Towards Vulnerability-Based Intrusion Detection with Event Processing

Amer Farroukh, Mohammad Sadoghi, and Hans-Arno Jacobsen

Middleware Systems Research Group

Department of Electrical and Computer Engineering

University of Toronto, Toronto, Canada

amer.farroukh@utoronto.ca, mo@cs.toronto.edu, jacobson@eecg.toronto.edu

ABSTRACT

Computer systems continue to be breached despite substantial investments in defense mechanisms to stop attacks from propagating. The accuracy of current intrusion detection systems (IDSes) is hindered by the limited capability of regular expressions (REs) to express the exact vulnerability. Recent advances have proposed vulnerability-based IDSes that parse traffic and retrieve protocol semantics to describe the vulnerability. Such a description of attacks is analogous to subscriptions that specify events of interest in event processing systems. However, the matching engine of state-of-the-art IDSes lacks efficient matching algorithms that can process many signatures simultaneously. In this work, we place event processing in the core of the IDS and propose novel algorithms to efficiently match vulnerability signatures. Also, we are among the first to detect complex attacks such as the Conficker worm which requires correlating multiple protocol data units (MPDUs) while maintaining a small memory footprint. Finally, we show that our algorithms are resilient to attacks through extensive testing of the IDS under different workloads. Our approach incurs negligible overhead when processing clean traffic and is faster than existing systems.

Categories and Subject Descriptors

H.3.3 [Information Search and Retrieval]: Information filtering

General Terms

Algorithms, Design, Performance, Security

Keywords

Vulnerability Signature, Intrusion Detection, Signature Matching, Deep Packet Inspection, and Complex Event Processing

1. INTRODUCTION

The current defensive mechanism in the IT world consists of an intrusion detection system (IDS) that examines traffic

for malicious content and raises alerts for any suspicious activity [19, 18, 1, 15]. Alerts are then forwarded to an event processing routine which correlates and prioritizes them and notifies the network administrator. However, such systems have severely failed to detect sophisticated attacks that require deep content inspection and state maintenance [6]. According to the Cyber Secure Institute, the economic loss due to the Conficker worm tallied up to \$9.1 billion as it infected more than 10 million hosts in 2008 [2]. The currently deployed IDSes have failed to stop the propagation of this attack which raises the need for more sophisticated detection mechanisms that parse and analyze traffic beyond the network layer.

Event processing systems, on the other hand, can match events accurately through deep content inspection of structured event predicates [10, 14, 3, 12]. However, the functionality of such systems in the security domain has been limited to event correlation of alerts generated by the intrusion detection system (IDS) [27]. The accuracy and speed of current IDSes [19, 18, 1, 15] can be significantly improved by using event processing techniques that can parse and analyze traffic beyond the network layer and detect more sophisticated attacks which require correlation between multiple protocol data units (MPDUs) [10, 31, 16, 12]. To illustrate the need for deep content inspection and state maintenance, we describe the Conficker worm in more detail below.

Example: Conficker exploits a server service vulnerability in the Windows system (MS08-067) by sending a specially crafted WINRPC request that overflows the server buffer [2]. A successful attack results in remote code execution in which the attacker takes full control over the infected machine. We model the vulnerability signature for Conficker in Figure 1. The signature requires correlation of different single protocol data units (PDUs).

A single PDU signature is composed of a Boolean formula of predicates. Each predicate holds an attribute-value-matcher triplet and predicates are combined using operators such as AND (&) or OR (||). The BIND PDU consists of three predicates while the REQUEST PDU consists of four predicates. Each predicate has an attribute name (rpc_ver), a value (3), and a matcher (=). As field values can vary in type, different matchers must be used for each predicate. The UUID field requires a string matcher while the PathName field requires length checking as well as regular expression (RE) matching.

Moreover, the RPC version (rpc_ver, rpc_ver_minor) and interface ID (UUID) must be matched in the BIND PDU; the server acknowledgement (Accept) must be detected in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS'11, July 11–15, 2011, New York, New York, USA.

Copyright 2011 ACM 978-1-4503-0423-8/11/07 ...\$10.00.

```

(BIND || ALTER_CONTEXT):
rpc_ver=3 & rpc_ver_minor=0 &
UUID="4b324fc8-1670-01d3-1278-5a47bf6ee188"

-> (BIND_ACK || ALTER_CONTEXT_RESPONSE):
rpc_ver=3 & rpc_ver_minor=0 &
result[UUID] = Accept

-> REQUEST:
rpc_ver=3 & rpc_ver_minor=0 & opnum=0x1f &
strlen(stub.PathName)> 256 &
matchRE(stub.PathName, "/^\\x05\\x00\\x00")

```

Figure 1: MPDU Signature of Conficker Worm.

the BIND_ACK PDU; and the malicious code is passed as part of the REQUEST PDU to overflow the 256 byte buffer. To identify the PDU sequence, the NEXT ($\Rightarrow$) operator ensures that a client is successfully bound to the server interface before issuing a request (BIND $\Rightarrow$ BIND_ACK $\Rightarrow$ REQUEST). The OR ($||$) operator is essential to stop attackers from eluding detection as the same attack can be carried out using an ALTER_CONTEXT PDU to change the interface binding number [6].

Our extensive study of the Conficker worm has shown that regular expressions are not sufficient to detect such attacks. The signature language must be expressive enough to model the exact vulnerability which requires arbitrary matchers such as length checking, ranges, and regular expressions. As a result, the predicate matching phase must be completely isolated from the signature matching phase to support arbitrary matchers. Moreover, the IDS should support the AND, OR, and NEXT operators to correlate different protocol data units (PDUs) and detect such sophisticated attacks which require state maintenance. However, the signature language used in current IDSes [19, 18, 1] models one version of the attack as it cannot express the exact vulnerability exploited by the Conficker worm.

To elude detection by the IDS, attackers craft polymorphic variants of the attack in which the byte code used is different while maintaining the same malicious functionality [7]. Current IDSes [19, 18, 1] fail to detect polymorphic attacks due to the limited capability of regular expressions to express the exact vulnerability [24]. Recently, researchers have proposed vulnerability-based signatures to enhance the accuracy of the IDS [29]. A vulnerability-based signature is a set of predicates and each predicate consists of an attribute-value pair which is similar to the event processing language of subscriptions and events. Despite the similarity of language between intrusion detection and event processing domains, we list some of the differences which impose some challenges as well as create opportunities for enhancing performance.

First, event processing systems aim to support millions of users that can subscribe or unsubscribe at any time [11] while rules in an IDS are in the order of a few thousands, and they are precompiled at boot-time [19]. Second, the protocol specification in IDSes determines the order of fields, which can be exploited to accelerate parsing as irrelevant fields can be skipped [24]. However, predicates in event processing systems exhibit no specific order, and all predicates must be parsed to match the event [10]. Third, the norm in event processing is high number of matches as users often subscribe to events that occur frequently [14]. On the

other hand, the matching engine of IDSes must be optimized for the non-matching case (attack-free) since normal traffic constitutes the vast majority of Internet traffic [28]. Forth, publishers and subscribers are disjoint in event processing applications while clients and servers are grouped into connections in IDSes [18]. An IDS must handle thousands of connections simultaneously as it protects the whole network backbone [19]. As a result, memory footprint is a crucial factor when handling large number of connections separately [15]. Each connection requires a separate memory slot for partial matches; thus, matching algorithms must be optimized for speed and memory.

Given the differences between the two domains, the signature language must be expressive enough to support multiple data types and express the exact vulnerability. The `rpc_ver` in the Conficker example (Figure 1) requires integer matching while the `PathName` requires RE matching and length checking. As a result, the IDS must support arbitrary matching dimensions that can handle a more expressive signature language such as string length checking or regular expressions which are not commonly supported in event processing applications. These matching dimensions include exact matching of numbers, range checking, exact string matching, string length checking, and regular expression matching. More matching dimensions can be augmented to the IDS as the signature language evolves to cover emerging attacks. The message fields are evaluated against the individual predicate matchers and the results of all matchers are aggregated using AND, OR, and other operators. As modularity of individual matchers and the overall matching algorithm is vital for the functionality of IDSes, we investigate matching algorithms that exhibit this distinction and study their applicability in this domain [12, 31]. We examine algorithms that can support arbitrary matchers where predicate matching is isolated from signature matching as is the case for the counting algorithm [31]. Moreover, to detect complex attacks, such as the Conficker worm, we present our Memory Conscious Network (MCN) which correlates signatures matched by different PDUs to detect MPDU attacks. Our contributions are summarized as follows:

1. We present two novel algorithms, Access Predicate Pruning (APP) and Early Elimination (EE), which efficiently match single PDU signatures while maintaining a small memory footprint. Our algorithms reduce the matching time by pruning partial matches of signatures that will eventually not be matched. (Section 4)
2. We are among the first to support MPDU matching in IDSes through our Memory Conscious Network (MCN) that uses bit encoding to minimize memory and correlate messages within a single connection or across multiple connections. (Section 5)
3. We integrate our algorithms into state-of-the-art IDS, and we show that our proposed solutions are resilient to attacks by providing thorough evaluations for different traffic types ranging from normal traffic to extreme attack scenarios for average to large rule sets. (Section 6)

2. RELATED WORK

In this section, we describe the limitations of signatures used in current IDSes, discuss the matching algorithms used

in the IDS domain, and survey event processing algorithms. We also provide an overview of the candidate selection algorithm [15] which serves as the baseline for our evaluation.

Current IDS Capabilities: The field of signature matching mainly focuses on three types of signatures: strings, regular expressions (REs), and vulnerability-based signatures [4, 19, 26, 25, 7, 24]. Despite continuous efforts to enhance the performance of the matching engine in current IDSes, the accuracy of such systems is hindered by the limited capabilities of regular expressions to represent all attacks as they incur a high number of false positives and false negatives. Such signatures are attack driven where a rule can only detect a single type of attack. Current IDSes such as Cisco [1], Snort [19] and Bro [18] use a combination of strings and REs to describe attacks in the signature database. The most widely used multi-pattern string matching algorithms in IDSes [19] are Aho-Corasick [4] and Wu-Manber [30] which scale much better when compared to single pattern matching algorithms that search for each pattern individually [13]. Regular expression (RE) signatures can be represented in deterministic finite automaton (DFA) [26] or non-deterministic finite automaton (NFA) [9]. Smith *et al.* [25] propose a combination of DFA and NFA called extended finite automaton (XFA) where a finite memory is manipulated by instructions attached to states and edges to track dependencies. An XFA is both memory-efficient and time-efficient. String matching algorithms and RE expression matching are orthogonal to our work as we focus on matching many vulnerability-based signatures simultaneously while supporting any type of predicates.

Vulnerability-based Signature Matching: Researchers have recently devoted many efforts on vulnerability-based signatures for IDSes [7, 24, 15, 29]. Wang *et al.* [29] first introduced vulnerability-driven signatures and several efforts focused on automatic generation of such signatures to match all polymorphic and metamorphic variants of an attack [7]. Schear *et al.* [24] present an architecture for integrating parsing and matching of vulnerability signatures. However, the complexity of manually modifying the parser specifications and the explicit maintenance of the protocol state machine by the signature author renders their system infeasible to support large numbers of vulnerability signatures. Netshield [15] researchers have proposed a framework for parsing and matching vulnerability-based signatures. They present the candidate selection algorithm to match many signatures simultaneously.

Candidate Selection (CS) Algorithm: The state-of-the-art algorithm for matching vulnerability signatures in IDSes is the candidate selection (CS) algorithm [15]. Matching is performed in a two-step fashion where predicates are first evaluated then forwarded to the CS algorithm which accumulates the results from the individual matchers [15] and outputs the matched signatures. The field order of the protocol specification is exploited to combine rules into blocks (*RBs*), decide the matcher order, and minimize matching time. At runtime, for every iteration RB_i , field matched, the results returned by individual matchers are merged as follows: $S_i = S_{i-1} \oplus A_i + B_i$ where S_i is the candidate list (CandList) of the current iteration and S_{i-1} is the CandList of the previous iteration. A_i is the set of candidates from RB_1 to RB_{i-1} , while B_i is the list of rules whose predicates are matched in the current RB_i . The operator $\oplus$ reduces the number of partial matches after every iteration but is

expensive to perform due to traversal of A_i and S_i lists as shown in Section 6. For more details, refer to [15].

The main network monitoring applications in event processing are Gigascope [8] which is a stream database for network monitoring and SNIF [17] which is a live stream matching framework that trades off accuracy for speed. In this work, we target algorithms that can support arbitrary matchers, rather than SQL queries, and can accurately match signatures [12, 31, 3, 10, 5, 20, 21]. Those algorithms fall under one of two categories: two-phase and single phase algorithms.

Two-phase algorithms such as counting [31] and GPX-Matcher [20] evaluate predicates in the first phase and match subscriptions in the second phase. Fabret *et al.* [10] use access predicates to group signatures based on common predicates and control access to subscription clusters. At runtime, all access predicates are scanned and the corresponding clusters are evaluated only if the access predicate is matched. Given the nature of the IDS matching problem in which arbitrary matchers must be supported with expressive operations such as regular expression matching and string length checking, two-phase algorithms best suit IDSes. The counting algorithm presented in [31] is the only algorithm that can support arbitrary matchers since phases 1 and 2 are completely distinct. The index structure of signatures and the abstract representation of predicate matches as counters allow for supporting arbitrary predicate matchers, such as regular expressions, substring matching, and Boolean expressions, which are essential to evaluate vulnerability-based signatures. Other algorithms, such as propagation [10], are domain specific and can only address specific types of predicates. The tight coupling between the algorithm and the predicate type renders such algorithms ineffective in the intrusion detection domain which requires supporting complex types such as regular expressions.

In single phase algorithms, typically novel tree organizations are used to match subscriptions [3, 21]; predicates are evaluated in internal nodes while leaf nodes represent subscriptions. These algorithms are fine-tuned for efficient event processing but are not optimized for space. Similarly, Rete [12] is another well-known algorithm; however, Rete [12] also trades off space for matching efficiency and matching expressiveness. Guoli *et al.* [14] proposed an extended and improved version of the Rete algorithm to support composite event processing in publish/subscribe systems. Given the memory constraint of the IDS which is susceptible to memory-explosion attacks, we present our Memory Conscious Network (MCN) that optimizes for efficiency as well as memory footprint to support multiple protocol data unit (MPDU) matching which requires correlation between different packet flows and connections.

Hardware Support for Matching Given the large number of signatures (subscriptions) and the wire-speed processing requirements, Sadoghi *et al.* [22, 23] proposed the use of Field-programmable gate arrays (FPGAs) to further accelerate the matching engine (based on propagation [10] algorithm) in order to meet such stringent requirements.

3. SYSTEM ARCHITECTURE

We integrated our matching engine and MPDU algorithm into the open-source Netshield IDS [15] as shown in Figure 2. Packets are first reassembled to form connections and then forwarded to the appropriate protocol parser. Once a pro-

Figure 2: System Architecture.

tolocol field is parsed, the corresponding matcher is invoked. Netshield supports five matching dimensions: exact matching and range checking of numbers using a binary search tree, exact string matching using a trie, string length checking using a binary search tree, and regular expression matching using a combined DFA. More matching dimensions can be easily supported if needed by the signature language. The results of the individual matchers are simultaneously sent to the core matching engine. We replace the CS algorithm of Netshield [15] with our own algorithms, Access Predicate Pruning (APP) and Early Elimination (EE), to reduce matching time. These algorithms accumulate the results of the partial matches produced by the individual matchers and indicate whether a whole signature is matched. The final component of the system is our own extension to the architecture to support MPDU matching. The MPDU signatures are compiled to build our Memory Conscious Network (MCN) that matches signatures efficiently. Our system can run in two different modes, simple signatures and MPDU signatures, based on the needed functionality of the IDS.

4. ACCESS PREDICATE ALGORITHMS

This section describes two novel algorithms that are used to accumulate partial matches from individual matchers and match a single PDU signature. We first present Access Predicate Pruning (APP) which minimizes the number of partial matches. Next, we present our Early Elimination (EE) algorithm that uses a two-list scanning technique to discard partial matches. Finally, we describe the data structures used for both algorithms and discuss their time and space complexities.

4.1 Access Predicate Pruning (APP)

Normal Internet traffic can cause partial matches of signatures where some predicates are satisfied, but no one signature is fully matched. For example, the signature $S_1:P_1 \& P_2 \& P_3$ where P_2 and P_3 are often matched but not P_1 incurs unnecessary overhead of processing partial matches as S_1 is never matched (P_1 is never satisfied). To reduce the overhead of processing a large partial matches list, we introduce the notion of access predicates to prune signatures that would not eventually be matched. Our definition of an access predicate is distinguished from earlier definitions [11] by maintaining a single access predicate per signature rather than per cluster [10]. Also, we push the access predicate distinction into the individual matchers to reduce runtime processing and to eliminate the expensive scan of all cluster access predicates as is the case for [10, 11]. Our approach ex-

ploits the pruning effect of access predicates used in [10] and supports arbitrary matchers as is the case for the counting algorithm [31].

The data structure for Access Predicate Pruning (APP) is shown in Figure 3, where signatures are controlled by an access predicate, and each signature has an initial counter that stores the number of predicates it holds including the access predicate. For example, signature S_1 has AP_1 as its access predicate and an initial counter of value three as it has two more predicates P_1 and P_2 . In the pre-computation phase, the data structure is built and counters are initialized.

As shown in Figure 3, the individual matchers maintain two lists: an access predicate list and a predicate list. To reduce computation and save overhead, the access predicate distinction is pushed into the individual matchers. For example, consider a tree-based structure for string matching where predicates are evaluated at leaf nodes. If the predicate $user="admin"$ is an access predicate for signature S_1 but not for signatures S_2 and S_3 , the predicate leaf node outputs the sets $\{S_1\}$ and $\{S_2, S_3\}$ instead of a single set $\{S_1, S_2, S_3\}$. At run time, each predicate is forwarded to the appropriate individual matcher based on its type. The individual matchers evaluate the predicate and output two lists that are processed by the APP algorithm.

The APP algorithm maintains a list of partial matches that counts the number of predicates matched by a single message for each signature. Elements of the access predicate list are added to the partial matches list without further processing since their access predicate is satisfied as per the individual matcher. However, elements of the predicate list in Figure 3 are discarded if their access predicates have not been satisfied. Signature pruning based on access predicates reduces the partial matches list which is evaluated at the end of the message after all its relevant fields have been parsed. If the number of predicates matched for each signature in the partial matches list is equal to its total number of predicates, the signature is output as a match. Consequently, access predicates play a significant role in minimizing partial matches, and we next discuss the selectivity of access predicates.

Figure 3: Access Predicate Pruning Data Structure.

Access Predicates: Consider the following signature which has three predicates: $(HTTP_RequestLine.uri.path.filename == "shtml.dll" \& len(HTTP_Headers["Content-Length"]) > 50 \& HTTP_Headers["Accept-Language"] = "en-us")$. $Accept-Language = "en-us"$ is a poor

choice of an access predicate as most HTTP requests satisfy this condition (significant amount of web sites support the English language). However, filename is a distinguishing factor of the signature as each signature usually protects one vulnerable filename (there is usually a single bug per web page). Filename is a selective access predicate that is rarely satisfied which reduces the number of partial matches. Moreover, we also exploit the selectivity of the individual matchers. For example, the string matcher is more selective than the RE matcher as `filename == "shtml.dll"` matches only one file while `matchRE(filename, " * .dll")` matches any file with a dll extension.

Since parsing and matching are intertwined and message fields have a pre-defined order, predicates whose fields appear first in a message are preferred as access predicates. Consider the following three predicates P_1 , P_2 , and P_3 whose fields have the same order in the protocol specification. P_2 cannot be an access predicate since P_1 is evaluated before P_2 . When P_1 is matched, APP will not include the signature in the partial matches list since its access predicate P_2 is not yet satisfied. As the second field of the message is parsed and P_2 is matched, the signature is added to the partial matches list but is never fully matched since P_1 was discarded earlier. To match the signature correctly, P_1 must be the access predicate. However, if P_1 (method="GET") is not selective while P_2 (filename="shtml.dll") is, the matcher allows for buffering P_1 and processing P_2 first; thus, P_2 can be selected as the access predicate. While buffering fields can enhance access predicate selectivity and reduce partial matches, only a few fields that are small in size, 4 bytes or less, can be buffered due to memory constraints of the IDS which must support thousands of connections simultaneously. Nevertheless, some fields in the protocol specification do not exhibit any specific order and any of them can be elected as an access predicate. Next, we discuss the APP algorithm in more detail.

APP Algorithm: Each signature has a counter associated with the number of predicates to be satisfied for it to be matched. These counters are called `initial_counters` and are precompiled from the signature database before runtime processing as shown in Algorithm 1.

At run time, the individual matchers output two sets per matched predicate: `AP_List` (list of signatures which have this predicate as an access predicate) and `P_List` (list of signatures which have this predicate as a regular predicate). The `Prune()` function in Algorithm 1 adds the `AP_List` to the `CandList` as the access predicate is the first predicate in the signature; thus, any signature in the `AP_List` cannot have been previously present in the `CandList` and the pertaining `runtime_counters` are set to one. Moreover, for every signature in the `P_List`, if the `runtime_counter` is greater than zero (access predicate of this signature has already been satisfied), the `runtime_counter` is incremented. Otherwise, the signature is discarded.

At the end of all iterations (i.e., the whole message is parsed), for every signature in the `CandList`, `CheckCounter()` is invoked and the signature is matched if its `runtime_counter` and its `initial_counter` are equal. Moreover, counters of the `CandList` are reset to zero and the `CandList` is discarded to process a new message.

APP Complexity: For each predicate matched, APP adds counters for elements in the `AP_List` and checks the access predicate condition for elements in the `P_List`. At the

Algorithm 1 Access Predicate Pruning (APP)

Function `Prune()`

```

for  $Matcher_i$  in  $AllMatchers_i$  do
  Match  $Matcher_i$  and get  $AP\_List_i$ ,  $P\_List_i$ 
  for  $SigID$  in  $AP\_List_i$  do
    Add  $SigID$  to  $CandList$ 
    Increment  $runtime\_counter[SigID]$ 
  for  $SigID$  in  $P\_List_i$  do
    if ( $runtime\_counter[SigID] > 0$ ) then
      Increment  $runtime\_counter[SigID]$ 

```

Function `CheckCounters()`

```

for  $SigID$  in  $CandList$  do
  if ( $runtime\_counter[SigID] = initial\_counter[SigID]$ ) then
     $SigID$  is matched

```

Function `ResetCounters()`

```

for  $SigID$  in  $CandList$  do
   $runtime\_counter[SigID] \leftarrow 0$ 
  Empty  $CandList$ 

```

end of the message, APP evaluates and resets the counters of its `CandList`. Thus, the time complexity of APP is $O(AP_List + P_List)$ for each predicated evaluated plus $O(CandList)$ for the final scan of counters pertaining to the `CandList`. The APP algorithm reduces the size of the `CandList` as only signatures whose access predicates are satisfied (less likely to be) are included in the `CandList`. As a result, the final scan at the end of the message is fast as only a few partial matches, rather than the whole signature set, are considered. The runtime memory of APP is determined by the size of the `CandList` whose counters are maintained in memory.

4.2 Early Elimination (EE)

The APP algorithm significantly reduces the size of partial matches but requires traversal of both lists output by the individual matchers: the access predicate list and the predicate list. However, the average size of the access predicate list is significantly smaller than the predicate list since access predicates are less likely to be satisfied. The access predicate list represents signatures whose access predicates have been satisfied and whose other predicates can potentially be matched. As only a few elements in the predicate list are potential matches (their access predicates have already been satisfied), we introduce the Early Elimination (EE) algorithm that selectively scans signatures in the predicate list and discards others.

Access predicates in the EE algorithm have a more effective role in which they not only reduce the number of partial matches, but also improve matching time by truncating the predicate list to be scanned. As only elements in the partial matches list have their access predicates satisfied, we use the partial matches list to control access to the predicate list. The intuition behind the EE algorithm is scanning both lists simultaneously as long as there are elements in the partial matches list that can potentially match elements in the predicate list. All other elements are discarded when the assumption no longer holds.

Figure 4 gives an overview of the EE algorithm and its signature structure. Signatures are controlled by access predicates and a counter is maintained to keep track of the total number of predicates per signature. The signature IDs, however, are generated in a way that maintains order in which

Figure 4: Early Elimination Data Structure.

signatures with smaller predicate IDs have smaller signature IDs. For example, the ID of the first signature (S_1) is less than the ID of the third signature (S_3) since the smallest predicate ID in S_1 is P_1 while it is P_4 in S_3 . Also, the ID of the first signature (S_1) is less than the ID of the second signature (S_2) since the second smallest predicate ID in S_1 is P_2 while it is P_4 in S_2 . Maintaining signature order during the compilation step of the algorithm is essential for early elimination of signatures based on their IDs.

At run time, the predicate list and the partial matches list are scanned simultaneously and the algorithm terminates as soon as all elements of either list have been scanned. In case an element is found in both lists (access predicate has been previously satisfied), the corresponding counter is incremented. Moreover, elements in the access predicate list are added to the partial matches list since their access predicates have been satisfied as per the nature of the list. At the end of the message, counters of the partial matches list are checked to find a match and are then reset.

EE Algorithm: The predicate list (P_List) and the access predicate list (AP_List) produced by the individual matchers are processed in the following fashion specified in Algorithm 2. The `EarlyEliminate()` function uses the partial matches list, `CandList`, to index the P_List. If the signature ID of an element in the P_List is equal to the signature ID of an element in the `CandList`, the signature `runtime_counter` is incremented. If the signature ID of an element in the P_List is greater than the signature ID of an element in `CandList`, the next element in the `CandList` is considered and vice versa. The scanning stops when all elements of either list have been scanned. Finally, the AP_List is simply added to the `CandList` and the corresponding `runtime_counters` are set to one.

At the end of the message, a final scan is performed on all counters in the `CandList` and a signature is matched if its `runtime_counter` is equal to its `initial_counter`. Moreover, `runtime_counters` are reset and the `CandList` is discarded.

EE complexity: The time complexity of the EE algorithm is dependent on the size of the partial matches list (`CandList`). For each predicate evaluated, a counter is created for every signature in the access predicate list. Also, the partial matches list and the predicate list (P_List) are scanned simultaneously until all elements of either list are scanned. At the end of the message, a final scan is required to check all counters in the `CandList`. Thus, the worst time of EE is $O(\{P_List + CandList\} + AP_List)$ in addition to the

Algorithm 2 Early Elimination (EE)

Function `EarlyElimination()`

```

for  $Matcher_i$  in  $AllMatchers_i$  do
  Match  $Matcher_i$  and get  $AP\_List_i$ ,  $P\_List_i$ 
   $j \leftarrow 0$ ,  $k \leftarrow 0$ 
  if ( $P\_List_i$  and  $CandList$  are not empty) then
    loop
      if ( $P\_List_i[j] > CandList[k]$ ) then
        Increment  $k$ 
        if ( $k = |CandList|$ ) then
          break
      else
        if ( $P\_List_i[j] < CandList[k]$ ) then
          Increment  $j$ 
          if ( $j = |P\_List_i|$ ) then
            break
        else
          Increment  $runtime\_counter[CandList[k]]$ ,  $j$ ,  $k$ 
          if ( $(j = |P\_List_i|) \text{ OR } (k = |CandList|)$ ) then
            break
    for  $SigID$  in  $AP\_List_i$  do
      Add  $SigID$  to  $CandList$ 
      Increment  $runtime\_counter[SigID]$ 

```

Function `CheckCounters()`

```

for  $SigID$  in  $CandList$  do
  if ( $runtime\_counter[SigID] = initial\_counter[SigID]$ ) then
     $SigID$  is matched

```

Function `ResetCounters()`

```

for  $SigID$  in  $CandList$  do
   $runtime\_counter[SigID] \leftarrow 0$ 
  Empty  $CandList$ 

```

final scan cost of $O(CandList)$. The advantage of the EE algorithm is evident in the case where no access predicates are matched; i.e., the `AP_List` is empty and no elements are added to the `CandList`; thus, the P_List is simply ignored. Therefore, on average, the time for EE per predicate evaluated is $O(\min\{P_List + CandList\} + AP_List)$.

The memory footprint of EE is determined by the number of partial matches in the `CandList`. A single counter for each element in the `CandList` must be maintained in memory to keep track of the number of matched predicates per signature. These counters are evaluated at the end of the message to find a match.

5. MEMORY CONSCIOUS NETWORK

In our Memory Conscious Network (MCN), signatures are compiled into a list of connected nodes to maintain partial state between different PDUs. While the Rete [12] algorithm requires maintaining the actual PDU message (partial match data) as part of its nodes, we present a novel technique of bit encoding to represent partial matches and minimize the memory footprint. Each MPDU signature is composed of a set of single PDU signatures that hold some semantic relation between each other. The network is composed of a set of nodes where a node can hold a single PDU or represent a relation between two PDUs. Figure 6 shows an example of a MCN in which a circle represents a single PDU signature and a square represents a relation between two PDUs. For instance, the left square holds an OR relation between S_1 and S_2 and the node is satisfied if S_1 or S_2 is matched. Nodes in the network are designed to exploit sharing among MPDU signatures in which shared predicates are evaluated

only once. As $(S_1 \& S_2)$ is a common predicate for S_4 and S_6 , the AND condition is evaluated only once in node JN_3 . The key concept of MCN is to exploit the network design to evaluate common predicates only once, thus improving performance.

The network is traversed as follows: When a single PDU is matched, the appropriate signature node is accessed and all join nodes connected to it are traversed. The condition in each join node is evaluated, and when it is satisfied, MPDU signatures associated with this node are output and all nodes connected to its output links are traversed. Network propagation ends when the condition in a join node is not satisfied or the node has no output links.

The condition in a join node represents a relation between two PDUs in a MPDU signature. A relation is expressed using one of the following operators : AND (&), OR (||), and NEXT ($\Rightarrow$). Most signatures require conjunction of predicates as is evident in this example (PDU1: *HTTP_RequestLine.uri.filename* == "shtml.dll" & PDU2: *len(HTTP-Headers["Content - Length"])* >= 100). Moreover, there might be multiple ways of triggering the vulnerability as is the case for the WINRPC protocol. The attacker can bind to an interface using a BIND PDU or an ALTER_CONTEXT PDU, thus the need for the OR operator. Nevertheless, sequence is crucial in MPDU attacks in which the attacker must be bound to an interface before launching a WINRPC attack which can be detected by the NEXT operator. A vulnerable webpage that requires the user to be logged in before exploiting the vulnerability is another example for the NEXT operator (PDU1: *user logs in* $\Rightarrow$ PDU2: *server authenticates user* $\Rightarrow$ PDU3: *user launches an attack*).

To detect MPDU attacks such as the login example, the Memory Conscious Network (MCN) is composed of two types of nodes: signature nodes and join nodes. A signature node holds a unique ID and has output links to a set of join nodes. A join node, however, has a unique join node ID and has output links to other join nodes. It combines two input nodes using one of the three operators (&, ||, $\Rightarrow$). It also holds a list of MPDU signatures which are matched when it is satisfied.

Figure 5 shows the data structure used for each type of node. An AND join node requires two state bits to maintain the state of left and right hand input nodes accordingly. An AND node can be accessed through its left or right input link. If the AND node is reached through its left input link, for example, the left bit is set to one (left side is matched). Consequently, the other bit, the right bit, is checked and if it is one (right side was matched earlier), the AND condition evaluates to true and the MPDU signatures associated with this node are output. Finally, when the AND condition of the join node is satisfied, all nodes connected to its output links are traversed.

An OR join node has a similar structure but no state bits are required. An OR condition is satisfied when either of the sides is matched and it acts as a forwarder. When the OR node is accessed from either side (left or right side is matched), the corresponding MPDU signatures are output and all nodes connected to its output links are traversed. A NEXT join node requires a single state bit to track sequence. The structure of the NEXT node dictates that the left side must be matched first and then the right side for the condition to be satisfied. When the NEXT node is accessed from

Figure 5: MCN Node Data Structure.

its left input link, the bit is set to one and the right side is ready to be matched. When the node is accessed through its right input link, the bit value is checked. If it is one (left side is already matched), the condition evaluates to true, MPDU signatures corresponding to this node are output, and all nodes connected to its output links are traversed. Otherwise, the condition evaluates to false as the left side was not previously matched, thus no further traversal of join nodes that are connected to the NEXT node output links.

Finally, a signature node has only one input which is the ID of the single PDU signature. The signature nodes are used to index join nodes with no condition to be evaluated.

MCN Consumption Policy: In the IDS domain, all attacks must be detected; thus, signatures must not be consumed once matched. In Figure 6, S_7 is output twice although S_3 came in once. We first run through the example, then explain the different consumption policies. When S_1 is matched, join nodes JN_1 , JN_5 , and JN_2 are traversed. JN_1 is an OR node which acts as a forwarder to JN_4 . As a result, JN_4 which is an AND node, is accessed through its left side and the left bit is set to one. JN_5 is also an AND node and its left bit is set to one. Finally, the state bit of JN_2 is set to one as it is a NEXT node, and it is accessed through its left input link.

When S_3 is matched, JN_4 is accessed first and its right bit is flipped to one. As a result, both bits of JN_4 are set to one and the AND condition evaluates to true; thus, S_7 is matched. Next, the right bit of JN_3 is set to one.

When S_2 is matched, join nodes JN_1 , JN_2 , and JN_3 are accessed. The OR Node, JN_1 , acts as a forwarder and since both bits of JN_4 have been previously set to one, S_7 is output once again. Next, JN_2 is accessed through its right link and since the state bit is one (S_1 previously matched), the condition evaluates to true and S_5 is matched. Finally, the left bit of JN_3 is set to one and the AND condition is satisfied as S_3 has been previously matched. As a result, S_4 is output as a match and JN_5 is accessed through its right input link. As both bits of JN_5 are set to one, S_6 is matched.

After S_7 was matched in the bottom left AND join node (JN_4), the bits of the JN_4 were not reset and any further occurrence of S_1 , S_2 , or S_3 can trigger the match of S_7 as the AND condition has already been satisfied. This notion of no-consumption policy is crucial to correlate all attack requests and to flag any MPDU combination that can trigger an attack. Nonetheless, a consume-once policy can be used in which join node bits are reset when the node is satisfied. For example, when S_7 is matched, both bits of JN_4 are reset to zero. When S_2 is matched, only the left bit of JN_4 will be set to one; thus, the AND condition is not satisfied. This

Figure 6: Memory Conscious Network Example.

policy will ensure that S_7 will be matched once as S_3 occurs only once. However, we adopt the first policy as it can detect larger number of attacks.

Consumption Policy Analysis: The no-consumption policy can result in deep network propagation as bits are not reset when the join node condition is satisfied. Once a condition is satisfied, it is always evaluated to be true which increases the average traversal depth of a signature match. Despite the disadvantage of longer traversal, no-consumption policy has a better attack coverage as all possible PDU combinations are considered. However, the consume-once policy has a shorter traversal path as once a join node condition is satisfied, the state bits are reset to zero. Future access to the same join node does not result in a match since the condition is evaluated again. The consume-once policy shortens the traversal path but incurs the overhead of resetting the state bits when a join condition is satisfied. Also, it detects only one MPDU combination for the attack. Consider a sample run of S_2 , S_3 , and S_2 for the same network given in Figure 6 and consume-once policy is used. S_4 is matched once since only the $\{S_2, S_3\}$ combination is detected. However, the no-consumption policy considers S_3 twice and can detect both combinations $\{S_2, S_3\}$ and $\{S_3, S_2\}$; as a result, S_4 is matched twice.

MCN Modes of Operation: The scheme described above can detect attacks within one connection or across multiple connections. Distributed denials of service (DDoS) attacks require examining multiple connections to correlate PDUs from different clients [18]. However, other attacks, such as Conficker, require isolation between connections [15]. Consider a login scenario in which an attacker first logs in to the server and then launches the attack. If attacker A is successfully logged in to the server and attacker B launches the attack, correlating the two PDUs can result in a false positive as attacker B is not logged in to the server and his request will eventually be denied.

Per-connection correlation is supported by maintaining separate join node bits per connection. The MCN network is shared among all connections but only the state bits are local to the connection being processed. To correlate attacks across multiple connections, join node bits are defined globally and are shared by all connections.

MCN Algorithm: The MPDU component is added on top of the single PDU matcher and the MPDU signature set is compiled to build the MPDU network. Each MPDU signature is composed of several single PDU signatures and every two PDU signatures are combined into one node using parenthesis to incorporate precedence. Algorithm 3 depicts the steps taken to evaluate a node. A signature node simply

triggers the evaluation of all join nodes without any processing. A join node, however, checks the corresponding bits based on its operator type and once its conditions are satisfied, it outputs all MPDU signatures associated with it and triggers further evaluation of all join nodes connected to its output links. As we adopt a no-consumption policy, the state bits are not reset when the join condition is satisfied.

Algorithm 3 Memory Conscious Network (MCN)

Function SignatureNodeEvaluation()

for *link* in *AllOutputLinks* do
call *JoinNodeEvaluation*(*SignatureNodeID*) of *link*

Function JoinNodeEvaluation(ID)

if AND Operator then
 $bit[ID] \leftarrow 1$
if both bits are set to 1 then
All corresponding signatures are matched
else
return
else
if NEXT Operator then
if ID is the first signature in the sequence then
 $bit[ID] \leftarrow 1$, return
if $bit[first\ signature] = 1$ then
All corresponding signatures are matched
else
return
else
if OR Operator then
All corresponding signatures are matched
for *link* in *AllOutputLinks* do
call *JoinNodeEvaluation*(*JoinNodeID*) of *link*

MCN Complexity: The total memory used by our algorithm to maintain state is proportional to the number of operators in the MPDU signature set which is equal to $O((AND_Nodes * 2 + NEXT_Nodes)/8)$ bytes since the AND operator requires two bits, the NEXT operator requires one bit, and the OR operator requires no bits. When processing 100 MPDU signatures, this number is in the order of twenty bytes per connection which is feasible to support given the memory capacity of nowadays servers (4+ GB) and it is significantly smaller than the memory requirement of the basic Rete algorithm which requires storing matched PDUs (approximately 1500 bytes per PDU) in the join nodes. A typical IDS can handle thousands of simultaneous connections where MCN consumes only few MB for all connections. The memory footprint of MCN is significantly smaller when compared to the memory capacity of current servers (4+ GB) or the requirement of the Rete algorithm (hundreds of MB).

The time complexity is determined by the depth of the network traversed. When a single PDU signature is matched, the ID is hashed to index the appropriate signature node. Next, all output links of the signature nodes are traversed and any join node whose condition is satisfied can trigger further evaluation of its output join nodes. The ID hash is $O(1)$ while the join node time complexity is $O(links * Nodes_Traversed)$.

6. EXPERIMENTS

We first evaluate the proposed matching algorithms over the HTTP protocol when processing a wide range of traffic and compare them to other existing techniques [15, 31].

Figure 7: 600 HTTP Attacks Scenario

Next, we study the matching time of the MPDU algorithm which was designed to detect more sophisticated attacks such as the Conficker worm. All algorithms are implemented in C++ and are integrated into the Netshield framework [15].

6.1 APP and EE Evaluation

We expose the matching engine of the IDS to different workloads to extensively study its performance. Given the largest available vulnerability-based signature set of 794 HTTP signatures [15], we generate new signatures by appending and truncating predicates to extend our database to 20,000 signatures and measure the scalability of the system. Moreover, we evaluate the IDS performance when processing normal traffic (attack-free), attacks, partial attacks, and well-known traffic such as the DARPA data set[28].

Attack Resiliency: To model the system behavior under stress attacks, we launch 600 HTTP attacks and vary the number of signatures from 1,000 to 20,000. Figure 7 shows degradation in performance of the CS algorithm due to the expensive calculation of the candidate selection list (S_i) and the previous rule blocks list (A_i) at every iteration. Since there is a match of one of the signatures, R_1 for example, S_i will always include R_1 for every iteration i and A_i will sometimes include R_1 whenever one of the predicates of R_1 is matched. The expensive operation, $\oplus$, of scanning A_i for every element in S_i hinders performance as R_1 will be carried forward all the way to the end. Furthermore, the CO algorithm requires evaluation of all `runtime_counters` only once at the end of the message but scanning all counters still hinders its performance since most of the counters are not modified and thus must not be checked.

The other algorithms (APP and EE) maintain a candidate list and only `runtime_counters` in this list are evaluated at the end of the message. As R_1 is carried forward across iterations, one time evaluation at the end of the message rather than for every iteration, as is the case for the CS algorithm, yields better performance and allows for matching more signatures. As the run time evaluation of APP and EE is determined by the number of partial matches and not the total number of signatures, they gracefully scale without exhausting the matcher as the number of signatures reaches 20,000. Under heavy-load attacks, the CS algorithm exhausts the matching engine and does not scale well as the matching time increases from 500 μ s to over 3 ms when going from 1,000 to 20,000 signatures. Our proposed algorithms exhibit 6-fold improvement over the CS algorithm when processing 20,000 signatures for the given traffic.

Partial Attacks: Normal traffic can still result in partial matches where no signature is matched but some of its predicates are. For example, the method field in the HTTP

Figure 8: 600 HTTP Partial Attacks Scenario

header is a non-selective predicate and will result in a high number of partial matches as significant number of signatures can have `HTTP_RequestLine.method == "GET"` as a predicate. Moreover, an attacker can exhaust the IDS and force it to drop packets by sending requests that result in a high number of partial matches without sending a full attack and thus will not be detected. Algorithmic attacks [9] that induce worst-case behavior on the IDS is an example of such attacks and it is crucial to study how the proposed algorithms behave.

Figure 8 shows the performance of the matching algorithms when processing 600 HTTP partial attacks against 1,000 to 20,000 signatures. The access predicates in this experiment are not matched while other predicates in the system are matched. The CO algorithm is not affected by the type of experiment run as it always performs the same steps regardless of the number of matches thus it behaves poorly in this case as well. However, the APP algorithm only increments counters of signatures whose access predicates have already been matched which reduces the size of the candidate list (zero in this case) and results in smaller matching time.

Despite the access predicate list (AP_List) being empty, the APP algorithm must still scan the predicate list (P_List) at every iteration to check if the `runtime_counter` of any of its signatures is greater than zero (i.e., access predicate is already matched). The EE algorithm scans the P_List and the candidate list simultaneously and stops when all elements of either list are scanned. The AP_List is always empty thus the candidate list is also empty and the EE algorithm does not scan the P_List which results in faster matching time. The CS algorithm exhibits a similar behavior as the candidate selection list (S_i) is always empty since no access predicate is matched and the current rule block list (B_i) is always empty. Since the candidate list (S_i) is empty, the previous rule block list (A_i) will not be scanned although A_i has many elements which improves the matching performance. The scenario described in this experiment is essential to the performance of the IDS as partial matches are the common trend in normal traffic and attackers tend to use this technique to overload the IDS. The EE algorithm behaves extremely well in this case where it requires less than 20 μ s to match 20,000 signatures, thus not affected by such attacks.

Netshield HTTP Signatures: Netshield researchers have converted snort signatures (973 HTTP signatures in version 11/2007) into 794 vulnerability-based signatures [15]. This is the largest publicly available dataset for vulnerability-based signatures which we use to evaluate our algorithms under different scenarios: clean traffic, attacks, and partial attacks. We also study the effect of access predicate selec-

tivity on the matching time of APP and EE. Figure 9 shows that the CO algorithm behaves poorly for all workloads due to the expensive operation of evaluating all `runtime_counters` for every message processed.

To model clean traffic, we capture a HTTP trace of the Middleware Systems Research Group (MSRG) lab at the University of Toronto where users are accessing legitimate websites with an average connection size of 28KB. All other algorithms incur negligible overhead when processing clean traffic as no matches occur resulting in an empty candidate list and no `runtime_counters` are evaluated.

The APP and EE algorithms outperform the CS algorithm when the IDS is under attack as shown in Figure 9. The CS algorithm requires calculation of the candidate list (S_i), the previous rule blocks list (A_i), and the current rule block list (B_i) at every iteration. When the system is under attack, the candidate list (S_i) will include at least one element which must be checked against all elements in A_i for every iteration i . The other algorithms maintain a candidate list whose `runtime_counters` are checked only once at the end of the message; thus, outperforming the CS algorithm and conforming with the results of the attack resiliency experiment discussed previously.

The third histogram in Figure 9 shows the matching time of all algorithms when no full attacks are launched but only partial matches occur as is the case in the partial attacks experiment. The access predicates are not matched but some of the regular predicates are matched. As mentioned earlier, partial matches can occur in regular traffic and an attacker can force partial matches to exhaust the IDS and later launch the real attack. The EE algorithm performs best in this case as the candidate list is empty (no access predicates matched) and thus the predicate list (`P_List`) is not scanned. The APP algorithm has a slower matching time as the predicate list (`P_List`) must be scanned despite the candidate list being empty. In the CS algorithm, although the candidate list (S_i) and the current rule block list (B_i) are empty, the elements in the previous rule blocks list (A_i) are scanned to check if the $field_i$ holds a don't care value. Scanning the previous rule blocks list (A_i) is analogous to scanning the (`P_List`) in the APP algorithm, and thus the two algorithms exhibit similar performance.

The forth histogram in Figure 9 shows the effect of having selective access predicates on the performance of the access predicate aware algorithms. To quantify the selectivity of access predicates, we study a different case of partial matches where some of the access predicates are matched as well as other regular predicates without resulting in a full signature match. This experiment shows the importance of the choice of access predicates in which more selective access predicates result in a better matching time. Since some of the access predicates are matched, the current rule block list (B_i) and consequently the candidate list (S_i) in the CS algorithm are not empty. Moreover, as other regular predicates are matched, the previous rule blocks list (A_i) is not empty as well which explains the slower performance of the CS algorithm in Figure 9. As access predicates are not selective (traffic matches access predicates), the APP and EE algorithms exhibit a comparable behavior since access predicates are matched and candidate lists of both algorithms are similar. The EE algorithm requires scanning the predicate list (`P_List`) and the candidate list simultaneously. Since the access predicates are not entirely selective, the candidate list

Figure 9: Netshield 794 HTTP Signature Set

is not empty and some elements of the predicate list (`P_List`) must be traversed, which explains the slower performance of the EE algorithm in this case when compared to the previous histogram where no access predicates are matched.

We finally test our algorithms against the most common benchmark, MIT 1998 intrusion detection data set (DARPA) [28], which is mainly composed of normal traffic with some intervals of attacks and partial attacks [15, 28]. The last histogram in Figure 9 indicates the poor performance of the CO algorithm due to the evaluation of all `runtime_counters` which is independent of traffic. All other proposed algorithms take on average $5\mu s$ to process one connection while the CS algorithm averages around $9\mu s$ per connection.

As shown in this experiment, the selectivity of access predicates is crucial to the performance of our proposed access predicate aware algorithms (APP and EE). Access predicates guard other predicates in the signature and control admittance of signatures to the candidate list. As a result, maintaining the size of the candidate list as small as possible, which is determined by the selectivity of the access predicates, yields the best performance.

6.2 MCN Evaluation

As there is no prior work to compare our MCN implementation to, we implement a sequential algorithm that loops over all MPDU signatures and matches them one at a time as a baseline for our evaluation. We use the Netshield 794 HTTP signature set to generate 1,000 MPDU signatures as today, no MPDU signature set exists. Given the nature of current attacks and the lack of MPDU signatures, we believe that MPDU signatures will constitute no more than 10% of the total number of single PDU signatures. We base our assumption on the Conficker worm signature which drove us to support MPDU signatures. Conficker exploits a vulnerability in the stateful WINRPC protocol for which 3,519 Snort (RE-based) signatures exist [19]. Those signatures are translated into only 45 vulnerability-based signatures which constitutes 1.3% of RE-based signatures [15]. Given the small number of WINRPC vulnerability signatures (45) and the significant reduction of the signature set when the signature language is more expressive, we believe that 10% is a reasonable value for a transition from single PDU to MPDU signatures. However, we also test our MCN algorithm against larger MPDU signature sets to measure scalability.

First, we devise two sets of experiments in which we study the effect of increasing the number of attacks on matching

Table 1: MPDU Workload Specification

MPDU Signatures (100)	High-Clustering		Low-Clustering	
	Sequential	MCN	Sequential	MCN
Algorithm	Sequential	MCN	Sequential	MCN
Signature Nodes	290	72	490	234
AND Nodes	80	58	180	114
NEXT Nodes	85	68	185	115
OR Nodes	30	20	30	5
Memory per connection (bytes)	31	24	69	44

time and show the efficiency of our MCN implementation when MPDU signatures exhibit high level of sharing single PDU signatures. Table 1 shows the specification used for both experiments for 100 MPDU signatures with an average of 4 predicates per signature (4 single PDU signatures per one MPDU signature). We use the high-clustering signatures for attack resiliency testing and we compare the two signature sets in the second experiment. Finally, we vary the number of signatures from 100 to 1,000 and show that our MCN algorithm scales better than the sequential scan.

Attack Resiliency: The high-clustering workload specified in Table 1 is used in this experiment in which 100 MPDU signatures are built from 72 unique single PDU signatures. This workload exhibits high sharing of single PDU signatures to create 100 MPDU signatures since the sequential algorithm requires 290 signature nodes while MCN requires only 72 signature nodes. As our MPDU algorithm can detect attacks within one connection or across multiple connections, we differentiate the two cases in Figure 10 by suffixing the algorithm name with G. Correlating attacks across multiple connections requires maintaining global state bits that are accessed by different connections, which explains the G suffix in the algorithm name. However, the algorithm with no G suffix only correlates PDUs within one connection and a separate set of state bits is maintained per connection.

Figure 10 shows the average matching time per connection as we increase the percentage of attacks ranging from clean traffic to full attack scenario. An attack is carried out by sending a PDU that matches one of the simple HTTP signatures, thus matching a predicate in the MPDU signature set. The attacks are launched as follows: An attack is chosen at random from a set of malicious requests and for every connection, 1.5 attacks are launched on average. For example, 33% attack means that one in every three connections is malicious while the other two connections carry clean traffic that matches no signatures (single or multiple PDU). The sequential algorithm results in a linear increase in matching time as the number of attacks increases, reaching 250 μ s when fully under attack. The poor performance of the sequential algorithm is expected as it does not exploit sharing of predicates and each signature is evaluated separately. The sequential algorithm is a baseline for our evaluation as no other MPDU solutions exist.

In the case of clean traffic (0% attack), there is no overhead for any of the algorithms as no processing is required since no single PDU signatures are matched. The MPDU algorithm in our architecture is added on top of the single PDU matching block and is only invoked when a single PDU signature has been successfully matched which is not the case for the clean traffic (0% attack). As the percentage of attacks increases from 30 to 100, our MPDU implementation requires less than 25 μ s per connection and is 10 times faster than the sequential algorithm. To maintain partial state of the current MPDU signature set, 24 bytes are needed for MCN while the sequential algorithm requires 29% more bytes as is evident in Table 1. Figure 10 shows

Figure 10: Attack Resiliency (100 MPDU Sigs).

no significant difference between the global and normal case of either algorithm. Correlating messages across different connections can result in more state transitions, while isolating connections requires resetting state bits for every new flow. On average 1.5 attacks are launched per malicious connection which results in about 3 state transitions for the global case. Moreover, isolating connections requires similar overhead to reset 24 or 31 bytes which explains the similar performance of the two implementations.

Clustering Effect: Our MCN implementation clusters common predicates among different signatures to speed up matching. To validate the efficiency of such clustering, we use two sets of 100 MPDU signatures that are built from 72 and 234 distinct single PDU signatures, respectively. The network size of the low-clustering signature set is significantly larger since the number of nodes increases as more distinct single PDU signatures are used which limits the amount of sharing between MPDU signatures. For each connection, we launch 15 attacks that trigger significant number of state transitions (partial matches of MPDU signatures).

Figure 11 shows about 7 μ s increase in the average matching time between the two signature sets when matching is carried out within one connection. The partial state bit number doubles between the two sets, which explains the increase in matching time due to the overhead of resetting 44 bytes for the low-clustering case. When matching is performed across connections (global matching), the matching time still increases as more nodes are traversed for a given attack since attacks from multiple connections are correlated. For the high-clustering workload, the global version performs worse than the per-connection version since a single match results in about 5 state transitions which takes longer than resetting 24 bytes for the other version.

Figure 11: MCN Clustering Effect

Also, the low-clustering workload favors the global version as the number of state transitions decreases to one as few predicates are shared while the per-connection version struggles to maintain low matching time due to the overhead of resetting 44 bytes for each new connection.

Scalability: To ensure the scalability of our MCN al-

Figure 12: MCN Scalability

gorithm, we vary the number of MPDU signatures from 100 to 1000 signatures and launch multiple single PDU attacks. Figure 12 shows that the sequential algorithm behaves poorly as it requires 4ms and 40ms per connection when processing 100 and 1,000 MPDU signatures, respectively. The sequential algorithm is directly proportional to the number of MPDU signatures as a ten-time increase in signatures results in a ten-time increase in matching time. However, our MCN algorithm scales well as the average matching time per connection only increases by a factor of two when going from 100 to 1,000 MPDU signatures. As shown in Figure 12, MCN is 29 times faster than the sequential algorithm when processing 100 MPDU signatures and scales five times more.

7. CONCLUSION

To achieve acceptable coverage of attacks, the intrusion detection system (IDS) must support different matching dimensions and correlate multiple data protocol units (MPDUs). In this work, we presented two algorithms for single PDU matching: Access Predicate Pruning (APP) and Early Elimination (EE). Both algorithms incur negligible overhead when processing clean traffic and exhibit 6-fold improvement over the candidate selection (CS) algorithm when processing 20,000 signatures against heavy-attack traffic. As far as we are aware, this work represents one of the first efforts to support MPDU matching of vulnerability-based signatures in IDSes. We also presented our Memory Conscious Network (MCN) which exploits bit encoding to represent state transitions while maintaining a small memory footprint. Our solution scales five times more when compared to the sequential scan and is 29 times faster when processing 100 MPDU signatures.

8. REFERENCES

- [1] Cisco IOS Intrusion Prevention System Deployment Guide, 2010.
- [2] Conficker's estimated economic cost? \$9.1 billion. Cyber Secure Institute. <http://www.zdnet.com>, 2009.
- [3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC '99*.
- [4] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *CACM '75*.
- [5] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. *ICDCSW'02*.
- [6] R. Bidou. IPS Shortcomings. <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bidou.pdf>.
- [7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *SP '06*.
- [8] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03*.
- [9] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *SSYM '03*.
- [10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01*.
- [11] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *DEBS '09*.
- [12] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence '82*.
- [13] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SICOMP '77*.
- [14] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware '05*.
- [15] Z. Li, G. Xia, Y. Tang, Y. He, Y. Chen, B. Liu, J. West, and J. Spadaro. Netshield: Massive semantics-based vulnerability signature matching for high-speed networks. In *SIGCOMM '10*.
- [16] M. R. N. Mendes, P. Bizarro, and P. Marques. A framework for performance evaluation of complex event processing systems. In *DEBS '08*.
- [17] A. Mukherji, E. A. Rundensteiner, D. C. Brown, and V. Raghavan. Snif tool: sniffing for patterns in continuous streams. In *CIKM '08*.
- [18] V. Paxson. Bro: a system for detecting network intruders in real-time. In *SSYM'98*.
- [19] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99*.
- [20] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. GPX-Matcher: a generic boolean predicate-based XPath expression matcher. In *EDBT'11*.
- [21] M. Sadoghi and H.-A. Jacobsen. BE-Tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space. In *SIGMOD'11*.
- [22] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB '10*.
- [23] M. Sadoghi, H. Singh, and H.-A. Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *DaMoN'11*.
- [24] N. Schear, D. R. Albrecht, and N. Borisov. High-speed matching of vulnerability signatures. In *RAID '08*.
- [25] R. Smith, C. Estan, S. Jha, and I. Siahaan. Fast signature matching using extended finite automaton (XFA). In *ICISS '08*.
- [26] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS '03*.
- [27] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *SP '10*.
- [28] C. Thomas, V. Sharma, and N. Balakrishnan. Usefulness of DARPA dataset for intrusion detection system evaluation. In *SPIE '08*.
- [29] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04*.
- [30] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical report, University of Arizona, 1994.
- [31] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *TODS '94*.