Generic Application-Level Protocol Analyzer and its Language

Nikita Borisov
U. C. Berkeley

David J. Brumley
Carnegie Mellon University

Helen J. Wang
Microsoft Research

Chuanxiong Guo
Institute of Communications Engineering, Nanjing

Abstract

Application-level protocol analyzers are important components in tools such as intrusion detection systems, firewalls, and network monitors. Currently, protocol analyzers are written in an ad-hoc fashion using low-level languages such as C, incurring a high development cost and security risks inherent in low-level language programming. Motivated by the large number of application-level protocols and new ones constantly emerging, we have architected and prototyped a *Generic* Application-level Protocol Analyzer (GAPA), consisting of a protocol specification language (GAPAL) and an analysis engine that operates on network streams and traces. GAPA allows rapid creation of protocol analyzers, greatly reducing the development time needed. It uses a syntax similar to that found in existing specification documents and supports both binary and text-based protocols. The GAPA design goals include expressiveness, ease of use, safety, and low overhead; it is intended to operate well in an adversarial environment. Our evaluation demonstrates that our GAPA language is expressive and easy to use for practical protocols, and our GAPA system is scalable and allows online analysis of protocol traffic. We have already found GAPA to be useful in intrusion detection, firewall, and networking monitoring contexts, and we envision additional applications, such as automatic vulnerability signature generation.

I. Introduction

Protocol analysis is the process of (re)constructing the *protocol context* of *communication sessions* from an ongoing network stream or trace. This involves translating a sequence of packets into protocol messages, grouping them into sessions, and modeling state transitions in the protocol state machine. The protocol context extracted by a protocol analyzer refers to a particular traversal of the state machine for a communication session, as shown in Figure 1. Figure 2 illustrates an example of analyzing the RPC-over-HTTP protocol: the protocol analzyer first analyzes the packets into HTTP messages, then further parses the HTTP payload into RPC messages, and finally groups the RPC messages into their respective sessions according to the RPC protocol's session semantics.

Protocol analysis has been widely used in intrusion detection systems such as Snort [39] and Bro [32] and firewalls such as Hogwash [22] and Shield [42]. Protocol analysis is crucial to these systems because precise reconstruction of the protocol context significantly reduces the number of false positives and false negatives. Another existing use of protocol analysis is to label network traffic trace with more protocol semantics, facilitating network monitoring and distributed system debugging. Ethereal [37] employs numerous protocol analyzers for this purpose.

Unfortunately, state-of-the-art practice for creating proto-

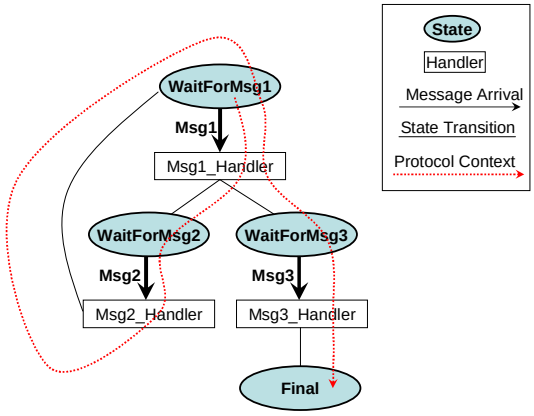

Fig. 1. Tracing Protocol Context

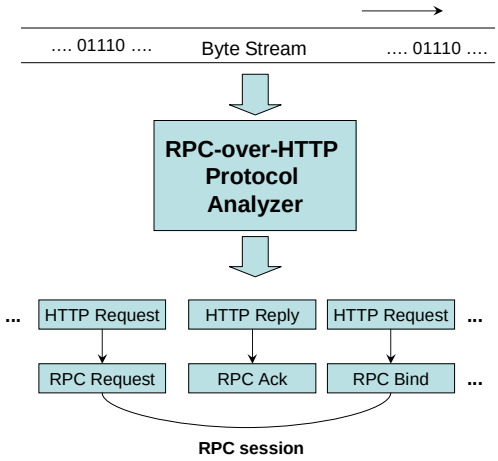

Fig. 2. Protocol Analyzer for RPC-over-HTTP

col analyzers is ad-hoc. To develop a protocol analyzer, the protocol specification must first be obtained or written. Then according to the specification, the protocol analyzer is developed using general-purpose, low-level languages like C, sometimes as a plugin to be statically loaded into the base framework [37]. Such development often requires understanding a large body of source code and involves thousands of lines of low-level language programming effort followed by comprehensive testing. This process must be repeated for each of the protocols to be analyzed. In addition, many protocols have multiple implementations with differences and extensions that need to be explicitly modeled in the analyzer. Given the multitude of the protocols and their variations, applying traditional development techniques simply does not scale.

In this paper, we tackle the problem of rapid development of protocol analyzers for application-level protocols that operate above the transport layer. We target these protocols because they are numerous, with new ones constantly emerging. Hence, scaling the development effort for such protocols is a worthwhile goal.

Our work is motivated by the observation that the task of analyzing various protocols shares a significant number of common functions, such as session dispatching, state machine operations, message parsing, protocol layering, and handling out-of-order or fragmented messages. Our approach is to architect these common and generic protocol elements into one analysis engine as part of a Generic Application-level Protocol Analysis (GAPA) framework. The other part of the framework is the GAPA language (GAPAL), a special-purpose language for describing individual protocols by configuring the common functions and specifying protocol-specific details such as message layout and state machine transitions. The configuration and customization semantics reflect the essential abstractions of protocol analysis. With GAPA, creating a new analyzer involves only specifying the protocol in GAPAL and testing its specification, replacing the much more strenuous process of low-level language development.

Our language, GAPAL, has the following goals and challenges:

- **Flexibility**: We should be able to express and analyze all common protocols in the GAPA framework. This requires us to encapsulate the full set of common protocol functions into GAPA and to have a flexible enough GAPA language for describing both binary and text-based protocols.
- **Ease of use**: GAPAL specifications should be easy to write and to read. This goal is in tension with the previous one: While the flexibility of the language allows us to be more expressive, it makes the language richer and harder to use. Similarly, a low-level language like C can be quite flexible, but difficult to write and read. The challenge here is to strike a balance between ease-of-use and flexibility by having the right set of common functions implemented in GAPA, while leaving enough flexibility to accommodate the distinctiveness of each protocol. A practical guideline we have followed here is to constrain the level of the details of the protocol descriptions in GAPAL to be no greater than that of today's RFC protocol specifications; and to design the syntax of GAPAL to naturally reflect various protocol abstractions for ease of writing.
- **Safety**: We want to reduce the chance of errors in protocol analyzers. Safety errors in an analyzer can cause crashes in the underlying system, or worse, introduce vulnerabilities exploitable by remote traffic. We therefore build type-safety into our language. Further, because GAPAL is special-purpose, we have the luxury to use static checks to ensure that the protocol analysis logic is properly specified, and to guarantee termination of protocol analyzers.
- **Modularity**: Protocols are often built on top of one another, and components from one protocol are reused in others. Modularity is needed to ensure the re-usability and readability of GAPAL specifications.

In addition, our analysis engine, which interprets the GAPAL specifications and performs protocol analysis, has the following goals and challenges:

- **Low performance overhead**: The overhead incurred by GAPA should be small in relation to the applications that implement the protocols being analyzed, in order to support online analysis.
- **Correct operation in an adversarial environment**: We must ensure that GAPA functions well even in the presence of attackers. When GAPA is used for real-time protocol analysis — as part of a firewall, for example — attackers can attempt to launch "state-holding" denial-of-service attacks [32] [33]. Even when GAPA is used for offline purposes such as trace analysis, attackers can respond with large amounts of decoy traffic, delaying detection of their malicious deeds. We must therefore protect online, real-time operations of GAPA by minimizing the amount of state it maintains for protocol analysis. Furthermore, we must also ensure that GAPA's interpretation of the protocol context is consistent with that of the application, even in the face of carefully crafted, malicious traffic [32] [33].

Our work is inspired by Shield [42] which is a vulnerability-driven end-host firewall. Shield analyzes the exploitable application protocols according to a vulnerability signature and detects then blocks the exploits. Shield's authors gave a preliminary and incomplete design of a generic protocol analyzer and a language. Our GAPA language is a new design and our analysis engine addresses numerous issues, such as timeout, exception handling, and pre-existing session handling, that were left out in Shield. GAPA also has numerous applications beyond application-level firewalls.

While we are the first to design and prototype a comprehensive generic application-level protocol analysis language and framework for creating protocol analyzers, many protocol specification languages for various purposes have been proposed in the literature or used in practice, although none is

suitable or specifically designed for the purpose of protocol analysis. Some languages (e.g., PacketTypes [27], ASN.1 [15], NDR [35]) specify only binary (but not text-based) packet formats while some other languages like StateChart [21] and Esterel [5] express state machines, but not data handling. Many languages (e.g., Estelle [7], StateChart [21] Promela++ [3], LOTOS [41], SDL [36], RTAG [1]) have been designed for formal reasoning and verification of protocol interactions, which is orthogonal to protocol analysis. There have also been languages like Prolac [25] proposed for programming the entire logic of a protocol, but they do not provide special-purpose abstractions for protocol analysis.

Another alternative approach is a protocol-analysis framework implemented as a C library; however, we believe it is advantageous to use a special-purpose language for protocol analysis. First, by offering appropriate protocol analysis-specific abstractions, it is easier to program in such a language and the programs are succinct and easy to read. In some other contexts, special-purpose languages offerred three- to four-fold reduction in code size [25], [27]. Second, a special-purpose language gives opportunities for protocol analysis-specific safety checks and optimizations.

In comparison with existing languages, our GAPA language expresses protocol abstractions specific for protocol analysis. GAPAL contains a number of interesting features to meet the special needs of protocol analysis. To make payload format specification easy for binary as well as text-based protocols, we adopt a syntax that is similar to Backus-Nauer Form (BNF) grammar, which has been widely used by many protocol specifications from standard bodies, such as RFCs. We also introduce the ability to direct parsing using computations based on previous fields, giving our language much more expressive power while staying close to the simple BNF syntax. We create a special "visitor" syntax; this allows easy access and manipulation on message components embedded in the grammar. Further, we support protocol analysis-specific safety checks and optimizations in addition to the traditional ones.

To achieve scalability and resiliency to state-holding attacks in the analysis engine, we minimize the memory footprint of the engine with *speculative execution*. Speculative execution allows us to process partially arrived messages and apply the analysis logic to them without having to buffer packets until a complete message arrives. We also carefully manage timeouts, exceptions, and pre-existing sessions, and minimize the chance of incorrect interpretation of the protocol context in GAPA. A particular novel aspect is the use of *outgoing message clocking* to synchronize the current protocol state of GAPA with that of the application.

We have prototyped our GAPA system. In our initial evaluation, we have specified a number of application-level protocols and found that GAPAL is expressive and easy to use, and the GAPA prototype is scalable for online protocol analysis on clients and can be potentially scalable for servers.

GAPA, used together with protocol specifications in GAPAL, provides us with knowledge of the precise protocol context of a communication session, giving us the ability to accurately label network traffic or detect intrusions. GAPA can also be used to "normalize" online communication traffic[1] — that is, ensure some invariants on the traffic that is delivered to the application. For example, GAPA can be used in an application-level firewall, such as Shield [42], to ensure that traffic that exploits vulnerabilities is never delivered to the application.

The detailed protocol knowledge obtained through GAPA has other uses as well. For example, GAPA can potentially enable the *automatic* generation of vulnerability signatures when combined with unknown-attack detection tools, such as TaintCheck [29], Minos [12], Vigilante [11], DynamicCheck [34], or Reactive Immune System [38], and enforce such signatures.

For the rest of the paper, we first give an overview of the GAPA system in Section II. Then, we present our GAPAL language in Section III, and the GAPA analysis engine in Section IV. We present our evaluations in Section V. We describe a number of applications of GAPA in Section VI. In Section VII, we compare and contrast with related work. We address future work in Section VIII and finally conclude in Section IX.

II. GAPA System Overview

The set of common protocol functions carried out in the analysis engine determines the flexibility of the GAPA language and the ease of programming in GAPAL. Naturally, we want to implement protocol-independent functions in the engine; and GAPAL syntax must support programming (or configuring and customizing) some of these common functions. We first briefly illustrate these functions and their respective abstractions that need to be supported by GAPAL; then we give an overview on how these functions make up our GAPA system.

The common functions are:

- *Session dispatching:*
  A session is an abstraction common to most protocols. The session is identified based on either the underlying transport connection (i.e. the source and destination IP address and ports), or based on some session identification in the message. The GAPA engine will need to keep track of active sessions and dispatch messages to the appropriate ones.
- *State machine operations:*
  For each session, GAPA must maintain the current protocol state. The state affects how the input messages are processed; arriving messages will cause transitions to a new state.
- *Message parsing:*
  A protocol analyzer will need to parse messages according to a protocol-specific message format. This parsing needs to be done incrementally, since application-

[1]This is in contrast with other "traffic normalizers" [20][26] that normalize transport protocols.

layer messages can be split among several packets. Correct parsing state must be maintained between packets, otherwise partial messages will be analyzed incorrectly [33][32][20][26].

- *Protocol Layering:*
  Application-level protocols can be layered on one another. For example, RPC can be layered over HTTP. Layering support needed in the analysis engine involves not only piping the payload to the next upper layer, but also maintaining respective session state at each layer.
- *Application-Level Datagrams:*
  Some application-level protocols use UDP as the transport protocol and implement their own datagram fragmentation, reassembly, and reordering. The GAPA engine uses layering to support such datagrams, with a lower layer directing GAPA how to perform reordering and reassembly before delivering an in-order byte stream to the upper layer.
- *Timeout handling:*
  Timeout events are used in many protocols. The analysis engine needs to have timer supports. (Timer support for analyzing network traces uses the timing information in the trace.) Timeout handling is complicated by the inability to stay completely synchronized with the application, hence a timeout event in GAPA may not exactly correspond to the same event in the application. Here, we use *outgoing message clocking*, which eliminates the need to maintain timers in the engine for some protocols, and helps resynchronize with the application in case of a timing mismatch for other protocols. Section IV-E gives more details on this.
- *Exception handling*:
  Protocol messages may be malformed, causing a parsing exception. Other exceptions may include explicit errors signaled by the protocol analysis, or errors in the buffering layers of the engine. Depending on the user policy, the GAPA engine can handle these exceptions by raising alerts, dropping packets and terminating connections, or simply ignoring them. We allow separate handling of each kind of exception.
- *Pre-existing session handling:*
  There may be sessions that have already started before GAPA's protocol analysis takes place. In the analysis engine, we need to handle messages belonging to such sessions carefully: They should not be treated as malformed messages; otherwise, the exception handling may undesirably disrupt the pre-existing sessions. Section IV-F gives more details on this.

Figure 3 shows how a packet (from a live stream or a trace) traverses through our analysis engine and gets analyzed. First, *Spec Dispatcher* uses the process image name that the packet belongs to (if available) and port numbers to locate the proper GAPAL specification (short-handed as *Spec*) from all the compiled and statically checked Specs that were previously loaded into the system. Then, the session identification logic

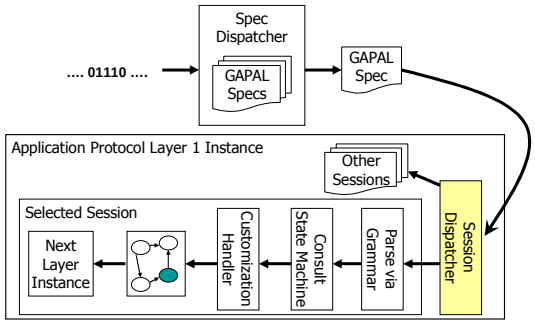

Fig. 3. GAPA System Architecture Overview

Variable type	Lifetime	Visibility
message-local vars	message	entire program
session-local vars	session	after session dispatching
handler-local vars	handler	handler, visitor blocks
local vars	{ } block	{ } block

TABLE I
THE LIFETIME AND SCOPE OF GAPAL VARIABLES.

in the Spec is interpreted to dispatch the packet to the proper session. For that session, the packet is parsed according to some message format specified in the Spec. Based on the current state and the direction of the packet, the corresponding handler is invoked to parse and process the payload. The handler always sets the next state for the session. If protocol layering is involved as specified in the Spec, the packet is further piped to the next upper layer going through the same session dispatching and message processing in that layer.

III. GAPA LANGUAGE

In this section, we present our GAPAL design and how we achieve its goals as described in Section I.

We first walk through the high-level layout of a GAPAL program, as shown in Figure 4. Most abstractions are enclosed by curly braces to support modularity and readability. The particular order in which each abstraction is specified is unimportant.

A `uses` statement indicates the next lower layer protocol from which message payload is piped to the protocol being specified. This allows a programmer to specify each layer in a separate GAPAL file and use layering as a means of composition. The `include` is a standard pre-processor directive that includes the specified file data in-place. A `transport` statement specifies the underlying transport protocols and port numbers used for base layer protocols.

We define several kinds of variables in GAPAL with different lifetimes and scopes. Variables defined before `grammar` section are used to track data across the lifetime of a session, spanning multiple messages. They can be accessed by all code blocks in GAPAL except those that are run before the `session-identifier` has run since the session instance is not yet determined at that point of execution. We do not allow

```
protocol <protoName> {
  uses <lowerLayerName>;
  include <fileName>;
  transport = { ([TCP|UDP]/<Port>)+ };

  // session-local variables
  (<base type> <varName>;)*

  grammar {
    // message-local variables
    (<base type> <varName>;)*

    // message-parsing rules
    NonTerminal[#maxBytes] ->
     [{<code>}]
     ([<name>:]<type>[#maxBytes] [{<code>}])+
     ("|" [alt(<alternation name>)] ... )*;
    ...
  };

  state-machine <name> {
    ((<state>, [IN|OUT|TIMEOUT])
        -> <stateHandlerName>)+
    initial-state = <stateName>;
    final-state = <stateName>;
  };

  session-identifier(<startNonTerminal>){
    <code>
    return <session ID>;
  };

  handler <name>(<startNonTerminal>){
    // handler-scoped variables
    (<base type> <varName>;)*
    (<visitor>)*
    <post-parsing code>
    return <nextState>;
  };
};
```

Fig. 4. The high-level layout of a GAPAL protocol specification; items enclosed in "[]" are optional; "()*" indicates 0 or more of the items enclosed in "()", and "()+" indicates 1 or more of the items.

global variables across different sessions: GAPAL sessions are more flexible than what may be defined in a protocol; for example, a GAPAL session can be defined to represent multiple RPC sessions and the respective state machine models the multiple sessions together; therefore, when a GAPAL programmer feels the need of sharing variables among several sessions, those sessions should be defined as a single session in GAPAL.

Variables defined in `grammar` are *message-local* variables that are used throughout the lifetime of parsing a message, which may be composed of multiple packets. These variables can be accessed by the entire program, including those code blocks that are executed before the `session-identifier`. These variables are re-initialized with each message and are intended to be used to assist parsing.

We also allow local variables inside any block, which have a lifetime of the duration of the block. A special case are handler-local variables, which are defined within a handler and are visible so long as the handler is executed; we will explain these variables in more detail below in Section III-B. Table I shows the lifetime and scope of each variable type.

Base types for the variable declarations are 8 to 64 bit integers (both signed and unsigned), uninterpreted bytes, floats, doubles, strings, and booleans. We also support safe arrays of base types, but we do not let the programmer manually allocate or free dynamic memory.

The rest of the `grammar` section specifies the protocol message formats using BNF-like grammar rules. A non-terminal refers to either a message component or an entire message. A non-terminal roughly corresponds to a C structure or union type name for binary messages, and a BNF non-terminal for text-based messages. The production rule of a non-terminal indicates the make-up of the non-terminal and can include alternation `|`. Programmers can use `alt(<alternation name>` to indicate the name of an alternation which is useful to visitors (to be explained later). The `<type>` could be a base type, a token type, such as a regular expression, or another non-terminal. The type is used for typing the message components for type safety in expressions and statements, as well as for grouping bytes into type instances during parsing. Programmers can add `<symbol>` before `<type>` to refer to a parsed message field when needed. `<maxBytes>` is for programmers to indicate the maximum number of bytes that `symbol:type` can take. Alternatively, programmers can also specify a default upper bound size for all fields (not shown in the figure) and enforce it at run-time — this is to ensure the correct GAPA operations in the face of malicious or malformed, runaway payloads. `<code>`'s embedded throughout a grammar rule contain parsing-related logic. At run-time, the analysis engine uses *recursive descent parsing* to parse an input byte stream according to the grammar. Like typical recursive-descent parsers, we disallow left-recursion. We present our design rationale and further details on `grammar` in Section III-A.

A `state-machine` specifies protocol states and their respective handlers, and indicates the initial and final state of a session. `IN` and `OUT` indicate the direction of an input packet, incoming or outgoing. There is one handler per state per packet direction. We also allow a `TIMEOUT` transition to be specified; timeouts have to be activated dynamically by the handlers (see Section IV-E).

The `session-identifier` parses the packet according to the grammar rule for `<startNonTerminal>` and extracts and returns a session ID. The session ID is used by the analysis engine to dispatch the input packet to the correct session instance. A previously unseen ID creates a new session instance. Each session instance has its own copy of the handlers, session state machine, and session variables. In `session-identifier` section, only message-local variables can be used, but not session-local ones, since session dispatching has not finished running.

A handler carries out the customization logic that program-

mer intended for protocol analysis and always returns the next state for the current session. The next state is returned by the handler rather than being specified in `state-machine` because it may be dependent on both the message content or the protocol analysis logic. We support assignment, conditional statements, common expression operators, and a `foreach` iterator in handlers. The `foreach` iterator can be used to iterate through local safe arrays or grammar array elements of a message. We allow only forward traversal of the arrays, like DPF [16]. Therefore, infinite loops cannot be created with our `foreach`. (Cycles in the grammar are statically detected and disallowed, which we illustrate in Section III-C.) Further, all statements and expressions are statically typed for safety. We also offer a variety of built-in functions. Common functions include byte order conversion routines (ntohs and ntohl, etc.), string routines (strlen, strtol, etc), and others. For layering, a lower-layer protocol can pipe data to the next upper layer via the `send` call. We give further details on visitor usage and scoping in handlers in Section III-B.

A. Message-Parsing Grammar

A key challenge in message parsing is to accommodate both binary and text-based protocol messages. Much of previous work (Section VII) addressed only binary protocols.

Binary messages are typically viewed as a C-like constructed type (e.g., structs, unions, etc.) overlayed on a byte stream. Text-based messages, on the other hand, are often represented by some sort of grammar, often in Backus-Nauer form (BNF) or some variant thereof.

In GAPAL, we are able to use the same parser for binary and text-based messages by observing that they follow a similar recursive structure, hence we use a BNF-like grammar to represent both. Expressing text-based messages in our grammar is very natural — in our experience, much of the grammar can be created by simply copying and pasting the BNF notations from the RFC or similar specification. Expressing binary messages is also straightforward: a structure is represented as a sequence of fields, while nested structures use non-terminals to refer to the component substructures. Unions are implemented using alternation, and we include special support for arrays, specified as `name:<type>[s]` where s may be a constant, a previously defined symbol of type integer, or an executable expression.

Array support represents a departure from the otherwise context-free nature of our grammar specification. We find, however, that they are a common enough idiom in both text-based and binary protocols that we need to support them in GAPAL. It is straightforward to integrate array support into a recursive-descent parser: whenever it encounters `<type>[s]`, it first evaluates s and keeps a counter to parse s copies of `<type>`.

Figure 5 gives a comparison between our RPC (binary) message grammar snippet and its corresponding specification [35]. Figure 6 shows that of HTTP (text) messages. (Some of notations such as “?” and “:=” will be explained shortly.)

```
HTTP_message -> Request | Response;
Response -> ResponseLine HeadersBody;
HeadersBody -> Headers CRLF Body ;
Body ->
      { if (chunked) {
          body := ChunkedBody;
        } else {
          body := NormalBody;
      } body:? ;
NormalBody -> data:byte[content_length]
Headers -> GeneralHeader Headers | ;
GeneralHeader -> name:"[A-Za-z0-9-]+"
              ":" value:"[^\r\n]*" CRLF
      { // == is for string comparison
        if (name == "Content-Length") {
           // convert value to base 10
          content_length = strtol(value,10);
        }
        ...
      };
ChunkedBody -> ...;
```

Fig. 6. GAPAL Code Snippet for HTTP.

1) Code Blocks Embedded in the Grammar: To simplify writing grammar rules, we allow programmers to embed C-like code blocks into the grammar to help direct parsing. This is particularly useful for messages that use a length field to indicate the size of the following data [2]. For example, the length of the body of an HTTP message is specified by the header field `Content-Length`; in Figure 6 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. Both of our GAPAL code snippets in Figure 6 and Figure 5(b) demonstate the usage of the resolve operator. It is possible to rewrite these grammars to avoid the resolve operator and be context free, but the resulting specification is much more awkward.

Code blocks can access message-local variables, `session-vars`, and locally defined temporary variables. When using `session-vars`, it means that message parsing is dependent on the session context.

To ensure type safety, we statically check that resolved symbols are *not* used in expressions, since we would not be able to determine their type.

Although these statement blocks are handy, for modularity and grammar reusability, we advise programmers to only include parsing-related logic in them and exclude protocol analysis functions that best belong in handlers.

[2] A size used to direct parsing of future data is called a synthesized attribute in automata theory. For this reason grammars such as ours are sometimes called attribute grammars.

```
/* bind header */
typedef struct {
  /* common header */
  u_int8 rpc_vers = 5;
  u_int8 rpc_vers_minor;
  /* bind iff PTYPE == 11 */
  u_int8 PTYPE;
  u_int8 pfc_flags;
  byte   packed_drep[4];
  u_int16 frag_length;
  u_int16 auth_length;
  u_int32  call_id;
  /* end common fields */

  u_int16 max_xmit_frag;
  u_int16 max_recv_frag;
  u_int32 assoc_group_id;

  /* var-size presentation context list */
  p_cont_list_t p_context_elem;
  /* optional authentication verifier */
  /* following field present iff auth_length!=0 */
  auth_verifier_co_t auth_verifier;
} rpcconn_bind_hdr_t;


typedef struct {
  u_int8          n_context_elem;
  u_int8          reserved;
  u_short         reserved2;
  p_cont_elem_t [size_is(n_cont_elem)] p_cont_elem[];
} p_cont_list_t;


typedef struct { ... } p_cont_elem_t;

typedef struct { ... } auth_verifier_co_t;
```

(a) The RPC BIND message from the OpenGroup specification [35].

```
int m_auth_len;

commonRPCHeaders ->
        rpc_vers:uint8
        rpc_vers_minor:uint8
        PTYPE:uint8
        pfc_flags:uint8
        packed_drep:byte[4]
        frag_length:uint16
        auth_lengh:uint16  {
          m_auth_len = auth_lengh;
        }
        call_id:uint32;

rpcconn_bind_hdr_t ->
        commonRPCHeaders
        max_xmit_frag:uint16
        max_recv_frag:uint16
        assoc_group_id:uint32
        p_context_elem:p_cont_list_t {
          if (m_auth_len !=0)
            auth_verifier := auth_verifier_co_t;
          else
            auth_verifier:= emptyRule;
        }
        auth_verifier:?;

emptyRule -> ;

p_cont_list_t ->
        n_context_elem:uint8
        reserved:uint8
        reserved2:uint16
        p_cont_elem:p_cont_elem_t[n_context_elem];

auth_verifier_co_t -> ... ;
p_cont_elem_t -> ... ;
```

(b) RPC BIND message in GAPAL

Fig. 5. The RPC (over TCP) BIND message layout

B. Visitors in the Handlers

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. The dot notation essentially requires duplicating the parsing logic present in the grammar, and can be tedious and error-prone.

We eliminate re-parsing with a much cleaner and clearer syntax by allowing the programmer to write grammar visitors [18]. A visitor is a block of code that is executed each time a rule is visited. The syntax for a visitor is:

```
@<non-terminal>(._<alternation name>)?
      -> { ... <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. Symbol names in the production of the non-terminal (or the alternation) can be accessed locally by the visitor. 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 protocol analysis so that the same parsing logic can be re-used for different protocol analyses. Essentially, the visitors in a handler represent the handler's customization of message parsing for the purpose of protocol analysis. Consequently, visitors are always executed before the rest of the handler code.

Visitors can declare local variables, which exist only for the duration of each code block. For longer-lived variables, visitors can refer to *handler-local* variables. Handler-local variables have a lifetime corresponding to the handler execution; they are initialized before any visitors are called and destroyed once the handler exits.

The example below counts the number of headers in an HTTP request for the grammar given in Figure 6:

```
handler hndl(HTTP_Message){
  int8 hdrcnt = 0;
  @GeneralHeader → { hdrcnt++; };
  print(''Total number of headers: %v\n'', hdrcnt);
}
```

The grammar rule symbols for non-terminal `GeneralHeader` are locally available to the visitor. `hdrcnt` is a handler-local variable that is initialized to zero every time the handler is called. Every time `GeneralHeader` is traversed during parsing, `hdrcnt` is incremented. When the entire message is parsed, the total number of headers is printed.

C. Safety Checks and Optimizations

GAPAL is type-safe. In addition, we also check for:

- *Dynamic safety:* We perform array bounds checking at runtime to avoid memory errors. In addition, we perform checks that are missing even from some memory-safe languages like Java [24] and OCaml [30], such as checking for integer math overflow and division by zero to eliminate logic errors caused by these so that GAPAL programs are more robust to run in the analysis engine.
- *Unreachable grammar rules:* A GAPAL author may inadvertently create unreachable grammar productions. A non-terminal is unreachable if it can never be reached during parsing. We warn the author on such productions.
- *Correct state machine use:* We perform control-flow analysis to make sure that every handler returns a state. We also make sure that the returned state is defined in the state machine abstraction. Furthermore, we alert programmers of the unreachable states.
- *Resolve safety:* Resolve statements (Section III-A) are naturally proceeded by a conditional statement, i.e., if some condition occurs, resolve the grammar symbol to some non-terminal. We check to make sure that symbols are properly resolved along all control paths. Also, we make sure that resolved symbols are not used in expressions, since we cannot statically determine their type.
- `session-identifier` *safety:* We ensure that only message-local variables are accessed in `session-identifier` section. Further, some of the code fragments that are executed during parsing may reference `session-vars`, and will therefore cause an error if they are executed before the `session-identifier` (and hence the session dispatching) has run. To prevent such errors at run-time, we perform a static analysis of which code blocks will be executed before the session dispatcher is complete and flag error if they refer to `session-vars`.
- *Ensuring termination:* We want to ensure that parsing will always terminate. We can find cycles in the grammar using standard techniques to find left recursion. The code blocks are also guaranteed to terminate because they cannot include infinite loops. However, with the resolve feature, it is possible to create a parsing cycle by combining grammar rules and code blocks. For example,

  ```
  G -> { s := G; } s:?
  ```

  creates a parsing cycle. We detect this by checking the code blocks and obtaining a list of all possible types that a symbol may resolve to. We then apply a left-recursion check to all of those types.

IV. The Analysis Engine

The analysis engine operates either in *normalizing mode*, forwarding potentially modified traffic to the application based on the handler actions, or *analysis mode*, used for monitoring traffic or processing network logs when no modifications to the network data are necessary.

Message parsing is used throughout the protocol analysis process in the engine. From GAPAL Spec dispatching (Section II), the analysis engine finds the respective GAPAL Spec for the current packet to be analyzed. The engine then follows the grammar that specifies the message format, performs recursive descent parsing and generates a parse tree. Since the engine may receive packets containing incomplete messages, it performs parsing incrementally, saving parsing state between packets.

The engine also executes code fragments during parsing, both those embedded in the message grammar and those resulting from the visitor pattern in the handlers (Section III-B). The resulting parse tree contains the components of the message parsed out of the packet. Later on, handlers use the parse tree, as well as other session state maintained by the code fragments, to carry out further analysis.

For the rest of section, we present other interesting techniques that we used in designing the engine.

A. Buffering

We want the memory footprint of the engine to be as small as possible, both to improve performance and to avoid state-holding attacks. The engine must buffer parts of a parse tree that will later be referred to by the handlers, incompletely parsed fields of a message, and packets that are about to be normalized.

Statically, we compute which parts of the parse tree are referenced by handlers and automatically discard the unreferenced parts of the parse tree. At parse time, we keep track of the parse tree variables that the handlers will reference in the future, as provided by the handler interpreter. As soon as we know that a variable will no longer be needed, we free the corresponding memory. This, combined with speculative handler execution (described in the next section), greatly reduces the parse tree buffering requirements.

The only kind of normalization that is currently supported is dropping a packet or a session based on an error. Therefore, we have optimized the memory management in the GAPA engine for this case. We buffer a packet in the engine only as long as its fields are being parsed by the grammar parser, and release it to the application immediately afterwards. This may cause potentially malicious packets to the application, however, these packets will be incomplete. The security assumption we are making is that if the GAPA engine does not have enough of the message to make a decision about whether it is malicious or not, the partial content is not yet dangerous enough to cause errors in the application. This assumption is made real by the

speculative execution mechanisms described below; however, it requires care on the part of the GAPAL programmer.

As described in Section III, message components have a maximum size by default or GAPAL programmers can explicitly specify the maximum size through `maxBytes` (Figure 4). For example, the URL field of HTTP might be specified to not exceed 2000 bytes. This gives an upper limit on the buffering needed for message components. When length of a field exceeds the limit at runtime, an exception will be triggered in the analysis engine.

B. Speculative Execution

We use a special technique to execute handlers as early as possible in the parsing process. In the firewall and intrusion detection scenario, this allows us to detect attacks earlier and to avoid buffering incomplete messages. It also helps us optimize parsing once it is clear that no attack exists.

The engine begins executing handlers as soon as it has parsed a single packet, even if the message is incomplete. We speculatively execute the handler until it references a component of a message that has not yet been processed. At that point, we save a continuation for the rest of the handler until the next packet. If the next packet includes the referenced component, we resume execution of the handler and continue until either another unreferenced component is encountered or the handler returns. If the handler is finished, we can switch to a light parsing mode for the rest of the message, skipping parsing any fields whose length can be determined in advance.

Early execution works well only if the order in which fields are referenced matches the order they are parsed. Otherwise, a lookup for a field that comes late in the message will delay the execution of the handler. In normalizing mode, this may result in a security hole, since a partial message may be passed on to the application even though later checks, blocked by some field late in the message, will flag it as malicious. However, it is possible to warn programmers of such scenarios at the compile time. In simple cases, where there are no data- or control-flow dependencies between statements, it may be possible to reorder code statements to avoid this problem. For complex cases with such dependencies, programmers should be conservative and block the early arriving field more aggressively, which may cause false positives, but no false negatives will result.

Early execution is also used during session dispatching. When a message is received, we first execute the `session-identifier` logic, followed by the appropriate handler. The transition between the dispatcher and the handler happens automatically at the point when enough of the message has been parsed for the dispatcher to complete.

C. Layering

An important component of the analysis engine is layering. Layers are implemented as essentially separate instances of the GAPA engine (Figure 3). When a lower layer sends data to an upper layer, it is treated as an incoming packet in the upper layer GAPA instance and parsed, dispatched, and processed the same way as regular communications. In particular, all the buffering and speculative execution mechanisms operate the same way, defending from malicious traffic and avoiding state-holding attacks.

In addition to supporting several protocols layered on top of each other, layering can be used to separate a complex protocol into component layers. In fact, we use layering to implement application-level fragmentation and datagram reordering. The lower layer parses the fragment headers and uses a special version of the `send` call — `sendFragment` — to indicate to the engine where in the fragment sequence the current datagram appears. The engine then performs fragment reassembly before passing the data to the upper layer, which parses the reassembled data into meaningful message components.

When GAPA is used in normalizing mode and it lies on the forwarding path, the packets must be forwarded at the lowest layer of the engine. However, upper layers may still signal errors to the engine, which will cause the underlying packet to be dropped. Speculative execution at the lower layers will cause as much data to be passed to the upper layers as possible, while speculative execution at the upper layers will cause analysis to proceed as far forward as possible. Once again, we rely on the assumption that if the GAPAL specifications do not have enough data to make a decision about the packet, the packet cannot harm the upper layer of the application.

D. Exception Handling

During message parsing, an exception might occur. This may be due to a malformed message or because of an exception in a handler. Another cause is a grammar field being too long: GAPA allows GAPAL authors to specify a maximum length for fields to avoid buffering large amounts of data (see Section III); this is particularly useful when GAPA is used in adversarial conditions.

The exception can be handled in several ways: by alerting the user of the error, by dropping the packet or terminating the communication session (in normalizing mode), or by simply ignoring it. Exceptions raised in the handlers are never ignored, however, parsing exceptions are dealt with according to a policy decision. This decision is made by the user of GAPA independent of the GAPAL specification, depending on whether it is more important to be conservative in the analysis or to avoid disruption. If it is important to be minimally disruptive, GAPA can be told to pass any packets it does not understand to the application, optimistically assuming that they will not cause any harm. However, the more conservative approach is to reject all such packets, potentially disrupting the application if there is an error in the grammar. Trace-based testing can help prevent such errors.

In either case, the session transitions to an error state after an exception. All future packets for the session will also cause an exception and will be handled according to the same policy, effectively terminating the analysis of the session. In the future, we plan to investigate using outgoing message clocking (described in the next section) to potentially resynchronize with the application after a parsing error by watching its response. (Of course, this is impossible if the

erroneous packets are discarded, since no response will ever be sent.)

E. Timeouts

Protocol state machines often have timeout events for retries or for session state cleanup, in case of remote host or connectivity failures. However, maintaining timing in GAPA is tricky because a timeout in GAPA may *not* correspond a timeout in the application and vice versa. Such inconsistencies can lead to incorrect analysis.

To address this challenge, we use *outgoing message clocking* to minimize the timer usage in GAPA and to synchronize GAPA's current protocol state with that of the application (when necessary). The intuition here is that an outgoing message (of a firewalled host or a monitored entity in a trace) or a sequence of outgoing messages reveals the current protocol state that the application is in. Please note that we cannot trust incoming messages in an adversarial environment, since incoming messages could be from an attacker. Furthermore, we are assuming here that the machine that the application is running on is not compromised — otherwise, the correct operations of GAPA cannot be guaranteed in the first place.

In timeout handling, there are two kinds of timeouts: The first kind triggers a network event, such as a retry message or a socket-closing event — they are observable from the network and consequently, by GAPA. The other kind of timeouts is "network-silent", such as session-cleanup kinds of timeouts.

For network-observable timeouts, we apply outgoing message clocking to eliminate the need to maintain timing in GAPA — instead of maintaining a timer and transitioning to a new state upon a timeout, the state transition is triggered by observable network events.

For network-silent timeouts, GAPA has no choice but to maintain a timer. In GAPA, a handler can set a timeout using the `timeout`(*time*) built-in function; if no state transition occurs when specified time has elapsed, a timeout handler is called. The time can be determined dynamically based on protocol context. To cope with timing inconsistencies between the GAPA and the application, one solution (for the normalizing mode operation) is to enforce a timeout event in GAPA with a conservatively early timeout: Even when a waited-for message arrives before the application times out, GAPA can discard the message and force timeout in the application. This solution is undesirable in that it changes application behavior. Furthermore, it is only applicable to normalizing mode, but not the analysis mode where such enforcement is not possible. Another solution is that GAPA maintains a conservatively long timeout: GAPA may enter an inconsistent state with the late-arriving, waited-for message, but it may be harmless to the application since the application would treat the message as an exception. However, if we are guarding a bug in the post-timeout exception handling code of the application, this inconsistency will be a problem.

In our solution, we design our analysis engine to be acknowledging about its ambiguity on the current protocol state: We create a *MaybeTimeout* state in the protocol, and transition to it conservatively before the application timeout. From this point on, we once again apply *outgoing message clocking* to infer and synchronize with the current protocol state in the application. In more detail, a message received in the *MaybeTimeout* state that would normally transition to state *B* would cause a transition to the state *TimeoutOrB*, indicating that there is still an ambiguity about the application state. The response from the application should resolve this ambiguity and let GAPA transition to the correct state. If no traffic is received, the protocol will transition from the *MaybeTimeout* state to a *Timeout* state after a large enough wait to account for a margin of synchronization error. If the GAPA is running on the same machine as the application, this margin of error can be quite small; in a network setting it may need to be as large as seconds. Figure 7 and Figure 8 shows the state machines maintained in the application and the analysis engine, respectively, for this example. In the general case, it may take more than one message to resolve the ambiguity in the application state and more ambiguous states will need to be introduced.

Fig. 7. Application state machine

Fig. 8. GAPA maintained state machine

GAPAL programmers can be agnostic of the timeout handling in the engine and specify the protocol state machine just as it is described in protocol specification documents using timeout events for both network-observable and network-silent timeouts. Then GAPA carries out the state machine transformation at the compile time.

F. Pre-Existing Sessions

From time to time, the GAPA policies of a running analyzer will need to be upgraded. We want to do this with minimal disturbance to the analysis and the application. We therefore use the old policy to process existing connections, and only apply the new policies to newly formed sessions. However, this becomes complicated if the new policy uses a different user-defined session identification function (Figure 4) than the old one. In this case, we perform a two-stage session identification process: first, the old session identification is used to see if the message corresponds to an existing session according to the old policy. If no such session exists, the new session identification function is run to dispatch the message according to the new policy. The two-stage identification process remains in effect as long as any of sessions using the old policy are active.

There can also be pre-existing sessions before GAPA is installed and runs on a system. As a compromise between security and disturbing these pre-existing sessions, we adopt a grace period when GAPA first starts. During the grace period, messages belonging to unknown sessions to GAPA are not treated as exception, but are parsed, observed and used to infer the current protocol state of the application. Again, we apply outgoing message clocking here to use a sequence of outgoing messages to infer the correct state of a pre-existing session. Certainly, in a normalizing mode, such a grace period gives opportunities to attacks; such vulnerable time windows can be eliminated by restarting the application.

V. Evaluation

We have prototyped the GAPA framework in C++. We use Lex and Yacc specifications for the syntax. All together, there is a little over 15000 lines of code, not including comments. About 56% of the code is for interpreting the type-safe language, 18% for grammar parsing, 9% for session and state management, and 17% for the language syntax files (Yacc and Lex files) and others.

Next, we present our evaluation results on the expressiveness of our language and the performance of our prototype.

A. Experience with GAPAL

We have specified a number of protocols using GAPAL: HTTP [17], RPC [35], SIP [19], DNS [28], BitTorrent [6], and TLS [14]. This represents a diverse collection, including text and binary protocols, both stream- and datagram-oriented. In all cases, we have found the specification process to be straightforward, as we can start with a BNF specification and then annotate it with additional parsing and protocol logic. We were able to specify most protocols within a few hours. The most difficult task was resolving some unclear parts of a protocol specification; this task cannot be helped by tools, and would be easier for someone with detailed protocol knowledge.

Table II summarizes our specifications. The “GAPAL LoC” column shows the number of uncommented lines of code in GAPAL specification. The “Session” column indicates whether a protocol uses an implicit session identification with IP addresses and ports or an explicit one with session ID embedded in the messages. The column of “Layering” indicates whether layering mechanism has been used for protocol layering, fragmentation, or out-of-order datagram handling.

The complexity of our specifications roughly corresponds to the complexity of the protocol definition in the RFCs that we used, as most lines are copied from the BNF-like specifications in the RFCs. For comparison, we studied the protocol analyzers included in Ethereal [37]. We found that Ethereal used about an order of magnitude more lines of code than GAPAL. The comparison is not entirely accurate, as a lot of the Ethereal code is dedicated to pretty-printing the protocol headers, but we expect that even without this functionality there would be a large difference between both the code size and the development effort involved in Ethereal and GAPAL.

For most of the protocols, we implemented a simple labeling analysis to print out the values of some relevant fields. We have found that the visitor syntax is very useful in this task, since we can directly reference fields in the grammar as they are parsed, rather than manually extracting them from the parse tree.

We also implemented a more complicated analysis to detect the CodeRed worm [8]. This analysis uses a layered composition of the HTTP protocol and a CodeRed-specific URL parser. The HTTP protocol identifies the URLs in the HTTP requests and passes them on to the URL parser using the layering mechanism. The URL parser looks for URLs interpreted by the IDA ISAPI filter and breaks them into their constituent components. The analysis handler then checks whether the buffer parameter exceeds a certain length (causing a buffer overflow), and raises an alert when it does.

The CodeRed URL parser specification is only 25 lines of GAPAL code. We have tested it with a CodeRed infection packet and it successfully detected the worm; we also tested the parser on a 1 GB web trace and did not detect any false positives. Note that we are able to use detailed protocol knowledge to specify the exact vulnerability, rather than a simple web signature. Our CodeRed analysis would have detected CodeRed II and other potential variants of the worm, including polymorphic variants that use URL escaping, as the HTTP protocol analyzer removes the escapes before passing them to the URL parser.

B. GAPA Performance

To evaluate the GAPA performance, we collected a web trace in front of a busy web server[3]. The trace contains 48,755 packets, not counting re-transmitted packets. Then, we ran our GAPA prototype with a HTTP Version 1.1 GAPAL Spec to analyze the trace on a 3.06 GHz CPU with 1 GB RAM running Windows XP. We first evaluate the parsing efficicency, using a HTTP GAPAL Spec that contains only parsing logic, and no analysis logic. GAPA parses the HTTP messages at a rate of

[3]The name of the server is not disclosed for the purpose of double-blind review.

Protocol	# of States	GAPAL LoC	Session	Layering
HTTP	3	55	Implicit	Fragmentation
RPC/TCP	9	122	Explicit	Fragmentation
RPC/UDP	6	72	Explicit	Fragmentation & out-of-order
SIP	6	276	Explicit	SDP
DNS	4	60	Explicit	Out of order (with UDP)
BitTorrent	3	38	Implicit	No
TLS	5	46	Implicit	No
SSH	4	39	Implicit	No
DHCP	2	14	Explicit	No

TABLE II
EXPERIENCE WITH GAPAL.

3340 packets per second, and a bit rate of 11.7 Mbps. Then, we further measured the impact of additional protocol analysis logic, using our CodeRed detector discussed above for the measurement. We obtained a rate of 3020 packets per second, and 10.5 Mbps.

Using profiling, we determined that the largest components contributing to the GAPA overhead were regular expression matching during the parsing, and execution of the interpreted language statements. For regular expressions, we plan to use advanced pattern matching techniques optimized for matching several regular expressions at once [13]. To speed up interpreted language execution, we plan to investigate compiling it to machine code.

VI. APPLICATIONS OF GAPA

GAPA can serve as an effective core mechnanism for a number of interesting applications which we sketch below:

- *Intelligent network trace labeling:*
  GAPA can enable rapid development of new protocol analyzers for network monitoring tools like Ethereal [37], allowing network trace to be intelligently labeled with the right amount of application level protocol semantics at GAPAL programmer's discretion.
  Further, strong typing and safety checks in GAPAL eliminate many potential software defects such as those discovered for `tcpdump` and Ethereal. Since year 2000, 15 different vulnerabilities have been identified in tcpdump[10] and about 45 in Ethereal [9]. Some of these vulnerabilities are buffer overruns due to C safety issues (caused by integer overflow, for example), and some are denial-of-service attacks caused by specially crafted packets triggering infinite loops. Such vulnerabilities are impossible in GAPAL specifications. (It is possible for the GAPA engine to have buffer overruns and infinite loops, but the engine core can be more thoroughly tested than the constantly evolving body of protocol-specific analyzers.)
- *Easy authoring of vulnerability signatures for known vulnerabilities:*
  Recent work Shield [42] uses vulnerability signatures to block known-vulnerability attacks on an endhost. A vulnerability signature of an application can be represented and recognized by a protocol analyzer. The signature encodes all possible sequences of the protocol messages that lead to the protocol state prior any potential exploitations (i.e., the protocol context), along with message parsing instructions for exploit detections. Without precise protocol specifications, authoring such vulnerability signatures can be difficult. Further, an application may have multiple vulnerabilities. Processing these vulnerability signatures in turn is inefficient; and merging them into one signature is non-trivial.
  With GAPA, authoring vulnerability signatures can be made easier. We believe it will be beneficial to maintain a complete and well-tested GAPAL specification of an application-level protocol, which can be evolved along with the application changes. (Such a specification can be useful for testing and debugging the application, as well as vulnerability signatures.) From that point on, authoring individual vulnerability signatures based on that protocol is reduced to annotating the existing specification with vulnerability-specific checks. For example, checking for buffer overrun of a protocol message component requires just adding a couple lines in a handler, checking the length of the component and reacting to buffer overruns when they happen. Merging vulnerability signatures also becomes trivial — it is a matter of customizing state handlers, adding new visitors to inspect some message component, for example. Such fast vulnerability signature authoring can close the critical time window between vulnerability disclosure and protection even further.
- *Automatic vulnerability signature generation for newly discovered vulnerabilities:*
  Much research work designs tools for detecting "zero-day" exploits, and hence discovering the associated new vulnerabilities. TaintCheck [29], Minos [12], Vigilante [11], DynamicCheck [34], and Reactive Immune System [38] are recently proposed and prototyped tools for detecting and tracing "tainted" control flows caused by external data or buffer overruns at run-time. While we know exactly where in the binary code a control flow

violation or buffer overrun occurs, this information is not sufficient to author a vulnerability signature, since the protocol context information is missing. Directly extracting protocol semantics (i.e., the protocol state machine and message formats) from either the binary code or source code is very difficult.

GAPA can be used to generate accurate vulnerability signatures. When the attack detection tools identify which packet caused the run-time violation, GAPA can reconstruct the protocol context and identify the protocol states and message components involved in causing the vulnerability. This can be done either by running GAPA in parallel with the attack detection tools and signaling it whenever an error occurs, or by having the tools simply keep a log of all packet sequences (for active sessions) and passing the sequence causing the violation to GAPA as a trace. Because of the protocol context reconstructed by GAPA, the generated signature will be more precise; they will cause fewer false positives and are more likely to catch polymorphic worms.

VII. Related Work

There is much literature on intrusion detection [32], [39] and firewalls [22]. However, none has addressed the rapid development of protocol analyzers with a *generic* protocol analysis framework. Packet filters [4] are programmable selection criteria for classifying or selecting packets from a packet stream in a generic and reusable fashion; but they are not meant to analyze protocol context, which requires interpreting packets into messages and messages into sessions. In Section I, we gave an overview of how our work compares and contrasts with other protocol description languages. In this section, we address each related work in turn and provide detailed comparisons.

Shield's [42] generic protocol analysis inspired us to design and develop a full-fledged GAPA for purposes beyond just shielding. In fact, Shield's design appears to be preliminary. The Shield language was mostly suitable for binary protocols such as RPC [35], but would be difficult to express text-based protocols such as HTTP [17]. 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. While the idea was novel, converting an existing protocol specification document to one that is expressed in Shield language becomes a difficult task. In contrast, the GAPAL design takes a more disciplined approach – instead of structuring a binary or text-based data stream rigidly, GAPAL uses BNF-like attribute grammars which are easy for both text and binary messages. Shield also did not address a number of issues in its analysis engine design such as exception, timeout, or pre-existing session handlings.

PacketTypes [27] is a packet specification language that automatically generate packet recognizers in C and type protocol messages into the constituent fields. PacketTypes was designed primarily for binary protocols at layer 3 or 4 (e.g., PacketType was used to implement a parser for Q.931 [40], an ISDN layer 3 protocol). Unlike GAPAL, text-based protocol messages and higher layer protocols are hard to express with PacketTypes.

Prolac [25] is a statically-typed, object-oriented language for network protocol implementation. Prolac provides a compiler for creating C code from a Prolac specification. The design of Prolac was driven by making the TCP implementation readable and extensible with good performance. While Prolac can be used to implement any network protocols, GAPAL is more special-purpose with explicit, built-in support for abstractions needed for protocol analysis. These abstractions would need to be manually implemented for each protocol implementation in Prolac.

StateCharts [21] and Esterel [5] are both languages for programming reactive systems (e.g., real-time systems, control systems, hardware design, distributed systems, communication protocols). Esterel also includes a compiler that translates Esterel programs into finite-state machines. While the protocol state machine specification part of our language corresponds to the control handling aspect of Esterel and StateChart, they offers minimal data handling support and other protocol analysis abstractions.

The *x*-Kernel [23] provides an explicit architecture for constructing and composing network protocols. Although the infrastructure is written in C, it enforces a minimal object-oriented style on protocol and session objects. The essential abstractions supported are protocol, session, and message objects along with a set of support routines for buffer management, identifier mapping, and timer support. Through this uniform interface among protocols, *x*-Kernel aims to improve the structure and performance of protocol layering. In comparison to *x*-Kernel [23], we provide finer-grained and more explicit protocol abstractions and eliminate redundant implementations across different protocols: For example, for the protocol object, we additionally support the abstraction of session dispatching; for the session object, we additionally support a state machine automaton that a session is supposed to follow. While a C-like modular design is possible for protocol analysis, we believe it is too low-level and unsafe. In contrast, our language GAPAL supports strong typing, and because it is special-purpose, we can easily carry out static checking at compile time for specific properties related to protocol analysis (Section III-C).

ASN.1 [15], NDR [35], and USC [31] are used to specify binary message formats, but not the protocol logic. These languages can be considered as a subset of GAPAL. We have built translators from NDR and ASN.1 into GAPAL.

Formal Description Languages [2] (FDL) such as Estelle [7], Promela++ [3], LOTOS [41], RTAG [1], and SDL [36] are designed to analyze and to reason and verify the properties of a particular protocol state machine design. This is orthogonal to our goal of a framework for parsing and analyzing protocol messages and reassembling messages into sessions.

In summary, in the arena of protocol analysis, we are the first to provide a comprehensive generic application protocol analysis framework; comparing with related work in protocol description languages, no existing languages offers sufficient

abstractions needed for analyzing both binary and text-based protocols.

VIII. Future Work

Our GAPA analysis engine currently interprets GAPAL programs. For higher speed, it may be necessary to compile them to C or machine code. This would involve translating both the grammar rules and the executable statement blocks; the key challenge is supporting incremental parsing and speculative execution in the translated code.

Our GAPA language could benefit from better support for modularity. We anticipate an inheritance model for refining a protocol specification with new `session-vars`, augmented handlers, and possibly an extended state machine. Such a model would better support designing various analyses on top of a base protocol specification. We are also gaining experience programming with GAPAL and finetuning its design.

GAPAL programming can be further made easier by having a "protocol analyzer development kit" (PADK), similar to some of the GUI design tools, where states and messages can be drawn on a canvas, from which PADK would generate code template. Debugging facilities would also facilitate GAPAL development.

IX. Concluding Remarks

In this paper, we have presented the design, implementation, and evaluation of a comprehensive generic application-level protocol analysis framework, GAPA. Our key contributions include the design of a novel protocol analysis language that is safe, easy-to-use, and expressive, and a number of techniques that we use for our run-time engine, such as speculative execution, outgoing message clocking for timeout handling and pre-existing session handling. GAPA is of great utility for a number of applications: intrusion detection, firewalling, intelligent network trace labeling, and vulnerability signature authoring and generation. Our evaluation indicates that our GAPAL is expressive and easy to use and our GAPA system prototype is capable of doing online analysis for clients and has potential for servers as well.

X. Acknowledgements

Andrew Begel, John Douceur, John Dunagan, Jon Howell, and Dawn Song gave us invaluable critiques on the drafts of our paper. Our work also benefitted 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. We are thankful to everyone's help.

References

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

[2] Fulvio Babich and Lia Deotto. Formal methods for specification and analysis of communication protocols. *IEEE Communications Surveys and Tutorials*, December 2002.

[3] Anindya Basu, Mark Hayden, Greg Morrisett, and Thorsten von Eicken. A language-based approach to protocol construction. In *Proceedings of the ACM SIGPLAN workshop on domain specific languages*, 1997.

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

[5] Gerard Berry. *The Esterel Primer*.

[6] BitTorrent. http://bittorrent.com/.

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

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

[9] Mitre corporation. Common vulnerabilities and exposures database (CVE), keyword search "ethereal". `www.cve.mitre.org`.

[10] Mitre corporation. Common vulnerabilities and exposures database (CVE), keyword search "tcpdump". `www.cve.mitre.org`.

[11] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms? In *HotNets III*, November 2004.

[12] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of 37th International Symposium on Microarchitecture*, October 2004.

[13] Neil Desai. Increasing Performance in High Speed NIDS.

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

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

[16] Dawson R. Engler and M. Frans Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM*, 1996.

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

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[19] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. Rfc 2543 - sip: Session initiation protocol, March 1999.

[20] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of USENIX Security Symposium*, August 2001.

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

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

[23] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 1991.

[24] Java. http://java.sun.com/.

[25] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable tcp in the prolac protocol language. In *Proceedings of ACM SIGCOMM*, 1999.

[26] G. Robert Malan, David Watson, and Farnam Jahanian. Transport and application protocol scrubbing. In *Proceedings of IEEE Infocom*, 2000.

[27] P. J. McCann and S. Chandra. PacketTypes: Abstract Specification of Network Protocol Messages. In *Proceedings of ACM SIGCOMM*, 2000.

[28] P. Mockapetris. Rfc 1035 - domain names - implementation and specification, November 1987. http://www.faqs.org/rfcs/rfc1035.html.

[29] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[30] O'caml. http://www.ocaml.org/.

[31] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. USC: A Universal Stub Compiler. In *Proceedings of ACM SIGCOMM*, 1994.

[32] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, Dec 1999.

[33] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection, January 1998. http://www.insecure.org/stf/secnet_ids/secnet_ids.html.

[34] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of ACSAC*, 2004.

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

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

[37] Richard Sharpe, Ed Warnicke, and Ulf Lamping. Ethereal. `www.ethereal.com`.

[38] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

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

[40] International Telecommunications Union. *Recommendation Q.931 - ISDN user-network interface layer 3 specification for basic call control*, may 1998.

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

[42] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, 2004.