Protomatching Network Traffic for High Throughput Network Intrusion Detection

Shai Rubin
Computer Sciences Dept.
University of Wisconsin, Madison
shai@cs.wisc.edu

Somesh Jha
Computer Sciences Dept.
University of Wisconsin, Madison
jha@cs.wisc.edu

Barton P. Miller
Computer Sciences Dept.
University of Wisconsin, Madison
bart@cs.wisc.edu

ABSTRACT

Before performing pattern matching, a typical misuse-NIDS performs protocol analysis: it parses network traffic according to the attack protocol and normalizes the traffic into the form used by its signatures. For example, consider a NIDS that attempts to identify an HTTP-based attack. The NIDS must extract the URL from the raw traffic, convert HEX encoded characters into their equivalent ASCII form if necessary, and only then perform matching on the normalized URL. Protocol analysis is time consuming, especially in a NIDS that analyzes and normalizes all traffic just to discover that the majority of the traffic does not match any of its signatures.

We develop a technique called *protomatching* that combines protocol analysis, normalization, and pattern matching into a single phase. The goal of the protomatching signatures is to exclude non-attack traffic quickly before the NIDS performs any further time-consuming analysis. Protomatching is based on a novel signature with two properties. First, the signature ensures that the attack pattern appears in the context that enables successful attack. This saves the need for protocol analysis. Second, the signature matches both encoded and normalized forms of an attack and this saves the need for normalization.

We empirically show that a Snort implementation that uses protomatching is up to 49% faster than an unmodified Snort.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Metrics—*invasive software,unauthorized access.*

General Terms

Security, Management.

Keywords

Intrusion detection, signatures, protocol analysis.

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

A misuse network intrusion detection system (NIDS) recognizes an attack via a signature, usually a string or a regular expression that matches a characteristic pattern of the attack. For example, to identify an HTTP-based attack called *DNS-tools* (CVE-2002-0613), which enables attackers to modify DNS entries, Snort [32] checks whether the network traffic contains the string "`dnstools.php`". However, not all traffic that contains this string is a real DNS-tools attack. To correctly identify the DNS-tools attack, Snort must ensure that the string "`dnstools.php`" is part of a URL in an HTTP request.

To achieve such level of accuracy, Snort performs *protocol analysis* before pattern matching. It parses HTTP traffic, finds the URL in any HTTP request, and looks for "`dnstools.php`" only in those places. To the best of our knowledge, commercial intrusion detection systems (e.g., [4, 5, 14, 44]) also perform protocol analysis for many commonly used protocols, such as FTP, HTTP, and SMTP.

During protocol analysis a NIDS also performs *traffic normalization*. Some protocols, such as FTP or HTTP, enable *multiple encodings* for the same payload. Most notoriously, HTTP allows URLs to be encoded using lower- or uppercase characters as well as hexadecimal ASCII values [9, 10]. A NIDS typically translates, or *normalizes*, the raw traffic into the form used by its signatures. For example, Snort translates hexadecimal encodings in a URL into lower-case characters. A NIDS that does not normalize network traffic is vulnerable to evasion attacks [12, 20, 23, 29, 46].

All common NIDS seem to use the methodology outlined above, which we call *analyze-normalize-match* (ANM). First, a NIDS encodes its signatures in a *normalized* form. Then, during runtime, the NIDS parses the traffic according to the protocol the attack uses and normalizes the traffic, if necessary. Last, the NIDS matches the normalized traffic against its normalized signatures.

Unfortunately, the ANM methodology incurs performance penalty. An ANM-based NIDS inspects the traffic twice: once during protocol analysis and once during matching. An ANM-based NIDS wastes time analyzing and normalizing benign traffic only to discover later that this traffic does not match any of its signatures. Indeed, our experiments reveal that Snort spends about 30% of its time analyzing and normalizing benign HTTP requests. Dreger *et al.* [8] noticed similar results for Bro [25] and recognized that HTTP analysis is a serious bottleneck for sites with a high volume of HTTP traffic.

We propose to replace ANM with a more efficient tech-

System	Size (MB)	% speedup (compared to default Snort configuration)	% speedup (compared to custom configuration)
Original Snort (ANM)	7	–	–
Snort+deterministic protomatcher (P–ANM)	30	45	25
Snort+hierarchical protomatcher (P–ANM)	13	49	27

Table 1: Summary of results on real HTTP traffic with all Snort's HTTP signatures (999 signatures).

nique: *protomatch—analyze-normalize-match* (P–ANM). As we explain below, the *protomatching* phase inspects the network traffic exactly once, but still performs protocol analysis, normalization, and pattern matching. The goal of the protomatching phase is to quickly identify traffic, either normalized or encoded, that can **never** match any signature in the NIDS database. In comparison, traffic that **might** match a signature, is further analyzed using a more computationally expensive technique, such as the ANM. A NIDS that is based on the P–ANM methodology is efficient because it inspects 99% of the traffic only once, instead of inspecting 100% of the traffic twice. Our experiments show that a P–ANM-based Snort is up to 49% faster than Snort that uses the ANM technique.

The first contribution of this paper is the concept of a *protomatching signature*, a regular expression with two properties. First, the expression ensures that the characteristics pattern of an attack appears in the context that is necessary for the attack to succeed. For example, based on the HTTP specification, we build a protomatching signature for the DNS-tools attack that identifies the string "`dnstools.php`" only if this string appears in a URL. Hence, a protomatching signature saves the need for protocol analysis.

Second, a protomatching signature matches both normalized and encoded versions of an attack. To do so, we represent alternate encodings as *substitutions* [13]: operations that map a character to a regular expression describing all possible encodings of that character. Then, we build a signature that matches all possible representations of an attack. For example, we change the signature to match encoded variants, such as "`GET dnstools.%70h%70`", in which the character '`p`' is replaced with its hex encoding "`%70`".

Since a protomatching signature is a regular expression, we automatically compile it into a *protomatcher*: a deterministic finite state machine that matches every possible encoding of "`dnstools.php`" and ensures that this string appears only in a URL of a valid HTTP method.

Ideally, we would like to construct a *full-coverage protomatcher*, a protomatcher that contains a protomatching signature for every signature in the NIDS database. While we have built a full-coverage protomatcher for more than 50% of the 1022 HTTP signatures in Snort's database, we noticed that a full-coverage protomatcher cannot fit in 2GB of memory if it contains signatures that require complicated regular expressions. Therefore, we developed the *superset protomatcher* that requires a smaller memory footprint.

A *superset protomatcher* recognizes a superset of the traffic matched by a full-coverage protomatcher. For example, instead of recognizing the string "`dnstools.php`", the superset protomatcher recognizes the string "`dnstools`". This means three things. First, a superset protomatcher consumes less memory. Second, since it recognizes a superset of the traffic, it never misses traffic matched by the full-coverage protomatcher. However, it might produce *false matches*: traffic that matches the superset protomatcher but does not match any of the NIDS signatures. Third, traffic that does not match the superset protomatcher also does not match any signature in the NIDS database, so it can be immediately accepted as benign.

The superset protomatcher is the second contribution of this paper and the core of the P–ANM technique. Traffic, either normalized or encoded, that *does not* match the superset protomatcher is immediately ignored because it is benign. Traffic that *does* match the superset protomatcher is forwarded to the second phase: the traditional ANM technique.

This two-phase design is time-efficient because the superset protomatcher accepts 99% of the benign traffic and rarely utilizes the slower ANM-based path. This design fits into memory because the combination of superset protomatcher and the ANM-based matching consumes an order of magnitude less memory than a full-coverage protomatcher.

We discuss two possible protomatcher implementations. First, we implemented a protomatcher as a deterministic finite state machine. Second, to reduce the protomatcher memory footprint, we implement it as a *hierarchical protomatcher* that is based on two automata: a matcher and a normalizer. Unlike the ANM method that first fully analyzes and normalizes the traffic and then performs matching, the hierarchical protomatcher performs protocol analysis and matching until it encounters an encoded representation. Then, it passes control to the normalizer, which translates the encoding into a normalized form and returns it to the matcher. The hierarchical protomatcher consumes only 13MB and improves Snort's performance by 49%, with the default Snort configuration, and by 27% with our customized configuration (Table 1).

In summary, this paper makes the following contribution:

1. **The idea of a protomatching signature**. We developed a signature that matches both encoded and normalized variants of the attack and guarantees that the attack pattern appears in the right context.

2. **The idea of the superset protomatcher**. We show that when it is infeasible to build a protomatching-signature for every signature in the NIDS database, the idea of protomatching signatures can still be very beneficial. We use shorter protomatching signatures to build a superset protomatcher that filters 99% of the benign traffic and therefore omits the need for expensive protocol analysis.

3. **Feasibility study of the P–ANM methodology**. We implemented the P–ANM methodology in Snort. We show that a deterministic protomatcher improved Snort's performance by 45% when we use Snort's default configuration. Even after customizing Snort's configuration to maximize performance (Section 5), our implementation performed 25% faster. The hierarchical protomatcher consumed only 13MB and im-

proved Snort's performance by 49%, with the default configuration, and by 27% with our customized configuration (Table 1).

Since protomatcher-based Snort consumes more memory than the original Snort, it might suffer more cache misses. We investigated a *cache-poisoning* attack in which an attacker attempts to degrade the protomatcher's performance by forcing it to generate many cache misses. While this attack degrades the protomatcher by 2%, it degrades the performance of the original Snort by 4.5% (Section 5.3).

2. RELATED WORK

We review related work in the areas of protocol analysis and traffic normalization, efficient pattern matching, and intrusion detection for high-speed links.

Protocol analysis and traffic normalization. Typically, modern NIDS are based on the ANM methodology. This can be easily seen in Snort [32] and Bro [25] because their source code is available. Based on our discussions with NIDS developers, user manuals we read, and our practical experience using proprietary NIDS, we believe that the ANM methodology is also common in other commercial systems [4, 5, 44]. Since the ANM boosts task separation, it is attractive from the software engineering viewpoint. In this work, however, we challenge the ANM methodology on the point of efficiency: we combine analysis, normalization, and matching into a single protomatching phase.

Ptacek and Newsham [29] were the first to recognize that a NIDS that does not perform normalization is susceptible to evasion. To defend against evasion, Handley *et al.* [12] investigated normalization techniques for the TCP, IP, UDP, and ICMP protocols. They proposed a normalizer that reverses transformations before the NIDS analyzes the traffic. In this work, we focus on the normalization of higher level protocols that support alternate encodings, such as HTTP, TELNET, SMTP, or FTP.

The problem of alternate encodings is particularly painful for HTTP traffic. HTTP allows three encodings of URLs: alphabetic characters, HEX encoding, and UTF-8 [10]. While the Apache web server supports only these encodings, the IIS server supports five more encodings that are Microsoft-specific [31]. This assortment of HTTP encodings complicates HTTP normalization, a situation that has been extensively exploited to evade detection systems [9, 20, 22, 23, 46]. In addition, Dreger *et al.* [8] found that the overhead of HTTP traffic analysis can increase Bro [25] runtime by a factor of five. Given the high cost of HTTP analysis, and the fact that HTTP contributes more than 60% of the overall traffic to many organizations [8], we chose to evaluate our technology using a protomatcher for HTTP traffic.

Alternate encodings are not unique to HTTP. Attackers have injected TELNET control characters in the middle of FTP commands [30]. Furthermore, FTP [27] and SMTP [28] commands are case insensitive. Even when protocol specifications do not allow multiple encodings, a protocol implementation might allow it. For example, implementations of many protocols encode the carriage-return line-feed sequence either as the sequence "`\r\n`" or as a single "`\n`". While we focus on HTTP encodings, we believe that alternate encodings of data in other protocols can be handled using a protomatcher-based design.

Fast pattern matching for NIDS. There is a significant amount of work on efficient *string matching* for intrusion detection purposes. Researchers have proposed either software-based [1, 3, 6, 11, 47] or hardware-based [18, 39, 45] matching algorithms. This previous work does not address the problem of matching in the presence of alternate encodings and none of these algorithms mention protocol analysis as part of the matching algorithm.

Recent research has suggested that strings alone are not sufficient to accurately detect attacks. Instead, researchers have proposed using *regular expression* matching [24, 48]. To match regular expressions, Sommer and Paxson [36] used a DFA. However, unlike our work, they performed matching on already-normalized traffic. They also noticed that their DFA might not fit into memory, so they constructed the relevant portions of their DFA during matching. They acknowledge that this incremental approach degrades Bro's performance and might not be needed because their DFA did not grow beyond 20MB. We construct our protomatcher (our DFA) once and do not modify it further.

The pattern matching mechanism in Snort (since version 2.0 [32]) is based on a concept similar to protomatching. First, Snort uses a fast set-wise string matching algorithm (e.g., [1]) that identifies any rule that **may** match the traffic. Then, if such rules have been identified, Snort invokes a slower matcher that fully match the traffic against those rules. This is the same pre-filtering principle that is the core of the protomatching technique. However, unlike protomatching, Snort applies this two-stage process after it normalizes the traffic while protomatching saves the normalization phase. Indeed, this difference results in up to 49% improvement in Snort performance.

Dealing with high-speed links. Vendors of commercial NIDS advertise that their products can monitor high-speed links of at least 1 Gbps (e.g., [5, 38]). However, anecdotal evidence suggests that this is not always the case [42]. Schaelicke *et al.* [35] conducted an evaluation of several systems and emphasized pattern matching and protocol analysis as the factors limiting NIDS performance.

To deal with high-speed links, researchers have suggested a distributed NIDS that balances the network traffic such that each sensor monitors a different portion of the protected network [16, 37]. Our work focuses on the performance of a single sensor. Since protocol analysis and pattern matching are usually done by each sensor, sensors that use a protomatcher would further increase the throughput of such distributed designs.

3. THE BASICS OF PROTOMATCHING

We illustrate the concepts behind protomatching. We first formulate how the ANM methodology relates protocol analysis and matching. Then, we formulate the concept of a protomatching signature. Last, we develop the concept of the superset protomatcher, which is the basis of the P–ANM methodology. Throughout this section we use the DNS-tools attack as our running example.

The DNS-tools exemplary attack. The DNS-tools attack allows attackers to bypass access to a popular DNS administration tool and gain administrative privileges on a DNS server (CVE-2002-0613 [41]). An attacker that launches this attack typically uses an HTTP request as il-

```
   input  : A string w, L_protocol, N normalization
            function, S a signature.
   output: Returns match if and only if
            (w ∈ L_protocol) ∧ (N(w) ∈ L(S)).
   //Input conforms to the attack protocol?
1  if w ∉ L_protocol then
2      return no-match
   //Compute normalized traffic
3  Compute w' = N(w) ;
   //Normalized traffic matches signature?
4  if w' ∈ L(S) then
5      return match;
6  else
7      return no-match;
```

Algorithm 1: The analyze, normalize, match (ANM) method.

lustrated below, denoted R_1:

R_1: `GET dnstools.php?section=hosts& user_logged_in=true HTTP/1.1`

Consider an attacker who wants to use the DNS-tools attack and also wants to avoid detection by a NIDS. The attacker first encodes parts of the substring "`dnstools.php`" using hexadecimal encoding. For example, the attacker changes one '`o`' and one '`p`' into their hexadecimal values, obtaining the string "`dnsto%4fls.ph%50`". Then, the attacker further obfuscates the substring by mixing upper- and lower-case characters, for example, by converting '`h`' into '`H`'. The result of this process is the following request:

R_2: `GET dnsto%4fls.pH%50?section=hosts& user_logged_in=true HTTP/1.1`

The NIDS goal is to identify both R_1 and R_2 as instances of the DNS-tools attack.

3.1 The ANM Methodology

We use Snort's signature for the DNS-tools attack to illustrate how the ANM methodology attempts to identify both R_1 and R_2. Snort searches the URL of every incoming HTTP request for the following regular expression over the ASCII input $\Sigma = \{0 \ldots, 255\}$:

$$S_{dns} = \Sigma^\star \cdot \text{“}\texttt{dnstools.php}\text{”} \cdot \Sigma^+ \cdot \text{“}\texttt{user\_logged\_in=true}\text{”}$$

S_{dns} matches R_1, but does not match R_2. Therefore, to determine that R_2 is a DNS-tools attack, Snort performs the following steps. First, it parses R_2 and verifies that it is a valid HTTP request. Second, during this parsing, Snort normalizes the URL in R_2: Snort converts "`%4f`" into '`o`', "`%50`" into '`p`', and '`H`' into '`h`'. Last, it checks whether the normalized URL matches S_{dns}.

Formalizing the ANM methodology. Let $L_{protocol}$ be a language that defines valid messages of the attack protocol (e.g., the syntax of HTTP [10]). Let $\mathcal{N} : \Sigma^\star \to \Sigma^\star$ be a *normalization* function, a function that translates a string into its normalized form. Let S be a signature (e.g., a regular expression), and denote the language that the signature defines as $L(S)$. Let w be the NIDS input (e.g., a TCP stream).

The ANM method first checks that w conforms to the syntax of the attack protocol, that is, whether $w \in L_{protocol}$. Second, ANM computes the normalized version of the input, $\mathcal{N}(w)$. Last, the ANM method checks whether the normalized input matches the signature language, that is, $\mathcal{N}(w) \in L(S)$. Formally, the ANM method returns match if and only if $(w \in L_{protocol}) \wedge (\mathcal{N}(w) \in L(S))$ (Algorithm 1).

It is easy to see why Algorithm 1 is inefficient. It performs a membership check (Line 1) and computes $\mathcal{N}(w)$ (Line 3) on every input, for example a network packet, regardless of whether the input is benign or malicious. The P–ANM methodology addresses exactly this inefficiency. P–ANM attempts to determine whether a packet is benign using a single membership check.

3.2 A Protomatching Signature for the DNS-tools Attack

The basic idea behind protomatching is to convert protocol analysis and normalization into a single membership check. First, we expand S_{dns} such that it accounts for all possible encodings of the DNS-tools attack. Second, we expand S_{dns} such that it conforms to the HTTP specification.

Expanding S_{dns} to match all encodings of the DNS-tools attack. This step is based on the observation that alternate encodings are substitutions: operations that map characters to regular expressions [13]. We process S_{dns} and substitute each character that can be encoded in multiple ways with a regular expression that describes all possible variants of an attack, **with respect** to a given set of substitutions. These substitutions preserve language regularity, producing a regular expression that matches both normalized and encoded versions of an attack (see [33] for a proof of this claim).

We illustrate a substitution for the alternate encodings our attacker used to obfuscate R_1: upper-/lower-case and HEX encodings. Our implementation also handles encoding called `Uencode`, which is unique to Microsoft IIS server. We discuss other possible encodings in Section 4.1.

Consider the character '`d`'. Our substitution, denoted $\mathcal{N}^{-1}$, maps '`d`' as follows:

$$\mathcal{N}^{-1}(\texttt{d}) = [\texttt{d|D|\%44|\%64}]$$

We denote the substitution $\mathcal{N}^{-1}$ because it computes the inverse of the normalization function in the ANM technique (Section 3.1). Instead of mapping encodings to a normalized representation, $\mathcal{N}^{-1}$ maps the normalized representation to an expression describing every possible encoding.

We replace each character that appears in a URL in S_{dns} with its corresponding regular expression. We replace the character '`d`' in the string "`dnstools`" with the regular expression [`d|D|%44|%64`], then we replace the character '`n`' with [`n|N|%4e|%4E|%6e|%6E`], and so on. In the end, we obtain a signature similar to the expression PS^1_{dns} (for brevity, we omit the substitutions for characters beside '`d`' and '`n`'):

$$PS^1_{dns} = \Sigma^\star \cdot [\texttt{d|D|\%44|\%64}] \cdot [\texttt{n|N|\%4e|\%4E|\%6e|\%6E}] \cdot \text{“}\texttt{stools.php}\text{”} \cdot \Sigma^+ \cdot \text{“}\texttt{user\_logged\_in=true}\text{”}$$

Formally speaking, $PS^1_{dns} = \mathcal{N}^{-1}(S_{dns})$. As long as we can express alternate encodings as regular substitutions, the set of all possible variants of an attack is a regular language [33].

Expanding PS^1_{dns} to conform to HTTP specification. PS^1_{dns} integrates multiple encodings into a regular

input : A string w, $L_{protocol}$, $\mathcal{N}$ normalization function, S a signature.
output: Returns match if and only if $w \in L_{protocol} \cap L(\mathcal{N}^{-1}(S))$.
`//1,2 are done once in a pre-computing phase`
1 Pre-compute $\mathcal{N}^{-1}(S)$;
2 Pre-compute $PS = L_{protocol} \cap L(\mathcal{N}^{-1}(S))$;
3 **if** $w \in$ PS **then**
4 return match;
5 **else**
6 return no-match;

Algorithm 2: Protomatching method.

expression, so it saves the need for a separate normalization phase. Now, our goal is to convert PS^1_{dns} to an expression that also conforms to the HTTP specification, so we can omit the protocol analysis phase.

The HTTP syntax is given in Backus-Naur form [26] in the HTTP specification [10]. To the best of our knowledge, whether the HTTP syntax can be expressed as a regular language has yet to be investigated. However, below we propose a regular expression that matches the HTTP *Request-Line*: the HTTP method followed by a URL followed by the HTTP version. We chose to model the Request-Line because it appears in more than 85% of Snort signatures. Currently, we construct the regular expressions for the HTTP protocol manually. In the future, we plan to use automated techniques [15] to construct a regular approximation directly from the BNF representation of the HTTP syntax.

We convert PS^1_{dns} into our final protomatching signature (Section 4.1 details the conversion process):

$$PS_{dns} = (\Sigma^\star \cdot \text{“}\texttt{\textbackslash n\textbackslash n}\text{”})^\star \cdot \text{“}\texttt{GET}\text{”} \cdot (SP)^+ (\texttt{U})^\star \cdot [\texttt{d|D|\%44|\%64}] \cdot [\texttt{n|N|\%4e|\%4E|\%6e|\%6E}] \cdot \text{“}\texttt{stools.php}\text{”} \cdot (\texttt{U})^+ \cdot \text{“}\texttt{user\_logged\_in=true HTTP/1.1\textbackslash n}\text{”}$$

In the PS_{dns} expression, ‘\n’ denotes a newline character, *SP* denotes white space characters, and U denotes characters that can appear in a URL according to the HTTP specification (e.g., ‘\n’ cannot appear in a URL).

PS_{dns} ensures that the string “dnstools.php”, is part of a URL of a valid HTTP method. It ensures that (i) the “GET” appears in the beginning of a line, (ii) only white spaces separate the “GET” from the URL, (iii) only valid characters appear in the URL, and (iv) the URL ends according to the HTTP protocol. Therefore, when the traffic we observe matches PS_{dns} we can be sure that the string “dnstools.php” (or any of its encoded versions) is part of a URL in a valid HTTP method. In other words, our conversion saves the need for protocol analysis.

Formalization of a protomatching approach. Algorithm 2 uses a protomatching signature for matching. First, the algorithm computes $\mathcal{N}^{-1}(S)$ (Line 1). Second, the algorithm computes $L_{protocol} \cap \mathcal{N}^{-1}(S)$ (Line 2), which is our protomatching signature, denoted by *PS*. In particular, in our example $PS_{dns} = L_{HTTP} \cap \mathcal{N}^{-1}(S_{dns})$. Note that the expression for $L_{protocol} \cap L(\mathcal{N}^{-1}(\hat{S}))$ can be pre-computed because it is independent of the input. Last, in Line 3 the algorithm checks whether the input matches our protomatching signature. It is possible to formally prove that protomatching is equivalent to the ANM methodology, that is, Algorithm 2 is equivalent to Algorithm 1 [33].

input : A string w, $L_{protocol}$, $\mathcal{N}$ normalization function, S a signature.
output: Returns match if and only if $w \in L_{protocol} \cap L(\mathcal{N}^{-1}(S))$.
1 Pre-compute $\hat{S}$ such that $L(\hat{S}) \supseteq L(S)$; `//Compute superset signature (pre computation phase).`
2 Pre-compute $\mathcal{N}^{-1}(\hat{S})$; `//Done once in a pre-computing phase`
3 Pre-compute $\hat{PS} = L_{protocol} \cap L(\mathcal{N}^{-1}(\hat{S}))$; `//Done once in a pre-computing phase`
4 **if** $w \in L(\hat{PS})$ **then**
5 return the result of Algorithm 1 on $(w, L_{protocol}, \mathcal{N}, S)$;
6 **else**
7 return no-match;

Algorithm 3: The protomatch, analyze, normalize, match (P–ANM) method.

In Section 4.2 we discuss two techniques to convert a set of protomatching signatures into a protomatcher: a deterministic finite state machine that can be used at runtime to detect whether the network traffic matches any of the underlying signatures. Ultimately, our goal is to construct a *full-coverage protomatcher*, a protomatcher that contains a protomatching signature for every signature in the NIDS database. The advantage of a full-coverage protomatcher is its efficiency: it requires a single inspection of each network byte during the membership check (Line 3 in Algorithm 2).

Unfortunately, a full-coverage protomatcher is difficult to achieve. While we show that it is feasible to build a full-coverage protomatcher for more than 500 Snort signatures (Section 5.1), these signatures are short regular expressions. They usually require matching of two strings: the HTTP method and an additional string. However, researchers have shown that accurate signatures require sophisticated expressions [7, 24, 34, 36, 48]. For example, PS_{dns} requires matching three strings. Our experience shows that a protomatcher for all of the HTTP signatures in Snort's database requires more than 2GB of memory.

3.3 A Superset Protomatching Signature

The goal of the superset protomatcher is to reduce the memory footprint of the full-coverage protomatcher. It does so, by trading matching efficiency for memory consumption. Instead of using a signature like PS_{dns}, it uses a less-specific, or a *superset*, signature for the DNS-tools attack.

A superset signature omits some portions of the original signature in the NIDS database. Hence, its corresponding protomatcher consumes less memory. At the same time, a superset signature causes *false matches*, cases in which the traffic matches the signature but does not match the original signature in the NIDS database. Hence, in cases where the traffic matches a superset signature, we need to verify whether or not the traffic also matches the original signature. In our implementation, this slower matching process is based on the traditional ANM methodology.

This two-phase approach is the core of the P–ANM methodology. P–ANM is time-efficient because, under normal conditions, most of the traffic is benign. It is memory efficient because it splits the detection into two processes that together consume less memory than a full-coverage protomatcher.

Consider again the DNS-tools attack. Its superset signature can be:

$$\hat{S}_{dns} = \text{“}\texttt{GET}\text{”} \cdot \Sigma^{\star} \cdot \text{“}\texttt{dnstools.php}\text{”}$$

We follow the steps from Section 3.2 and construct the following superset protomatching signature (for brevity, we omit encodings of characters beside ‘d’ and ‘n’):

$$\hat{PS}_{dns} = (\Sigma^{\star} \cdot \text{“}\texttt{\textbackslash n\textbackslash n}\text{”})^{\star} \cdot \text{“}\texttt{GET}\text{”} \cdot (SP)^{+} (\texttt{U})^{\star} \cdot \texttt{[d|D|\%44|\%64]} \cdot \texttt{[n|N|\%4e|\%4E|\%6e|\%6E]} \cdot \text{“}\texttt{stools.php}\text{”}$$

There are several reasons to construct $\hat{S}_{dns}$ as we did above.

1. **$\hat{S}_{dns}$ preserves false negative correctness of S_{dns}.** The language of $\hat{S}_{dns}$ is a superset of the language of S_{dns}: that is, $L(\hat{S}_{dns}) \supset L(S_{dns})$. This property remains true even if we apply our substitutions, that is $\mathcal{N}^{-1}(\hat{S}_{dns}) \supset \mathcal{N}^{-1}(S_{dns})$. Therefore, a superset protomatcher recognizes any attack instance recognized by the full-coverage protomatcher and some additional traffic. By constructing $\hat{S}_{dns}$ as a superset of S_{dns} we ensure lack of false negatives.

2. **A protomatcher based on $\mathcal{N}^{-1}(\hat{S}_{dns})$ consumes exponentially less memory than a protomatcher based on $\mathcal{N}^{-1}(S_{dns})$.** The reason is that $\mathcal{N}^{-1}(\hat{S}_{dns})$ contains only two explicit substrings separated with the general term $\Sigma^{\star}$. Essentially, every $\Sigma^{\star}$ followed by a string doubles the memory that a DFA consumes. For example, to match `user_logged_in=true` in $\mathcal{N}^{-1}(S_{dns})$, a DFA must identify that the input stream already contains `GET` and `dnstools.php`, in that order. In a DFA, such knowledge is expressed by adding states. When we consider a set of signatures, each signature doubles the size of the protomatcher, resulting in an exponential growth of states.

3. **The majority of benign traffic does not match the superset protomatcher**. Like other researchers [2, 19], we have noticed that 99% of the HTTP requests in traffic we monitor does not match both S_{dns} and $\hat{S}_{dns}$. This seems intuitive because most HTTP traffic does not target the organization's DNS server.

4. PROTOMATCHING IMPLEMENTATION

We implemented the P–ANM methodology in Snort. We chose Snort because it is widely used and its source code as well as its signatures are publicly available. We describe how to convert Snort HTTP signatures into protomatching signatures and how to construct a protomatcher that can match those signatures during runtime. We focus on HTTP signatures because they account for 46% of all Snort signatures, and because Snort performs HTTP analysis and normalization that we would like to save.

While we use Snort, our protomatching signatures and runtime protomatcher can be used in the context of other NIDS. After all, our signatures are regular expressions that conform to the HTTP specification and our protomatcher is based on a finite state machine.

4.1 Automatically Converting Snort Signatures into Protomatching Signatures

Snort enables two types of patterns in an HTTP signature: one that must appear in the attack URL, denoted `uri-content`, and one that can appear anywhere in the HTTP request, denoted `content`. We split Snort signatures into four types, based on the combinations of these two patterns (Table 2).

We constructed two signatures from each Snort signature: A *full-coverage* signature, used by our full coverage protomatcher, and a *superset* signature, used by our superset protomatcher. For Type 3 signatures, we used the `uri-content` pattern as our superset signature. For Type 4 signatures, we used the longest pattern, under the assumption that longer patterns would cause fewer false matches. We leave other strategies to build superset signatures as future work.

In our current implementation, we do not translate other fields of a Snort's signature into a regular expression. For example, we ignore the *depth* and *offset* fields that specify portions of the packet in which the pattern should be found. Therefore, when a signature uses such a field (less than 10% of the signatures use these fields), we always treat its corresponding protomatching signature as a superset signature.

Expanding Snort signatures according to the HTTP syntax. To convert a Snort signature into a protomatching signature, we first expand it according to the syntax of HTTP. Before each pattern (e.g., a `uri-content` pattern), we add a regular expression that matches a valid HTTP method. In the case that we know the method necessary for the attack to succuss, for example “`GET`”, we add only this method. In all other cases, we add a regular expression that matches either “`GET`”, “`HEAD`”, or “`POST`”. We believe that these are the most common methods used in HTTP attacks; clearly other methods can be added as needed.

We ensure that an HTTP method always appears in the beginning of a line, as required by the HTTP specification. Between the method and the pattern, we allow only characters that are permitted in a URL; for example, we disallow white space characters. These expressions can be automatically added as defined by the regular expressions denoted by `M`, `L`, and `P`, in Table 2.

Alternate Encodings as substitutions. We implemented three types of HTTP alternate encodings: upper-/lower-case switching, HEX encoding, and the Microsoft-specific U-encoding [9]. We denoted these transformations as `UL`, `HEX`, and `Uencode`, respectively. The `uri-content` pattern requires normalization of the `Uencode`, `HEX`, and `UL` transformations, while the `content` pattern only requires normalization of the `UL` transformation.

In general, like the HEX encodings, the `Uencode` maps a character to a string containing the character's hexadecimal ASCII value, but does so using 4 instead of 2 bytes. We chose these transformations because they have been widely used for evasion [20, 23, 46] and represent transformations used by two popular web servers: Apache and IIS. To embed these multiple encodings into our protomatching signatures, we just replaced each character in the `uri-content` pattern with all of its possible encodings. Table 3 illustrates the three encodings defined as substitutions, and a simple DFA that identifies both encoded and normalized versions of the character ‘o’.

UTF-8 is another transformation allowed by the HTTP specifications [10], and the only other transformation supported by Apache. We do not foresee any problems in supporting UTF-8. In the case of a protomatcher for Apache, UTF-8 would replace the `Uencode`. In the case of a proto-

Type	Patterns (re denotes a regular expression).	# of sig. in Snort database	Convert into a regular expression M: HTTP method (e.g., "GET", "HEAD"). S: White space. U: A valid URL character [10]. L: match beginning of a line i.e., $\Sigma^\star \cdot$'\n' $\mathcal{I}^{-1}$: UL substitution (Table 3). $\mathcal{N}^{-1}$: UL+HEX+Uencode substitutions (Table 3). Σ: $\{0 \dots 255\}$
1	$uri = re_1$	749	full-coverage: $\mathtt{L} \cdot \mathtt{M} \cdot \mathtt{S}^+ \cdot \mathtt{U}^\star \cdot \mathcal{N}^{-1}(re_1)$ superset: $\mathtt{L} \cdot \mathtt{M} \cdot \mathtt{S}^+ \cdot \mathtt{U}^\star \cdot \mathcal{N}^{-1}(re_1)$
2	$content = re_1$	130	full-coverage: $\mathtt{L} \cdot \mathtt{M} \cdot \mathtt{S}^+ \cdot \Sigma^\star \cdot \mathcal{I}^{-1}(re_1)$ superset: $\Sigma^\star \cdot \mathcal{I}^{-1}(re_1)$
3	$uri = re_1$, $content = re_2$	122	full-coverage: $\mathtt{L} \cdot \mathtt{M} \cdot \mathtt{S}^+ \cdot \mathtt{U}^\star \cdot \mathcal{N}^{-1}(re_1) \cdot \mathtt{U}^\star \cdot \mathtt{S} \cdot$ "HTTP") $\cap (\Sigma^\star \cdot \mathcal{I}^{-1}(re_2) \cdot \Sigma^\star)$ superset: $\mathtt{L} \cdot \mathtt{M} \cdot \mathtt{S}^+ \cdot \mathtt{U}^\star \cdot \mathcal{N}^{-1}(re_1)$
4	$content_1 = re_1$, $\dots$, $content_n = re_n$	21	full-coverage: $\mathtt{L} \cdot \mathtt{M} \cdot \bigcap_i \Sigma^\star \cdot \mathcal{I}^{-1}(re_i) \cdot \Sigma^\star$ superset: $\mathtt{L} \cdot \mathtt{M} \cdot \Sigma^\star \cdot \mathcal{I}^{-1}(re_1)$

Table 2: Converting Snort signatures into protomatching signatures.

matcher for IIS, UTF-8 would increase the size of a protomatcher, and it might be that only a hierarchical protomatcher would be feasible.

There are five other esoteric transformations only supported by the IIS server [31]. Since they have a distinctive pattern, are rarely used, and their usage is highly suspicious, we believe that the best way to handle those is through signatures rather than normalization. For example, the *double HEX encoding* encodes the character '%' using the HEX encoding. That is, the substring "%2520" is first decoded into "%20" and then into the space character. In this case, we used the signature $\Sigma^\star \cdot$%25, as also done by Snort. For other Microsoft-specific encodings the reader is referred to [31].

4.2 Converting Protomatching Signatures into a Protomatcher

We describe two possible implementations for a protomatcher that can be used in practice to match our protomatching signatures, based on a *deterministic* finite state machine and based on an *hierarchical* state machine. We used both techniques to implement our protomatchers in our experiments (Section 5). To handle possible false matches of our superset protomatcher, we invoked Snort's analyzer, normalizer, and matcher. While this might not be the most time-efficient method, it was sufficient to illustrate the benefits of the P–ANM methodology.

Implementing a protomatcher using a deterministic FSM. Recall that a protomatching signature is a regular expression. Since regular expressions are closed under the union operation, there exists a single regular expression that recognizes all signatures in parallel. We used publicly available tools [21, 49] to construct this expression and then to automatically convert it into a deterministic finite state machine. We implemented the machine as an $M \times 256$ table where M is the number of states and 256 is the size of our alphabet. The table represents a function $f(i, j) = k$, that is, from state i with the input j the automaton moves to state k. We incorporate this table into Snort during compilation time.

Note that an implementation based on a deterministic FSM is highly efficient because it inspects each network byte exactly once. There is no need for any protocol analysis or normalization. Ultimately, our goal is a full-coverage protomatcher that only uses full-coverage signatures. Unfortunately, this was not always feasible because a deterministic protomatcher based only on full-coverage signatures consumed more than 2GB of memory when we added full-coverage versions of signatures of Type 2, 3 and 4. Hence, when a full-coverage protomatcher was infeasible, we constructed a superset protomatcher using the superset signatures (Section 5.1).

The *hierarchical protomatcher*, implementing a protomatcher using an hierarchical FSM. As we show in Section 5.1, a deterministic superset protomatcher is feasible but can consume more than 20MB of memory. Because of its large size, the protomatcher might cause many cache misses. Therefore, we investigated an implementation that reduces the memory footprint of our protomatcher.

The hierarchical protomatcher is a memory-efficient implementation of a protomatcher. It splits the protomatcher into two machines: a *matcher* and a *normalizer*. The matcher is responsible for protocol analysis and pattern matching. The normalizer is responsible for handling multiple encodings. Unlike the ANM method that first normalizes the whole HTTP request, the hierarchical protomatcher consults with the normalizer only when necessary. For example, when a matcher encounters the character '%' in "dnsto%4fls" (R_2 in Section 3), it calls the normalizer that interprets the string "%4f" and returns the character 'o' to the matcher, which continues with the matching process. Formally, the normalizer is implemented as a transducer [40], a finite state machine that outputs a symbol on each transition.

To implement a hierarchical protomatcher, we build a protomatching signature according to the rules in Table 2, but we do not apply the $\mathcal{N}^{-1}$ substitution. Hence, in the hierarchical protomatcher case, our protomatching signature accounts only for the UL encoding, while the normalizer implements the HEX and Uencode transformations.

The main advantage of a hierarchical protomatcher is its memory efficiency, it consumes an order of magnitude less memory that a deterministic protomatcher. The reason is that a deterministic implementation inlines the normalization of the HEX and Uencode encodings in every pattern, while in the hierarchical implementation we abstract this normalization as a function call.

We can use some of the memory we save to improve the accuracy of the underlying superset protomatcher. We pinpoint superset signatures that cause a significant number of false matches (Section 3.3) and convert them back to full coverage signatures. Since we do so only for a few signatures, we only increase the size of the hierarchical protomatcher to 6MB (Section 5.2.2), which is equivalent to the size of Snort's matcher. More importantly, this process significantly reduces the number of false matches of the under-

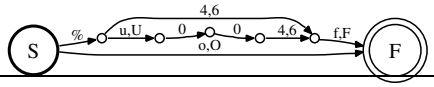
Name	$\mathcal{N}^{-1}$(o)	A DFA that identifies the HEX, Uencode, and UL substitutions for the character 'o'
HEX	([%6f|%6F|%4f|%4F])	
Uencode	([%u006f|%u006F|%u004f| %u004F|%U006f|%U006F|%U004f|%U004F])	
UL	([o|O])	

Table 3: Representing encodings as substitutions. The $\mathcal{I}^{-1}$ substitution is analogous to $\mathcal{N}^{-1}$ but without the Uencode substitution.

lying superset protomatcher. We believe that such a signature refinement process can be automated. For example, a NIDS can use an administrator responses to discover the signatures that cause the most false matches. However, such study is beyond the scope of this paper.

5. FEASIBILITY STUDY

We investigated the ability of the P–ANM methodology to improve the performance of Snort (version 2.3.3). We were interested in the following questions.

1. **Is a protomatcher-based Snort feasible?** Especially, we were interested in understanding the memory consumption of various protomatchers and the time it takes to construct them.

   In Section 5.1, we show that a protomatcher-based Snort is feasible. We show that, based on 999 protomatching signatures, our protomatchers consume up to 22.09MB or 6.47MB, when implemented using the deterministic or hierarchical implementations, respectively.

2. **How does a protomatcher affect Snort performance?** In Section 5.2, we compared the performance of a protomatcher-based Snort to a vanilla Snort: the original Snort that uses the ANM methodology.

   In Section 5.2.1, we show that a protomatcher-based Snort is 45% faster that a vanilla Snort. After we tuned Snort to use a Wu-Manber pattern-matching algorithm [47], our protomatcher-based Snort was still 25% faster. We show that, on real HTTP traffic, our protomatcher classified more than 99% of the traffic as benign. In other words, our protomatcher determined that more than 99% of the traffic was benign without utilizing the expensive analysis-normalizing-matching phases.

   In Section 5.2.2, we reduced the number of false matches using an hierarchical protomatcher. We identified superset signatures that cause the majority of false matches and replaced them back with their full-coverage versions. We reduced the number of false matches by more than 50% and improved our performance by an additional 2% (or 4% without the Wu-Manber pattern matching), beyond the 25% obtained with the deterministic protomatcher.

3. **How does the memory size of a protomatcher affect its performance?** Since a protomatcher consumes more memory than a vanilla Snort, a protomatcher-based Snort might suffer more cache misses. We investigated a *cache-poisoning* attack in which an attacker attempts to degrade the protomatcher's performance by forcing it to generate many cache misses. While this attack degrades the a protomatcher by 2%, it degrades the performance of the vanilla Snort by 4.5%.

Experimental methodology. Our Snort distribution contained 1022 HTTP signatures. We removed 23 (2.25%) of them from the experiments because they caused thousands of false alarms in our environment.

To compare the performance of our protomatcher-based Snort to a vanilla Snort, we used traces of live HTTP traffic that we collected from the gateway router of our department web server. Each trace contained between five and seven million packets with their full HTTP payload. Two traces, T_1 and T_2, were collected during the day, starting at 9:00am and 2:00pm, respectively. Trace T_3 was collected during the night, starting at 1:00am. All traces where collected during September 2005. On average, 4% of the TCP packets in each trace are requests.

Our performance metric is Snort's *Average per-Packet Processing Time* (ApPPT). We measured this time in CPU cycles. We started counting cycles when Snort gets a packet from libpcap [43] and stopped counting when Snort finishes handling the packet and is ready to accept the next one. We ran all our experiments on a Pentium 4, 3GHz, 1MB L2 cache, 2GB of memory, running Tao Linux 1.0.

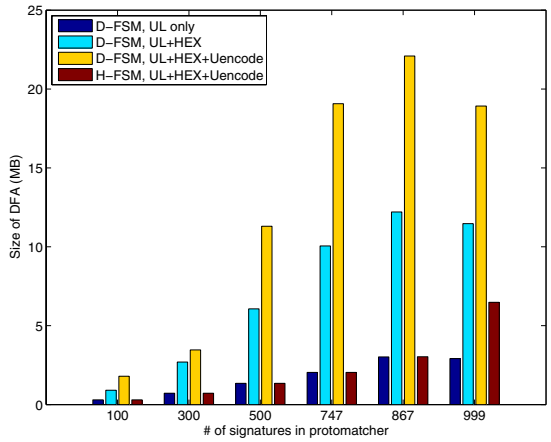
5.1 Feasibility and Memory Consumption

We experimented with protomatchers containing different numbers of signatures. First, we built full-coverage protomatchers, using full-coverage Type 1 signatures (Table 2). Since we have 747 Type 1 signatures, we built protomatchers with 100, 300, 500, and 747 signatures. Second, we tried to add another 120 full-coverage Type 2 signatures. However, this attempt failed because building a full-coverage protomatcher consumed more than 2GB of memory. Hence, we switched to a superset protomatcher, using 120 Type 2 superset signatures and 747 full-coverage signatures. Finally, we added 132 Type 3 and Type 4 superset signatures, constructing a superset protomatcher with 999 signatures.

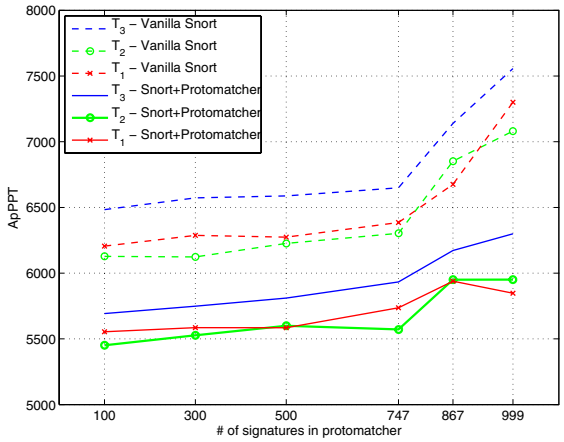
We build the protomatchers once using the deterministic FSM and once using an hierarchical FSM. Building times for the deterministic protomatchers span from 4 minutes for the smallest to 92 minutes for largest, when constructed with the UL, HEX, and Uencode transformations (Figure 1a). Building times for an hierarchical implementation are considerably lower. Recall that we implement a protomatcher as a separate table (Section 4.2) and we incorporate the table into Snort during the compilation process. Therefore, from the administrator viewpoint, the only differences between a protomatcher-based Snort and a vanilla Snort, is their compilation time. According to our measurements, given a protomatcher as a header file, the compilation time of a protomatcher-based Snort is longer by at most 10 minutes.

Number of signatures	100	300	500	747	867	999
Type of signatures (Table 2):	1	1	1	1	1,2	1,2,3,4
Protomatcher type:	full-coverage	full-coverage	full-coverage	full-coverage	superset	superset
Building time (min):	4	9	18	60	80	92
Memory size D-FSM (MB):	1.79	3.46	11.30	19.06	22.09	18.92
Memory size H-FSM (MB):	0.29	0.71	1.34	2.03	3.03	6.47

(a) Summary of feasibility study: building times and memory sizes are for protomatchers built using the `UL`, `HEX`, and `Uencode` transformations (Section 4.1).



(b) Protomatchers memory size.



(c) Average Cycle per Packet.

Figure 1: Performance comparison of a protomatcher-based Snort with a vanilla Snort. (D-FSM and H-FSM denote a deterministic and hierarchical protomatcher, respectively.)

Figure 1a also presents the memory consumption of protomatchers that support normalization for the `UL`, `HEX`, and `Uencode` transformations. Protomatchers based on a deterministic implementation consume up to 22.09MB, or less than 1% of our workstation memory. Protomatchers based on the hierarchical implementation consume an order of magnitude less memory.

Figure 1b presents the memory consumption of protomatchers that support different types of normalization: the `UL+HEX+Uencode` that we used in our ApPPT experiment (Section 5.2.1), a `UL+HEX` protomatcher that is suitable to use with the Apache web server, and a `UL` protomatcher for comparison purposes. The `HEX+UL` protomatcher is about half the size of its `HEX+UL+Uencode` counterpart, indicating the heavy memory consumption imposed by the `Uencode` normalization. In comparison, the `UL` protomatchers consume less than 3MB, or half the size of Snort's matcher.

Note that the protomatcher with 867 signatures consumes more memory than a protomatcher with 999 signatures. The reason is that sometimes adding a signature reduces the number of states in an automaton. Consider, for example, an automaton with a signature $\Sigma^\star \cdot$ `cgi_bin/perl` to which we add the signature $\Sigma^\star \cdot$ `perl`. When we add the latter signature we reduce the number of states because the new automaton does not need to ensure the existence of substring "`cgi_bin/`".

In summary, our results show that a protomatcher-based NIDS is feasible. The memory consumption of our protomatchers is reasonable. Since construction of a protomatcher is done in a separate process, one can build a protomatcher without halting Snort. Then, incorporating a protomatcher into Snort just increases the compilation time by 10 minutes.

5.2 Performance Improvements

We first compared the ApPPT of our protomatcher-based Snort to a vanilla Snort. Then we investigated the capability of a protomatcher to quickly filter benign traffic.

5.2.1 *Improved ApPPT*

We ran our protomatcher-based and vanilla Snorts on our three traces and measured the ApPPT. Our protomatcher normalizes the `UL`, `HEX` and `Uencode` transformations. We report the results against a vanilla Snort configured with the Wu-Manber pattern matching algorithm. By default, Snort uses the Aho-Corasick algorithm [1]. When compared to the Aho-Corasick algorithm, our protomatcher-based Snort further improved the performance by additional 10%-15% **above** the numbers reported below.

Up to 25% improvement of Snort's ApPPT. The ApPPT of Snort with a superset protomatcher containing 999 signatures is lower by up to 25%, 19%, and 18% on T_1, T_2, and T_3, respectively (Figure 1c). Note the steep increase in Snort ApPPT as we add Type 3 and 4 signatures (the 867 and 999 marks). Inherently, matching complex regular expressions is a time consuming task. This illustrates the benefits of the P–ANM approach: the majority of the traffic does not match any signature and a protomatcher saves expensive analysis, normalization, and matching times.

Recall that an hierarchical protomatcher is less efficient than a deterministic protomatcher, because it must implement communication between the matcher and the normalizer. However, our hierarchical protomatcher further improved the ApPPT by 2%, on average (compare the "Hierarchical" and "Deterministic" lines in Figure 3). With 999 signatures the hierarchical protomatcher improved Snort's

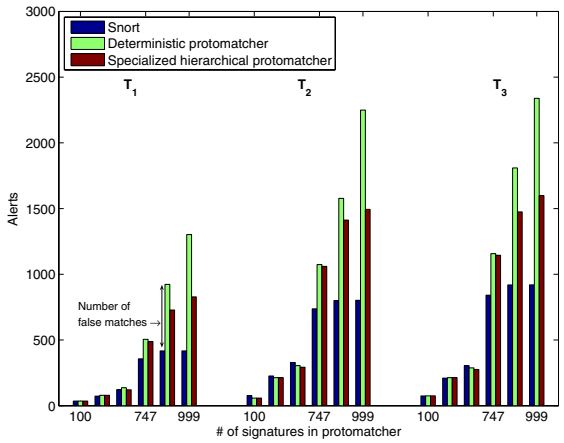


Figure 2: Snort alerts vs. protomatcher alerts. We define a false match as a case when traffic matches a protomatcher but does not produce an alert.

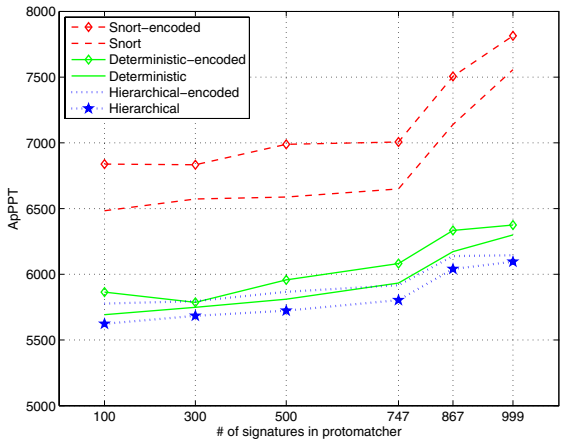


Figure 3: Effects of a cache-poisoning attack on the T_3 trace. The hierarchical protomatcher exhibits the best ApPPT across all traces.

ApPPT by 27%, 21%, and 23% on T_1, T_2, and T_3, respectively. The reason for this improvement is the smaller size of the hierarchical protomatcher, which reduced the number of cache misses. Therefore, Snort with a hierarchical protomatcher can handle up to 27% more traffic than a vanilla Snort.

Separate normalization is expensive. Although only 4% of the packets contain HTTP requests, our superset protomatcher improved Snort's ApPPT by up to 25%. This illustrates the large toll of normalization on the performance of a NIDS. The benefit from the superset protomatcher increases as we add complex signatures, that is, Type 3 and 4 (Table 3) signatures, to the system. This is evident in Figure 1c and Figure 3.

5.2.2 *Utilization of the P–ANM Methodology*

On average, a protomatcher classified 99% of the HTTP requests as benign. The superset protomatcher with 999 signatures classified as benign 99%, 98.8%, and 99% of the HTTP requests in T_1, T_2, and T_3, respectively. The ability of a protomatcher to quickly accept benign traffic, either normalized or encoded, is the core of its efficiency.

However, our superset protomatchers still produce false matches (Section 3.3). For example, our superset protomatcher with 999 signatures matches 885 HTTP requests in T_1 while Snort produces only 417 alerts (Compare "Snort" and "Deterministic protomatcher" bars in Figure 2). Since the hierarchical protomatcher is a compact implementation of a protomatcher, it provides plenty of opportunity to further reduce the number of false matches.

We discovered three Type 4 signatures that caused more than 50% of the false matches across all the three traces. We changed these signatures from superset signatures back to their full-coverage signatures. This increased the size of the hierarchical protomatcher from 3MB to 6MB (Figure 1b) but also decreased the number of false matches by more than 50% in the case of a hierarchical protomatcher with 999 signatures (compare "Deterministic protomatcher" and "Specialized hierarchical protomatcher" bars in Figure 2).

5.3 Sensitivity to Cache Poisoning Attack

A protomatcher is oblivious to the content of the input: the time it spends on a packet with n bytes is always equal to the time it takes to perform n table lookups. One way to degrade the performance of a protomatcher is to increase the table lookup time. To achieve this, we investigated a cache poisoning attack in which we increase the table lookup time by causing cache misses. We study one type of such attack, using URL HEX encoding. Of course, this initial study does not imply that our protomatcher is robust against all cache poisoning attacks; such a study is beyond the scope this paper. However, our study does show that our current protomatcher implementation can effectively sustain our chosen attack.

Our cache poisoning attack attempts to cause cache misses by forcing a protomatcher to visit more states. To do so, we encode all URLs in our traces using only hexadecimal representation. For example, we change a URL like `www.example.com` into `%77%77%77%2e...%2e%63%67%6d`. Note that the hexadecimal representation visits three times more states than the normalized request. This technique generates traffic that accesses real web pages, so it is less likely to be noticed. Note that our attack changes the character distribution inside URLs, so it could be detected using anomaly detection techniques [17]. However, currently Snort does not use such techniques.

We assumed that the attack would have a larger effect on a protomatcher-based Snort than on vanilla Snort. Recall that Snort normalizes all URLs before matching. Hence, in the vanilla Snort case, we assumed the same cache behavior during matching. Furthermore, Snort's normalizer consumes less that 2KB, so it easily fits into L1 cache. Hence, we assumed no significant difference in the cache behavior between the encoded and original traces during normalization. In the protomatcher case, however, we assumed that the cache misses would increase by a factor of three. Since we did not take any effort to increase the spatial locality of the protomatcher states, we believe that this is a reasonable assumption.

Figure 3 presents the effects of our cache poisoning attack.

Two observations should be noted:

1. The attack increases the ApPPT of all three systems (in Figure 3, compare "Snort-encoded" and "Snort", "Deterministic-encoded" and "Deterministic", and "Hierarchical-encoded" and "Hierarchical"). This increase is expected because the encoded URLs are three times longer than the original ones.

2. Snort is more affected by the attack than our protomatcher-based implementation. Snort's ApPPT increased by 4.5% (compare the "Snort-encoded" and "Snort" lines). In comparison, the ApPPT of Snort with a deterministic protomatcher only increased by 2% (compare the "Deterministic-encoded" and "Deterministic" lines).

   This result is surprising since it contradicts our assumptions above. There might be two reasons for this result. First, the attack was ineffective in increasing the number of cache misses. It means that a more sophisticated cache poisoning attack is needed. Second, the attack was effective, but cache performance is only a minor component of the ApPPT. Further investigation of these issues is left for future work.

6. CONCLUSION AND FUTURE WORK

We formulated the concept of a protomatcher: a deterministic finite automaton that performs protocol analysis, normalization, and matching. We studied the performance of a protomatcher-based Snort and showed that it is both feasible and beneficial. We envision two main research directions that extend our work.

To become widely accepted, protomatching should be further automated. Currently, we manually convert a full-coverage signature into a superset one. We also manually refine a protomatcher to reduce the number of its false matches. We plan to investigate ways to automate these tasks in the future.

We also plan to further study the resiliency of protomatching and the P–ANM methodology against resource-consumption attack. Attackers may attempt to induce a protomatcher to generate many false matches. Such an attempt would excessively utilize the four phases in the P–ANM methodology, which would become more expensive than the three phases in the ANM methodology. Therefore, it is important to develop a mechanism that chooses the least expensive methodology based on the current network load.

We presented only an initial study on a protomatcher resiliency against cache poisoning attack. This issue should be study further. For example, it will be interesting to develop a cache-aware protomatcher whose states are organized in a way that increases their locality in memory.

Acknowledgments. We deeply thank Vinod Ganapathy and the anonymous referees for their useful comments that have helped us refine the concepts and experiments presented in this paper.

7. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, **18**(6), June 1975.

[2] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, and E. P. Markatos. Piranha: Fast and memory-efficient pattern matching for intrusion detection. In *IFIP International Information Security Conference*, Chiba, Japan, May 2005.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, **20**(10), Oct. 1977.

[4] CheckPoint Software Technologies. InterSpec Internal Security. Available at `www.checkpoint.com`.

[5] Cisco Systems. Cisco IPS 4200 Series Sensors. Available at `www.cisco.com`.

[6] C. J. Coit, S. Staniford, and J. McAlemey. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, Anaheim, CA, June 2001.

[7] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, Nov. 2005.

[8] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *ACM Conference on Computer and Communications Security*, Washington, DC, 2004.

[9] eEye Digital Security. %u encoding IDS bypass vulnerability, 2001. Available at `www.eeye.com/html/Research/Advisories/AD20010705.html`.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616 - Hypertext Transfer Protocol*. The Internet Engineering Task Force, June 1999.

[11] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. Technical Report CS2001-0670, University of California San Diego, May 2001. Updated version available at `http://woozle.org/\~mfisk/`.

[12] M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, Washington, DC, Aug. 2001.

[13] J. Hopcroft, R. Motwani, and J. Ullman. ***Introduction to Automata Theory, Languages, and Computation***. Addison-Wesley, 2 edition, 2001.

[14] Internet Security Systems. RealSecure Network 10/100. Available at `www.iss.net`.

[15] J. C. Junqua and G. van Noord, editors. **Robustness in Language and Speech Technology**. Springer, 2001.

[16] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[17] C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *ACM Symposium on Applied Computing*, Madrid, Spain, March 2002.

[18] R.-T. Liu, N.-F. Huang, C.-N. Kao, and C.-H. Chen. A fast pattern matching algorithm for network processor-based intrusion detection system. In *IEEE International Conference on Performance, Computing, and Communications*, Phoenix, AZ, Apr. 2004.

[19] E. Markatos, S. Antonatos, M. Polychronakis, and

K. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *IASTED International Conference on Communications and Computer Networks*, Cambridge, MA, Nov. 2002.

[20] R. Marti. THOR: A tool to test intrusion detection systems by variations of attacks. Master's thesis, Swiss Federal Institute of Technology, Mar. 2002.

[21] M. Mohri, F. C. N. Pereira, and M. D. Riley. AT&T Finite-State Machine Library. Available at `www.research.att.com/sw/tools/fsm`.

[22] D. Mutz, C. Krügel, W. Robertson, G. Vigna, and R. R. Kemmerer. Reverse engineering of network signatures. In *The AusCERT Asia Pacific Information Technology Security Conference*, Gold Coast, Australia, May 2005.

[23] D. Mutz, G. Vigna, and R. A. Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In *Annual Computer Security Applications Conference*, Las Vegas, NV, Dec. 2003.

[24] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[25] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, **31**(23/24), Dec. 1999.

[26] N. Peter. Revised report on the algorithmic language ALGOL 60. *Comm. of the ACM*, **3**(5), 1960.

[27] J. Postel and J. Reynolds. *RFC 959 - File Transfer Protocol*. The Internet Engineering Task Force, 1985.

[28] J. B. Postel. *RFC 821 - Simple Mail Transfer Protocol*. The Internet Engineering Task Force, 1982.

[29] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Inc., Calgary, AB, Canada, 1998.

[30] Robert Grahm. SideStep: IDS evasion tool, Jan. 2000.

[31] D. J. Roelker. HTTP IDS evasions revisited, Jan. 2003. Available at `www.idsresearch.org`.

[32] M. Roesch. Snort: the Open Source Network Intrusion Detection System. Available at `www.snort.org`.

[33] S. Rubin. *Formal Models and Tools to Improve NIDS Accuracy*. PhD thesis, University of Wisconsin-Madison, 2006.

[34] S. Rubin, S. Jha, and B. P. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[35] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the performance of network intrusion detection sensors. In *International Symposium on Recent Advances in Intrusion Detection*, Pittsburgh, PA, Sep. 2003.

[36] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Conference on Computer and Communications Security*, Washington, DC, Oct. 2003.

[37] R. Sommer and V. Paxson. Exploiting independent state for network intrusion detection. In *Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2006.

[38] SourceFire Inc. SourceFire IS3000 Series. Available at `www.sourcefire.com`.

[39] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture*, Madison, WI, June 2005.

[40] R. Teitelbaum. *Minimal Distance Analysis of Syntax Errors in Computer Programs*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Sep. 1975.

[41] The National Institute of Standards and Technology (NIST). National vulnerability database. Available at `nvd.nist.gov`.

[42] The NSS Group. Intrusion prevention systems (IPS) group test (Edition 3), Aug. 2005. Available at `www.nss.co.uk`.

[43] The Tcpdump Group. TCPDUMP/LIBPCAP. Available at `www.tcpdump.org`.

[44] TippingPoint, a Division of 3Com. UnityOne, Intrusion Prevention Systems. Available at `www.tippingpoint.com`.

[45] G. Tripp. A finite-state-machine based string matching system for intrusion detection on high-speed networks. In *European Institute for Anti-Virus Research (EICAR) Annual Conference*, Malta, May 2005.

[46] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communications Security*, Washington, DC, Oct. 2004.

[47] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR94-17, Department of Computer Science at the University of Arizona, May 1994.

[48] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. In *USENIX Security Symposium*, Washington, DC, Aug. 2005.

[49] S. Yu. Grail+: A symbolic computation environment for finite-state machines, regular expressions, and finite languages. Available at `www.csd.uwo.ca/research/grail/grail.html`.