High-speed Matching of Vulnerability Signatures

Nabil Schear*, David R. Albrecht†, Nikita Borisov†

*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Illinois at Urbana–Champaign
{nschear2,dalbrech,nikita}@uiuc.edu

Abstract. Vulnerability signatures offer better precision and flexibility than exploit signatures when detecting network attacks. We show that it is possible to detect vulnerability signatures in high-performance network intrusion detection systems, by developing a matching architecture that is specialized to the task of vulnerability signatures. Our architecture is based upon: i) the use of high-speed pattern matchers, together with control logic, instead of recursive parsing, ii) the limited nature and careful management of implicit state, and iii) the ability to avoid parsing large fragments of the message not relevant to a vulnerability.
We have built a prototype implementation of our architecture and vulnerability specification language, called VESPA, capable of detecting vulnerabilities in both text and binary protocols. We show that, compared to full protocol parsing, we can achieve 3x or better speedup, and thus detect vulnerabilities in most protocols at a speed of 1 Gbps or more. Our architecture is also well-adapted to being integrated with network processors or other special-purpose hardware. We show that for text protocols, pattern matching dominates our workload and great performance improvements can result from hardware acceleration.

1 Introduction

Detecting and preventing attacks is a critical aspect of network security. The dominant paradigm in network intrusion detection systems (NIDS) has been the *exploit signature*, which recognizes a particular pattern of misuse (an *exploit*). An alternative approach is to use a *vulnerability signature*, which describes the *class* of messages that trigger a vulnerability on the end system, based on the behavior of the application. Vulnerability signatures are exploit-generic, as they focus on how the end host interprets the message, rather than how the particular exploit works, and thus can recognize polymorphic and copycat exploits.

Exploit signatures are represented using byte-string patterns or regular expressions. Vulnerability signatures, on the other hand, usually employ protocol parsing to recover the semantic content of the communication and then decide whether it triggers a vulnerability. The semantic modeling allows vulnerability signatures to be both more general and more precise than exploit signatures. However, this comes at a high performance cost. To date, vulnerability signatures have only been considered for user on end hosts, severely limiting their deployment.

In our work, we observe that full and generic protocol parsing is *not necessary* for detecting vulnerability signatures. Using custom-built, hand-coded vulnerability signature recognizers, we show that these signatures can be detected 3 to 37 times faster than the speed of full protocol parsing. Therefore, there is no *inherent* performance penalty for using vulnerability signatures instead of exploit signatures.

Motivated by this, we design an architecture, called VESPA[1], for matching vulnerability signatures at speeds adequate for a high-performance enterprise NIDS, around 1 Gbps. We build our architecture on a foundation of fast string and pattern matchers, connected with control logic. This allows us to do deep packet inspection and model complex behavior, while maintaining high performance. We also minimize the amount of implicit state maintained by the parser. By avoiding full, in-memory semantic representation of the message, we eliminate much of the cost of generic protocol parsing. Finally, in many cases we are able to eliminate the recursive nature of protocol analysis, allowing us to skip analysis of large subsections of the message.

We have implemented a prototype of VESPA; tests show that it matches vulnerability signatures about three times faster than equivalent full-protocol parsing, as implemented in binpac [1]. Our architecture matches most protocols in software at speeds greater than 1 Gbps. Further, we show that our text protocol parsing is dominated by string matching, suggesting that special-purpose hardware for pattern matching would permit parsing text protocols at much higher speeds. Our binary protocol parsing is also well-adapted to hardware-aided implementation, as our careful state management fits well with the constrained memory architectures of network processors.

The rest of this paper is organized as follows: Section 2 gives some background on vulnerability signatures and discusses the context of our work. Sections 3 and 4 describe the design of VESPA and the vulnerability signature language. We present the implementation details of VESPA in Section 5. Section 6 contains a performance evaluation of our prototype. We discuss some future directions in Section 7 and related work in Section 8. Finally, Section 9 concludes.

2 Background

2.1 Vulnerability Signatures

Vulnerability signatures were originally proposed by Wang et al. [2] as an alternative to traditional, exploit-based signatures. While exploit signatures describe the properties of the exploit, vulnerability signatures describe how the vulnerability gets triggered in an application. Consider the following exploit signature for Code Red [3]:

```
urlcontent:"ida?NNNNNNNNNNNN..."
```

The signature describes how the exploit operates: it uses the ISAPI interface (invoked for files with extension "`.ida`") and inserts a long string of `N`'s, leading to a buffer overflow. While effective against Code Red, this signature would not match Code Red II [4]; that variant used `X`'s in place of the `N`'s. A vulnerability signature, on the other hand, does not specify how the worm works, but rather how the application-level vulnerability is triggered. An extract from the CodeRed signature in Shield [2] is:

[1] VulnErability Signature Parsing Architecture

```
c = MATCH_STR_LEN(>>P_Get_Request.URI,"id[aq]\?(.*)$",limit);
IF (c > limit)
  # Exploit!
```

This signature captures *any* request that overflows the ISAPI buffer, making it effective against Code Red, Code Red II, and any other worm or attack that exploits the ISAPI buffer overflow. In fact, this signature could well have been written before the release of either of the Code Red worms, as the vulnerability in the ISAPI was published a month earlier [5]. Thus, while exploit signatures are reactive, vulnerability signatures can proactively protect systems with known vulnerabilities until they are patched (which can take weeks or months [6]).

2.2 Protocol Parsing

Traditionally, exploit signatures are specified as strings or regular expressions. Vulnerability signatures, on the other hand, involve some amount of protocol parsing. Shield [2] used a language for describing C-like binary structures, and an extension for parsing text protocols. The follow-on project, GAPA [7], designed a generic application-level protocol analyzer to be used for matching vulnerability signatures. GAPA represented both binary and text protocols using a recursive grammar with embedded code statements. The generated GAPA parser, when guided by code statements, performed context-sensitive parsing. GAPA aimed to provide an easy-to-use and safe way to specify protocols and corresponding vulnerabilities.

Binpac [1], another protocol parser, was designed to be used in the Bro intrusion detection system [8]. Binpac is similar to GAPA: both use a recursive grammar and embedded code for parsing network protocols, and both are intended to minimize the risks of protocol parsing. Binpac, however, is designed only for parsing, with other parts of Bro performing checks for alarms or vulnerabilities. Binpac uses C++ for its embedded code blocks, and compiles the entire parser to C++ (similar to yacc), whereas GAPA uses a restricted, memory-safe interpreted language capable of being proven free of infinite loops. Binpac trades some of GAPA's safety for parsing speed; consequently, it achieves speeds comparable to hand-coded parsers written for Bro.

Since the implementation of GAPA is not freely available, we use binpac as our prototypical generic protocol parser generator in comparing to our work. Binpac is significantly faster than GAPA, yet it is not able to parse many protocols at speeds of 1 Gbps (though sparing use of binpac, where most data passing through the NIDS is not analyzed, can be supported.)

2.3 Vulnerability Complexity

Although Shield and GAPA used protocol parsing for vulnerability signatures, Brumley et al. suggest that vulnerability signatures could be represented across a spectrum of complexity classes [9]. They consider the classes of regular expressions, constraint satisfaction languages, and Turing machines, and provide algorithms to derive automatic vulnerability signatures of each class. As increasingly complex specifications of signatures are used, the precision of signature matching improves.

We make a different observation: most vulnerability signatures can be matched *precisely* without full protocol parsing. And such precise matching can be carried out at much greater speeds. In Table 1, we compare the performance of binpac to hand-coded implementations of several vulnerability signatures. We wrote the hand-coded implementations in C and designed them to match one specific vulnerability only. These would fall into the Turing machine class according to Brumley et al., but they are optimized for speed. Notice that the hand-coded implementations operate about *3x to 37x faster* than equivalent binpac implementation.

Table 1: The throughput (Mbits/s) of binpac parsers vs. hand-coded vulnerability matchers

Protocol	binpac	hand-coded
CUPS/HTTP	5,414	20,340
DNS	71	2,647
IPP	809	7,601
WMF	610	14,013

To see why this is the case, consider the following CUPS vulnerability (CVE-2002-1368 [10]). CUPS processes the IPP protocol, which sends messages embedded inside HTTP requests. CUPS would crash if a negative `Content-Length` were specified, presenting a denial-of-service opportunity. Our binpac implementation to check for this vulnerability is based on the binpac HTTP specification, which parses the HTTP header into name–value pairs. We add a constraint that looks for header names that match `Content-Length` and verifies that a non-negative value is used. Our hand-coded implementation, on the other hand, is built upon an Aho–Corasick [11] multi-string matcher, which looks for the strings "`Content-Length:`" and "`\r\n\r\n`" (the latter indicating the end of the headers). If "`Content-Length:`" is found, the following string is parsed as an integer and checked for being non-negative.

The parsers operate with equal precision when identifying the vulnerability, yet the hand-coded approach performs much less work per message, and runs more than 3 times as quickly. Of course, not all vulnerabilities can be matched with a simple string search. However, what this vulnerability demonstrates is that an efficient vulnerability signature matching architecture must be able to handle such simple vulnerabilities quickly, rather than using heavy-weight parsing for all vulnerabilities, regardless of complexity. The architecture will surely need to support more complex constructs as well, but they should only be used when necessary, rather than all the time. We next present a new architecture for specifying and matching vulnerability signatures that follows this principle. Our architecture shares some of the goals of binpac and GAPA; however, it puts a stronger focus on performance, rather than generality (GAPA) or ease-of-authoring (binpac).

3 Design

To make vulnerability signatures practical for use in network intrusion detection systems, we developed VESPA, an efficient vulnerability specification and matching architecture. The processes of writing a protocol specification and writing a vulnerability signature are coupled to allow the parser generator to perform optimizations on the generated code that specialize it for the vulnerabilities the author wishes to match.

Our system is based on the following design principles:

- Use of fast matching primitives
- Explicit state management
- Avoiding parsing of irrelevant message parts

Since text and binary protocols require different parsing approaches, we describe our design of each type of parser and how we apply the design principles listed above. We first give a brief outline of how the system works, and then go into detail in the subsequent sections on how our approach works.

We use fast matching primitives—string matching, pattern matching (regular expressions), and binary traversal—that may be easily offloaded to hardware. The signature author specifies a number of matcher primitive entries, which correspond to fields needed by the signature to evaluate the vulnerability constraint. Each matcher contains embedded code which allows the matching engine to automatically extract a value from the result of the match. For example, the HTTP specification includes a string matcher for "`Content-Length:`", which has an extraction function that converts the string representation of the following number to a integer.

Along with each matcher, the author also specifies a handler function that will be executed following the extraction. The handlers allow the signature author to model the protocol state machine and enable additional matchers. For example, if a matcher discovers that an HTTP request message contains the POST command, it will in turn enable a matcher to parse and extract the message body. We also allow the author to define handlers that are called when an entire message has been matched.

The author checks vulnerability constraints inside the handler functions. Therefore constraint evaluation can be at the field level, intra-message level, and inter-message level. Depending on the complexity of the vulnerability signature, the author can choose where to evaluate the constraint most efficiently.

3.1 Text Protocols

We found that full recursive parsing of text protocols is both too slow and unnecessary for detecting vulnerabilities. However, simple string or regular expression matching is often insufficient to express a vulnerability constraint precisely in cases where the vulnerability depends on some protocol context. In our system, we combine the benefits of the two approaches by connecting multiple string and pattern matching primitives with control logic specialized to the protocol.

Matching Primitives. To make our design amenable to hardware acceleration we built it around simple matching primitives. At the core, we use a fast multi-string matching algorithm. This allows us to approximate the performance of simple pattern-based IDSes for simple vulnerability signatures. Since our system does not depend on any specific string matching algorithm, we have identified several well-studied algorithms [11, 12] and hardware optimizations [13] that could be employed by our system. Furthermore, hardware-accelerated regular expression matching is also becoming a reality [14]. As discussed later, this would further enhance the signature author's ability to locate protocol fields.

Minimal Parsing and State Managment. We have found that protocol fields can be divided into two categories: core fields, which define the structure and semantics of the protocol, and application fields, which have meaning to the application, but are not necessary to understand the rest of the message. An example of a core field is the `Content-Length` in HTTP, as it determines the size of the message body that follows in the protocol, whereas a field such as `Accept-Charset` is only relevant to the application.

Our approach in writing vulnerability signatures is to parse and store only the core fields, and the application fields relevant to the vulnerability, while skipping the rest. This allows us to avoid storing irrelevant fields, focusing our resources on those fields that are absolutely necessary.

Although many text protocols are defined in RFCs using a recursive BNF grammar, we find that protocols often use techniques that make identification of core fields possible without resorting to a recursive parse. For example, HTTP headers are specified on a separate line; as a result, a particular header can be located within a message by a simple string search. Header fields that are not relevant to a vulnerability will be skipped by the multi-string matcher, without involving the rest of the parser. Other text protocols follow a similar structure; for example, SMTP uses labeled commands such as "`MAIL FROM`" and "`RCPT TO`", which can readily be identified in the message stream.

3.2 Binary Protocols

While some of the techniques we use for text protocol parsing apply to binary protocols as well, binary protocols pose special challenges that must be handled differently from text.

Matching Primitives. Unlike text protocols, binary protocols often lack explicit field labeling. Instead, a parser infers the meaning of a field from its position in the message—relative to either the message start, or to other fields. In simple cases, the parser can use fixed offsets to find fields. In more complicated cases, the position of a field varies based on inter-field dependencies (e.g., variable-length data, where the starting offset of a field in a message varies based on the length of earlier fields), making parsing data-dependent. Thus, parsers must often traverse many or all of the preceding fields. This is still simpler than a full parse, since the parser only examines the lengths and values of structure-dependent fields.

Since binary protocols are more heavily structured than text protocols, we need a matching primitive that is sufficiently aware of this structure while still maintaining high performance. We call this type of parser a binary traverser.

Designing an efficient binary protocol traverser is difficult because binary protocol designs do not adhere to any common standard. In our study of many common binary protocols, we found that they most often utilize the following constructs: C structures, arrays, length-prefixed buffers, sentinel-terminated buffers, and field-driven case evaluation (switch). The binpac protocol parser generator uses variations on these constructs as building blocks for creating a protocol parser. We found binpac to have sufficient expressive power to generate parsers for complex binary protocols. However, binpac parsers perform a full protocol parse rather than a simple binary traversal, so we use a modification to improve their performance.

Minimal Parsing and State Management. We reduced overhead of original binpac parsers for state management and skipped parsing unimportant fields. Because binpac carefully separates the duties of the protocol parser and the traffic analysis system which uses it, we were able to port binpac specifications written for the Bro IDS to our system. We retain the protocol semantics and structure written in the Bro versions but use our own system for managing state and expressing constraints. While we feel that additional improvements may be made in generating fast binary traversers, we were able to obtain substantial improvements in the performance of binpac by optimizing it to the task of traversal rather than full parsing. Furthermore, the binpac language provides exceptional expressiveness for a wide range of protocols, allowing our system to be more easily deployed on new protocols.

3.3 Discussion

By flattening the protocol structure, we can ignore any part of a message which does not directly influence properly processing the message or matching a specific vulnerability. However, some protocols *are* heavily recursive and may not be flattened completely without significantly reducing match precision. We argue that it is rarely necessary to understand and parse *each and every* field and structural construct of a protocol message to match a vulnerability. Consider an XML vulnerability in the skin processing of Trillian (CVE-2002-2366 [10]). An attacker may gain control of the program by passing an over-length string in a `file` attribute, leading to a traditional buffer overflow. Only the `file` attribute, in the `prefs/control/colors` entity can trigger the vulnerability, while instances of `file` in other entities are not vulnerable. To match this vulnerability with our system, the signature author can use a minimal recursive parser which only tracks entity open and close tags. The matcher can use a stack of currently open tags to tell whether it is in the `prefs/control/colors` entity and match `file` attributes which will cause the buffer overflow. The generated parser is recursive but only for the specific fields that are needed to match the vulnerability. This type of signature is a middle-ground for our system—it will provide higher performance than a full parser while requiring the user to manipulate more state than a simpler vulnerability.

In rare cases it may be necessary to do full protocol parsing to properly match a vulnerability signature. While our system is designed to enhance the performance of

```
parser HTTP_Request {
  dispatch() %{   deploy(vers);   deploy(is_post);   deploy(crlf);   }%

  int vers = str_matcher "HTTP/1."
        handler handle_vers()
        %{      end = next_whitespace(rest);
                vers = str_to_int(rest,end);    }%

 handle_vers() %{  // handle differently depending on version... }%

  bool is_post = str_matcher "POST"
        handler handle_post()
        %{   is_post=true;   }%

  handle_post() %{    if(is_post) { deploy(content_length); }   }%

  int content_length = str_matcher "Content-Length:"
        handler handle_cl()
        %{   end = next_line(rest);
             content_length = str_to_int(rest,end);   }%

  handle_cl() %{   if(this->content_length < 0) { // EXPLOIT! }
                   else  { deploy(body); }      }%

  bool crlf = str_matcher "\r\n\r\n" || "\n\n"
        %{ // do nothing explicit here }%

  Buffer body = extended_matcher crlf
        handler handle_body()
        %{   body = Buffer(rest, this->content_length);
             stopMachine();   }%

  handle_body() %{  // process body using another layer   }%
}
```

Fig. 1: Sample Specification for HTTP Requests (simplified)

simpler vulnerability signatures, it is still able to generate high-performance full recursive parsers. The drawback to our approach versus binpac or GAPA in this situation is that the user must manage the parser state manually, which may be error prone.

We do not yet address the problem of protocol detection. However, our system can be integrated with prior work [15] in an earlier stage of the intrusion detection system. Furthermore, the high-speed matching primitives used by VESPA may also be used to match protocol detection signatures.

4 Language

We have developed a vulnerability signature expression language for use with our system. We give an example vulnerability specification for the CUPS negative content length vulnerability in Figure 1.

Writing a signature involves specifying the matchers for the core fields of the protocol message and then specifying additional matchers to locate the vulnerability. We specify a single protocol message using a *parser* type. The code generator maps this message parser to a C++ class that will contain each state field as a member variable.

Inside a message parser, the vulnerability signature author defines handler function declarations and field variable declarations with matching primitives. The author can specify additional member variables that are not directly associated with a matcher using `member_vars %{ ... }%`.

Each underlying matching primitive always searches for *all* the requested strings and fields with which the matcher is initialized. For example, an HTTP matcher might search for "`Content-Type:`" in a message even though this string should only be expected in certain cases. This allows the primitive matcher to run in parallel with the state machine and constraint evaluation, though we have not yet implemented this. It also prevents the matching primitives from needing to back up to parse a newly desired field. We provide a utility for keeping track of which fields the matcher should expect and perform extraction and which to ignore. This state is controlled using the `deploy(`*`var`*`)` function. This function may be called from any handler function, and initially by the `dispatch` function. `deploy` marks a variable as expected in a state mask stored inside the parser. This will cause the matcher to execute the variable extraction function and handler when it is matched. A handler function may in turn enable additional matchers (including re-enabling itself) using the `deploy` function. The parser ignores any primitive match that is not set to be active using `deploy`.

The parser automatically calls the `dispatch` function each time the parser starts parsing a new protocol message. This allows the author to define which fields should be matched from the start of parsing. It also allows the initialization of member variables created using `member_vars`. Conversely, the parser automatically calls `destroy` to allow any resources allocated in `dispatch` to be freed.

4.1 Matcher Primitives

Protocol fields and matcher primitives are the heart of a vulnerability specification. The format of matcher primitive specification is:

```
var_type symbol = matching_primitive meta-data
        handler handler_func_name()
        %{
                // embedded C++ code to extract the value
        }%
```

The `var_type` specifies the storage type of the field; e.g., `uint32`. The symbol is the name of the field that will be stored as a member of the C++ parser class. There are three types of matching primitives.

1. `str_matcher` (string matcher primitive): The meta-data passed to this matcher are a string or sequence of strings separated by ||, and this instructs the underlying multi-string matching engine to match this string and then execute its extraction function. It supports matching multiple different strings that are semantically identical using *or* ("||").
2. `bin_matcher` (binary traversal primitive): The meta-data passed to this matcher are the file name of a binpac specification. This is followed by a colon and the name of a binpac `record` type. The meta-data end with the name of a field inside that `record`

that the author wishes to extract (*e.g.*, IPP.binpac: IPP_Message.version_num). The generated binpac parser will then call back to our system to perform the extraction and run the handler for the requested field.

3. `extended_matcher` (extension to another matcher): This construct allows us to perform additional extractions after matching a single string or binary field. This is often useful when multiple fields are embedded after a single match. It also allows the author to specify a different extraction function depending on which state is expected. The meta-data passed to this primitive are the name of another variable that uses a standard matching primitive.

Each variable match also specifies an extraction function within braces, %{ and }%, which extracts a relevant field from the message. We have provided a number of helper functions that the author can use in the extraction function, such as string conversion and white space elimination. In a string matcher extraction function, there are two predefined variables the signature author can use and modify: `rest` and `end`. The `rest` variable points to the first byte of input after the string that was matched. The parser also defines `end`, which allows the extraction function to store where the extraction ends. Extended matchers run immediately following the extraction function of the string matcher on which they depend and in the same context. Hence, any changes to the state of `rest` and `end` should be carefully accounted for in extended matcher extraction functions.

There are two additional functions that the author can use inside the extraction function of a string matcher: `stopMachine()` and `restartMachine(`*`ptr`*`)`. These functions suspend and restart pattern matching on the input file. This is useful, for example, to prevent the system from matching spurious strings inside the body of an HTTP message. The `restartMachine(`*`ptr`*`)` function restarts the pattern matching at a new offset specified by *`ptr`*. This allows the matcher to skip portions of the message.

4.2 Handlers

Each matcher may also have an associated handler function. The handler function is executed after the extraction and only if the matcher is set to be active with `deploy`. The signature author defines the body of the handler function using C++ code. In addition to calling the `deploy` function, handler bodies are where vulnerability constraints can be expressed. We do not yet address the reporting mechanism when a vulnerability is matched. However, since any C++ code may be in the handler, the author may use a variety of methods, such as exceptions or integer codes. The author may also use the handler functions to pass portions of a protocol message to another parser to implement layering and encapsulation.

While structurally different from existing protocol parser generators like GAPA and binpac, our language is sufficiently expressive to model many text and binary protocols and vulnerabilities. Porting a protocol specification from an RFC or an existing spec in another language (like binpac or GAPA) is fairly straightforward once the author understands the protocol semantics.

5 Implementation

5.1 Compiler

We designed a compiler to generate machine-executable vulnerability signature matchers from our language. We implemented the compiler using the Perl programming language. Our implementation leverages the "Higher Order Perl" [16] Lexer and Parser classes, which kept down the implementation complexity: the entire compiler is 600 lines. Approximately 70% of the compiler code specifies the lexical and grammatical structures of our language; the balance performs symbol rewriting, I/O stream management, and boilerplate C++ syntax.

Our compiler operates on a single parser file (e.g., `myparser.p`), which defines a signature matcher. The generated code is a C++ class which extends one of the parser super classes. The class definition consists of two files (following the example above, `myparser.h` and `myparser.cc`), which jointly specify the generated parser subclass.

5.2 Parser Classes

Generated C++ classes for both binary and text parsers are structurally very similar, but differ in how they interface with the matching primitives. We have optimized the layout and performance of this code. We use inlined functions and code whenever possible. Many extraction helper functions are actually macros to reduce unnecessary function call overhead. We store the expected state set with `deploy` using a bit vector.

For string matchers, we use the sfutil library from Snort [17], which efficiently implements the Aho–Corasick (AC) algorithm [11]. Because the construction of a keyword trie for the AC algorithm can be time-consuming, we generate a separate reusable class which contains the pre-built AC trie. Our text matcher is not strongly tied to this particular multi-string matching implementation, and we have also prototyped it with the libSpare AC implementation [18].

We use binpac to generate a binary traverser for our parsers. As input, the compiler expects a binpac specification for the binary protocol. This should include all the `record` types in the protocol as well as the basic `analyzer`, `connection`, and `flow` binpac types. We then use the `refine` feature of binpac to embed the extraction functions and callbacks to our parser. Since binpac does simple extractions automatically, it is often unnecessary to write additional code that processes the field before it is assigned. Like the AC algorithm for text parsers, the binary parser is not heavily tied to the binary traversal algorithm or implementation. For a few protocols, we have developed hand-coded replacements for binpac binary traversal.

5.3 Binary Traversal-Optimized Binpac

We have made several modifications to the binpac parser generator to improve its performance for binary traversal. The primary enhancement we made is to change the default model for the in-memory structures binpac keeps while parsing. The original binpac allocated a C++ class for each non-primitive type it encountered while parsing.

This resulted in an excessive number of calls to `new`, even for small messages. To alleviate this problem, we changed the default behavior of binpac to force all non-primitive types to be pre-allocated in one object. We use the `datauint` type in binpac to store all the possible subtypes that binpac might encounter. To preserve binpac semantics, we added a new function, `init(`*`params...`*`)`, to each non-primitive type in binpac. The `init` function contains the same code as the constructor, and we call it wherever a new object would have been created. It also accepts any arguments that the constructor takes to allow fields to be propagated from one object to another. We restrict binpac specifications to be able to pass only primitive types from object to object. While this reduces our compatibility with existing binpac specifications, it is easy to change them to support this limitation.

Some objects in binpac *must* be specified using a pointer to a dynamically created object and cannot be pre-allocated. For example, in the Bro DNS binpac specification, a `DNS_name` is composed of `DNS_label`s. A `DNS_label` type also contains a `DNS_name` object if the label is a pointer to another name. This circular dependency is not possible with statically sized classes. We added the `&`*`pointer`* attribute modifier to the binpac language to allow the author to specifically mark objects that must be dynamically allocated.

The final modification we made to binpac was to change the way that it handled arrays of objects. The original version of binpac created a vector for each array and stored each element separately. Because binary traversal only needs to access the data as it is being parsed, we do not need to store the entire array, only the current element. We eliminated the vector types entirely and changed binpac to only store the current element in the array using a pre-allocated object. If the author needs to store data from each element in the array, he must explicitly store it outside of binpac in the VESPA parser class using a handler function.

6 Evaluation

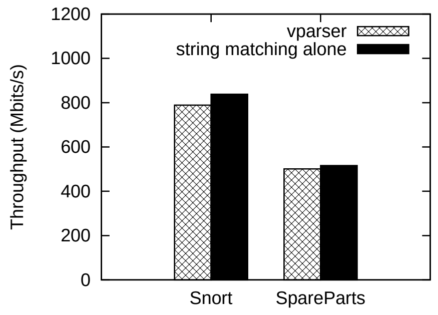
We evaluated VESPA with vulnerabilities in both text and binary protocols. We implemented matchers for vulnerabilities in the HTTP, DNS, and IPP protocols. We searched for exploitable bugs in network-facing code, focusing especially on scenarios where traditional exploit signatures would fail. Like Cui et al. did with GAPA [19], we found the process of writing a vulnerability signature for a protocol very similar to writing one for a file format. Thus, we used our system develop to a binary parser for the Windows Meta-file Format (WMF).

We ran all our experiments on an Ubuntu 7.10 Linux (2.6.22-14-x86_64) system with a dual-core 2.6 GHz AMD Athlon 64 processor and 4GB of RAM (our implementation is single-threaded so we only utilized one core). We ran the tests on HTTP and DNS on traces of real traffic collected from the UIUC Coordinated Science Laboratory network. We collected WMF files from freely available clipart websites. Since we did not have access to large volumes of IPP traffic, we tested using a small set of representative messages. We repeated the trace tests 10 times, and we repeated processing the IPP messages 1 million times to normalize any system timing perturbations. We show the standard deviation of these runs using error bars in the charts.

6.1 Micro-benchmarks of Matching Primitives

To evaluate the performance of using fast string matching primitives, we implemented our parser using two different implementations of the Aho–Corasick (AC) algorithm and compared their performance (Figure 2a). We used the sfutil library, which is part of the Snort IDS [17], and the Spare Parts implementation of AC [18]. We used those base implementations to search for the same strings as our vulnerability matcher does, but without any of the control logic or constraint checking. We found that for either AC implementation, the performance of a basic HTTP vulnerability matcher (which handles optional bodies and chunking) was very close to that of the string matching primitive.

The performance of string matching alone approximates (generously) the performance of a simple pattern-based IDS. If the vulnerability signature is simple enough to be expressed using a simple string match (e.g., the IPP vulnerability for a negative `Content-Length`), our system is able to match it with comparable performance to a pattern based IDS.



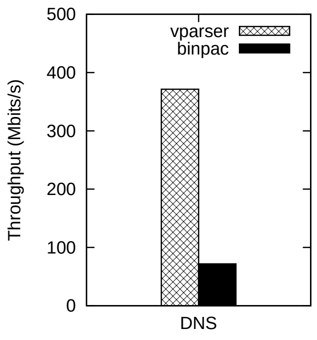
(a) Comparison between string matching primitive and parsing for HTTP requests

Parser Type	Bytes allocated	Num calls to `new`
DNS (binpac)	15,812	539
DNS (traversal)	2,296	14
IPP (binpac)	1,360	33
IPP (traversal)	432	6
WMF (binpac)	3,824	94
WMF (traversal)	312	6

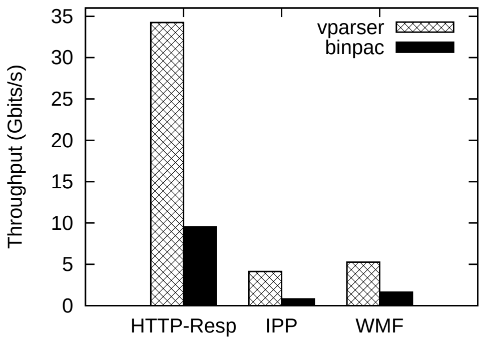
(b) Dynamic memory usage for a single message for standard binpac vs. binary traversal

Fig. 2: Micro-benchmarks

We next investigated the performance of binary traversal in binpac. One of the primary changes we made to binpac was to change its default memory and allocation behavior. We instrumented the original version of binpac and a parser built with our binary traversal-optimized version to assess the effectiveness of this change (Figure 2b). We saw an overall reduction in memory usage despite pre-allocating types that may not be present in the message. We were also able to cut the number of calls to `new` by a substantial factor for all three binary protocols we implemented. Our IPP and WMF traversers do not contain any explicit pointer types (specified with `&pointer`), so the number of allocated blocks is constant for *any* protocol message. The number of times the DNS parser calls the `new` allocator is proportional to the number of name pointers in the message.



(a) DNS Throughput (Mbps)



(b) Parser Throughput (Gpbs)

Fig. 3: Vulnerability Signature Matcher Performance

6.2 Signature Matching Performance

We evaluated the throughput of our vulnerability signature matching algorithms compared to the binpac parser generator. Binpac is the most efficient freely available automated protocol parser generator. We do not evaluate against GAPA because it has not been publicly released. Furthermore, binpac far exceeds GAPA in performance because it directly generates machine code rather than being interpreted [1]. Since binpac is not specifically designed for vulnerability signatures, we added vulnerability constraint checking to the binpac protocol specifications. In each of the following sections we describe the protocol and vulnerabilities we tested against. We show the results in Figure 3.

HTTP/IPP. The Common Unix Printing System (CUPS), with its protocol encapsulation and chunk-capable HTTP parser, illustrates several design choices which confound exploit-signature writers. The vulnerability given in CVE-2002-0063 [10] occurs because of the way the Internet Printing Protocol (IPP) specifies a series of textual key–value pairs, called attributes. The protocol allows attribute lengths to vary, requiring the sender to use a 16-bit unsigned integer to specify the length of each attribute. CUPS reads the specified number of bytes into a buffer on the stack, but the buffer is only 8192 bytes long, allowing an attacker to overflow the buffer and execute arbitrary code with the permissions of the CUPS process. A signature for this attack must check that each attribute length is less than 8192. IPP is a binary protocol but it is encapsulated inside of chunked HTTP for transport. Attackers can obfuscate the exploit by splitting it across an arbitrary number of HTTP chunks, making it very hard to detect this attack with pattern-based signatures. We also tested the negative content length vulnerability that we have discussed previously.

We designed a text-based vulnerability signature matcher for HTTP. In addition to vulnerabilities in HTTP itself, many protocols and file formats which are encapsulated inside of HTTP also have vulnerabilities. We use VESPA to match the `Content-Length`

Table 2: HTTP Message Rate

HTTP Message Type	Message Rate (msgs per sec)
Requests	370,005
Responses	196,897
Chunked	41,644
Overall	314,797

vulnerability in CUPS/IPP, as well as to extract the body of the message to pass it to another layer for processing. We support standard and chunked message bodies and pass them to a null processing layer. Unfortunately, we were unable to make a direct comparison to binpac for chunked HTTP messages due to a bug in binpac's buffering system: binpac will handle such a message but fail to extract data from each individual chunk. Despite this, we found that VESPA was considerably faster than the equivalent binpac parser. Since much of the HTTP message body is ignored by both VESPA and binpac, the throughputs we observed are very high because the size of the body contributes to the overall number of bytes processed. We also measured the message processing rates for various types of HTTP messages and found them to be adequate to process the traffic of a busy website (Table 2).

We implemented a binary IPP vulnerability matcher to be used in conjunction with our HTTP parser. The VESPA IPP matcher ran four times as fast as the binpac version, largely due to the improved state management techniques we described earlier. We also developed a hand-coded drop-in replacement for our binpac binary traverser of the IPP protocol. Using this replacement, we were able to achieve an order of magnitude improvement over the performance of the binpac binary traversal (see Table 1). Therefore, our architecture stands to benefit from further improvements of the base matching primitives of binary traversal as well.

DNS. The DNS protocol includes a compression mechanism to avoid including a common DNS suffix more than once in the same message. Parsing these compressed suffixes, called name pointers, is best done with a recursive parser, but doing so introduces the possibility of a "pointer cycle," where a specially-crafted message can force a parser to consume an infinite amount of stack space, leading to a denial of service [20].

DNS name pointers can occur in many different structures in DNS, so the binary traversal must parse and visit many of the fields in the protocol. Therefore, parsing DNS is usually much slower than other protocols. Indeed, DNS is the worst-performing of our vulnerability signature matchers, though it is still several times faster than binpac, as can be seen in Figure 3. Pang et al. suggest that this is due to an inherent difficulty of parsing DNS, pointing to the comparable performance of their hand-implemented parser to binpac [1]. We have found this not to be the case, as our hand-implemented DNS parser that finds pointer cycles can operate at nearly 3 Gbps (see Table 1). As part of our future work, we will investigate what part of our current design is responsible for the much worse performance of DNS; our hope is that we will be able to achieve speeds in excess of 1 Gbps with an automatically-generated parser.

WMF. Vulnerabilities are increasingly being found in file formats (so called "data-driven attacks") rather than just network messages. The WMF format allows specification of a binary "abort procedure," called if the rendering engine is interrupted. Attackers began to misuse this feature in late 2005, using the abort handler for "drive-by downloads," where an attacker could run arbitrary code on a victim's computer by simply convincing them to render a WMF, requiring only a website visit for clients using Internet Explorer (CVE-2005-4560 [10]).

This vulnerability has been problematic for intrusion detection systems, Snort in particular. Snort normally processes only the first few hundred bytes of a message when looking for vulnerabilities; however, a WMF vulnerability can be placed at the end of a very large media file. However, matching the Snort rule set over an entire message exhausts the resources of most intrusion detection systems, requiring most sites to resort to a convoluted configuration with two Snort processes running in concert. Our architecture allows for a much cleaner approach: after an HTTP header has been parsed, the WMF vulnerability matcher would be called in the body handler, while other string matchers and handlers would be turned off. Figure 3 shows that WMF files can be parsed at multi-gigabit rates, so this would not put a significant strain on the CPU resources of the NIDS.

7 Future Directions

Although our prototype shows that high-performance vulnerability signature matching is possible in software, to achieve speeds in excess of 1 Gbps for all protocols, a hardware-accelerated approach is likely needed. Our plan is to use hardware implementations of fast pattern-matching algorithms [14, 21] to replace the software implementations. This should dramatically increase the performance of text protocol parsing, as discussed in Section 6.1. We will also investigate the use of network processors, such as the Intel IXP family [22], to bring vulnerability processing closer to the network interface, and to exploit the inherent parallelism in matching signatures. Previous work has shown that using network processors can be nearly two orders of magnitude faster than similar implementations in software [23]. Network processors achieve such speedups in part by using a complex memory hierarchy; our careful management of limited state makes our architecture well-adapted to being ported to a network processor.

There are also performance gains yet to be realized in software matching as well. Our hand-coded matchers for vulnerabilities in binary protocols, in particular, are significantly fasters than those implemented using VESPA (see Table 1). The extra performance is likely due to eliminating the abstractions that ensue from representing a binary protocol structure in binpac. Our future work includes faster implementation of those abstractions, as well as the design of abstractions better suited to fast matching. One challenge that we will face is the fact that binary protocols exhibit much less consistency of design than text protocols.

Our eventual goal is to create a network intrusion *prevention* system (NIPS), which will sit as a "bump in the wire" and filter attacking traffic. In addition to throughput, another challenge that a NIPS will face is reducing latency, since, unlike intrusion detection systems, filtering decisions must be complete before the traffic can be forwarded to its destination. Furthermore, a NIPS must be able to recognize a large collection of vulnerability signatures at once. Our use of multi-pattern search as a base primitive will make parallel matching of several signatures easier to implement, but our design will need to incorporate constructs that will allow the reuse of common components (e.g., HTTP `Content-Length` extraction) between multiple signatures.

Authoring of effective signatures is a complex and error-prone process; this is true for exploit signatures, and more so for vulnerability signatures. Although our architecture was optimized for performance, rather than ease of authorship, we have found that expressing vulnerability constraints using VESPA was not appreciably more difficult than using binpac or GAPA. However, as we gain more experience with VESPA, we plan to improve the interface between the programmer and our architecture by, for example, introducing more reusable constructs and modularity. We also plan to develop better architectures for testing vulnerability signatures, to ensure that they do not generate false positives or false negatives.

Finally, automatic generation of vulnerability signatures can make them useful for not only known vulnerabilities, but new ones just observed ("zero-day"). Previous work has used annotated protocol structure [24, 19], program analysis [9, 25], or data flow analysis [26] to automatically generate vulnerability signatures. We will explore to what extent these approaches may be used to automatically generate signatures in our architecture. This will present a significant challenge to an automated approach, given that our architecture relegates more of state management to the programmer.

8 Related Work

8.1 Pattern Matching

The Wu–Manber [12], Boyer–Moore [27], and Aho–Corasick [11] algorithms provide fast searching for multiple strings. Their superior performance has made them natural candidates for IDS pattern-matching; in addition to our system, Snort [17] uses Aho–Corasick to match static strings.

Although slower than string matching, regular expression-based matching provides considerably more expressive power. Regular-expression matching is well-studied in the literature; broadly, deterministic matching (e.g., flex [28]) offers linear time but exponential space complexity, while nondeterministic matching (e.g., pcre [29]) offers linear space but exponential time complexity. Smith et al. attempt to combine the advantages of deterministic and nondeterministic matching using Extended Finite Automata [30]. Rubin et al. have developed protomatching to heuristically reduce matching complexity by discarding non-matching packets as quickly as possible, while keeping a low memory footprint [31]. Special-purpose hardware achieves sustained pattern matching at 4 Gbps [14]. Clark et al. [13] used application-specific FPGA cores to exploit the parallelism inherent in searching for many patterns simultaneously in a single body of text.

8.2 Vulnerability Signatures

The Shield project at Microsoft Research [2] pioneered the idea of vulnerability signatures; Borisov et al. extended the idea with a generic protocol parser generator [7]. Brumley et al. explained the complexity of various approaches to matching [9].

The binpac project at UC Berkeley and the International Computer Science Institute [1] focused on implementing a yacc-like tool for generating efficient protocol parsers from high-level definitions. binpac abstracts away much error-inducing complexity (e.g., network byte ordering). Its performance for many protocols is adequate for many intrusion detection tasks, but the VESPA architecture significantly improves on it, as shown in our evaluation.

The ongoing NetShield project [32] shares our goals of high-speed vulnerability signature detection. It has resulted in novel techniques for fast binary traversal, as well as efficient multi-signature matching, which may provide promising approaches for addressing some of the same challenges in VESPA.

8.3 Intrusion Detection

Intrusion detection requires attention to both algorithmic efficiency, and systems / implementation issues. Ptacek and Newsham [33] have detailed several strategies for evading intrusion detection by shifting packet TTLs, among others. Snort [34, 17] and Bro [8], two popular IDS platforms, have addressed many systems-level issues, but are intended only to detect, not prevent intrusion. So-called intrusion prevention systems go further, by being deployed inline with the forwarding path; these systems take a more active stance against hostile traffic by dropping malicious or otherwise anomalous packets. The SafeCard [35] project used an Intel IXP network processor to perform intrusion protection in real-time up to 1 Gbps. It used high-speed matching of regular expressions, as well as an early implementation of Prospector [26] signatures, finding vulnerabilities within HTTP headers. The project shows that special-purpose hardware is a promising direction for high-performance intrusion prevention systems.

9 Conclusion

We have proposed an architecture, called VESPA, for fast matching of vulnerability signatures. VESPA relies on the fact that full protocol parsing is often not necessary to match vulnerability signatures and as a result is able to match signatures several times faster than existing work. We have built a prototype implementation of our architecture, and we showed that we can match vulnerabilities in many protocols at speeds in excess of 1 Gbps, thus demonstrating that vulnerability signatures are practical for high-performance network intrusion detection systems. We plan to continue to improve the performance of our system by improved implementation of base primitives and hardware acceleration, and to develop a full-fledged implementation of a high-performance network intrusion prevention system based on vulnerability signatures.

10 Acknowledgments

We would like to thank David Nicol and William Sanders for their guidance on this project and the anonymous referees for their suggestions on an earlier version of this draft. This work was supported by NSF grant CNS 06–27671.

References

1. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: A yacc for Writing Application Protocol Parsers. In: Proceedings of the Internet Measurement Conference. (2006)
2. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In: ACM SIGCOMM Computer Communications Review. (2004)
3. CERT: "Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. www.cert.org/advisories/CA-2001-19.html (July 2001) CERT Advisory CA-2001-19.
4. Friedl, S.: Analysis of the New "Code Red II" Variant. www.unixwiz.net/techtips/CodeRedII.html (August 2001)
5. Microsoft: Unchecked Buffer in ISAPI Extension Could Enable Compromise of IIS 5.0 Server. www.microsoft.com/technet/security/bulletin/ms01-023.mspx (June 2001) Microsoft Security Bulletin MS01-033.
6. Rescorla, E.: Security Holes... Who Cares? In Paxson, V., ed.: USENIX Security Symposium. (August 2003)
7. Borisov, N., Brumley, D.J., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: A Generic Application-Level Protocol Parser Analyzer and its Language. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium. (2007)
8. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-time. Comput. Netw. **31**(23-24) (1999) 2435–2463
9. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards Automatic Generation of Vulnerability-Based Signatures. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy. (2006)
10. CVE: Common Vulnerabilities and Exposures http://cve.mitre.org/.
11. Aho, A.V., Corasick, M.J.: Efficient String Matching: an Aid to Bibliographic Search. Commun. ACM **18**(6) (1975) 333–340
12. Wu, S., Manber, U.: A Fast Algorithm for Multi-Pattern Searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona (1994)
13. Clark, C., Lee, W., Schimmel, D., Contis, D., Koné, M., Thomas, A.: A Hardware Platform for Network Intrusion Detection and Prevention. In: Proceedings of the Third Workshop on Network Processors and Applications. (2004)
14. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In: ISCA. (2006) 191–202
15. Dreger, H., Feldmann, A., Mai, M., Paxson, V., Sommer, R.: Dynamic Application-layer Protocol Analysis for Network Intrusion Detection. In: USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2006) 18–18
16. Dominus, M.J.: Higher Order Perl: Transforming Programs with Programs. Morgan Kaufmann (2005)
17. Sourcefire, Inc.: Snort. www.snort.org
18. Watson, B.W., Cleophas, L.: SPARE Parts: a C++ Toolkit for String Pattern Recognition. Softw. Pract. Exper. **34**(7) (2004) 697–710

19. Cui, W., Peinado, M., Wang, H.J., Locasto, M.E.: ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In Pfitzmann, B., McDaniel, P., eds.: IEEE Symposium on Security and Privacy. (May 2007) 252–266
20. NISCC: Vulnerability Advisory 589088/NISCC/DNS (May 2005) www.cpni.gov.uk/docs/re-20050524-00432.pdf.
21. Clark, C.R., Schimmel, D.E.: Scalable Pattern Matching for High-Speed Networks. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, California (2004) 249–257
22. Intel: Intel Network Processors. www.intel.com/design/network/products/npfamily/index.htm
23. Turner, J.S., Crowley, P., DeHart, J., Freestone, A., Heller, B., Kuhns, F., Kumar, S., Lockwood, J., Lu, J., Wilson, M., Wiseman, C., Zar, D.: Supercharging PlanetLab: A High Performance, Multi-application, Overlay Network Platform. SIGCOMM Computing Communications Review **37**(4) (2007) 85–96
24. Liang, Z., Sekar, R.: Fast and Automated Generation of Attack Signatures: A Basis for Building Self-protecting Servers. In Meadows, C., ed.: ACM Conference on Computer and Communications Security, New York, NY, USA, ACM (November 2005) 213–222
25. Brumley, D., Wang, H., Jha, S., Song, D.: Creating Vulnerability Signatures Using Weakest Pre-conditions. In: Proceedings of the 2007 Computer Security Foundations Symposium, Venice, Italy (July 2007)
26. Slowinska, A., Bos, H.: The Age of Data: Pinpointing Guilty Bytes in Polymorphic Buffer Overflows on Heap or Stack. In Samarati, P., Payne, C., eds.: Annual Computer Security Applications Conference. (December 2007)
27. Boyer, R.S., Moore, J.S.: A Fast String Searching Algorithm. Commun. ACM **20**(10) (1977) 762–772
28. Flex: The Fast Lexical Analyzer. www.gnu.org/software/flex
29. PCRE: Perl Compatible Regular Expression Library. www.pcre.org
30. Smith, R., Estan, C., Jha, S.: XFA: Faster Signature Matching with Extended Automata. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. (2008)
31. Rubin, S., Jha, S., Miller, B.P.: Protomatching Network Traffic for High Throughput Network Intrusion Detection. In: Proceedings of the 13th ACM conference on Computer and communications security. (2006)
32. Li, Z., Xia, G., Tang, Y., He, Y., Chen, Y., Liu, B., West, J., Spadaro, J.: NetShield: Matching with a Large Vulnerability Signature Ruleset for High Performance Network Defense. Manuscript (2008)
33. Ptacek, T.H., Newsham, T.N.: Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6 (1998)
34. Roesch, M.: Snort—Lightweight Intrusion Detection for Networks. In Parter, D., ed.: Proceedings of the 1999 USENIX LISA Systems Administration Conference, Berkeley, CA, USA, USENIX Association (November 1999) 229–238
35. de Bruijn, W., Slowinska, A., van Reeuwijk, K., Hruby, T., Xu, L., Bos, H.: SafeCard: A Gigabit IPS on the Network Card. In: Proceedings of the 9th International Symposium On Recent Advances in Intrusion Detection. (2006)