HIGH-PERFORMANCE NETWORK INTRUSION DETECTION: A NEW PARADIGM IS NEEDED

BY

DAVID ROBERT ALBRECHT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Assistant Professor Nikita Borisov

ABSTRACT

Fast data rates and complicated protocols have outpaced network intrusion detection systems. Administrators are forced to choose between breadth and depth: systems either deeply analyze traffic for a small handful of vulnerabilities, or search for many in parallel using more primitive (and easily evadable) techniques. We present a new parser architecture called VESPA, which uses the concept of *vulnerability signatures* to offer both speed and accuracy. VESPA is informed by a study of network protocols, which precedes the design. We conclude by reviewing several trends in computer architecture, and their impact on future intrusion detection systems. We believe a system which offers both speed and accuracy is possible, but requires rethinking how network intrusion detectors are designed, in light of trends in computer architecture.

To Stephanie and Steve, for convincing me to start,
To John, Nikhil, and my research group—Nikita, Prateek, Nabil, Robin,
Amir, and Shishir—for showing me the way,
And Kate, for helping me to finish.

ACKNOWLEDGMENTS

Chapter 4 is derived from “High-speed matching of vulnerability signatures,” work previously published at the 2008 Recent Advances in Intrusion Detection Symposium. The work was co-authored with Nabil Schear and Nikita Borisov. Apart from the noted exception, this thesis is original work of the author. It has not been published or used previously for any purpose.

This work was supported by NSF grant CNS 06-27671.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In the past decade, the Internet has become mission-critical. Once peripheral to other communication systems, today the Internet is used to execute financial transactions, control utility infrastructure, and even serve court summons [1], [2]. Given the network's centrality to its users' personal and professional lives, service interruptions are quite disruptive. In 2007, the Estonian government became a victim of cyberterrorism when a distributed denial of service attack disabled access to several of its web sites for several hours [3].

Despite the high cost of disruptions, software vulnerabilities persist. Code and protocol defects in operating systems, servers, and recently, client applications (e.g. web browsers [4]) offer plenty of opportunities for misuse. Combined with fast, anonymous broadband access, vulnerabilities in network-facing code have created an extremely challenging threat landscape [5].

The need for systems that detect and respond to misuse—network intrusion detection systems (NIDS)—is greater than ever, yet existing systems have not kept up. IT organizations with big budgets have made some headway using expensive commercial solutions; those with fewer resources sacrifice detection accuracy to keep up with the incoming traffic stream.

Setting out, we wanted to know whether a NIDS could be built on commodity hardware while still offering reasonable performance. Ideally, the system would offer full understanding of protocol semantics, and be capable of operation at around 1 Gbps. After a year and a half of research and study, we believe such a system is possible, but it requires rethinking how to construct a NIDS.

We begin by characterizing the NIDS workload with a detailed study of network protocols. We continue with experiments on protocol parsing, which demonstrate that today's systems leave significant performance on the table in the name of usability and object-orientation. Finally, we describe why current designs are misaligned with fundamental trends in computer architecture.

This work focuses on signature-based intrusion detection systems positioned at an enterprise ingress/egress point. We do not study anomaly-based systems or positioning a NIDS at another point in the topology, although in principle our insights should apply to both of these scenarios.

CHAPTER 2

LITERATURE REVIEW

NIDS design is an exercise in systems engineering. The system must reconstruct the flow of incoming data, convert the stream of bytes into protocol events, and apply policy to the traffic. With so many related concerns, the scope of NIDS literature is large.

2.1 Pattern Matching

Pattern matching is a key building block of NIDS. High-speed pattern matching is especially important, as we want to design a system that operates at 1 Gbps.

NIDS use substring search (the most primitive search operation) to locate static strings in network data. The most naive algorithm, compare-and-backtrack, is the slowest. Compare-and-backtrack requires no preprocessing, but this is not a major advantage as signature-based NIDSes use a static pattern set. Knuth-Morris-Pratt [6] does better than compare-and-backtrak by precomputing a "partial match table," which avoids full backtracking. The Aho-Corasick algorithm [7] goes further by searching for a multi-element "dictionary" of strings in a single, linear-time pass of the text. Boyer-Moore [8] also offers linear-time search, but its runtime is scaled by $\frac{1}{m}$, where m is the length of the pattern to match. Wu and Manber's "fuzzy string search" [9] offers the ability to find "approximate" matches (based on Levenshtein distance [10]) of a string within a corpus of text.

Beyond static strings, *regular languages* define a set (possibly infinite) of patterns to match.[1] The set of patterns is called a regular language, and is

[1]While the number of strings in a regular language is potentially infinite, the number of states in the corresponding automaton is necessarily finite. Consider the regular expression which matches all strings: its automaton contains a single "accept" state, but it matches

specified by a *regular expression*. Finite automata recognize membership in a regular language; given a regular language, constructing an automaton to recognize it is trivial, from the correctness standpoint. However, the technique used to construct the automaton affects its performance characteristics. “Nondeterministic” constructions (e.g. pcre [11]) attempt to find a path through the automaton by backing out when they hit a “dead end.” The nondeterministic approach requires a modest amount of memory, but can suffer from long runtimes if excessive backtracking occurs. “Deterministic” approaches (e.g. flex [12]), on the other hand, generate a new machine with a state corresponding to every member of the power set of the nondeterministic machine’s states. Deterministic matchers linearly bound runtime (they never backtrack), but have huge memory requirements—potentially exponential in the nondeterministic machine’s number of states.

Context-free languages generalize regular expressions with production rules. Our study of network traffic (protocols and vulnerabilities) revealed few, if any, protocol structures that mapped naturally to context-free languages (Pang et al. [13]). Context-sensitive languages offer the most general pattern recognition, but recognizing them lacks the computational boundedness of context-free languages, which limits their usefulness in IDS.[2]

Some nontraditional approaches increase performance by exploiting particular characteristics of the NIDS workload. Smith et al. attempted to combine the advantages of deterministic and nondeterministic matching using Extended Finite Automata [14]. Rubin et al. developed protomatching to heuristically reduce matching complexity by discarding non-matching packets as quickly as possible, while keeping a low memory footprint [15]. Hardware-based approaches offer high-speed matching using custom hardware; Section 2.4.1 explores this further.

an infinite number of strings.

[2]As an aside, we note that language complexity is, at best, a rough indicator of hardware complexity; the fact that a traffic pattern is expressable as a regular language in no way implies its ease of recognition, relative to a context-free language.

2.2 Protocols, and Protocol Parser Generators

Similar to yacc/bison [16], [17], protocol parser generators automatically generate protocol parsers from declarative grammars. Protocol parser generators augment the functionality of basic parser generators with features useful for parsing network data: automatic reordering of multibyte fields, state checkpointing, etc.

The "Generic Application-Layer Protocol Analyzer" [18] was the first of at least two major efforts to produce a protocol parser generator. GAPA offers a type-safe, declarative language called GAPAL, in which users declare protocol grammar. From the GAPAL specification, GAPA generates a recursive-descent parser to consume protocol data, which runs as a layered service provider in the Windows Sockets (Winsock) stack. GAPA's strong type-safety hardens generated parsers against exploitation, but the reference implementation achieved an overall throughput of only 11.7 Mbps on moderately powerful hardware.[3] GAPA's performance is adequate for end-host protocol parsing, but unsuitable for bulk (in-network) analysis.

Binpac, a project similar to GAPA, also generates network protocol parsers [13]. However, the binpac authors elected to sacrifice GAPA's strong safety for improved performance. Rather than targeting compilation to an interpreted environment, binpac emits native C++ code suitable for bulk monitoring and analysis applications. Binpac is used as the standard protocol parser in the Bro Intrusion Detection System [19].

The VESPA project [20] offers a third alternative for generating protocol parsers. The VESPA authors noted that intrusion detection policies often rely on a limited subset of protocol fields, even though most designs unconditionally parse all fields. By performing parsing in a policy-aware way, the system realizes order-of-magnitude speedups in some protocols. VESPA is explained in detail in Chapter 4.

2.3 Software Intrusion Detection Systems

Both commercial vendors and free/open-source groups (FOSS) have developed software-based NIDS. Snort [21] and Bro [19] have received the

[3]A Windows XP machine with a 3 GHz CPU and 1 GB of RAM.

lion's share of attention in the literature; both feature wide deployment, and freely available codebases.

Bro has been under development by researchers at Lawrence Berkeley National Laboratory and the International Computer Science Institute several years before a paper describing the system was published in 1998 [19]. Bro features a modular, object-oriented construction designed for in-network (cf. host-based) monitoring. To control Bro's operation, users express policy in the Bro Scripting Language, a high-level, purpose-built interpreted language. Bro uses binpac (Section 2.2) to parse protocols, and ships with binpac grammars for many popular protocols (FTP, HTTP, SMTP, SMB/CIFS, etc.). Protocol grammars define "protocol events," which trigger execution of "handlers" written in the Bro scripting language. Bro's main code branch uses a single-threaded, non-blocking design. Efforts to parallelize Bro have produced the Bro cluster, a multi-machine system with network-based synchronization [22], and "superlinear" Bro, a multithreaded implementation [23].

Snort [21] was developed by Martin Roesch, and is currently maintained by Sourcefire, a commercial entity. The code is available under an open-source license; signatures are available commercially via Sourcefire, and through the Snort community.

Commercial vendors including TippingPoint, Cisco, and Radware also produce intrusion detection and prevention systems, but these are not widely studied in the research community.

2.4 Hardware Approaches

Intrusion detection's challenging performance requirements have motivated several groups to explore hardware-based solutions. Approaches range from developing accelerators to support existing software solutions, to full systems implemented on special-purpose hardware.

2.4.1 Hardware: Accelerators

Hardware accelerators increase performance by offloading a narrowly-targeted piece of the workload onto purpose-built hardware.

In some cases, a NIDS need only analyze the beginning of a flow to determine what action (drop, allow, inspect) should be taken for subsequent traffic. The ICSI Shunt [24] keeps a per-flow hashtable in a fast hardware cache. If the system determines that a flow is either safe or unsafe (meaning no further analysis is required), the Shunt avoids routing traffic to the NIDS, reducing its workload.

Clark et al. [25] explored using Intel IXP network processors and Xilinx Vertex FPGAs to accelerate string matching operations. The group claimed 1 Gbps attainable throughput, but at a coarse level of semantic analysis. Brodie [26] demonstrated regular expression matching at 4 Gbps using a highly optimized FPGA, which consumed the input text two characters at a time.

2.4.2 Hardware: Systems

The SafeCard [27] project used an Intel IXP network processor to perform intrusion protection in real time at up to 1 Gbps. Owing to implementation on the Intel IXP network processor, SafeCard developed novel techniques for (1) managing a highly vertical (five-level) memory hierarchy, and (2) partitioning the IDS workload into a form which effectively utilized the unique execution resources of the IXP architecture: small microengines coupled with a general-purpose processor.

2.5 Classification

Classification splits network traffic into predefined classes—often based on application protocol—to control quality of service. A closely related topic to intrusion detection, classification also requires semantic understanding of protocols to operate effectively.

Moore and Papagiannaki [28] measured the effectiveness of several forms of classification. Starting from port-only classification, the group used progressively more stateful and computationally intensive techniques, culminating in full per-flow tracking. Port-based classification (the most basic type) correctly identified the application protocol in approximately

70%[4] of their flows; inspecting the protocol of the first kilobyte raised correctness to 80%. Per-flow stateful protocol parsing classified over 99% of traffic correctly.

BLINC [29], the Blind Classifier, identifies application protocol without looking at payloads. BLINC takes an omniscient view of the network, and uses social and functional characteristics of network traffic[5] to determine application protocol. PISA [30], another payload-blind classifier, uses k-means clustering in an eleven-dimensional space to identify application protocols.

[4]Their study has no notion of "recall": every flow was classified correctly, otherwise, it was marked as wrong (even if unknown).

[5]Number of ports for incoming and outgoing traffic, number of IP addresses, and global topology.

CHAPTER 3

NETWORK PROTOCOLS AND SOFTWARE VULNERABILITIES

Network protocols govern the exchange of information over computer networks. Trends in protocol design have mirrored broader trends in software engineering; as programming languages have come in and out of fashion, so have ideas about network protocol design. Economic factors have also made their mark: while early designs favored efficient on-the-wire messaging formats, today's protocols stress cross-platform portability and human readability.[1]

Protocol parsing for NIDS is a delicate juggling act. An effective NIDS must accurately recover protocol semantics in the face of multi-Gbps data rates, without adding noticeable latency to network flows (if the NIDS is on the forwarding path). This chapter discusses common idioms used by protocol designers, and follows with a review of several representative vulnerabilities.

3.1 Protocol Archetypes

As a preliminary step in designing a high-performance parser, we made a detailed study of the most popular Internet protocols. The study yielded many useful insights about the tradeoffs protocol designers face, as well as what kind of parser would be suitable for the general task of recovering protocol semantics (events, pieces of data, etc.) from a man-in-the-middle perspective. In the following section, we describe the four general design patterns encountered in our study.

[1]Most of the insights in this section apply equally to file formats, which (like network protocols) aim to communicate state from one instance of a program to another. Likewise, designing a network intrusion detection system shares many goals with a file-based virus scanner.

Struct-Style Protocols

The first class of protocol, characterized by packed, undelineated binary data, we call *struct-style* protocols. These protocols are basically `struct` types in C, cast to opaque `void*` buffers encapsulated in transport-layer segments for transit.

Struct-style protocols are compact. Like `struct` types in C, fields are implicitly labeled by their offset relative to the start of the structure, avoiding the need for explicit labeling. Binary types remain in their in-memory representation, rather than a human-readabe format (ASCII/Unicode). In the simplest case, sending application data over the network is as simple as copying a structure from memory into a transport-layer buffer, which the application hands off to the operating system for transit via a system call. Owing to the minimal work necessary to interchange application and network data, struct-style protocols are amenable to very high-performance implementation. Additionally, provided the endpoint application environments (programming language, compiler, machine architecture, etc.) are sufficiently similar, this kind of protocol is the most straightforward to program.

The benefits of struct-style protocols lean heavily on the network's (assumed) homogeneity. Unfortunately, the success of many early struct-style protocols meant they were used to network everything from microcontrollers to mainframes, a decidedly heterogeneous collection of computers. Sending raw binary data over a network makes assumptions about host byte ordering, word alignment, and word size, none of which are broadly standardized. Additionally, the benefits of `memcpy`-like data movement are lost on receivers, which must walk protocol data field-by-field to ensure data integrity (on pain of opening the system to compromise). Further, struct-style protocols are minimally extensible, and, owing to their use of C-style strings, do not handle variable-length fields well. Add the fact that binary formats are not human readable (requiring the use of analysis tools to inspect protocol messages), and it becomes clear why struct-style protocols are undesirable for all but legacy, and very high-performance applications.

Owing to their original implementation in the C programming language, many of the Internet's core protocols (e.g. IP, TCP, DNS) are struct-style.

Unfortunately, the difficulties of implementing binary protocol parsers by hand has led to many code defects; DNS, in particular, has been plagued with a long history of exploitable code defects (e.g. [31], [32], [33]). Further complicating matters, the difficulties of parsing DNS apply to intrusion detection systems, making polymorphic exploits against DNS very hard to detect. The challenges of parsing struct-style protocols are addressed more fully in Chapter 4.

IETF-Style Protocols

The Simple Mail Transfer Protocol (SMTP), Hypertext Transfer Protocol (HTTP), and File Transfer Protocol (FTP) fall into the second group of protocols, which we call *IETF-style* after the committee that designed them.[2] These protocols feature production rules identified by English-language ASCII keywords, which make them human-readable. IETF-style protocols deliberately mimic context-free languages; like CFLs, IETF-style protocols feature variable-length production rules, and grammars defined using Augmented Backus-Naur Form (see, for example, RFC 2616 [34]). Parsing context-free languages is a well-studied topic in computer science; by designing protocols this way, the IETF accelerated application development by allowing reuse of insights from compiler design in network application programming.

By virtue of their CFL-like production rules, IETF-style protocols offer great extensibility without the complexity of struct-style protocols. Variable-length parameters and multipart messages are both well-supported in the IETF paradigm. Also, IETF-style protocols rely on simple ASCII sequences to identify production rules and delineate fields, which increases portability and human-readability. The chief disadvantage of these protocols is the heavyweight translation layer required to send and receive data to/from the network.

It is worth noting that all classes of attacks against websites—SQL injection, cross-site scripting, cross-site request forgery, etc.—use HTTP, an IETF-style protocol. The ability to extract and apply constraints to HTTP GET/POST parameters at line rate is a key enabler in the battle against

[2]The Internet Engineering Task Force

web-based attacks.

Structured Binary Protocols

The third class of protocol, *structured binary*, combines some characteristics of struct-style with IETF-style design. The archetype of this style is Abstract Syntax Notation 1 (ASN.1 [35]). ASN.1 finds use in the Secure Sockets Layer (SSL) protocol [36], the Simple Network Management Protocol [37], X.509 [38], LDAP [39], and some proprietary interbank financial networks.[3]

Strictly speaking, ASN.1 is a machine-independent grammar used to specify the structure of binary messages; ASN.1 encoding rules[4] specify how to serialize and deserialize network data. Like IETF-style protocols, ASN.1 messages have nodes, which can be arranged flexibly; the serialization rules permit variable-length data fields and arbitrary nesting of child nodes.

ASN.1 message nodes are arranged in *Type-Length-Value* (TLV) form: each node has its numerically assigned type and length prepended. With each node's length prepended, a fully optimized parser can offer fast sequential lookup by skipping uninteresting nodes.

Structured Text Protocols

The final class of protocol is *structured text*. Similar to structured binary protocols, structured text protocols use a tree-like structure, but lack length-prefixing. The most heavily used structured text protocol today is the Extensible Markup Language (XML), and the related suite of XHTML/DHTML technologies used to author webpages.

XML is the newest of the data formats discussed in this section, and represents the extreme of making data formats user-friendly at the expense of efficiency. XML documents are tree-like with very loosely constrained structure; the format permits variable-length child nodes, to arbitrary

[3]Sun XDR is another, albeit less popular, example of a structured binary protocol.

[4]ASN.1 provides several of these "encoding rules," among them BER, the "Binary Encoding Rules," and DER, the "Distinguished Encoding Rules." One of the major complaints users have about ASN.1 is that it fails to take a single position on how messages should be represented on the wire.

depth. XML also offers robust support for international character sets with Unicode, forcing parsers to handle variable-length character encodings (e.g. UTF-8 [40]). On the other hand, XML documents are very verbose, which limits how much semantic content a parser must extract at a given data rate.

To date, XML *per se* has not been a fertile ground for software vulnerabilities. While some vulnerabilities have manifested in XML parsers, the data format itself (like any other protocol) is merely a vehicle by which data flows between endpoints. However, application writers are continually demanding greater flexibility, human readability, and support for internationalization in their protocols and data formats. As these challenges become more important in Internet engineering efforts, we believe the design of future data formats will trend toward XML, especially given its support for internationalization.

The Failure of Regular Expressions

In this work, we define a *vulnerability* as a defect in a piece of network-facing software which offers some potential for misuse. An *exploit*, then, is a particular way of misusing a vulnerability. Parsing for intrusion detection involves a fundamental tradeoff between accuracy and performance.

To this author's knowledge, there is no intrusion detection system which can perform full parsing at 1 Gbps or greater on commodity hardware. On the other hand, some intrusion detection systems have sidestepped full parsing by, for example, searching for static strings known to correspond to a particular exploit in a network flow. Fast (sometimes hardware-accelerated) static string searches and regular-expression matchers have been built, and while these systems can identify particular instances of an exploit (e.g. the Code Red Worm's [41] exploit against the CVE-2001-0500 vulnerability [42]), their inability to recover protocol semantics generally, hinders their ability to look past different exploits which misuse the same vulnerability. In the case of an HTTP-based exploit, even trivial rearrangement of GET/POST parameters can confound a static string matcher.

Building an effective NIDS requires a parser. While some techniques (Protomatching [15], VESPA [20]) can optimize the workload, trying to shortcut around parsing allows an attacker to evade detection with even a small amount of polymorphism in their attacks.

3.2 Vulnerabilities

This section presents the details of several remotely exploitable software vulnerabilities. The vulnerabilities are chosen to illustrate the breadth of detection complexity, from the trivial to the very difficult.

3.2.1 CVE-2002-1368: CUPS Negative Content-Length Vulnerability

CVE-2002-1368 [43] was triggered by incorrect handling of a signed integer in HTTP headers. The code defect led to a remotely exploitable denial-of-service vulnerability in the Common Unix Printing System (CUPS) implementation of the Internet Printing Protocol (IPP) [44].

As noted in the previous section, Hypertext Transfer Protocol (HTTP) is an IETF-Style protocol. A HTTP session begins with a request from the client, followed by a response by the server. The client initiates the exchange by writing a request similar to what follows over a transport-layer socket (ellipses represent text omitted for brevity):

```
GET / HTTP/1.1
Host: localhost:80
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X (...) )
Accept: application/xml,application/xhtml+xml,text/html (...)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

As the reader can see, HTTP requests consist of a command (`GET`), followed by a path (`/`) and protocol version (`HTTP/1.1`).[5] In the example

[5]No translation tools were used to "decode" the above message; the message format is flat text (as shown), with ASCII-encoded text and TCP for reliable transport.

given above, the client has appended request metadata in the form of textual key-value pairs.

When a client submits a job to CUPS for printing via IPP, the client sends the job to the CUPS daemon using an HTTP POST request. The HTTP POST header includes a `Content-Length` key-value pair, which specifies the number of octets in the POST request, e.g., `Content-Length: 312`. A message cannot have negative length; the shortest possible message (the empty message) would have length zero. Unfortunately, some versions of CUPS parsed negative `Content-Length` "correctly" (they did not signal an error), and subsequently wrote a negative value into a signed integer within the server process.

After parsing the length, CUPS passes the signed quantity to `malloc()`. Since `malloc()` takes an unsigned argument, passing a signed `int` to `malloc()` results in an improper implicit cast. Due to two's complement encoding, the `malloc()` call would interpret the "negative" content lengths as a very large unsigned quantity, causing an error.[6] On failure to allocate the huge amount of memory associated with the "negative" value, the `malloc()` call fails, returning a null pointer to the application. When dereferenced, the null pointer causes a segmentation fault, crashing the process.

Exploits against CVE-2002-1368 conform to a regular language. The NIDS protocol parser must be able to recognize complex exploit attempts, but also simple exploits (such as this one) with high performance.

3.2.2 CVE-2002-0063: CUPS Attribute Length Bug

CVE-2002-0063 exposed a remotely exploitable buffer overflow in the CUPS implementation of IPP [44]. When an IPP client issues a print job to CUPS via HTTP POST (see the previous section for details), the protocol permits the client to embed a variable number of "attributes" which specify details of the job (e.g. paper orientation). The client prepends the length of each attribute as an unsigned, big-endian 16-bit quantity.

[6]Two's complement is a convenient way of representing negative numbers, which allows digital arithmetic units to ignore the sign of their operands and still operate correctly. To convert a positive integer into its negative two's complement representation, simply complement all its bits, and add 1 to the result. A 32-bit `-1` is encoded as `0xffffffff`.

While 16-bit lengths permit a 64 KB (65,536-byte) attribute, CUPS declares a buffer of only 8 KB to hold the attribute values. A crafted request can write past the end of the buffer, allowing arbitrary code execution using standard buffer overflow techniques.

If the HTTP message carrying the IPP request were constrained to arrive contiguously, a regular expression could detect an exploit attempt by scanning for an attribute length over 8192 bytes. However, HTTP provides functionality to "chunk" POST data, which permits splitting a POST request across several discontinuous blocks. To detect exploits against CVE-2002-0063 in the presence of chunking, the protocol parser must scan across chunk boundaries, which is beyond the capabilities of an unmodified regular expression engine. Understanding (and potentially parsing) HTTP syntax is necessary to detect exploits against CVE-2002-0063 effectively.

3.2.3 DNS Pointer Cycles

The DNS protocol [45], a struct-style protocol, is used to resolve domain names to IP addresses on the Internet.

The overall structure of DNS is a multiply rooted tree, where each organization controls its own *zone of authority.* As one traverses downward through the tree, a *domain name* is constructed right-to-left, using textual identifiers called *labels.* Domain names have no explicit, textual representation in DNS proper; however, the convention of using periods to delineate labels has led to the familiar notation of a hostname: `tindalos.crhc.uiuc.edu`. Note that, owing to DNS's tree-like structure, hosts near each other in the tree share a common suffix.

To conserve bandwidth, DNS includes a mechanism to avoid duplicating a common suffix within a single message. When writing a DNS response, the application writes the full hostname the first time it appears—but after the name appears in full, subsequent names with a common suffix can refer back to it using a binary sentinel followed by a numeric offset.

DNS permits names with dozens of labels. Unfortunately, allowing such long names allows an attacker to exploit a DNS parser by crafting packets with very deep recursive cycles. At best, such packets can slow DNS parsers to a crawl, or worse, run the parser out of stack space, causing it to crash.

To detect a pointer cycle, the NIDS protocol parser must fully walk each DNS packet that enters the network, and check it for cycles. Performing such thorough analysis is a daunting task. The prevalence of DNS, combined with the high performance cost of analyzing it, makes DNS performance a crucial test of NIDS performance.

3.2.4 CVE-2005-4560: WMF SetAbortProc() Bug

Windows Metafile (WMF) [46] is a vector graphics format developed in the early 1990s by Microsoft. The format permits specification of a series of Windows GDI[7] calls, which when replayed, draws the contents of the file onto a GDI rendering surface. The format offered the standard benefits of vector graphics formats, including small file size and the ability to resize without pixelation.

When WMF was designed, cooperative (non-preemptive) multitasking was prevalent. Also, graphics files were not as frequently interchanged as today, making security less of a concern. In any case, the 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 victims computer simply by convincing them to render a WMF, requiring only a web site visit for clients using Internet Explorer [47].

Like DNS pointer cycles, this vulnerability is hard to detect because it requires scanning a variable field-length binary structure for a particular pattern of bytes. The WMF SetAbortProc() bug is even more difficult to detect, due to (1) the huge number of protocol exchanges which could potentially embed a WMF (e.g. HTTP, instant message chats, email, etc.), and (2) the length of WMF files, which can reach into the tens, or even hundreds, of kilobytes in length.

[7]The Graphical Device Interface, the API Windows exposed to userspace programs for doing drawing.

3.3 Forward

As this section has shown, protocol design embodies many tradeoffs. Older protocols tend to be more compact, whereas newer ones, while more "wasteful," favor ease of human comprehension, and extensibility.

Detecting vulnerabilities is intimately related to protocol parsing. While it seems attractive to use simple primitives, such as static string searches and regular expressions, to detect vulnerabilities, these mechanisms are trivially evaded by even the simplest of polymorphic exploits.

CHAPTER 4

VESPA: THE VULNERABILITY SIGNATURE PARSER

In this section, we introduce VESPA, an architecture for high-speed protocol parsing.

4.1 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 use 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 [13]. 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.

4.2 Design

This section presents the design of the parser.

4.2.1 Background

Vulnerability signatures were originally proposed by Wang et al. [48] 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 [49]:

[1]Vulnerability Signature Parsing Architecture

```
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 [41]; 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 [48] is:

```
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 [50]. 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 [51]).

4.2.2 High-Level Objectives

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.

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 [7, 52] and hardware optimizations [53] that could be employed by our system. Furthermore, hardware-accelerated regular expression matching is also becoming a reality [26]. 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.

4.2.3 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.

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 [54]). 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 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.

```
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  }%
}
```

Figure 4.1: Sample Specification for HTTP Requests (Simplified)

We do not yet address the problem of protocol detection. However, our system can be integrated with prior work [55] 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.

4.2.4 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 4.2.4.

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.

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.

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.

4.2.5 Implementation

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" [56] 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 insertion.

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 superclasses. The class definition consists of two files (following the example above, `myparser.h` and `myparser.cc`), which jointly specify the generated parser subclass.

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 [21], which efficiently implements the Aho–Corasick (AC) algorithm [7]. 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 [57].

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.

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.

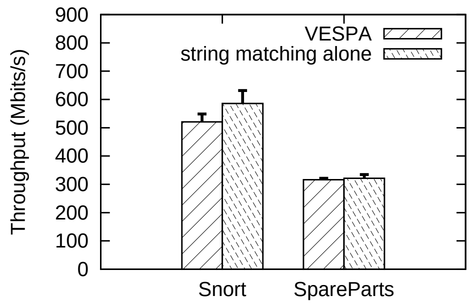
4.2.6 Evaulation

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. As Cui et al. did with GAPA [58], 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 to develop a binary parser for the Windows Meta-file (WMF) format.

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 4 GB 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 web sites. 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.

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



(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

Figure 4.2: Micro-benchmarks

Aho–Corasick (AC) algorithm and compared their performance (Figure 4.2.6a). We used the sfutil library, which is part of the Snort IDS [21], and the Spare Parts implementation of AC [57]. 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.

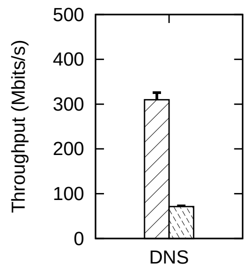
We next investigated the performance of binary traversal in binpac. One of the primary changes we made to binpac was to change its default memory and allocation behavior. We instrumented the original version of binpac and a parser built with our binary traversal-optimized version to assess the effectiveness of this change (Figure 4.2.6b). 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.

Signature Matching Performance

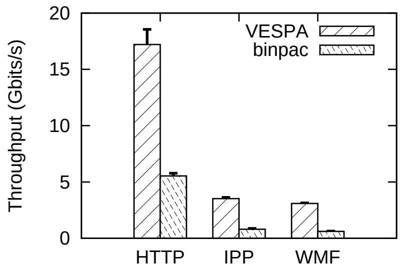
We evaluated the throughput of our vulnerability signature matching algorithms compared to the binpac parser generator. Binpac is the most efficient freely available automated protocol parser generator. We do not evaluate against GAPA because it has not been publicly released. Furthermore, binpac far exceeds GAPA in performance because it directly generates machine code rather than being interpreted [13]. 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 4.2.6.

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 [54] 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` 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



(a) DNS Throughput (Mbps)



(b) Parser Throughput (Gpbs)

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

(c) HTTP Message Rate

Figure 4.3: Vulnerability Signature Matcher Performance

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 web site (Figure 4.2.6c).

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. 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 [59].

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 4.2.6. 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 [13]. 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. 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 web site visit for clients using Internet Explorer (CVE-2005-4560 [54]).

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 4.2.6 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.

CHAPTER 5

SYSTEMS ISSUES

A network intrusion detection system (NIDS) has three principal modules:

- A mechanism for retrieving link-layer data from a file or network interface, and transforming it into a format the rest of the system can understand (which might involve TCP reassembly)
- Protocol parsers, which transform an application-layer flow into a series of semantically rich "application events" (e.g. login, retrieving a file, sending a message, etc.)
- A policy engine, which analyzes the event stream and looks for abnormalities.

Our vulnerability signatures work demonstrated the redundancy of full protocol parsing: recovering the *entire* application event stream is useful only insofar as the policy engine cares about every protocol event, which it normally does not.

Although our technique demonstrated aggressive parse speedup, we made no attempt to study how a faster parser would affect overall system performance. Accurately characterizing IDS performance requires carefully considering a multitude of factors, such as the nature of the traffic (protocols, packet sizes, fragmentation, etc.), machine characteristics (interconnect bandwidth/latency, processor characteristics, etc.), and how the software interacts with the hardware. Absent a thorough understanding of how these factors interact, micro-optimizing one part of the system risks running afoul of the cardinal principle of performance engineering: *always base optimization decisions on data.*

The experiments in this chapter use the Bro IDS [19], the system produced by the Networking Group at the International Computer Science Institute. Recently, the group has produced a cluster- and thread-based version of Bro, but both are still very experimental. We present two results. First, we found that protocol parsing does not dominate the IDS workload. Second, we learned that on a machine with a smaller cache, per-code file processor time and cache misses are more highly correlated than on a machine with a larger cache, which suggests that explicit management of the memory hierarchy must be a first-class design issue in next-generation NIDS.

5.1 Sensitivity Analysis

As the introduction to this chapter explained, accurately characterizing IDS performance is a high-dimension problem. In lieu of trying to infer the entire n-dimensional performance surface of the system, another technique is to use *sensitivity analysis*, which studies how the system reacts locally to changes in a subset of its parameters.[1] As shown by the vulnerability signatures work, the variable-length, recursive binary structure of the Domain Name System (DNS) protocol makes it especially difficult to parse, making it an ideal candidate to (1) measure sensitivity to complexity, and (2) establish an upper bound on the amount of time a NIDS spends parsing protocols.

We developed a tool, DNS-THRASH,[2] which uses a randomized traffic model to construct syntactically valid, but hard to parse DNS traffic. A notable feature of the tool is its ability to create deep "waterfalls" of recursion within a DNS packet, where names recursively point to other names with arbitrary depth (our experiments used a maximum recursion depth of eight).

5.1.1 Variable-Length DNS Messages

This experiment was our first attempt to measure runtime sensitivity to parse complexity.

[1]Many of the insights in this section resulted from two particularly good courses offered by Profs. David Nicol and R. Srikant given over the 2008-2009 academic year at the University of Illinois.

[2]Available via the author's web site, http://davidralbrecht.com.

The experiment used the standard (single-threaded) distribution of Bro 1.4, with debugging disabled. The traces used for each trial contained one million UDP segments, each containing a structurally identical (yet different) DNS message. The number of response records (RRs) in each packet was varied from zero to eight. Each response record recursively pointed to the record before it, giving a maximum recursion depth of eight.

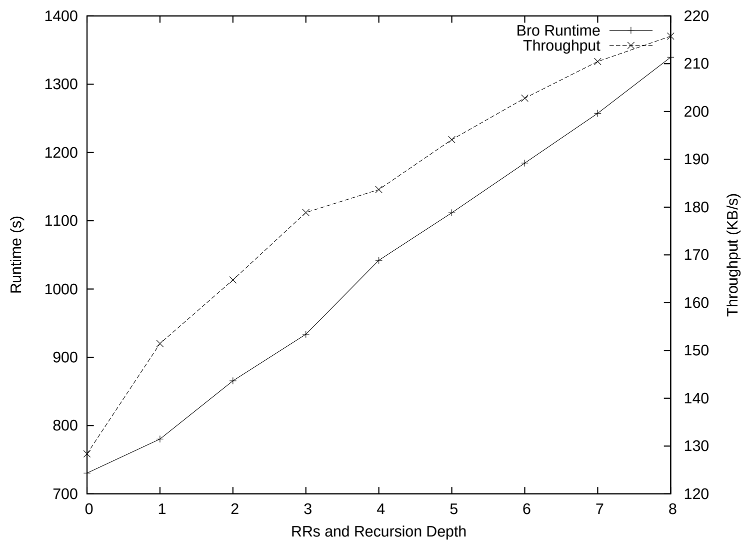


Figure 5.1: Variable-Length DNS Messages: Runtime and Throughput

Figure 5.1 shows that, as predicted, overall runtime increases with greater packet complexity. On the other hand, it is important to note that *the DNS messages in this experiment contain progressively more data*, leading to a curious result: while runtimes increase, overall system bandwidth actually *increases* as packet complexity rises, which suggests that *protocol parsing does not dominate the overall workload*.

5.1.2 Constant-Length DNS Messages

Building on the results of the last experiment, we designed a second experiment which used constant-sized DNS messages. This experiment used a configuration identical to the first experiment, except the number of response records was kept constant at 55 (the maximum number that would fit into a UDP segment); we varied only the depth of the recursion used, from one to eight.

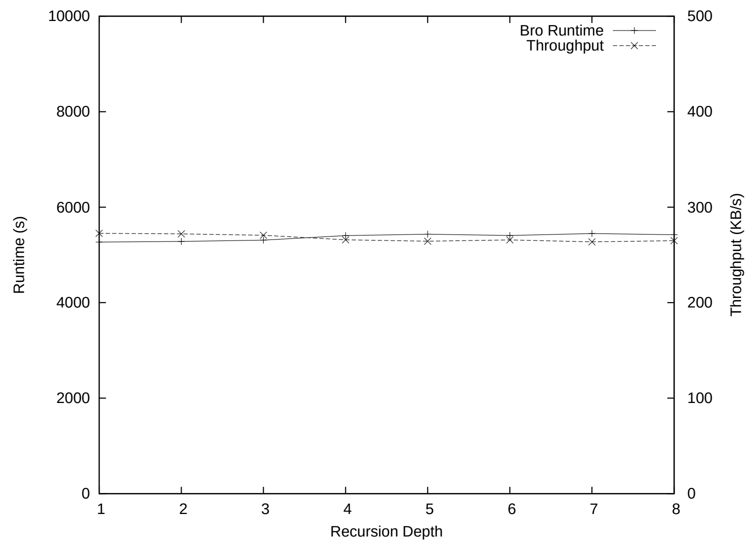


Figure 5.2: Constant-Length DNS Messages: Runtime and Throughput

Figure 5.2 shows a positive, but very slight, change in overall runtimes as complexity increases. Clearly, the variable parse complexity of the DNS packets does not dominate the workload.

5.1.3 Profiling Runs

Working from the insights gained in the DNS experiments, we dug deeper using Shark, a sampling profiler. In a third flight of tests, we used the same configuration as the constant-length DNS messages, while running Bro under Shark. In the most complex configuration (55 reponse records with eight-deep recursive nesting), we observed the binpac-based DNS parser consuming a maximum of 16.5% of the CPU time.[3] Based on this result, and the observation that DNS is traditionally one of the most difficult protocols to parse, we believe 16-17% CPU time constitutes a reasonable upper bound for the time the system spends parsing protocols. After seeing this result, it became clear that future efforts should focus on other areas of the system than protocol parsing.

[3]We performed ten trials, observing similar results with each trial.

5.1.4 Other Approaches

Dreger et al. [60] also studied resource usage of NIDS, but they focused on system flow capacity, rather than analyzing the system's workload. Similar to our work, Schaelicke et al. [61] studied resource usage on a single machine by a NIDS, but their approach assumed a single-threaded system. "In particular," Schaelicke notes, "processor speed is not a suitable predictor of NIDS performance, as demonstrated by a nominally slower Pentium-3 system outperforming a Pentium-4 system with higher clock frequency. Memory bandwidth and latency is the most significant contributor to the sustainable throughput."

5.2 The Hardware Mismatch

Since its inception, the semiconductor industry has delivered greater application performance with every product cycle. Despite sweeping microarchitectural and process changes, the basic programming model has not changed: a single von Neumann processor, executing instructions at ever-greater speed, with ever-increasing amounts of memory.

Memory latency and parallelism represent two threats to the ongoing hardware-agnosticism of software developers. While processor speeds have scaled at near-exponential rates, memory manufacturers have failed to deliver comparably accelerated access latencies. The widening gap between memory and execution speeds has necessitated complex multilevel caching strategies, which attempt to address the main memory's inability to keep up with the CPU's need to load and store data [61]. In addition, hardware parallelism represents a break in how processor performance has scaled: instead of delivering a single, faster processor, manufacturers have turned to many-core architectures to continue the march toward greater instruction throughput.

Despite intense research efforts, developer tools (languages, compilers, etc.) have yet to emerge which address these concerns, while retaining the simplicity of today's general-purpose, single-threaded languages. By continuing to use design idioms and tools which assume a single-threaded, flat-memory machine, developers risk stagnant performance even as hardware moves forward.

5.2.1 Preliminary Benchmarks

Porting an existing codebase of Bro's size to a multithreaded programming model is quite challenging. On one hand, the developer must exercise care to ensure data consistency; in the ideal case, several separate pieces of the program can execute independently, with minimal interaction. In most cases, however, synchronizing access to data structures involves locking, which forces threads to wait when they need access to a shared resource. Waiting on locks can cause sublinear scaling, or in some cases, even degrade overall performance, relative to the single-threaded version of the program.

Table 5.1: Mean runtimes in seconds over ten trials, with standard deviation in parens. Adding threads does not necessarily improve performance.

Scripts	0	1:1	1:2	1:2,3
tcp	93 (1)	77.76 (2)	67 (1)	64 (1)
mt	181 (72)	172 (58)	140 (62)	142 (44)
scan, trw	196 (94)	173 (8)	135 (5)	122 (8)
udp, icmp	1930 (48)	2946 (53)	2197 (44)	2169 (448)
http	2579 (70)	3929 (80)	2920 (54)	2600 (34)
ssh, pop3, irc, smtp	3530 (110)	5508 (122)	4028 (90)	3676 (105)

The results given in Table 5.1 were averaged over ten runs of superlinear Bro on hatswitch, a dual-processor AMD Opteron machine with 4 GB of RAM. Superlinear Bro always uses one "main" thread to reassemble TCP flows but allows creating additional parser and event engine threads. Thread configurations are specified by listing the processors (cores) on which additional parser threads should run, then a colon, and the cores on which event engine threads should run. In thread configuration "0", all tasks run in the same thread, on the same CPU.

Script complexity is cumulative; each trial adds to the complexity of the one before it. The "tcp" script reassembles TCP flows, and logs them to a file. The "mt" script adds analysis of several rudimentary internet services (finger, ident, ftp, ntp, tftp, and the RPC portmapper), "scan" performs rudimentary scan checks, and "trw" looks for scans using a more sophisticated threshold random walk algorithm.

5.2.2 The Memory Hierarchy

Memory bandwidth and latency are crucially important to program performance. Traditionally, software designers left memory access decisions up to the compiler, operating system, and hardware, but it is unclear whether this will be possible moving forward.

We begin with an experiment showing superlinear Bro running on rinseng, a machine with two Intel E5405 processors. The E5405 features two pairs of two cores, with a 6 MB shared cache between each pair of cores. The experiment measured the Bro runtime on a 700 MB trace of enterprise traffic captured at the University of Illinois Coordinated Science Laboratory. We measured the number of lowest-level (closest to memory) cache accesses and misses using oprofile, and cachegrind.

Cachegrind [62], part of the Valgrind project, uses an x86 virtual machine to simulate program execution. As it is a simulator, Cachegrind is designed to give consistent results across multiple trials—but each trial took 10-20 times as long as a native run. Also, our experiments show that Cachegrind's results significantly departed from those measured on actual native runs.

Oprofile [63] works by instrumenting the Linux kernel. The tool measures system events (cache events, machine faults, etc.) using on-die performance counters. Oprofile gives extremely accurate results, but at the expense of sensitivity to parameters which may be beyond the system designer's control (how the program interacts with libraries and the operating system, etc.) Further, optimizing to oprofile risks over-tuning to the particular machine architecture, in which case speedups will not readily transfer to different architectures.

Table 5.2 compares Cachegrind to oprofile results; the experimental configuration is the same as that in Figure 5.1. Table 5.3 gives the same data, but in percentage form.

With further study and analysis, we determined that the observed disparity between oprofile and Cachegrind was attributable largely to thread scheduling. Cachegrind simulates multi-threaded programs using very fine interleaving, and does not account for the differences in machines with shared vs. split caches, instruction interleaving from multiple cores, or cache misses not visible at the instruction level (e.g. translation lookaside misses).

Table 5.2: Cachegrind vs. oprofile L2 results. Results mostly agree, but differ markedly on the 1:1 and 1:2,3 trials.

Scripts	Threads	cg L2 refs	cg l2 misses	op L2 refs	op L2 misses
tcp	0	92.8 M	116 K	86.31 M	390 K
	1:1	75 M	1.18 M	42.98 M	0.95 M
	1:2	78.3 M	213 K	74 M	5.72 M
	1:2,3	77.92 M	206 K	85 M	12 M
tcp, mt	0	171.8 M	292 K	166.5 M	730 K
	1:1	133 M	3.4 M	69.4 M	1.83 M
	1:2	130.8 M	3.5 M	130 M	7.50 M
	1:2,3	132 M	5.12 M	152 M	19.89 M
tcp, mt, scan, trw	0	92.8 M	116 K	86.31 M	390 K

Table 5.3: Cachegrind vs. op L2 miss %.

Configuration	thread conf	cg L2 miss %	op l2 miss %
tcp	0	1.25	0.45
	1:1	1.57	2.21
	1:2	0.27	7.72
	1:2,3	0.26	14.12
tcp, mt	0	0.17	0.43
	1:1	2.55	2.63
	1:2	2.68	5.76
	1:2,3	3.88	13.09
tcp, mt, scan, trw	0	0.13	0.45

For our final experiment, we sought to determine whether adding a larger cache would mitigate the effects of misses on runtime. We used the same experiment design as the prior L2 miss study, and set oprofile to measure (1) L2 misses, and (2) time samples, over every file in the entire Bro superlinear codebase. We treated these two quantities (time and misses) as two random variables, and calculated their correlations over two machines (specify machine configurations).

Our results are very preliminary; however, using a scheme for statically mapping flows to CPUs (based on a hash of the connection key), we found correlations of $\rho = 0.6081$ on hatswitch[4], and $\rho = 0.4926$ on rinseng[5]. Likewise, using round-robin hashing, we found the time/L2 correlation to be $\rho = 0.7713$ on hatswitch, and $\rho = 0.5067$ on rinseng. These results

[4]A dual-processor, four core AMD Opteron machine with 4GB RAM.

[5]A dual-processor Intel Xeon E5405 machine with 32 GB RAM.

should not be treated as definitive, as we held the type of traffic constant, and instrumented only at the granularity of code files (not individual instructions). However, we do feel these results support further inquiry into the relationship between cache behavior and runtime performance.

CHAPTER 6

FUTURE DIRECTIONS

This section outlines the architecture of an envisioned next-generation intrusion-detection system, and identifies problems for further study.

6.1 Future Architecture

Over the course of our work, it became clear that fine-grained object-orientation is not appropriate for bulk packet processors. Bro, in particular, creates a new object corresponding to every stack frame in its interpreted policy language. Additionally, a major source of speedup in VESPA is its avoidance of object creation and deletion overhead. In our DNS experiments, we noticed Bro spending more than 25% of total CPU time in object constructors and destructors.[1]

Further, we believe that a next-generaion IDS must address the two major hardware challenges identified in the last section: (1) the proliferation of multicore processors, and (2) the growing cost of moving data to and from main memory. Future IDS designs, therefore, will require fine-grained threading, and explicit management of how data moves through the system. These two concerns must be addressed simultaneously; addressing locality requires knowledge of when data-dependent code executes. The eventual result will require a coordinated effort between operating system designers (locality-aware algorithms for thread and process scheduling), processor architects (cache and memory design), compiler authors (to provide the right abstractions for this new programming model), and software engineers, who will need to learn to program a machine radically different from the traditional, single-threaded, flat-memory model machine offered

[1]This figure includes *malloc()* and *free()* calls, but even still, we believe it is unreasonably high.

by the C programming language (and its descendants).

Stream processing is a step in the right direction. Projects such as the Click Modular Router [64], MIT Streamit [65], and IBM's System S initiative approach computation as a "software pipeline," where the workload is decomposed into a number of small, simple functional kernels. Data flows into the pipeline where it is operated on and transformed by the functional kernels, and eventually is logged or discarded at the end of the pipeline. Ideally, a streaming system's functional kernels map nicely onto operating system threads, exposing as much concurrency to the hardware as it can handle. Additionally, constraining how data moves through the system allows optimizing how the kernels are arranged on the processor, and how data moves from memory into the cache hierarchy.

Constructing a streaming NIDS would require a lot more work. For one, streaming systems assume a predictable flow of data through the inter-kernel communication channels, which is hardly the case with bursty network packets. Additionally, the stages of an IDS pipeline are far from a streaming system's "computation kernels"; they are functional units, many of which require session state tracking. Nevertheless, the potential for compiler-guided data flow management (including when and how threads are scheduled to execute) seems too huge to ignore.

6.2 Other Concerns

This section explains some of the challenges encountered in the course of this work. In the author's opinion, any of these would make a great topic for future research.

The need for data. A persistent problem in networking research is the lack of real-world data for experimentation. Header-only traces are available, but in general, privacy policies bar system administrators from recording traces, even for research purposes.

The security community has extensively studied trace anonymization, in hope of striking some balance between users' right to privacy, and the need for legitimate scientific research to advance the state of the art. The University of Illinois' excellent IT department, CITES (Campus Information Technologies and Educational Services), described a past project to set up a "vault," where researchers could perform experiments, but were limited in what they could remove from the system after their work.[2] With well-articulated policies and good enforcement, such a "data vault" could offer a fair middle ground between users and researchers.

Mapping a sequential workload onto parallel hardware. The parallelism movement in computer architecture is actively studying this problem, but security presents its own challenges. Load-balancing schemes must be robust against resource depletion attacks. Also, determining how to prevent resource starvation by attackers in multi-threaded code generally will become important as more code is parallelized.

Ambient authority. n-tier applications, such as database-backed web sites, are riddled with ambient authority. Database administrators routinely bypass database security by using superuser accounts for daily operation. Many web attacks, such as SQL injection, would be much harder if existing protection mechanisms (operating system security, database security, etc.) used n-tier application principals directly, rather than relying on application semantics to regulate user capabilities.

Where to place the NIDS. Many NIDS designers assume a single ingress/egress point for organizational Internet traffic, but this is increasingly not the case with personal VPN connections, single-site multihoming, and wide-area organizational intranets. In the past, sensors have been placed at Internet connection points, inside intranet infrastructure (e.g. LAN switches), and directly on user machines. A thorough study of the costs and benefits of each approach would be useful.

Design for parseability. When protocols like DNS were designed, computers were fast relative to their network interconnects; today, the situation is reversed. Even given today's extremely verbose protocols (such as XML), high-end servers struggle to keep pace with incoming traffic streams. To date, there has been no explicit study of how to design a protocol for high-performance parsing.

Special-purpose hardware. Although research IDS efforts focus on software systems, high-performance networking vendors such as Cisco and TippingPoint make extensive use of custom hardware. Custom hardware has been studied piecemeal, with some limited systems work (see Chapter

[2]Josh Stone, Mike Corn, and Roy Campbell worked on this initiative.

2), but further research could explore programming paradigms and tools for these relatively heterogeneous (or customized) platforms.

Forensic studies of intrusion. Over the course of this work, we came to believe that many security vulnerabilities were due to misconfiguration and operator error, not software defects *per se*. A study examining how real-word intrusions happened in the past could be invaluable for focusing future research efforts, as well as directing organizational resources to the highest-value uses, in terms of keeping intruders out.

What to do when misuse is detected? The standard action is a connection reset, followed by the insertion of a firewall rule which blocks further communication from the offending host. However, the author believes systems could go further, perhaps contributing to a distributed threat monitoring system.

CHAPTER 7

CONCLUSION

The state of the art in intrusion detection is inadequate. Signatures are too difficult to construct, evasion is too easy, and parsers cannot keep up with today's data rates. We wanted to understand what design choices would be necessary to enable rich semantic analysis (which hardens the system against evasion) at wire speed.

We began by studying protocol parsing, which required a thorough study of the protocols themselves. We discovered that, despite the diversity of protocols on the Internet, designers followed consistent idioms when constructing protocols. These idioms were influenced by hardware architectures, programming languages, and economic factors (e.g. the relative cost of bandwidth vs. compute time). Defects—whether in the protocols themselves, or in the way that software handles the protocols—give rise to exploits. Network protocol characteristics strongly influence what kind of exploits are available against a piece of network-facing software.

Once we familiarized ourselves with protocols, we turned our attention to protocol parsing. Parsing, in general, is well-studied in the computer science literature. Unfortunately, many of the traditional insights the literature offers (such as complexity classes) are not applicable to high-speed intrusion detection, a topic which leans heavily on how algorithms are implemented in hardware.

We studied the Bro Intrusion Detection System and found that binpac, Bro's protocol parser, parsed the entire application protocol exchange, regardless of whether the events were important to the policy engine. VESPA, our parser architecture, trades off hand-coded signatures for more than an order of magnitude speedup in protocol parsing. We attribute VESPA's gains to (1) its use of a flat, procedural data model (versus binpac's nested objects), and (2) eliminating redundant parsing by combining policy and parsing into a single module. VESPA's chief drawback is its difficulty of use; it required hand-coded signatures for each protocol/policy combination. However, we feel a valuable contribution has been made in identifying the potential for speedup using a different approach.

Beyond parsing, we noticed that both Bro and Snort—the two major research IDSes—expressed their application code in a flat, monolithic programming model. That model is not well-aligned with future trends in computer architecture, which suggests that it is time to rethink how intrusion detection systems are designed. The well-exposed parallelism and explicit dataflow modeling of streaming systems offers the potential to better utilize next-generation hardware, by letting the compiler make decisions about how the application should interact with the hardware.

Although we would have liked to implement a next-generation IDS, doing so was not within the scope of a master's project. We encourage further research to complete what we have left undone.

REFERENCES

[1] "Pirate Bay Served with Dutch lawsuit via Twitter and Facebook." [Online]. Available: http://www.thelocal.se/20244/20090624/

[2] "Court papers served over facebook." [Online]. Available: http://www.computerweekly.com/Articles/2008/12/16/233938/court-papers-served-over-facebook.htm

[3] The Economist, "The mouse that roared." [Online]. Available: http://www.economist.com/world/international/displaystory.cfm?story_id=E1_JSVNGNV

[4] C. Grier, S. Tang, and S. T. King, "Secure web browsing with the op web browser," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 402–416.

[5] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *IMC '09: Proceedings of the 2009 Internet Measurement Conference.* New York, NY, USA: ACM Press, November 2009, pp. 90–102.

[6] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal of Computation*, vol. 6, pp. 323–350, 1977.

[7] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[8] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[9] S. Wu and U. Manber, "Fast text searching: Allowing errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[10] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Cybernetics and Control Theory*, vol. 10, pp. 707–710, 1966.

[11] "PCRE: Perl-compatible regular expressions." [Online]. Available: http://www.pcre.org/

[12] "Flex: The Fast Lexical Analyzer." [Online]. Available: http://flex.sourceforge.net/

[13] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "Binpac: A yacc for writing application protocol parsers," in *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement.* New York, NY, USA: ACM, 2006, pp. 289–300.

[14] R. Smith, C. Estan, and S. Jha, "Xfa: Faster signature matching with extended automata," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 187–201.

[15] S. Rubin, S. Jha, and B. P. Miller, "Protomatching network traffic for high throughputnetwork intrusion detection," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2006, pp. 47–58.

[16] "Yacc: Yet Another Compiler-Compiler." [Online]. Available: http://dinosaur.compilertools.net/

[17] "Bison: Gnu parser-generator." [Online]. Available: http://www.gnu.org/software/bison/

[18] N. Borisov, D. Brumley, H. Wang, J. Dunagan, P. Joshi, and C. Guo, "Generic application-level protocol analyzer and its language," in *Proceedings of the 14th Annual Network & Distributed System Security Symposium*, 2007.

[19] V. Paxson, "Bro: A system for detecting network intruders in real-time," in *Proceedings of the 7th USENIX Security Symposium*, 1998.

[20] N. Schear, D. Albrecht, and N. Borisov, "High-speed matching of vulnerability signatures," in *Recent Advances in Intrusion Detection*, 2008, pp. 155–174.

[21] "Snort: An open-source network intrusion prevention and detection system by sourcefire." [Online]. Available: http://www.snort.org/

[22] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware," *Recent Advances in Intrusion Detection*, pp. 107–126, 2007.

[23] R. Sommer, V. Paxson, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion detection," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 1255–1279, May 2009.

[24] J. M. Gonzalez, V. Paxson, and N. Weaver, "Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2007, pp. 139–149.

[25] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high-speed networks," in *Proceedings of the 2004 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 249–257.

[26] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 191–202, 2006.

[27] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos, "SafeCard: A gigabit IPS on the network card," in *Recent Advances in Intrusion Detection*, 2006, pp. 311–330.

[28] A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *Passive and Active Network Measurement*, 2005, pp. 41–54.

[29] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel traffic classification in the dark," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA: ACM, 2005, pp. 229–240.

[30] R. Dhamankar and R. King, "Protocol identification via statistical analysis," 2007. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-07/Dhamankar_and_King/Presentation/bh-usa-07-dhamankar_and_king.pdf

[31] "Common vulnerabilities and exposures project: Cve-2007-0035." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0035

[32] "Common vulnerabilities and exposures project: Cve-2009-0025." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0025

[33] "Common vulnerabilities and exposures project: Cve-2008-1447." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1447

[34] "Hypertext transfer protocol – http/1.1." [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[35] ITU, "ITU-T Recommendation X.680." [Online]. Available: http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

[36] IETF, "Rfc 2246: The tls protocol." [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[37] IETF, "Rfc 1157: A simple network management protocol (snmp)." [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt

[38] ITU, "ITU-T Recommendation X.509." [Online]. Available: http://www.itu.int/rec/T-REC-X.509-200508-I/en

[39] "Rfc 4522: Lightweight directory access protocol (ldap): The binary encoding option." [Online]. Available: http://tools.ietf.org/html/rfc4522

[40] IETF, "Rfc 2279: Utf-8, a transformation format of iso 10646." [Online]. Available: http://www.ietf.org/rfc/rfc2279.txt

[41] S. Friedl, "Analysis of the New "Code Red II" Variant," www.unixwiz.net/techtips/CodeRedII.html, Aug. 2001.

[42] "Common vulnerabilities and exposures project: Cve-2001-0500." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2001-0500

[43] "Common vulnerabilities and exposures project: Cve-2002-1368." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1368

[44] IETF, "Rfc 2568: Rationale for the structure of the model and protocol for the internet printing protocol." [Online]. Available: http://tools.ietf.org/html/rfc2568

[45] IETF, "Rfc 1034: Domain names – concepts and facilities." [Online]. Available: http://www.ietf.org/rfc/rfc1034.txt

[46] Microsoft Corporation, "Windows metafile format (wmf) specification," 2007. [Online]. Available: \url{download.microsoft.com/download/0/B/E/0BE8BDD7-E5E8-422A-ABFD-4342ED7AD886/WindowsMetafileFormat(wmf)Specification.pdf}

[47] "Common vulnerabilities and exposures project: Cve-2005-4560." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4560

[48] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," in *ACM SIGCOMM Computer Communications Review*, 2004.

[49] CERT, "'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL," www.cert.org/advisories/CA-2001-19.html, Jul. 2001.

[50] Microsoft Corporation, "Unchecked buffer in ISAPI extension could enable compromise of IIS 5.0 server," www.microsoft.com/technet/security/bulletin/ms01-023.mspx, Jun. 2001, microsoft Security Bulletin MS01-033.

[51] E. Rescorla, "Security holes... who cares?" in *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[52] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Department of Computer Science, University of Arizona, Tech. Rep. TR-94-17, 1994.

[53] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Koné, and A. Thomas, "A hardware platform for network intrusion detection and prevention," in *Proceedings of the Third Workshop on Network Processors and Applications*, 2004.

[54] MITRE Corporation, "Common vulnerabilities and exposures." [Online]. Available: \url{cve.mitre.org}

[55] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "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, pp. 18–18.

[56] M. J. Dominus, *Higher Order Perl: Transforming Programs with Programs.* San Francisco, CA: Morgan Kaufmann, 2005.

[57] B. W. Watson and L. Cleophas, "SPARE Parts: a C++ toolkit for string pattern recognition," *Softw. Pract. Exper.*, vol. 34, no. 7, pp. 697–710, 2004.

[58] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto, "ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007, pp. 252–266.

[59] NISCC, "Vulnerability advisory 589088/NISCC/DNS." [Online]. Available: www.cpni.gov.uk/docs/re-20050524-00432.pdf

[60] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Predicting the resource consumption of network intrusion detection systems," in *Recent Advances in Intrusion Detection*, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., 2008, pp. 135–154.

[61] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland, "Characterizing the performance of network intrusion detection sensors," in *Recent Advances in Intrusion Detection*, G. Vigna, E. Jonsson, and C. Kruegel, Eds., 2003, pp. 155–172.

[62] "Cachegrind: A cache and branch-prediction profiler." [Online]. Available: http://valgrind.org/docs/manual/cg-manual.html

[63] "Oprofile: A system profiler for Linux." [Online]. Available: http://oprofile.sourceforge.net/

[64] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[65] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.