wants to ensure a certain ordering among evaluation of fields.

Derivative fields are also used to insert custom computation (such as event generation for the `Bro` NIDS) into the parsing process, as discussed in Section 4.3.

4.2 State Management

Up to this point we have explored various issues in describing the syntax of a byte segment. To model the state of a continuous communication, `binpac` introduces notions of *flow* and *connection*. A *flow* represents a sequence of *messages* and state to maintain between messages. A *connection* represents a pair of *flows* and state between flows. Note that here *connections* are not only TCP or UDP connections, but any two-way communication sessions. For example, a DCE/RPC *connection* may correspond to a TCP connection on port 135, a UDP session to the Windows messenger port, or a CIFS "named pipe" (a DCE/RPC tunnel through the CIFS protocol).

As shown in the HTTP example (line 38), the declaration of a connection consists of definitions of flow types for each flow. The "upflow" refers to the flow from the connection originator to the responder, and the "downflow" refers to the flow in the opposite direction. Like types, connections and flows can be parameterized, too.

Without loss of generality, we assume a flow consists of a sequence of messages of the same `binpac` type. (If a flow consists of messages of different types, we can encapsulate the types with a case type.) Thus one message type is specified for each flow, which we term *flow unit type*.

When specifying the flow unit type, we also specify how input data arrive in a flow: it may arrive as *datagrams*, each containing exactly one message, or in a *byte stream*, where the boundary of data delivery does not necessarily align with message boundaries, though the bytes are guaranteed to arrive in order.[3] The two input delivery modes are specified with keywords `datagram` and `flowunit`, respectively, as we see in the examples of DNS and HTTP parsers (lines 64 and 41 respectively).

4.2.1 Per-Connection/Flow State

While type parameterization allows types to share information *within* a message, in some scenarios we have to keep state at per-connection or per-flow level. For instance, a DCE/RPC parser needs to remember onto which interface a connection is bound, so that requests and replies can be parsed accordingly. As Figure 5 shows, a SUN/RPC parser keeps a per-connection table that maps session ID's to call parameters, and when a reply arrives, the parser can find the corresponding call parameters by looking up the reply message's session ID in the table. Connection/flow state is specified with embedded C++ code and corresponding access functions defined in `binpac` types.

Further abstraction of state is an important aspect of future work, as the abstraction can then expose data dependencies in the protocol

[3]Because the flows represent abstract flows, the delivery mode of a flow does not always indicate whether the underlying transport protocol is TCP or UDP. For example, while the DNS abstract flow takes input as datagrams, it is used for both TCP and UDP, whereas in the case of TCP, an additional thin layer between the DNS and TCP protocol delimits one DNS message from another in the TCP byte stream.

```
connection RPC_Conn(bro_conn: BroConn) {
   %member{
       typedef std::map<uint32, RPC_Call *> RPC_CallTable;
       RPC_CallTable call_table;
   %}
   # Returns the call corresponding to the xid. Returns
   # NULL if not found.
   function FindCall(xid: uint32): RPC_Call
       %{
       RPC_CallTable::const_iterator it = call_table.find(xid);
       if ( it == call_table.end() )
           return 0;
       return it->second;
       %}
   function NewCall(xid: uint32, call: RPC_Call): void
       %{
       if ( call_table.find(xid) == call_table.end() )
           call_table[xid] = call;
       %}
    #...
};

type RPC_Call(msg: RPC_Message) = record {
   # ...
} &let {
   # Register the RPC call by the xid
   newcall: void = $context.connection.NewCall(msg.xid, this);
};

type RPC_Reply(msg: RPC_Message) = record {
   # ...
} &let {
   # Find the corresponding RPC call.
   call: RPC_Call = $context.connection.FindCall(msg.xid);
};
```

Figure 5: SUN/RPC per-connection state.

analysis and enable better parallelization or hardware realization. The main challenge in abstracting state lies in understanding which data structures, such as hash tables, FIFO queues, and stacks, are commonly used in protocol parsers and providing ways to abstract them.

4.2.2 The `$context` *Parameter*

For types to access per-connection/flow state, the references to the corresponding connection and flow have to be given to the type parse functions through function parameters. As the connection and flow might be accessed by multiple types, we can propagate them as implicit parameters to relevant types, just as the byte order flag does. More generally, state can also be maintained at granularity other than connection or flow, e.g., at a multi-connection "session" level. We aggregate all such parameters as members of an implicit *context* parameter. The members of the context parameter are declared with `analyzer <name> withcontext` at the beginning of a `binpac` specification (Figure 2, line 1). The member values are instantiated in the `withcontext` clause in the flow unit definition (Figure 2, line 42).

4.3 Integrating Custom Computation

In a `yacc` grammar one can embed user-defined computation, such as syntax tree generation, in the form of C/C++ code segments, which the parser executes when reducing rules. `binpac` takes a slightly different approach in integrating custom computation with parsing. The computation (e.g., generating an event in the `Bro` NIDS) is embedded through adding *derivative fields* (discussed in Section 4.1.4). As the definition of type `HTTP_Header` in Figure 2 shows (lines 94-96), a `Bro` event for a HTTP header is generated by calling an external function `bro_event_http_header` in the definition of derivative field `bro_gen_hdr`. The function is invoked after parsing the data fields it depends on, `name` and `value` of the header. Note that these sorts of links are the only tie between the `binpac` specification for HTTP and the `Bro` system.

```
refine typeattr HTTP_Header += &let
  process_header: bool =
      $context.flow.bro_event_http_header(name, value);
;
```

Figure 6: Separating `Bro` event generation from protocol syntax specification with `refine`.

4.4 Separation of Concerns

"Separation of concerns" is a term in software engineering that describes "the process of breaking a program into distinct features that overlap in functionality as little as possible." [45] In the case of `binpac`, one would want to separate the definition of a protocol's syntax from specifications of additional computation (such as `Bro` event generation) on parsing results, because such separation allows us to reuse the protocol definitions for multiple purposes and across different systems. For the same reason, one may also want to separate specification of sub-protocols (e.g. RPC Portmapper and NFS) from the underlying protocol (e.g., RPC) and from each other.

`binpac` supports a simple but powerful syntactic primitive to allow separate expression of different concerns—parsing vs. analysis, a lower-level protocol vs. higher-level ones—and yet make the separated descriptions semantically equivalent to a unified one. The language includes a "refine typeattr" primitive for appending new type attributes, usually additional derivative fields, to existing types. For example, the generation of `http_header` event in the HTTP example (lines 94-96) can be separated from the protocol syntax specification, as Figure 6 shows.

Such separation allows us to place related-but-distinct definitions in different `binpac` source files. A similar `refine casetype` primitive allows insertion of new case fields to a case type definition (e.g., NFS_Params as a new case for RPC_Params), facilitating syntactical separation between closely related protocols.

Note that the support for separation of concerns in `binpac` is not complete in two ways. First, one cannot easily change the set of parameters of a type (or function), which can limit extension of protocol analyzers in some cases, an area for future exploration. Second, `binpac` does not *enforce* separation of concerns, or make it easier to describe things separately than describing them together. Thus, we rely on `binpac` users practicing a discipline of separating their concerns for better code maintenance and reuse.

5. PARSER GENERATION

Two main considerations in parser generation are (1) handling incremental input on many flows at the same and (2) detecting and recovering from errors. Below we examine them in turn.

5.1 Incremental Input

One approach to handle incremental input is to make the parsing process itself fully incremental, i.e., to make the parse function ready to stop anywhere, buffer unprocessed bytes at elementary type level, return, and resume on next invocation. The parsing state of a composite type, such as a `record`, can be kept by an indexing variable pointing to the member to be parsed next and a buffer storing unprocessed raw data.

However, incremental parsing at elementary type granularity is expensive, because boundary checks of adjacent fields can no longer be combined. It is also unnecessary for all the protocols we have encountered. As protocols are designed for easy processing, they often have a natural unit for buffering. Binary protocols (such as DCE/RPC) often have a "length" header field that denotes the total message length. ASCII protocols are usually either line-based

```
type DCE_RPC_Header = record
  ...
  frag_length: uint16;     # length of the PDU
  ...
;

type DCE_RPC_PDU = record
  header: DCE_RPC_Header;  # A 16-byte-long header
  ...
 &length = header.frag_length;
```

Figure 7: Specifying buffering length of a type.

(such as SMTP) or alternate between length-denoted and line-based units (such as HTTP). Given such parsing boundaries, we still require support for incremental parsing, but can carry it out at larger granularity and with reduced overhead.

Thus, `binpac` provides the attributes `&length` and `&oneline` to specify buffering units.[4] `&oneline` triggers line-based buffering while `&length` gives a message's length in bytes. `&length` usually points to a corresponding length field in the header (Figure 7) but can generally take any expression to compute the length. The `binpac` compiler performs data dependency analysis to find out the initial number of bytes to buffer before the length expression can be computed (in the case of a DCE/RPC message, the first 16 bytes). The generated code will buffer the message in two steps, first the initial bytes for computing the message length, then buffer up to the full length before parsing the remaining fields.

5.2 Error Detection and Recovery

Protocol parsers have to robustly detect and recover from various kinds of errors. Errors can be caused by irregularity in real-world traffic data, including small syntax deviations from the standard, incorrect length fields, corrupted contents, and even payloads of a completely different protocol running on the standard port of the parsed protocol. Errors can also result from incomplete input, such as due to packet drops when capturing network traffic. In these cases, the parser might not know in the specific state of the dialog, e.g., whether what it now sees on HTTP flow is inside a data transfer or not. Errors may also arise through incorrect `binpac` specifications, e.g., through missing cases or trying to access an unparsed case field, or due to adversarial manipulation, as discussed earlier.

Parsers generated by the `binpac` compiler detect errors of various aspects, as we discuss below. When an error is detected, the code throws a C++ run-time exception, which can then be caught for recovery.

5.2.1 Error Detection

Efficient Boundary Checking. Conceptually, boundary checking (whether scanning stays within the input buffer) only need take place before evaluating every elementary integer or character type field, because all other types are composed of elementary types. While it would be easy to generate the boundary checking code this way, the generated code would be quite inefficient, too. Instead, the `binpac` compiler tries to minimize the number of boundary checks. The basic idea is: before generating boundary checking code for a record field, check recursively whether we can generate the checking for the next field. If so, we can combine them into one check. In this way, the compiler can determine the furthest field for which the boundary checking can be performed at a given point of parsing.

[4] `binpac`'s incremental analysis depends on the existence of these attributes. Viewing the record definitions as a tree of types, each path from the root type to a leaf must contain one of them at a non-leaf node.

Handling dropped packets. When capturing network traffic, packet drops cannot always be avoided. They can be caused by a high traffic volume, kernel scheduling issues, or artifacts of the monitoring environment. Such drops lead to *content gaps* in application-level data processed by protocol parsers. Facing content gaps, parsers not only are unable to extract data for the current message, but also may not even know where the next message starts.

A particular, very common case of a content gap is one located inside a byte sequence of known length. For example, within an HTTP entity body, a content gap can be handled without creating uncertainty for subsequent protocol elements. If a byte sequence is defined as `&chunked` in a `binpac` specification—and thus only passed to a potential `&processchunk` function, but not further referenced by other expressions—then the generated parser can simply skip over such a gap. (If `&processchunk` is defined for the sequence, the function is called with a specially marked "gap chunk" so it can take note of the fact.) This mechanism allows us to handle most content gaps for protocols in which the majority of data is contained in long byte sequences. Hand-written protocol parsers in `Bro` handle content gaps in a similar way, but on an individual basis; the chunked byte string abstraction in `binpac` allows them to be handled universally for all protocols.

In general, it is trickier to handle content gaps which do not fully fall into a byte sequence of known size. We discuss these below in Section 5.2.2.

Run-time type safety. The only access to parsing results provided to `binpac` parsers is via typed interfaces. These leaves two aspects of type safety to enforce at run-time: (1) among multiple case fields in a case type, the generated code ensures that only the case that is selected during parsing can be accessed, otherwise it throws a run-time exception; (2) access to array elements is always boundary-checked. On the other hand, note that `binpac` cannot guarantee complete safety, as it allows arbitrary embedded C++ code which it cannot control.

User-defined error detection A user may also define protocol-specification error checking, using the `&check` attribute. For example, one may check the data against some protocol signature (e.g., the first 4 bytes of a CIFS/SMB message should be "`\xffSMB`") to make sure the traffic data indeed reflects the protocol.

5.2.2 Error Recovery

Currently errors are handled in a simple model: when the flow processing function catches an exception, it logs the error, discards the unfinished message as well as the unprocessed data, and initializes to resume on the next chunk of data.

One potential problem with this approach is that, for stream-based protocols, the next message might not be aligned with the next payload chunk. In the future we plan to add support for re-discovering message boundaries in such cases. Having such a mechanism will also help to further improve parsing performance, as we can then skip large, semantically uninteresting messages, and re-align with the input stream afterwards.

6. EXPERIENCES

We have used `binpac` to add protocol parsers for CIFS/SMB, DCE/RPC (including its end-point mapping protocol) and NCP to `Bro`'s traffic analysis engine.[5] To compare `binpac` with hand-

[5] Given the complexity of CIFS, the parser does not yet cover the entire protocol, but only the commonly seen message types.

Protocol	Hand-written			binpac		
	LOC	CPU Time (seconds)	Throughput	LOC	CPU Time (seconds)	Throughput
HTTP	1,896	538–541	244 Mbps / 36.7 Kpps	676	442–444	298 Mbps / 44.7 Kpps
DNS	1,425	37.3–37.5	18.6 Mbps / 13.3 Kpps	698	44.7–44.8	15.6 Mbps / 11.1 Kpps

Table 2: Performance.

written protocol parsers, we also rewrote the parsers for HTTP and DNS (and SUN/RPC, which we have not yet evaluated) in `binpac`. We use these latter to provide a comparison in terms of code size and performance between `binpac`-based and hand-written parsers.

As Table 2 shows, the `binpac`-based parsers for HTTP and DNS have code sizes of roughly 35–50% that of the hand-written parsers, measured in lines of code (and the same holds in source file sizes), respectively. We also note that for both protocols, the Bro-specific semantic analysis comprises well over half of the `binpac` specification, so for purposes of reuse, the specifications are significantly smaller than shown.

To test the performance of the parsers, we collected a one-hour trace of HTTP and DNS traffic at Lawrence Berkeley National Laboratory's network gateway. The HTTP subset of the trace spans 19.8M packets and 16.5 GB of data. The DNS subset spans 498K packets and 87 MB. The drop rate reported by tcpdump when recording the trace was below $4/10^6$.

Table 2 shows the CPU time required for each type of analysis, giving the minimum and maximum times measured across 5 runs, using a 3.4 GHz Xeon system running FreeBSD 4.10 with 2 GB of system RAM. We also show the throughput in bits/sec and packets/sec, observing that on a per-packet basis, DNS analysis is much more expensive than HTTP analysis, since many HTTP packets are simply entity data transfers requiring little work.

For these numbers, we disabled Bro's script-level analysis of the protocols, so the timings reflect the computation necessary for the parser to generate the Bro events corresponding to the application activity (including TCP/IP processing and TCP flow reassembly), but no further processing of those events. Specifically, the HTTP parser generates an event for (1) every request line (with the method, the URL, and the HTTP version as event parameters), (2) every response line (with the response number and the reason phrase), (3) every HTTP header (with the name and the value, in either request or response), and (4) the end of every request/reply message (with the length of the body). The DNS parser generates an event for every request and reply and for every answer in the reply. Thus, the generated events allow almost all essential analysis on HTTP and DNS, except that HTTP data bodies are not exported through events.

We see that the `binpac` HTTP parser performs significantly better than the hand-written one. This gain came after tuning the specification by adding a `&transient` attribute to `HTTP_header` fields, which instructs `binpac` to not create a copy of the corresponding bytestring. (Transient strings are visible only within the parsing function of the type, while non-transient ones, which are copied, can be accessed after parsing.) We have not yet applied the same tuning to the DNS specification; as a result, it allocates many more dynamic objects, and copies more strings than the hand-written one does. We do, however, believe that tuning it will prove straightforward.

Both the hand-written and the `binpac` HTTP parsers handle content gaps by skipping over gaps within data bodies, which are common in large traces. However, it is worth noting that in the hand-written case, content-gap skipping must be crafted in each case and thus is applied in very limited scope beyond HTTP bodies (such as SMTP mail bodies). For `binpac`-based parsers, gap skipping is automatically handled for every ***`string`*** type with a `&chunked` attribute.

We also note that in developing our DNS parser we found two significant bugs in the hand-written parser's processing. These related to using incorrect field widths and non-portable byte-ordering manipulations, and provide direct examples of the benefit in terms of correctness for specifying analyzers in a high-level, declarative fashion.

7. SUMMARY AND FUTURE DIRECTIONS

This paper presents `binpac`, a declarative language for generating parsers of application-layer network protocols from high-level specifications. Such parsers are a crucial component of many network analysis tools, yet coding them manually is a tedious, time-consuming, and error-prone task, as demonstrated by the numerous severe vulnerabilities found in such programs in the past.

`binpac` reflects a different paradigm for building protocol parsers: abstracting their syntax into a high-level meta-grammar, along with associated semantics. A parser generator then translates the specification into low-level code automatically. By providing such an abstraction, a programmer can concentrate on high-level protocol aspects, while at the same time achieve correctness, robustness, efficiency and reusability of the code.

In spirit, this approach is similar to that embodied in the use of `yacc` for writing parsers for programming languages, but many elements of the network analysis problem domain require significantly different underlying mechanisms. First, there are critical differences between the syntax and grammar of network protocols and context-free languages. In addition, processing network traffic requires a fundamentally different approach in terms of handling input, namely the ability to incrementally parse many concurrent input streams.

Our domain-specific `binpac` language addresses these issues with a set of network-specific features: parameterized types, variable byte ordering, automatic generation of boundary checking, and a hybrid approach of buffering and incremental parsing for handling concurrent input. `binpac` supports both binary and ASCII protocols, and we have already used it to build parsers for HTTP, DNS, SUN/RPC, RPC portmapper, CIFS, DCE/RPC (including the endpoint mapper), and NCP. We integrated all of these into the Bro NIDS, replacing some of its already existing, manually written ones. Our evaluation shows that `binpac` specifications are 35–50% the size of handcoded ones, with the protocol description (independent of the user's analysis semantics) comprising less than half of the specification. Our HTTP parser runs faster than the handcrafted one it replaces (and with equal memory consumption), and we are confident that the DNS will likewise soon exhibit performance equal to the one it replaces. `binpac` is open-source and now ships as part of the Bro distribution.

In the future, along with specifying further protocols in `binpac`, we envision exploiting its power in two areas. First, we wish to

explore the reusability of `binpac`-generated parsers by integrating them into additional network tools. Second, we intend to add back-ends other than C++ to `binpac` to generate parsers for different execution models. As proposed in [32], we specifically aim to build highly parallel parsers in custom hardware.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF Awards CNS-0520053, STI-0334088, ITR/ANI-0205519, and NSF-0433702, as well as a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD). Our thanks to John Dunagan and the anonymous reviewers for numerous valuable comments.

9. REFERENCES

[1] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[2] M. Arlitt, B. Krishnamurthy, and J. C. Mogul. Predicting short-transfer latency from TCP arcana: A trace-based validation. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2005.

[3] *Abstract Syntax Notation One (ASN.1)*. ISO/IEC 8824-1:2002.

[4] G. Back. Datascript—a specification and scripting language for binary data. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 66–77, London, UK, 2002. Springer-Verlag.

[5] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[6] T. P. Blumer and J. C. Burruss. Generating a service specification of a connection management protocol. In *PSTV*, pages 161–170, 1982.

[7] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. Generic application-level protocol analyzer and its language. Under submission.

[8] Common Internet File System. http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf.

[9] D. Crocker. *RFC 2234: Augmented BNF for Syntax Specifications: ABNF*.

[10] DCE 1.1: Remote procedure call. http://www.opengroup.org/onlinepubs/9629399/toc.htm.

[11] DSniff. www.monkey.org/ dugsong/dsniff.

[12] *The Ethereal Network Analyzer*. http://www.ethereal.com/.

[13] A. Feldmann, N. Kammenhuber, O. Maennel, B. Maggs, R. D. Prisco, and R. Sundaram. A Methodology for Estimating Interdomain Web Traffic Demand. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2004.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[15] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–304, New York, NY, USA, 2005. ACM Press.

[16] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, New York, NY, USA, 2006. ACM Press.

[17] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of USENIX Security Symposium*, 2001.

[18] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[19] V. Jacobson, C. Leres, and S. McCanne. *TCPDUMP*. ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[20] S. C. Johnson. YACC - Yet Another Compiler-Compiler. Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[21] J. Jung and E. Sit. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of the Internet Measurement Conference (IMC)*, Taormina, Italy, October 2004.

[22] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–13, Cambridge, MA, August 1999.

[23] C. Kreibich. *NetDude (NETwork DUmp data Displayer and Editor)*. http://netdude.sourceforge.net/.

[24] C. Kreibich. Design and implementation of netdude, a framework for packet trace manipulation. June 2004.

[25] A. Kumar, V. Paxson, and N. Weaver. Exploiting Underlying Structure for Detailed Reconstruction of an Internet-scale Event. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2005.

[26] P. J. McCann and S. Chandra. Packet Types: Abstract specifications of network protocol messages. In *Proceedings of the ACM SIGCOMM Conference*, pages 321–333, 2000.

[27] P. Mockapetris. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION, Section 4.1.4, November 1987. RFC 1035.

[28] *NFR Security*. http://www.nfr.com.

[29] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet Background Radiation. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2004.

[30] T. Parr and R. Quong. ANTLR: A predicated-ll (k) parser generator. *Software, Practice and Experience*, 25, July 1995.

[31] V. Paxson. BRO: A system for detecting network intruders in real time. In *Proceedings of USENIX Security Symposium*, San Antonio, TX, January 1998.

[32] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *Proceedings of Workshop on Hot Topics in Security (HotSec)*, Vancouver, B.C., Canada, July 2006.

[33] *NetWare Core Protocol*. http://forge.novell.com/modules/xfmod/project/?ncp.

[34] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[35] M. Roesch. SNORT: Lightweight intrusion detection for

networks. In *Proceedings of USENIX LISA*, 1999.

[36] The SNORT network intrusion detection system. http://www.snort.org.

[37] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[38] C. Shannon and D. Moore. The Spread of the Witty Worm. http://www.caida.org/analysis/security/witty, 2004.

[39] R. Srinivasan. *RFC 1831: RPC: Remote Procedure Call Protocol Specification*.

[40] R. Srinivasan. *RFC 1832: XDR: External Data Representation Standard*.

[41] *Ethereal OSPF Protocol Dissector Buffer Overflow Vulnerability*. http://www.idefense.com/intelligence/vulnerabilities/display.php?id=349.

[42] *Snort TCP Stream Reassembly Integer Overflow Exploit*. http://www.securiteam.com/exploits/5BP0O209PS.html.

[43] *Symantec Multiple Firewall NBNS Response Processing Stack Overflow*. http://www.eeye.com/html/research/advisories/AD20040512A.html.

[44] *tcpdump ISAKMP packet delete payload buffer overflow*. http://xforce.iss.net/xforce/xfdb/15680.

[45] Separation of concerns. http://en.wikipedia.org/wiki/Separation_of_concerns.

[46] C. Wong, S. Bielski, J. M. McCune, and C. Wang. A study of mass-mailing worms. In *Proceedings of the 2005 ACM Workshop on Rapid Malcode (WORM)*, pages 1–10, New York, NY, USA, 2004. ACM Press.