A Generic Application-Level Protocol Analyzer and its Language

Nikita Borisov
UIUC
nikita@uiuc.edu

David J. Brumley
Carnegie Mellon University
dbrumley@cs.cmu.edu

Helen J. Wang
Microsoft Research
helenw@microsoft.com

John Dunagan
Microsoft Research
jdunagan@microsoft.com

Pallavi Joshi
UC Berkeley
pallavi@eecs.berkeley.edu

Chuanxiong Guo
ICE Nanjing
xguo@ieee.org

Abstract

The Shield project relied on application protocol analyzers to detect potential exploits of application vulnerabilities. We present the design of a second-generation generic application-level protocol analyzer (GAPA) that encompasses a domain-specific language and the associated run-time. We designed GAPA to satisfy three important goals: safety, real-time analysis and response, and rapid development of analyzers. We have found that these goals are relevant for many network monitors that implement protocol analysis. Therefore, we built GAPA to be readily integrated into tools such as Ethereal as well as Shield.

GAPA preserves safety through the use of a memory-safe language for both message parsing and analysis, and through various techniques to reduce the amount of state maintained in order to avoid denial-of-service attacks. To support online analysis, the GAPA runtime uses a stream-processing model with incremental parsing. In order to speed protocol development, GAPA uses a syntax similar to many protocol RFCs and other specifications, and incorporates many common protocol analysis tasks as built-in abstractions. We have specified 10 commonly used protocols in the GAPA language and found it expressive and easy to use. We measured our GAPA prototype and found that it can handle an enterprise client HTTP workload at up to 60 Mbps, sufficient performance for many end-host firewall/IDS scenarios. At the same time, the trusted code base of GAPA is an order of magnitude smaller than Ethereal.

1 Introduction

The Shield project [54] introduced the concept of vulnerability-specific, exploit-generic signatures for intrusion prevention. These signatures differ from traditional exploit-specific signatures by describing all paths that lead to a network-exploitable vulnerability. In doing so, they require a much more detailed understanding of both network protocols and application context. Therefore, each signature performs some form of *protocol analysis*, parsing network message formats and reconstructing context.

The original Shield prototype included a domain-specific language for performing protocol analysis necessary for a signature. However, the language was incomplete and difficult to use for some protocols, in particular, text-based ones and complex ones involving multiple layers. This paper describes the design, implementation, and evaluation of the second-generation language for Shield, supporting rapid development of memory-safe and DoS-resilient protocol analyzers.

In building the language and its runtime, we had several goals. The most important one was *safety*: as Shield is intended to ensure the reliable operation of applications in a potentially adversarial environment, it is imperative that Shield analyzers do not introduce new sources of failure or vulnerabilities. We therefore wanted to avoid memory corruption errors and provide resistance to denial-of-service attacks. All of the analysis is performed within a memory-safe language, and we introduced further restrictions into the language to limit the amount of memory and CPU used by protocol analysis.

The second goal was one of *real-time analysis and response*. Shield acts as a “bump in the wire”; therefore, it must not buffer data for too long to avoid application delays or even deadlock, and at the same time it must not deliver data to the application until it can be sure that it is safe and will not result in a vulnerability. The language and runtime are therefore designed based on a streaming data model, parsing, analyzing, and enacting decisions on potentially incomplete application messages. In terms of performance, our prototype implementation can process data at speeds matching the traffic demands of busy web servers, allowing for online operation.

Our third goal was to support *rapid development* of protocol analyzers and vulnerability signatures. The timely deployment of vulnerability signatures is essential to Shield's effectiveness. To facilitate the development of protocol analyzers in Shield, we incorporated common tasks involved in protocol analysis, such as session management and buffering of incomplete data, as features of the language and the associated runtime. We structured the domain-specific language to be similar to the BNF specifications found in many RFCs that describe protocols. We also introduced a visitor syntax to separate vulnerability-specific analysis logic from message parsing and protocol context reconstruction, so that the latter can be reused among many vulnerability signatures for a single protocol or application.

Although our initial motivation came from the Shield project, we recognized that the above goals are important for other tools that perform application-level protocol analysis. In particular, network monitoring tools such as Ethereal [50] and tcpdump [31], intrusion detection systems such as Snort [51] and Bro [45], and application-level firewalls such as Hogwash [28] all perform some form of protocol analysis, but each tool involves hand-coding the analyzers in a general-purpose, low-level language such as C. This approach is both expensive and error-prone, resulting in dozens of security vulnerabilities that have been found in recent years in the popular tools Ethereal and tcpdump. We therefore designed our language and runtime to be a generic component that can be incorporated into such tools and perform the protocol analysis in a secure fashion. We called our tool GAPA, the Generic Application-level Protocol Analyzer, and its associated language GAPAL. The use of GAPA could reduce the trusted code base of tools like Ethereal by over an order of magnitude and greatly reduce the risk of worms that exploit the parsing logic of security tools, such as Witty [49].

The rest of the paper is organized as follows. We describe the GAPA language in section Section 2, and the language runtime in Section 3. We present our evaluation in Section 4. In Section 5, we compare and contrast GAPA with related work. Finally, we conclude in Section 6.

2 GAPA Language

A protocol analyzer specification in the GAPA language (GAPAL), called a *Spec*, takes care of three tasks. First, it specifies how to parse the message format used by a protocol. Second, it needs to correctly track session and application context. Finally, it needs to perform analysis based on the message content and the application context, and potentially carry out decisions, such as terminating a connection. In this section, we discuss the language features for these three tasks and then present some of the important safety features built into the GAPA language. We will illustrate many of the features using the specification for the HTTP protocol in Figure 1.

2.1 Message Parsing

Each protocol defines a particular message format for exchanging data. These formats can roughly be classified into text and binary. Text formats use ASCII text to encode both the structure and the content of messages, following some sort of grammar that is often formalized in Backus-Naur form (BNF). Binary formats use machine structures to encode data, overlaying C-like constructed types onto the data stream. A generic protocol analyzer must, of course, be able to parse both types of protocols.

The original Shield prototype language was mostly oriented towards parsing binary protocols, with the language describing data structures that form messages. Text protocols were supported by switching from byte to word or token positions; a decision that we ultimately found too cumbersome for easy protocol specification. In the second-generation language, we have adopted a BNF-like grammar for specifying message formats. This grammar makes it easy to represent text protocols, and there is a natural mapping of binary protocols to this grammar as well, making the language very versatile.

The grammar section of a GAPAL specification consists of production rules that specify a mostly (see below) context-free language. Each production maps a non-terminal (variable) to a sequence of terminals (tokens) and non-terminals; additionally, an alternation operation can be used to select among multiple such sequences. This structure closely mirrors BNF notation, and we found that much of the task of specifying text protocols can be accomplished merely by copying the BNF specification out of an RFC. Each item in the sequence on the right-hand side of a production can also be annotated with a symbol name, in the form of `<symbol>:<type>`. The symbol name can be used to refer to that part of the message later during protocol analysis.

Expressing binary protocols is also straightforward. In this case, the terminals represent base types, such as bytes or k-bit integers, non-terminals represent structures, and alternation is used to encode unions. We also support array notation, with both statically and dynamically-sized arrays.

RFCs often use BNF notation to describe a protocol, but these specifications are not in general context-free grammars; they contain ambiguities that are resolved based on other fields in a message or protocol state. Rather than force GAPAL authors to rewrite specifications to be context-free, we incorporate the concept of *directed parsing*, where a programmer can use an interpreted C-like language and embed code to specify how a certain field should be parsed based on variables computed from parsing previous fields. For

```
protocol HTTPProtocol {
  transport = (80/TCP);

  /* Session variables */
  int32 content_length = 0;
  bool chunked = false;
  bool keep_alive = false;

  /* message format specification
     in BNF-like format */
  grammar {
    WS = "[ \t]+";
    CRLF = "\r\n";

    %%

    HTTP_message -> Request | Response;
    Request-> RequestLine HeadersBody;
    Response-> ResponseLine HeadersBody;

    HeadersBody ->
      {
        chunked = false; keep_alive = false;
        content_length = 0;
      }
      Headers CRLF
        {
          /* message_body's type is resolved
     (:=) at runtime based on
             Transfer-Encoding */
       if (chunked)
            message_body := ChunkedBody;
          else
            message_body := NormalBody;
        }
       message_body:?;

    Headers -> GeneralHeader Headers | ;
    GeneralHeader->
      name:"[A-Za-z0-9-]+" ":"
      value:"[^\r\n]*" CRLF
        {
          if (name == "Content-Length") {
            content_length = strtol(value,10);
          } else if (name=="Transfer-Encoding"
            && value==" chunked") {
            /* slight simplification */
            chunked = true;
          } else if (name == "Connection"
            && value == " keep-alive") {
            keep_alive = true;
          }
        };

    NormalBody ->
      bodypart:byte[content_length]
      {
        /* ``send'': sending "bodypart" to
           the upper layer (e.g., RPC)
           for further parsing */
        send(bodypart);
      } ;
    [...]
  };  // Grammar

  state-machine httpMachine
  {
    (S_Request,IN)->H_Request;
    (S_Response,OUT)->H_Response;

    initial_state=S_Request;
    final_state=S_Final;
  };

  /* Always expect a response after a request */
  handler H_Request (HTTP_message) {
    int headerCount = 0;

    /* visitor syntax */
    @GeneralHeader->{
      print ("header name = %v \n", name);
      headerCount ++;
    }
    print(``Total number of headers: %v\n'',
          headerCount);
    return S_Response;
  };

  handler H_Response(HTTP_message) {
    if (keep_alive) {
      return S_Request;
    } else {
      return S_Final;
    }
  };

}; // protocol
```

Figure 1. Abbreviated HTTP specification in GAPAL.

example, the length of the body of an HTTP message is specified by the header field `Content-Length` as part of `GeneralHeader`; in Figure 1 the content length value is saved inside a variable and retrieved in the `NormalBody` production. Code blocks are also helpful when the type of a symbol is best determined at runtime. We introduce a *resolve* operator, denoted `:=`, which allows the statements to specify how to parse subsequent fields. A resolve assigns a type (or a non-terminal), specified on the right-hand side, to a symbol name on the left-hand side. A dynamically resolved symbol name is denoted with the '`?`' type. For example, in our HTTP specification (Figure 1), the type of `message_body` (as part of `HeadersBody`) depends on the value of `Transfer-Encoding` header. It is possible to rewrite these grammars to avoid the resolve operator and be context-free, but the resulting specification is much more awkward. Directed parsing is also useful in binary protocols to support such idioms as tagged unions.

2.2 Tracking Context

A typical protocol interaction involves the exchange of multiple messages from both sides. Therefore, a GAPAL specification must track the protocol context (and the corresponding application context) across multiple messages within a single interaction, or *session*. We define a session to be a group of related messages; for many protocols this corresponds directly to a single TCP connection, and this is the default session grouping used by GAPA. Other protocols, such as those running over UDP, may encode explicit session identifiers inside messages or have other implicit session identification mechanisms. To support such protocols, we allow a programmer to define a session identification handler, which parses a message (perhaps partially) and returns the session identifier that the message belongs to. Such handlers can be used in both TCP and UDP protocols.

Once a session has been selected, parsing occurs according to a state machine. The machine specifies what kind of messages may be received from which end of the connection in a given state. Once a message is received and fully parsed, GAPA calls a *handler*, written in the interpreted language, to determine the next state. The handlers are necessary because the state transition may depend on the content of a message. Handlers are also used for protocol analysis, as described below.

The HTTP state machine, shown in Figure 1, is conceptually simple, alternately expecting a request message from the client or a response from the server. The response handler in this case will either go to a final state or start waiting for another request based on a variable saving the value of the HTTP `Connection` header. Other protocols may have more complicated state machines, with messages potentially arriving from both sides.

2.3 Visitors

To perform analysis, handlers will need to refer to fields of a message according to the recursive grammar. In simple cases, dot notation such as $a.b.c$ could be useful. However, the dot notation becomes cumbersome in cases with deep recursion or alternation, which occur in both binary and text-based protocols. For example, RPC may have up to 11 different alternations with each alternation 4 levels deep. In the case of alternation, one must explicitly check which case was chosen in the current message in order to avoid referring to fields that are not present.

To address the difficulty of dot notation, we allow the programmer to write grammar visitors [25] inside handlers. A *visitor* is a block of code that is executed each time a rule is visited. The syntax for a visitor is:

```
@ <non-terminal> -> { ... <code block> ... }
```

The syntax assigns the non-terminal (or its alternation) a code block to run every time after the non-terminal (or the alternation) is parsed.

These code blocks work similarly to the blocks inserted into the grammar, however, we want to enable a clean separation between the parsing logic and the specific protocol analysis tasks so that the same parsing logic can be re-used for different tasks. Essentially, the visitors in a handler represent the handler's customization of message parsing for the purpose of a protocol analysis task. Consequently, visitors are always executed before the rest of the handler code.

As an example, in our HTTP specification in Figure 1, handler `H_Request` contains a visitor statement that "visits" non-terminal `GeneralHeader`, counts the number of headers, and prints all header names. Every time `GeneralHeader` is traversed during parsing, `headerCount` is incremented, and the header name is printed. When the entire message is parsed, the total number of headers is printed.

2.4 Layering

Protocols can be layered on top of other protocols, with a GAPAL Spec at each layer performing protocol analysis and sending data up to the next layer. A Spec can indicate a lower layer Spec with a `uses` statement, or directly bind to the transport layer with a `transport` statement. The lower layer Specs use the `send` call to pass data to upper layers, where it appears as an incoming data packet. Each layer has its own session identifiers, grammar, state machine, and handlers.

We also use our layering mechanism as a general way of composing data processing logic, very much in the same spirit as the Unix pipe. In particular, we use layering to implement application-level protocol fragmentation and datagram reordering. The lower layer parses out the fragment data and uses a special version of the `send` call to indicate to GAPA the fragment sequence metadata for the current datagram. GAPA then performs fragment reassembly before passing the data to the upper layer, which parses the reassembled data into meaningful message components.

The HTTP Spec in Figure 1 shows an example usage of `send` for protocol layering: the HTTP message body, parsed into the `bodypart` variable under the non-terminals `NormalBody` and `Chunk`, is "sent" to the next protocol layer, say, RPC, for further processing. As another example, we used layering to implement the vulnerability filter for CodeRed: the HTTP protocol identifies the URLs in the HTTP requests and pipes them on to a CodeRed URL parser which detects and blocks CodeRed.

2.5 Safety

Safety is a primary goal for the GAPA language. First and foremost, we want to ensure that GAPA analyzers do not suffer from crashes or buffer overruns due to memory errors. To avoid this, the GAPA interpreted language is strictly typed, with bounds checks on array accesses and no dynamic memory allocation. The lack of dynamic memory allocation is also helpful to prevent unbounded memory usage. This places a restriction on programmers to implement certain kinds of logic, but in our experience, we have not found this to be an inconvenience.

We also make sure that storing the partially-parsed message does not result in unbounded memory usage. The GAPA engine is designed to free memory for those message fields that will no longer be referenced, and to apply computation to message fields as early as possible through incremental execution (see Section 3.1). This allows us to parse variable-sized message components such as arrays in a streaming fashion: the programmer writes code to be run on each element using the visitor syntax or a `foreach` loop, and this code gets executed as each array element is being parsed, after which the memory for that element is released. The programmer can retain some of the information from the variable-sized arrays by storing it in a statically bounded buffer. For other types of message components, such as tokens specified by regular expressions, we enforce a static bound on their length.

To avoid excessive CPU consumption, we allow only a single looping construct within GAPAL — the aforementioned `foreach` statement. The `foreach` statement iterates over all items inside a safe array. It is limited to forward traversal [36]. The purpose of our `foreach` loop is to allow parsing of iterative structures in messages or to perform a constant number of iterations of certain tasks. It is possible to nest `foreach` statements for parsing nested, iterative structures in messages, but nesting `foreach` on the same array is disallowed as this is incompatible with the streaming model. In our experience using GAPAL, we find this limited iteration is sufficient. The `foreach` design allows the CPU cost per byte of network traffic to be statically bounded.

3 The Analysis Engine

The analysis engine is the GAPAL runtime. It parses messages using a recursive descent parser. The analysis engine first finds the appropriate GAPAL Spec for the current packet to be analyzed. The engine then follows the grammar that specifies the message format to generate a parse tree. Since the engine may receive packets containing incomplete messages, it performs parsing incrementally, saving parsing state between packets. During parsing, the engine executes code fragments, both those embedded in the message grammar and those resulting from the visitor pattern in the handlers (Section 2.3). The handlers use both the parse tree and any other session state updated by the code fragments to carry out task-specific logic and to update the current protocol state.

Whenever the analysis engine fails to parse a message, it alerts the application containing GAPA, which can then react appropriately. For example, a firewall would likely drop the message, while a network monitor such as Ethereal could display an uninterpreted byte stream.

In the remainder of the section, we focus on our techniques to limit the state in order to resist state-holding attacks and to achieve fidelity in the engine's interpretation of the current communication state of an application.

3.1 Limiting State

To prevent state-holding, we structure our analysis engine to perform filtering decisions as quickly as possible with the use of *incremental execution*. After receiving each packet, the appropriate handler is executed, even if the application-level message is incomplete. The handler is run until it references a message field that is not yet filled with a value. At that point, its execution is suspended and a continuation is saved, to be resumed when the next packet arrives. If the next packet contains the referenced field, the handler execution continues, otherwise, it is suspended once again. If the handler completes, the rest of the message is parsed without saving any state.

This approach allows the handlers to make filtering decisions on incomplete messages. If, for example, filtering is based on the content of a certain field, a handler performing such a check will be executed as soon as that field is fully parsed. We therefore pass packets containing incomplete messages to the application as soon as the incremental execution of all the handlers is complete.

One subtle issue raised by incremental execution in the firewall scenario is that a partial field that is passed to the application during incremental execution could trigger an application vulnerability such as a buffer overrun to be exploited before the full field is parsed and examined by GAPA. To address this issue, GAPA uses a stream-based rather than buffer-based implementation for built-in functions (e.g., length functions or regular expressions) for message fields whose sizes are unpredictable, such as a byte sequence field that ends with some terminator symbol. In the following example,

```
  if (strlen(field) > 1024) ...
```

`strlen` is stream-based: the comparison completes as soon as the engine parses a packet that causes `field` to have more than 1024 characters even if `field` has more bytes to come.

The programming semantics exposed by incremental execution to GAPAL handlers is that a message field is passed to the application after it is parsed and the handler has been given an opportunity to process it. This works well when a programmer processes message fields in the order of their arrival, which is typically the case. However, when a handler implementation processes multiple message fields in a different order from their parsing order, this can lead to unintended behavior: a handler may be suspended as it is waiting for a later field while the earlier potentially malicious field is passed up to the application. We make the design decision to place responsibility of avoiding such unintended behavior on GAPAL programmers because otherwise, the engine would need to buffer packets until all the message fields have arrived, resulting in unbounded state accumulation. To help programmers follow our programming semantics, we apply static analysis to identify such occurrences in GAPAL programs. In the future, we plan to investigate the possibility of automatically reordering code when there are no control- or data-flow dependencies between statements.

With this semantics of the incremental execution, only one partial field (or token) is saved across packets; and the other completed fields that are carried by the current packet are released immediately after the handler execution for the packet. To bound the saved state of the partial field, we chose to mirror the behavior of the applications, which usually involves imposing an artificial limit on the size of such tokens (e.g., Apache has a configurable maximum header token size [52]). We allow the programmer to specify the maximum length of any token, as well as a global maximum for all tokens in the message, and stop parsing the incomplete token once this length is exceeded.

Incremental execution applies naturally to session dispatching as well. When a message is received, we incrementally execute the session identifier logic, followed by the appropriate handler of the dispatched session. The transition between the dispatcher and the handler happens automatically at the point when enough of the session information has been parsed to decide which session's handler to run.

When multiple layered specs are used, we incrementally execute the handlers at each layer before passing a packet onto the application, since each layer may decide to filter the current message or session. We avoid data being buffered at a lower layer both to reduce the memory footprint and to ensure that filtering decisions at the upper layers can happen as early as possible. To support this, we incrementally execute the `send` operation on incomplete fields by sending the partial field up to the upper layer after each packet.

An attacker can also perform a state-holding attack by

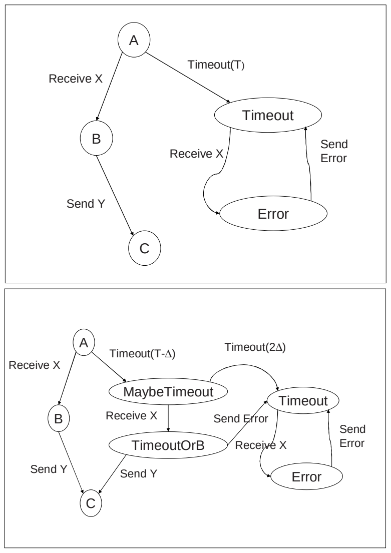

Figure 2. Application state machine and GAPA-maintained state machine

creating many concurrent sessions and causing our analysis engine to maintain session state for each. The monitored application can itself be attacked this way even when GAPA is absent. Our goal is to make the engine at least as scalable as the application. The per-session state maintained by the engine for parsing should be substantially less than the per-session state in the application. To deal with an attack using many concurrent sessions, we mirror the application behavior. For example, if an application explicitly closes outdated sessions, GAPA observes the socket close events and follows suit by discarding all the corresponding session state. The application may also use a timer to expire unused sessions with no externally-visible action; in this case, we use the timer support in GAPA, described below, to copy the application behavior.

3.2 Achieving Fidelity

GAPA's interpretation of the application communication state must stay synchronized with that of the application process. In the majority of cases, application state changes are either caused by or followed by a network message, and therefore it is easy to maintain fidelity in the GAPA state. However, in cases of timeouts there may be a state transition without any network messages, posing a challenge for GAPA. In this section, we discuss how we achieve better fidelity for timeout events.

Protocol state machines often have timeout events for retries or for session state cleanup, in case of remote host or connectivity failures. If the timeout in the application triggers a network event, such as a retry message or a closed socket, GAPA can monitor for such events and avoid maintaining the timing itself. However, for network-silent timeouts, GAPA has no choice but to maintain a timer. Maintaining timing in GAPA is tricky because the timers in GAPA may not be precisely synchronized with those used by the application. Such inconsistencies can lead to incorrect analysis. Our implementation of network-silent timeouts is incomplete, but we include our design here.

A handler in GAPA can set a timeout using the `timeout`(*time*) built-in function. If no state transition occurs when the specified time has elapsed, a timeout handler is called. To cope with timing inconsistencies between GAPA and the application, we apply bifurcating analysis as used in Bro [45]; in addition, we assume that one direction of the traffic comes from a trusted source. Instead of transitioning to a *Timeout* state, GAPA can take a transition to a *MaybeTimeout* state early enough to account for timing error tolerances (specified by the programmer). From this point, all messages from one direction are processed along an ambiguous path, until a message from the other direction resolves the ambiguity. Such bifurcating analysis can be incorporated into the protocol state machine automatically.

For example, if an incoming message would cause a transition to state B from a non-timed out state, we use it to transition from *MaybeTimeout* to *TimeoutOrB*. If GAPA is used to filter malicious traffic, this transition must check for messages that are potentially dangerous either on the timed out or the normal path in the application. The response from the application will let GAPA resolve the ambiguity and transition to the correct state.

If no traffic is received in the *MaybeTimeout* state, GAPA can transition to the *Timeout* state after waiting long enough to make sure that the application timeout has certainly expired, accounting for synchronization errors. Since GAPA is run on the same host as the application, we expect these timing tolerances can be quite small, but we have not yet performed experiments to study this in more detail. Figure 3.2 shows the state machines maintained in the application and the analysis engine for this example.

4 Evaluation

4.1 GAPA Safety

We evaluate the safety of GAPA using the metric of trusted code base size. Though crude, this metric has been applied in the past to guide research in securing other critical systems, such as the Java Virtual Machine [3]. A smaller trusted code base is easier to analyze and is less likely to contain security-critical bugs.

The GAPA trusted code base includes the GAPA engine, written in C++, but excludes protocol specifications written in GAPAL. The reason for this is that while bugs in the engine can potentially cause application compromise, bugs in GAPAL Specs will, at worst, cause incorrect protocol analysis.

We contrast this with the traditional protocol analysis approach used in tools such as the widely-used Ethereal [50] suite of protocol analyzers. In Ethereal, both the engine and protocol analyzers are written in C, and bugs in either can cause application compromise. Therefore, we need to classify both components as part of the trusted code base.

A direct comparison of Ethereal and GAPA is inappropriate, since Ethereal includes many more components, such as a UI and support for numerous file components. We therefore eliminate all engine components and focus only on the protocol analyzers. Luckily, the Ethereal code base uses naming conventions to distinguish between broad classes of functionality: much of the protocol-analysis specific code is in a specific sub-directory, *epan/dissectors*.

Looking at only the code in that sub-directory, we count 779 thousand lines of code that are used for protocol analysis. This number dwarfs the size of the GAPA trusted code base by over an order of magnitude: the GAPA engine is only 24 thousand lines of code. Therefore, securing Ethereal will be a much more difficult task than securing GAPA. This is not just a theoretical concern, either; a search for "ethereal dissector" reveals 69 vulnerabilities recorded in the Common Vulnerabilities and Exposures database [16]. Furthermore, this problem is likely to only get worse, as new protocols are added to the Ethereal dissector collection.

In contrast, adding new Specs to GAPA does not affect its security, and although our current GAPA implementation could have some security defects, the significantly smaller trusted code base presents a much more appealing target for standard security best practices, such as code reviews, penetration testing, fuzz testing, and static analysis.

Rewriting the Ethereal protocol analyzers in a general-purpose memory-safe language, such as Java, would eliminate many, but not all, of the above vulnerabilities. In particular, about a quarter of the dissector vulnerabilities result in denial of service through excessive memory or CPU consumption; these vulnerabilities would still exist. This highlights the importance of the resource limitations imposed by GAPAL that make it impossible to introduce such vulnerabilities in protocol analyzer Specs.

4.2 GAPA Language Ease-of-Use

The goal of our GAPA language evaluation is to show that the language is complete enough to express many important protocols and vulnerabilities in these protocols, that the amount of effort required for the specification is reasonable, and that the language features are helpful in this task. To this end, we have specified a number of protocols: HTTP [21], RPC [47] (over both TCP and UDP), DNS [37], SIP [26], BitTorrent [10], DHCP [18], SSH [55] and TLS [17]. This represents a diverse collection, including text and binary protocols, both stream- and datagram-oriented. In all cases, we specified the protocol in at least the amount of detail as is contained in the description of the message format in the RFC. We also implemented vulnerability signatures on top of the HTTP, RPC-over-TCP, and DNS specifications. For HTTP, we implemented signatures for both the HostHeader [29] and the CodeRed [15] vulnerabilities. For RPC, we implemented a signature for the MSBlast [38] vulnerability. For DNS, we implemented a signature for a DoS vulnerability common to multiple DNS implementations resulting from pointers in the DNS record forming a loop [40], which we refer to as the pointer-cycle vulnerability. Filtering for the pointer-cycle vulnerability required parsing the values of the pointers and following them until a cycle is found. This is an example of a GAPAL filter that could not have been implemented using regular expressions, even when the pointer values are exposed by a DNS protocol analyzer, as is the model in numerous IDSs, e.g., Snort [51].

In all cases, we found the specification process to be straightforward, as we can start with an RFC using BNF, copy-and-paste, and then annotate it with additional parsing and protocol logic. We were able to construct an initial specification for most protocols within a few hours. None of the 9 protocol specifications required more than 300 semicolons (the GAPAL end-of-statement marker).

Our experience with GAPAL is that the language features of embedded code blocks, visitors, and layering were quite helpful. Embedded code blocks were essential in every protocol where the length of a later field is specified by an earlier field (e.g., BitTorrent, HTTP, and RPC). We used the visitor syntax in all our vulnerability signatures to avoid modifying the protocol specification itself. Finally, the layering feature made composition of logic easy. For example, the HTTP protocol identifies the URLs in the HTTP requests, parses and processes escapes in the URL encoding, and passes them on to a URL parser using the layering mechanism. The URL parser only requires 25 lines of GAPA code to filter for CodeRed. We verified that the filter detects a real CodeRed infection packet and produces no false positives on a 500MB trace from a high volume commercial web site. Additionally, the signature naturally handles CodeRed II and even polymorphic variants of CodeRed, because it occurs after the URL has been parsed and escapes removed.

Our experience with GAPAL is that it is a significant improvement over the Shield language. The Shield language was mostly suitable for binary protocols such as RPC [47], not text-based protocols such as HTTP [21]. Shield's approach was to treat text messages like binary ones, using a C-like *struct*, but to allow units of "offset" and "size" to be defined as *words* (made of characters), in addition to bytes. This requires manually converting the recursions and alternations in BNF rules of text-based protocols to these rigid structs, which is often very difficult. GAPAL's use of BNF eliminates this difficulty.

The Shield language also failed to cleanly separate protocol analysis from vulnerability filtering. The GAPA design allows maintaining a complete and well-tested protocol specification. When a new vulnerability is discovered, the visitor syntax makes adding a corresponding filter as easy as adding a few vulnerability-specific checks. The visitor syntax also makes merging vulnerability filters trivial (just run all the visitors), and it avoids Shield's requirement that the specification indicate fields that do not require parsing using the "SKIP" keyword (the GAPAL compiler instead can use the visitors to derive the unneeded fields).

4.3 GAPA Performance

We evaluated the performance of GAPA using traces collected in two different production networks. The `WebServer` trace contains 500 megabytes of traffic from a link connecting to a high-volume commercial web site. The `EdgeRouter` trace contains 1 gigabyte of traffic from a link connecting a subnet containing hundreds of PCs to the rest of a large corporate intranet.

We chose three protocol analyzers for our evaluation, HTTP, RPC, and DNS. HTTP is primarily a text-based protocol, while RPC is primarily a binary protocol. DNS is of particular interest because it contains both variable-length fields and pointers. We manually verified each analyzer's correctness on small traces.

The primary metric we use in the evaluation is throughput for the specific protocol's traffic after extracting it from our trace. We chose this metric to avoid allowing the exact fraction of the various protocols in our traces to influence our results.

We conducted the measurements on a PC with a 3GHz Xeon microprocessor and 2 GB of RAM. Averages and standard deviations were taken over 10 runs. The CPU was saturated in all cases. All measurements were taken on already reconstructed TCP streams, to focus on the throughput of our GAPA prototype, which integrates into an end-host above the transport layer. A larger system incorporating GAPA, such as a web server protected by a GAPA-driven firewall on top of a host TCP stack, would naturally have some lower total throughput reflecting the additional work performed by the web server.

In Figure 3 we show the throughput of each protocol analyzer with and without vulnerability signatures on the `EdgeRouter` trace. (We use a box plot [14] to show the variability between several measurements; the box contains

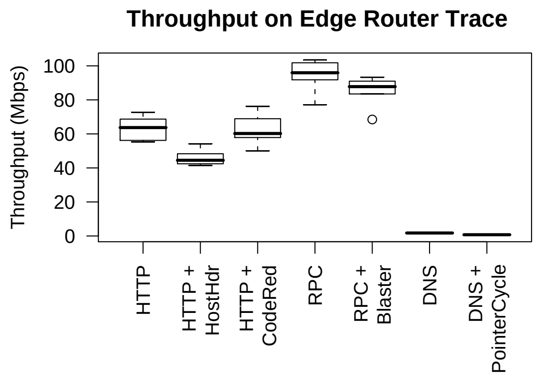

Figure 3. *EdgeRouter* **Trace**

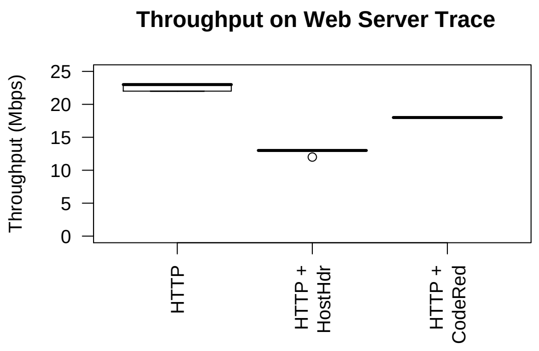

Figure 4. *WebServer* **Trace**

the middle 50% of the data and the line represents the median.) We found that the performance of our implementation of our HTTP analyzer was within a factor of 3 of published numbers on commercial HTTP protocol analyzers in firewall products [24]. The addition of either the CodeRed or the HostHeader vulnerability filters caused only minor drops in total throughput. The throughput for RPC was significantly greater than for the HTTP analyzer. The throughput for the DNS analyzer was quite low, and the addition of the pointer cycle vulnerability filter caused it to decrease even further. The order-of-magnitude difference in parsing rate between HTTP and DNS is consistent with the findings of other research on protocol analyzers [43]; DNS is more complicated to parse for protocol analyzers written in C as well.

In Figure 4 we show the throughput of the HTTP protocol analyzer with and without vulnerability signatures on the `WebServer` trace. The addition of vulnerability signatures caused minor degradations in throughput, as before. Throughput is approximately half what it was on the *EdgeRouter* trace.

The dramatic variation in bps throughput for the different

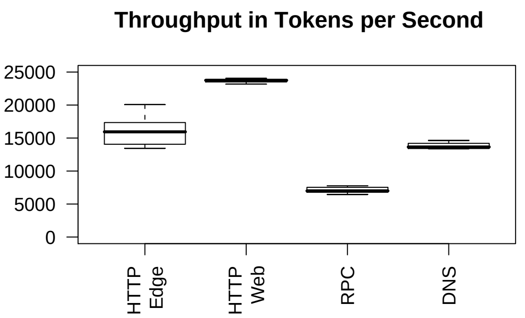

Figure 5. Comparison of throughput using tokens per second

protocol analyzers, and for the same protocol analyzer on different traces, can be explained by the complexity of the analysis being performed. In Figure 5, we show the number of tokens generated each second by the various protocol analyzers. The throughput in tokens per second is much closer across protocol analyzers than it is in bps. This suggests that the current dominant performance costs in our design are all per-token costs.

We first analyze the difference in throughput of the HTTP analyzer on the two different traces. Although the throughput in bps is higher on the `EdgeRouter` trace, the analyzer is parsing more tokens per second in the `WebServer` trace. Manual inspection revealed that the average packet length in the `WebServer` trace is only 446 bytes, approximately half that of the `EdgeRouter` trace. Because the HTTP analyzer treats the body of the HTTP request as an opaque array of bytes, the body is much faster to parse than the headers. In the `WebServer` trace, the bodies were much shorter, leading to less throughput under the bps metric. A similar effect explains the higher throughput in bps for the RPC analyzer when compared to the HTTP analyzer: the RPC bodies are longer than the HTTP bodies, leading to a higher throughput in bps.

The DNS analyzer's parsing rate in tokens per second is comparable to the parsing rate of HTTP. The difference in tokens per second seems largely due to the need for additional code blocks for determining the meaning of parsed fields. In particular, distinguishing between record names and record pointers, and switching appropriately, required examining individual bytes using the interpreted language much more often than was the case for HTTP headers. This explains the difference between the tokens per second rates for the two analyzers. The much larger difference in bps is explained simply by the fact that DNS has no large "body" against which the parsing costs can be amortized.

In Figure 6, we analyze the current costs of the various

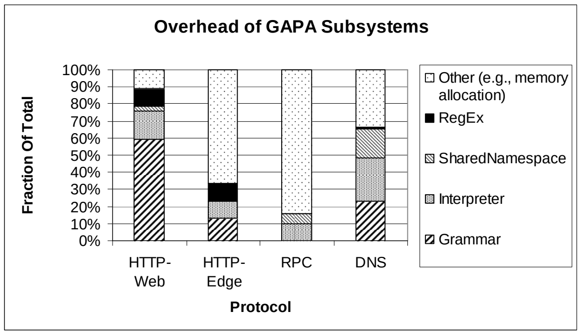

Figure 6. GAPA performance breakdown

components of the GAPA engine. The label "Grammar" refers to BNF-directed parsing. "Interpreter" refers to time executing handlers and embedded code blocks. "Shared-Namespace" refers to time spent reading and writing the grammar variables by the handlers and embedded code blocks. "RegEx" refers to time spent in the regular expression matching library. "Other" refers to memory allocation, copying, and initialization costs.

In the two lower throughput cases (HTTP-Web and DNS), we find that a significant fraction of time is going to the BNF-directed parsing and to the interpreter. In the two higher throughput cases (HTTP-Edge and RPC), we find that most of the CPU time is going to memory allocation, copying, and initialization costs. The CPU cost of interpreter initialization already incorporates the optimization of sharing interpreted code across sessions, so the remaining cost is simply allocating and initializing data. Although previous work on interpreters [46] has found that they often impose slowdowns ranging from a factor of 10 to over a factor of 100, the combined cost of the grammar interpreter and the C-subset interpreter is the dominant problem only for the lower throughput cases (HTTP-Web and DNS). For the higher throughput cases (HTTP-Edge and RPC), the major bottleneck is the classic problem of memory allocations and copies. In all cases, regular expression matching was a small fraction of overall CPU overhead. Our estimation is that all of the areas except regular expression matching are open to significant additional optimization. Compilation techniques seem like a particularly promising approach to performance improvement.

5 Related Work

5.1 Languages and Frameworks for Protocol Design, Verification and Implementation

Many languages and frameworks have been developed for protocol design, verification and implementation. Protocol analysis only aims to decode communication sessions of already-implemented and deployed protocols, and so can forgo functionality required in these other domains, e.g., dynamically allocating new ports or generating response packets. In this subsection, we explain how GAPA is better suited to the task of protocol analysis.

Languages based on formal methods [4], such as Estelle [12], Promela++ [6], LOTOS [53], and SDL [48] were originally targeted at protocol specification, emphasizing validation and verification of protocol logic through the use of finite state machines. StateCharts [27] and Esterel [8] are formal method-based languages that are more suitable for implementation. RTAG [2] (real-time asynchronous grammars) is also such a language, though it uses a context-free grammar to define the protocol behavior. GAPAL's use of finite state machines with code-directed transitions has close parallels in some of these languages, e.g., Esterel's use of code to decide the next state. However, these languages do not provide built-in abstractions for specifying protocol message formats. For example, in both the Esterel-based [13] and RTAG-based [2] TCP implementations, the TCP protocol state machine is specified in Esterel/RTAG, but packet access and manipulation have to be coded in C. GAPAL provides BNF with embedded code blocks for message format specification. Compared to C-based data manipulation, GAPAL provides memory safety and DoS resilience, e.g., GAPAL does not allow general-purpose loops or dynamic memory allocation.

Kohler *et al* [34] proposed Prolac, a statically-typed, object-oriented language for network protocol implementation, with the goal of improving readability, implementability, and extensibility. Prolac can be used to implement any network protocols, while GAPAL is more special-purpose with explicit, built-in support for abstractions needed for protocol analysis (message parsing, protocol state machine, visitors, layering, session dispatching). Like other protocol implementation languages, Prolac supports arbitrary recursion, while GAPAL does not.

The *x*-Kernel framework provides protocol, session, and message objects along with a set of support routines for buffer management, identifier mapping, and timers. Through this uniform interface among protocols, *x*-Kernel aims to improve the structure and performance of protocol layering. In accordance with GAPA's focus on protocol analysis rather than implementation, GAPA provides very different operations on protocols, sessions and message. For example, GAPA supports message parsing using BNF with embedded code, while *x*-Kernel supports message parsing by providing library functions that efficiently add or remove headers as a message moves down or up the protocol stack; GAPA uses the abstraction of a protocol state machine to track protocol state, while *x*-Kernel uses the protocol object to organize functionality (e.g., the

IP layer code is in the IP protocol object). Furthermore, GAPAL's type safety and DoS resilience are absent in the C-based *x*-Kernel.

Protocol conformance testing tools also share some similarities with GAPA, as they also must parse protocol messages and verify certain properties. However, testing scenarios differ from the context of firewalls or intrusion detection systems, as there is no need to worry about adversarial traffic trying to compromise or DoS the testing tool. Indeed, testing tools do not even need to operate online: Bishop *et al* [9] perform detailed conformance checking on TCP traces at 500 bps. The different emphasis in domain also manifests itself in the language design: Bishop *et al* choose a language offering logic operations, such as quantifiers, while GAPAL includes BNF with embedded code and other features to facilitate protocol analysis.

5.2 IDS/Firewall Languages

Snort and HogWash [51] use regular expressions as rules for matching malicious network input. To obtain more protocol context for such matching, they support protocol analyzers written in C as plug-ins. As discussed in Section 1, using C incurs both a high development cost and a high security risk.

The Bro [45] language is designed to specify network monitoring and intrusion detection policies based on known attack behaviors, such as port scanning. The Bro language does not support implementing protocol analysis, but the Bro system allows incorporating protocol analyzers written in C.

The Binpac [43] language has similar goals to GAPAL and naturally shares many of its features (the Binpac authors reference our early design document [11] in their work). Binpac is intended to perform message parsing only, providing an interface for analyzers written in a language such as C++, where GAPAL encompasses both parsing and reaction. Binpac is also compiled to C++ and uses C++ for parts of its parsing logic. For our purposes, C++ poses an unacceptable safety risk and we use our interpreted language for this purpose instead.

5.3 Data Description Languages

GAPAL is more than a data description language; it also includes protocol analysis-specific features such as the protocol state machine, layering, and visitors. In this section, we compare the message format part of the GAPAL design to previous work.

ASN.1 [19], NDR [47], and USC [42] target describing binary data structures. Datascript [5] and PacketTypes [35] specify binary data and enhance the specification with predicate-directed parsing. We found that text-based protocol messages can be more easily expressed using BNF with embedded code blocks; a more detailed discussion can be found in our ease-of-use evaluation in Section 4.2. Parser generators such as Yacc [33], ANTLR [44] and JavaCUP [30] use BNF and embedded code for parsing; GAPA additionally provides support for protocol analysis specific abstractions and a type-safe DoS-resilient language for the embedded code blocks. A somewhat orthogonal piece of work is Erlang's bit syntax [41], an Erlang language extension that supports binary pattern matching.

Recently, PADS [22] was proposed as a domain-specific language for specifying both text and binary data. Their goal is rapid parser generation for many kinds of ad hoc data such as web logs, finance records, firewall rules, etc.. Despite some differences in language appearance, PADS is similar to the message format specification part of GAPAL: GAPAL's use of BNF with embedded code block is mirrored by PADS's use of "switched PUnion" and code-directed parsing. Both GAPAL and PADS target text and binary data, and both arrived at a similar parsing design.

5.4 Packet Filters and Other Systems

Packet filters [36, 20, 7, 32] are programmable criteria for classifying or selecting packets from a packet stream based on headers for protocol layer 4 or below. Because of packet filters' emphasis on speed, they expose only simple rules for packet classification, such as predicates on byte values at fixed offsets. This makes them inappropriate for protocol analysis above the transport layer, which requires parsing multi-packet messages and keeping track of protocol state.

SPINE [23] and FLAME [1] are systems designed to allow untrusted code to process network messages in the kernel. They maintain isolation from arbitrary memory access within the kernel by using type-safe languages (Modula-3 and Cyclone, respectively), and they guard against unbounded CPU consumption by using timeouts. GAPAL's protocol-analysis specific abstractions and the analysis engine's built-in support for a stream processing model are absent from SPINE and FLAME. Also, SPINE and FLAME's timeout approach to DoS-resilience was inappropriate for our target scenario of a firewall; timeouts may lead to false positives.

Proof-Carrying Code [39]is a technique for conveying program properties like type safety or bounded execution time. If GAPA were compiled to machine code, instead of being interpreted, Proof Carrying Code would be a promising technique for avoiding a trusted compiler.

6 Concluding Remarks

Protocol analysis is important in numerous applications, such as intrusion detection, firewalling, and network monitoring. We have presented the design, implementation, and evaluation of a generic application-level protocol analyzer, GAPA, and its language. GAPA is the first system that allows rapid development of protocol analyzers while providing memory safety and DoS resilience. To achieve these properties, GAPAL uses techniques such as BNF with embedded code blocks, visitors for separating parsing-specific and task-specific logic, incremental execution, and layering. Our evaluation indicates that GAPAL is safe, expressive and easy to use and our GAPA system prototype can handle an enterprise client HTTP workload at up to 60 Mbps, sufficient performance for many end-host firewall/IDS scenarios.

7 Acknowledgments

Andrew Begel, John Douceur, Jon Howell, and Dawn Song gave us invaluable critiques on the drafts of our paper. Our work also benefited from discussions with Ronnie Chaiken, Jon Pincus, Dan Simon, Zhendong Su, and Zhe Yang. Geoff Nordland kindly provided us some of the network traces used in our evaluation upon our short-noticed requests. The anonymous referees offered many helpful suggestions. We are thankful to everyone's help.

References

[1] K. Anagnostakis, M. Greenwald, S. Ioannidis, and S. Miltchev. Open packet monitoring on FLAME: Safety, performance and applications. In *4th International Working Conference on Active Networks*, 2002.

[2] D. Anderson. Automated protocol implementation with RTAG. *IEEE Transactions in software engineering*, 14(3), March 1988.

[3] A. Appel and D.C.Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS TR-647-02, Princeton University, Apr. 2002.

[4] F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *IEEE Communications Surveys and Tutorials*, December 2002.

[5] G. Back. DataScript — A specification and scripting language for binary data. In *Proceedings of Generative Programming and Component Engineering*, volume 2487. LNCS, 2002.

[6] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In *Proceedings of the ACM SIGPLAN workshop on domain specific languages*, 1997.

[7] A. Begel, S. McCanne, and S. Graham. Bpf+: Exploiting global data-flow optimizations in a generalized packet filter architecture. *Computer Communication Review*, 29(4), 1999.

[8] G. Berry. *The Esterel Primer*.

[9] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of ACM SIGCOMM*, 2006.

[10] BitTorrent protocol specification. http://www.bittorrent.org/protocol.html/.

[11] N. Borisov, D. Brumley, H. Wang, J. Dunagan, P. Joshi, and C. Guo. Generic application-level protocol analyzer and its language. Technical Report MSR-TR-2005-133, Microsoft Research, Feb. 2005.

[12] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 1991.

[13] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *Proceedings of the ACM SIGCOMM*, 1996.

[14] J. Chambers, W. Cleveland, B. Kleiner, and P. Tukey. *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole, 1983.

[15] Microsoft Security Bulletin MS01-033, November 2003. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS01-033.asp.

[16] M. corporation. Common vulnerabilities and exposures database (CVE). `www.cve.mitre.org`.

[17] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1.0, January 1999. http://www.ietf.org/rfc/rfc2246.txt.

[18] R. Droms. Rfc 2131 - dynamic host configuration protocol, March 1997. http://www.faqs.org/rfcs/rfc2131.html.

[19] O. Dubuisson. *ASN.1 - Communication Between Heterogeneous Systems*. Morgan Kaufmann Publishers, 2000.

[20] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM*, 1996.

[21] 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.

[22] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of PLDI*, June 2005.

[23] M. Fiuczynski, R. Martin, B. Bershad, and D. Culler. SPINE: An operating system for intelligent network adapters. Technical Report TR-98-08-01, University of Washington, 1998.

[24] M. Fratto. Application-level firewalls: Smaller net, tighter filter, March 2003. http://www.nwc.com/1405/1405f32.html.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[26] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. RFC 2543: SIP: Session initiation protocol, March 1999.

[27] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[28] Hogwash. http://sourceforge.net/projects/hogwash/.

[29] Malformed HOST header causes IIS DoS, October 2002. http://www.securiteam.com/windowsntfocus/6C00C1F5QA.html.

[30] S. Hudson. CUP: LALR parser generator for Java. http://www2.cs.tum.edu/projects/cup/.

[31] V. Jacobson, C. Leres, and S. McCanne. Tcpdump. `www.tcpdump.org`.

[32] M. Jayaram and R. Cytron. Efficient demultiplexing of network packets by automatic parsing. In *Proceedings of the Workshop on Compiler Support for Systems Software*, 1996.

[33] S. Johnson. *Yacc — Yet Another Compiler-Compiler*, 1979. UNIX Programmer's Manual.

[34] E. Kohler, M. Kaashoek, and D. Montgomery. A readable TCP in the Prolac protocol language. In *Proceedings of ACM SIGCOMM*, 1999.

[35] P. J. McCann and S. Chandra. PacketTypes: Abstract specification of network protocol messages. In *Proceedings of ACM SIGCOMM*, 2000.

[36] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Symposium*, 1992.

[37] P. Mockapetris. RFC 1035 — Domain names — implementation and specification, November 1987. http://www.faqs.org/rfcs/rfc1035.html.

[38] Microsoft Security Bulletin MS03-026, September 2003. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-026.asp.

[39] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[40] NISCC Vulnerability Advisory DNS - 589088, May 2005. http://www.niscc.gov.uk/niscc/docs/al-20050524-00433.html.

[41] P. Nyblom. The bit syntax — the released version. In *Sixth International Erlang/OTP User Conference*, October 2000.

[42] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. USC: A universal stub compiler. In *Proceedings of ACM SIGCOMM*, 1994.

[43] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *ACM Internet Measurement Conference*, Oct. 2006.

[44] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[45] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, Dec 1999.

[46] T. H. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J.-L. Baer, B. Bershad, and H. Levy. The structure and performance of interpreters. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 31:9, pages 150–159, New York, NY, 1996. ACM Press.

[47] *DCE 1.1: Remote Procedure Call*. http://www.opengroup.org/onlinepubs/9629399/.

[48] http://www.sdl-forum.org/Publications/index.htm.

[49] C. Shannon and D. Moore. The spread of the Witty worm. http://www.caida.org/analysis/security/witty, 2004.

[50] R. Sharpe, E. Warnicke, and U. Lamping. Ethereal. `www.ethereal.com`.

[51] The Open Source Network Intrusion Detection System. http://www.snort.org/.

[52] Apache core features. http://httpd.apache.org/docs/1.3/mod/core.html.

[53] P. H. J. van Eijk, C. A. Vissers, and M. D. (Editors). *The formal description technique LOTOS*.

[54] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, 2004.

[55] T. Ylonen and C. Lonvick. *Internet Draft - SSH Transport Layer Protocol*, March 2005.