NetShield: Massive Semantics-based Vulnerability Signature Matching for High-speed Networks

Zhichun Li[§] Gao Xia[†] Hongyu Gao[§] Yi Tang[†] Yan Chen[§] Bin Liu[†] Junchen Jiang[†] Yuezhou Lv[†]
[§] *Northwestern University* [†] *Tsinghua University, China*

Abstract **–** Accuracy and speed are the two most important metrics for Network Intrusion Detection/Prevention Systems (NIDS/NIPSes). Due to emerging polymorphic attacks and the fact that in many cases regular expressions (regexes) cannot capture the vulnerability conditions accurately, the accuracy of existing regex-based NIDS/NIPS systems has become a serious problem. In contrast, the recently-proposed vulnerability signatures [10, 29] (*a.k.a.* data patches) can exactly describe the vulnerability conditions and achieve better accuracy. However, how to efficiently apply vulnerability signatures to high speed NIDS/NIPS with a large ruleset remains an untouched but challenging issue.

This paper presents the first systematic design of vulnerability signature based parsing and matching engine, NetShield, which achieves multi-gigabit throughput while offering much better accuracy. Particularly, we made the following contributions: (i) we proposed a candidate selection algorithm which efficiently matches thousands of vulnerability signatures simultaneously requiring a small amount of memory; (ii) we proposed an automatic lightweight parsing state machine achieving fast protocol parsing. Experimental results show that the core engine of NetShield achieves at least 1.9+Gbps signature matching throughput on a 3.8GHz single-core PC, and can scale-up to at least 11+Gbps under a 8-core machine for 794 HTTP vulnerability signatures.

1. INTRODUCTION

Keeping networks safe has been a grand challenge for the current Internet. The outbreak of the Conficker worm/botnet [2] at the end of 2008 shows that remote exploits are still a major threat to the Internet today. The Conficker worm mainly exploited a WINRPC remote code execution vulnerability (MS08-067), infected $9 \sim 15$ million hosts [2]. For such attacks, network-based Intrusion Detection/Prevention Systems (NIDS/NIPSes) are of critical importance because they protect the enterprise or an ISP as a whole including the users who do not apply patches or host-based defense schemes for various reasons (reliability, overhead, conflicts, *etc.*). Operating on routers/gateways, NIDS/NIPS can prevent attacks such as the Conficker worm from spreading.

Two metrics are extremely important for signature-based NIDS/NIPS: accuracy and speed. Accuracy is of particular importance, especially for an NIPS that throttles the connections which are identified as malicious by matching pre-defined signatures. It requires the signatures to be accurate enough, so that the NIPS can drop the packets with full confidence. Meanwhile, NIDSes/NIPSes have to maintain high speed. Most modern enterprises today have 1Gbps links and are moving towards 10Gbps.

1.1 State of the Art

For high speed NIDS/NIPSes, the *de facto* standard approach is to employ regex (regular expression) based matching engines. On the other hand, vulnerability signatures [9, 10, 29] have been proposed, but still have not been used in NIDS/NIPSes due to the low matching efficiency.

Regexes can be easily combined and matched simultaneously in a single pass over the input. Given this nice feature, most commercial NIDSes/NIPSes, *e.g.*, the products from Cisco and Juniper, as well as some open source ones, *e.g.*, Bro [21], use a regex-based matching engine to keep up with line speed. In such engines, each signature is a regex. The content of a connection is treated as a string and is matched against multiple regex signatures simultaneously. The latest research efforts [7, 8, 17, 25, 26, 30] assume regex can provide enough accuracy and focus on increasing its speed while reducing the memory occupation.

However, their assumption "regexes can provide enough accuracy" is questionable. Attackers have already developed polymorphic and metamorphic variations to evade detection [10]. In many cases, the protocol/application semantics and states are required to express the vulnerability conditions [29], which needs context-sensitive parsing. To say the least, even context-free languages have insufficient expressive power. Therefore, theoretically, regexes are *infeasible* to fully avoid false positives and false negatives, which is indeed true in practice. For example, due to the complexity of the NetBIOS/SMB/WINRPC protocol stack, it is almost impossible to write an accurate regex signature to detect the polymorphic versions of the Blaster worm or the recent Conficker worm. Besides, we also find many other similar cases in different protocols such as HTTP and DNS.

In comparison, the seminal work [10, 29] proposed the concept of vulnerability signatures (*a.k.a.*, data patches [13]) that achieve better accuracy than regex-based approaches. As advocated in [9, 29], a vulnerability signature is based on the thorough understanding of both the network protocol and the application context. It leverages semantic information to exactly describe all the possible paths that exploit a vulnerability over the network.

Most previous work on vulnerability signatures focuses on automated vulnerability signature generation [10, 12]. However, how to match a large number of vulnerability signatures efficiently has not been well studied. To the best of our knowledge, all the exist-

SIGCOMM'10, August 30–September 3, 2010, New Delhi, India.
Copyright 2010 ACM 978-1-4503-0201-2/10/08 ...$10.00.

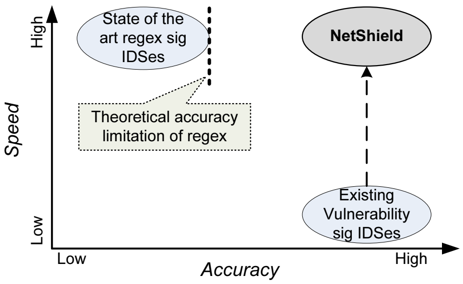

State of the art regex sig IDSes
NetShield
Theoretical accuracy limitation of regex
Existing Vulnerability sig IDSes
Speed
High
Low
Accuracy
Low
High

Figure 1: Comparing NetShield with existing approaches.

ing approaches [9, 22, 29] use *sequential matching*, *i.e.*, matching signatures one by one, leading to low speed.

Since a high-speed NIDS/NIPS protects a large number of diverse hosts, it usually has a large signature ruleset to cover all possible vulnerabilities. For example, Snort has more than 6,000 signatures. Cisco IPS has about 2,000 signatures. For both systems, even some single protocol, such as HTTP, has hundreds or even thousands of signatures. Therefore, matching a large vulnerability signature ruleset at high speed is a practical requirement.

In Figure 1, we compare our approach with existing ones in terms of accuracy and speed. Due to the theoretical limitation, to improve the accuracy of a regex-based approach is extremely hard, if not impossible. On the other hand, the existing vulnerability signature approaches with sequential matching cause low throughput. The *key challenge* is how to speed up vulnerability signature matching with large vulnerability rulesets. Furthermore, another challenge is to parse the traffic and to recover the protocol semantic information fast enough for signature matching.

1.2 Our Contributions

To address the challenges above, we design and implement a vulnerability based NIDS/NIPS prototype (named $NetShield$). NetShield obtains high throughput comparable to that of the state-of-the-art regex-based NIDS/NIPS while offering much better accuracy. In particular, we made the following contributions:

1) An efficient multiple signature matching scheme for a large number of vulnerability signatures. By formulating the multiple vulnerability signature matching problem, we devise a tabular presentation of vulnerability rulesets. This promotes us to design a novel algorithm called *Candidate Selection* (in short, CS) to match multiple vulnerability signatures simultaneously. The CS algorithm enables high-speed massive vulnerability signature matching with small memory requirement. As far as we know, this is the first research effort to formulate and solve the multiple vulnerability signature matching problem (§4).

2) Fast stream-fashioned lightweight parsing. We make two observations: (i) buffering and parse tree traversal are not necessary when the parsing is solely for signature matching; and (ii) These two overhead can be eliminated with proper design. We design the UltraPAC, an automatic parsing state machine generator. The generated parsing state machine can accurately parse out all the required fields avoiding unnecessary cost. Evaluation with real trace shows UltraPAC parser is about 3 $\sim$ 12 times faster than the BinPAC parser [20] (§5).

3) Evaluation and Methodology. By analyzing the vulnerabilities that the Snort ruleset targets, we create the vulnerability signatures for those vulnerabilities. We implement a software NetShield prototype (§6) and release to public [4]. In §7, experimental results show that, on a single-core 3.8Ghz PC, our core engine can achieve 6.7+Gbps parsing speed on HTTP, and 1.9+Gbps parsing plus matching speed for 794 HTTP vulnerability signatures with 2.3MB memory for the matching data structures. On a 8-core machine we boost the matching throughput to 11+Gbps.

```
BIND:
  rpc_vers==5 && packed_drep=='\x10\x00\x00\x00'
  && abstract_syntax.uuid==UUID_IRemoteActivation
  && abstract_syntax.version=="0.0"
BIND-ACK:
  rpc_vers==5
CALL:
  rpc_vers==5 && packed_drep=='\x10\x00\x00\x00'
  && opnum==0x00 && stub.RAbody.actual_length>=40
  && matchRE(stub.RAbody.buffer,
     /^\x5c\x00\x5c\x00/)
```

Figure 2: Vulnerability signature for MS03-026.

After §7, we discuss related work in §8. Finally we present discussions and conclusions in §9 and §10 respectively.

2. BACKGROUND AND MOTIVATIONS

2.1 What Is a Vulnerability Signature?

Vulnerabilities that can be exploited remotely are the result of faulty program logic. They may be triggered when the program handles inputs from networks. Wang *et al.* first propose the concept of *vulnerability signature* [13, 29], and point out that protocol semantic information is particularly useful for specifying such signatures. Brumley *et al.* argue that a perfect vulnerability signature has to be a Turing machine, but unfortunately matching such signatures is undecidable in general [10]. They propose to use symbolic constraints as vulnerability signatures. Similar to their definition, in this paper, we define a vulnerability signature as a set of symbolic predicates based on the protocol semantic information. This form of vulnerability signatures can express most known vulnerability conditions precisely. Based on the principle of optimizing common cases, our design mainly speeds up the matching speed of symbolic predicate signatures. In §9, we show that NetShield can be easily extended to support more complex cases.

To recover the protocol semantic information, we need to parse the input. In addition, a protocol state machine (*a.k.a.*, vulnerability state machine in [29]) is required for adjusting the protocol states when sending/receiving different *protocol data units (PDUs)*. PDUs are the atomic data units that are sent between two application endpoints. A PDU can be dissected into multiple protocol fields. Here, a *field* means a sub-sequence of bytes inside the PDU with certain semantic meaning or functionality. For a multi-PDU protocol, the protocol parser associates the related PDUs to different sessions. The PDUs in one session correspond to a single instance of the protocol state machine. The predicates of vulnerability signatures are defined on the sequence of PDUs in one session, one for each PDU. They are written as a set of *conditions* based on the PDU's protocol fields. If all the predicates required by the vulnerability signature are true, the signature is *matched*.

An example for illustration: As an example, we consider the MS03-026 vulnerability exploited by the Blaster worm. It is a stack buffer overrun vulnerability in the WINRPC protocol. WINRPC is a stateful protocol. A typical WINRPC call starts with a BIND PDU from the client side, asking to bind to a particular API. The server receives the request and responds with the BIND-ACK PDU for acknowledgement. After that, the client issues a remote function call (CALL PDU) using `opnum` as the ID of the function, followed by the required parameters. If the length of the buffer `actual_length` is longer than 40, a buffer overrun will be triggered. Figure 2 shows the corresponding vulnerability signature[1]. Since the signature captures the vulnerability conditions exactly, it can stop the Blaster worm in addition with *all possible* exploits (including future ones) of this vulnerability.

Moreover, given vulnerability signatures are more expressive, the number of signatures can be reduced when covering the same

[1] It is the same as the one generated in ShieldGen [13].

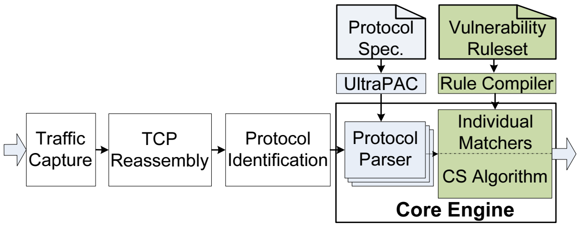


Figure 3: NetShield System Architecture.

vulnerability set. This is especially true for complex binary protocols, which is harder to use regex to express. For example, on average 1.2 Snort HTTP signatures can be reduced to one vulnerability signature, but for WINRPC the ratio is as high as 67.6:1. To some extent, this reduction also helps to improve the performance of signature matching.

2.2 Only Relying on Regex Is Insufficient

Can we accurately express vulnerability signatures in regexes only? The answer is *no*. The reason is twofold.

Regex cannot achieve accurate parsing: Fundamentally, to recover protocol semantics requires context-sensitive parsing. Regex language is a special subset of context-free language. Thus, in theory, it lacks the expressive power to accurately parse complex protocol grammars. In practice, real world protocols are sophisticated enough to render regex incapable. For example, the HTTP protocol encodes the length of chunked HTTP body in another field. Regexes fail to identify the boundary of the body in this case. In the DNS protocol, DNS labels can be either a string with variable length or a pointer. Regexes cannot dereference a pointer. There are similar cases in SSL, SNMP and other protocols.

Regex cannot help with multi-field matching: Regexes assume the input as one single raw byte string. In contrast, vulnerability signatures need to match multiple protocol fields in different data types (number or strings) and combine the matching results to make the decision. For the multi-PDU protocols, the protocol state machine further adds another layer of complexity. It is very hard to extend any regex approaches for vulnerability signatures.

We argue that, although regexes are useful, *only* relying on them is not enough. NIDS/NIPS vendors also realize this problem and add limited semantic processing power to their products. However, these add-ons are relatively ad-hoc and work as “patches” to the systems. These systems are still limited by the regex engines and cannot take the full benefit of vulnerability signatures. In this paper, we advocate that a *systematic and clean slate* design is necessary for protocol parsing and vulnerability signature matching.

3. NETSHIELD FRAMEWORK

Figure 3 depicts the framework of NetShield. There are efficient hardware techniques for traffic sniffing [3], TCP reassembly [14] and protocol identification (classify the traffic to different application protocols) [5]. Our work is focused on the design an efficient core engine for matching vulnerability signatures rather than building a full-featured NIDS/NIPS, which requires an additional heavy hardware/software engineering effort.

For each application protocol, we invoke the corresponding protocol parser which is generated by UltraPAC (our automated parser generator) from the protocol specification. Provided the required protocol fields, the matching engine matches all the vulnerability signatures for the given protocol *simultaneously*. The protocol parser and the matching engine are tightly coupled and work in a pipelined fashion. Whenever a protocol field is parsed, it is immediately sent to the matching engine where the incremental matching process is invoked. The pipelined processing helps reduce the memory consumption, processing delay and maximize the throughput. A full-featured NIDS/NIPS also needs to handle protocol normalization. The current design of UltraPAC only considers the application-level reassembly normalization. Encoding normalization can be incorporated to our design but will remain as part of the future work.

4. EFFICIENT MATCHING DESIGN

We first formulate the vulnerability signature matching problems. After analyzing the nature of the problem, we present the idea of our CS (Candidate Selection) algorithm, followed by the attack resilience analysis and the further enhancement to the basic scheme. Finally, we extend our algorithm to the multiple PDU (Protocol Data Unit) cases. Moreover, in this paper, we use signature and rule interchangeably.

4.1 The Vulnerability Signature Matching Problem

Characteristics and challenges of vulnerability signature matching: Five major characteristics make the vulnerability signature matching problem unique: (i) NIDSes/NIPSes have to keep per-flow state as minimum as possible, because they need to support a large number of sessions and avoid state-holding attacks. Buffering a PDU or even a single string-type field is uneconomical (up to hundreds of KB) and sometimes is subject to state-holding attacks (§4.3). Many variable-length string fields do not have length upper-bounds defined in the protocol. Buffering them is dangerous because an attacker can generate crafted traffic with very long fields to overwhelm the NIDS/NIPS. Moreover, we want to make the decision as soon as possible; thus, it is better to parse a PDU (or a partial PDU) and match the signatures in a streamline fashion, instead of waiting for all the fields available together. (ii) Vulnerability signatures require integer range checking or string length checking. Handling arbitrarily overlapping ranges is nontrivial. (iii) In a signature, different operations, such as integer range checking, string matching and regex matching, operate on different fields and have different nature. It is hard to combine them. (iv) For NIDSes/NIPSes, when multiple rules match a single session, the reporting order of matched rules does not matter. This behavior is different from firewall rules. (v) We have to handle the field dependencies as well as a large number of fields. Furthermore, different signatures may depend on a different set of fields.

These five characteristics in the same time are also research challenges. For example, (i) requires us to design a scheme keeping the per-flow memory usage as small as possible. (ii) and (iii) make the problem a hard multi-field matching problem.

Tabular signature representation: To tackle the problem, we convert the set of signatures to a two-dimensional table, namely *signature table*. This transformation is the *key* to unlock the potential of matching multiple vulnerability signatures simultaneously.

We take two steps to achieve this. First, we normalize signatures to only use $\&\&$ (AND). Any signature that uses $||$ (OR) operator is split into multiple signatures. Second, we convert the normalized signatures to the signature table. Each unique two-tuple ($<$fieldname$>$ $<$operator$>$) is a matching dimension (called a *matcher*), where $<$fieldname$>$ is a protocol field (defined in §2.1) and $<$operator$>$ is the corresponding operator. For example, the (filename ==) is a matcher defined by the filename field in the HTTP URI and the exact string matching operator. Different signatures may require different right operands for the filename field, *e.g.*, filename==“header.php” or filename==“ads.cgi”. One exception is that some protocol fields may form associative array (Perl Hash). For example, the HTTP headers are name and value pairs, which can be treated as an associative array. First, the name of a header needs to be matched. If it is matched, we then match the

RuleID	**RB**	**Matcher 1** **Method ==**	**Matcher 2** **Filename ==**	**Matcher 3** **Filename RE**	**Matcher 4** **VARS ==RE**	**Matcher 5** **Headers ==LEN**
1	1	DELETE	*	*	*	*
2	1	TRACE	*	*	*	*
3	1	POST	header.php	*	*	*
4	2	*	ads.cgi	*	name=“file"; value ∼ “.*\.\./"	*
5	2	*	awstats.pl	*	name=“configdir"; value ∼ “.*7C"	*
6	2	*	fp40reg.dll	*	*	name=“host"; len(value)>300
7	3	*	*	.*\.id[aq]$	*	*
8	4	*	*	*	name=“name"; value ∼ “.*GLOBAL"	*
9	5	*	*	*	*	name=“User-Agent"; len(value)>512

Table 1: A simplified example with nine HTTP signatures on five matchers (matching dimensions).

value condition. Although it is possible to treat them as separate matchers, we find that treating them as a single matcher is easier since most of rules have binding relationships between the names and values.

For N signatures defined on K matchers, we build a $N \times K$ table. This applies to both the single PDU and the multiple PDU case. A row represents a signature, a column represents a matcher, and a cell represents the right operand of the matcher on the signature. If the signature j does not depend on matcher i, we use a wildcard “*” to indicate in the cell. Since both N and K are large, the signature table is usually sparse and has many wildcards. A simplified example is given in Table 1[2]. This example includes five matchers on four protocol fields. Here, operator “`==`” means exact matching, operator “`RE`” regex-matching, Operator “`LEN`” string length checking. `VARS` is the list of variable assignments (name and value pairs) in the HTTP URI, `VARS` and `Headers` can be treated as associative arrays; thus, they are four tuples.

SPMSM: To simplify the discussion, we first define the single PDU multiple signature matching problem (SPMSM)—given a set of signature $\mathcal{S} = \{S_1, S_2, \ldots\}$ and a PDU $\mathcal{P}$, find the signatures in $\mathcal{S}$ that match $\mathcal{P}$. Once solving SPMSM, we extend the solution to the multiple PDU case in §4.5.

Hardness of SPMSM: It is known that general multidimensional range search over N ranges in K dimensions (NKQUERY) has $\Omega((\log N)^{K-1})$ worst-case time with linear memory, or $O(N^K)$ memory for linear search time [11]. This problem can be reduced to the SPMSM problem (runtime excluding the construction time). Therefore, the SPMSM problem's worst case bound will be at least as bad as that of the NKQUERY problem. This implies for the worst case rulesets it is impossible to have a fast linear time algorithm.

Observation on real world rulesets: Since the worst case rulesets have bad theoretical results, we study the nature of real-world rulesets, because, after all, the attackers do not have control over the vulnerabilities in a ruleset. Vulnerabilities are solely determined by the bugs in programs. To design an algorithm with good performance on real-world rulesets is still very important.

After examining a protocol field against a given matcher for all the signatures in the ruleset, the signatures that match the protocol field on the matcher except those with wildcard are called *candidates*. Our observation is that, for a real world vulnerability ruleset, most matchers are selective, *i.e.*, producing small candidate sets.

The observation is made by studying real-world vulnerabilities, mainly the vulnerability signatures corresponding to Snort and Cisco rulesets. String-related matchers are selective, because strings used in signatures are usually long and unique. For the number-related matchers, in most cases this observation still holds. However, it is not true for some matchers that are not crucial to the signatures and are used solely for reducing false positives, *e.g.*, the matcher for checking the WINRPC version field. In other words, they are not rule filters but traffic filters. Fortunately, the protocol fields of such non-selective matchers usually take little space, *e.g.*, four bytes or less. We can always buffer the fields and match them later. In §4.2.3, we show that buffering and matching them later will reduce the matching overhead.

4.2 The Candidate Selection Algorithm

Based on the characteristics of vulnerability signature matching and the observation from real-world rulesets, we design the CS algorithm that want to keep the per-flow state small. We adopt a decomposition design, *i.e.*, to match each matcher separately and then combine the results. For each matcher, we search the candidates for all the rules simultaneously. The key challenge is how to efficiently merge candidates from different matchers.

One possible approach is to use bit-vector to encode matching results. However, for a large ruleset, *e.g.*, 1000 signatures, it needs 1000 bits (125 bytes) per connection, which is memory inefficient. Alternatively, we propose the CS algorithm. By exploiting the rule order, the CS algorithm tracks the rules with “*” cases without explicit states, and thus saves the per-flow memory usage. The CS algorithm only needs to track a small number of candidates for each matcher. We match each matcher separately upon the protocol field's arrival and then iteratively merge the possible candidate rules to produce the final result. Our experiments show the CS algorithm has good throughput and memory usage.

The CS algorithm consists of the pre-computation to decide rule order, the pre-computation to decide matcher order and the runtime process. The pre-computation helps reduce the overhead at runtime. We assume the protocol field arriving order as the matcher order when we introduce rule ordering in §4.2.1. We defer the description of matcher ordering to §4.2.3, because it requires a deep understanding of the candidate selection process at run time shown in §4.2.2.

4.2.1 *Pre-Computation: Deciding the Rule Order*

Based on the characteristic (i) mentioned in §4.1, we exploit the degree of freedom in signature ID order. The key advantage of rule ordering is to track the don't care cases implicitly and save the per-flow states.

We use M_i to denote the ith matcher, *i.e.*, the ith column in the signature table. We call the rules that do not have wildcard in M_i the rules *requiring* M_i. Algorithm 1 shows the rule ordering process. R is initialized as the complete list of rules. We iterate over

```
Algorithm 1 **RuleOrdering**()
$R$ is the list of rules;
**For** $M_i$ in $AllMatchers$
    $RB_i \leftarrow$ the rules in $R$ requiring $M_i$;
    $R \leftarrow R - RB_i$;
**Endfor**
$NR \leftarrow []$;
**For** $i$ from 1 to $K$ append the rules in $RB_i$ to $NR$;
**Return** $NR$;
```

[2] Please note that since we reduce the number of matchers from 31 to 5 for simplifying the example, these rules are not 100% accurate and can only be used as illustration.

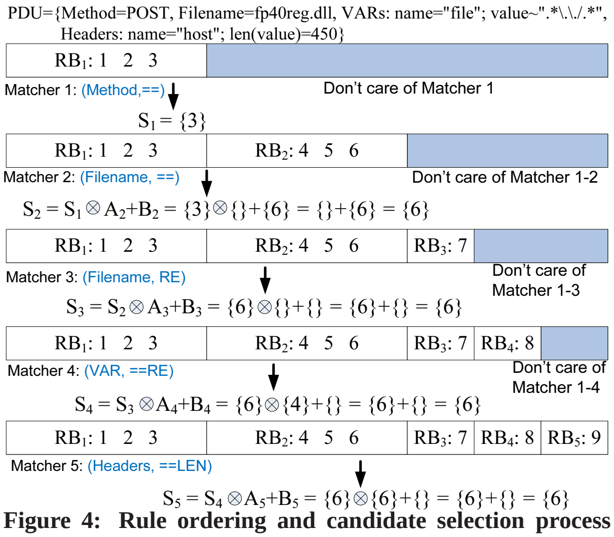

PDU={Method=POST, Filename=fp40reg.dll, VARs: name="file"; value~".*\.\./.*",
Headers: name="host"; len(value)=450}
RB1: 1 2 3
Matcher 1: (Method,==)
Don’t care of Matcher 1
S1 = {3}
RB1: 1 2 3
RB2: 4 5 6
Matcher 2: (Filename, ==)
Don’t care of Matcher 1-2
S2 = S1 ⊗ A2+B2 = {3}⊗{}+{6} = {}+{6} = {6}
RB1: 1 2 3
RB2: 4 5 6
RB3: 7
Matcher 3: (Filename, RE)
Don’t care of Matcher 1-3
S3 = S2 ⊗ A3+B3 = {6}⊗{}+{} = {6}+{} = {6}
RB1: 1 2 3
RB2: 4 5 6
RB3: 7
RB4: 8
Matcher 4: (VAR, ==RE)
Don’t care of Matcher 1-4
S4 = S3 ⊗A4+B4 = {6}⊗{4}+{} = {6}+{} = {6}
RB1: 1 2 3
RB2: 4 5 6
RB3: 7
RB4: 8
RB5: 9
Matcher 5: (Headers, ==LEN)
S5 = S4 ⊗A5+B5 = {6}⊗{6}+{} = {6}+{} = {6}

Figure 4: Rule ordering and candidate selection process of rules in Table 1.

each matcher. At the beginning of the ith iteration, the remaining rules in R have wildcards in all the previous matchers M_1 to M_{i-1}. We select all the rules in R that requiring M_i to construct RB_i, which is the ith rule block. We remove the selected rules from R before the next iteration starts. Finally, we concatenate the RB_i to form the list of ordered rules. Table 1 shows the rules after rule ordering. the RB column is the rule block ID. For example, the rules 1–3 require Matcher 1, so they form the first rule block. Then, because the rules 4–6 require Matcher 2, they form the second rule block.

DataType	Operation	Implementation
number	exactly match	balanced binary search tree
number	range check	balanced binary search tree
string	exactly match	trie
string	regular expression	combined DFA
string	length checking	balanced binary search tree

Table 2: Typical implementations of different matchers.

4.2.2 Runtime Process

In this section, we first illustrate the method to obtain the candidate rules for a single matcher. After that, we propose the iterative approach to combine the candidate rules among multiple matchers. At last, we analyze the complexity of the algorithm.

Single Matcher Matching: For each matcher (column) in the signature table, we check the conditions (cells) requiring that matcher simultaneously by using a searching data structure. For example, when checking which string in different conditions (cells) is equal to the string X, we can build a trie in pre-computation and lookup the string X in the trie simultaneously. As shown in Figure 4, for matcher 1 (method ==), we look up the string “POST” in the trie, and decide rule 3 matches based on Table 1. In Table 2, we describe the example implementations for all types of matchers.

One exception is that, if in a signature the right operand of a matcher is a protocol field (a variable) rather than a constant, we have to match it separately instead of using the implementations in Table 2. Nevertheless, we can still add the result to the candidate set and follow the same candidate selection process (§4.2.2). Fortunately, For the real-world vulnerabilities, those cases are very rare. Thus, the performance will not degrade much. In such a case, the right operand field has to be buffered beforehand. To our knowledge, in all existing cases, the fields are integers, so the buffer overhead is not high.

Algorithm 2 **CandidateSelection_Runtime**()

```
$S \leftarrow \phi$; $T \leftarrow 0$;
encode don't care information into bitmap $MAP$;
For $Matcher_i$ in $AllMatchers$
    Match $Matcher_i$ and get $A_i$ and $B_i$;
    $T \leftarrow T + |RB_i|$;
    $S \leftarrow$ Otimes($S, A_i, Matcher_i, MAP$);
    append $B_i$ to $S$;
    If ($S == \phi$ And $T == N$) Return $\phi$;
Endfor
Return $S$;
Function Otimes($S$,$A$,$MID$,$MAP$) //⊗ operator
    $NS \leftarrow \phi$;
    For $RuleID$ in $S$
        If ($MAP[RuleID][MID]$==1)
            If ($RuleID$ ∈A) add $RuldID$ to $NS$;
        Else
            add $RuleID$ to $NS$;
        Endif
    Endfor
    Return $NS$;
```

Candidate Selection Process: After matching a matcher, we iteratively combine the results with previous matchers’ together efficiently. We define A_i, B_i and S_i as follow. After matching the arrived protocol field against the corresponding matcher M_i, we obtain the set of candidates as $A_i + B_i$. A_i is the candidates of M_i from $RB_1 \ldots RB_{i-1}$, *i.e.*, the rule blocks shared with previous matchers. B_i is the candidates of M_i from RB_i, *i.e.*, the rule block extended by M_i solely. No candidate is from the rule blocks later than RB_i because M_i has wildcards on those rule blocks. S_i is the set of rule candidates after matching matcher $M_1 \ldots M_i$. For S_i, we only need to care the $RB_1 \ldots RB_i$, because the rules after RB_i all have wildcards for $M_1 \ldots M_i$. S_1 can be directly obtained by matching the arrived protocol field against M_1 on rule block RB_1. In general, we use the iteration $S_i = S_{i-1} \otimes A_i + B_i$ to obtain S_i from S_{i-1}. Since S_{i-1} is also from $RB_1 \ldots RB_{i-1}$, we need to merge S_{i-1} and A_i by using a special operation $\otimes$. $\otimes$ is a “special” set intersection with wildcard support. For each element e in S_{i-1}, two ways lead it to $S_{i-1} \otimes A_i$: either e don’t care M_i (has a wildcard) or e in A_i. Since B_i and $S_{i-1} \otimes A_i$ are from different rule blocks, they are mutually exclusive. We simply append B_i to get S_i (achieve set union +). In Figure 4, the arrows show how we obtain the S_1 to S_5 upon the corresponding protocol field arrives. The whole PDU is given at the beginning of the figure. For instance, from Table 1 we know, “POST” will match rule 3, so $S_1 = \{3\}$. Next, we check “fp40reg.dll” against the second column (matcher) in Table 1. When it matches RB_1, we get $A_2 = \phi$, and when it matches RB_2 we get $B_2 = \{6\}$. Then we calculate $S_2 = S_1 \otimes A_2 + B_2$. $S_1 = \{3\}$, and rule 3 requires M_2 but not in A_2; therefore, $S_1 \otimes A_2 = \phi$. Finally, we get $S_2 = \{6\}$.

In Algorithm 2, $|RB_i|$ represents the number of rules in RB_i. The bitmap MAP encodes 0 if a cell in the signature table is a wildcard; otherwise 1. To check whether an element in A_i set can be achieved in $O(1)$ by hash table or TCAM, or in $O(\log(|A_i|))$ by balanced binary search tree. One optimization we add is that, if the candidate set from all signatures ($T == N$) is already empty, we stop the matching right away, before applying the others matchers.

Complexity Analysis: We analyze the complexity of Algorithm 2. In the iteration $S_i = S_{i-1} \otimes A_i + B_i$, A_i reduces the size of S_{i-1} by filtering out some elements, and B_i enlarges the S_{i-1} to get S_i. Because $S_{i-1} \otimes A_i$ and B_i are mutually exclusive, appending B_i to S_i has negligible overhead. This shows another advantage of our scheme that is to decouple the candidate set addition and deletion. The main overhead of the iteration comes from $S_{i-1} \otimes A_i$ which is $O(|S_{i-1}|)$. Therefore, we can use the average of $|S_i|$ ($i \in [1, N-1]$) as the metric to optimize the speed. We denote it as $avg(|S_i|)$. As long as we manage $avg(|S_i|)$ to be small, the overhead will be small. We find $|B_i|$ can be used to bound the $avg(|S_i|)$. $|S_i| \leq \sum_{j=1}^{i} |B_j|$, so $avg(|S_i|) \leq \sum_{j=1}^{N} \frac{N-j}{N-1}|B_j|$. The proof is in our tech. report [19]. The bound is not tight, but it gives us a hint that the matchers at beginning are more crucial since their $|B_i|$ contribute more to $avg(|S_i|)$. In our evaluation (§7.2), we find $avg(|S_i|) < 1.5$ and $max(|S_i|) < 8$ in all the rulesets and traces we evaluate.

For N signatures defined on K matchers, in the worst case ruleset, $avg(|S_i|)$ may have $O(N)$ candidates, requiring $O(K \times N)$ operations in total. However, based on our observation (§4.1), a matcher will usually only have no more than C candidates ($(|A_i| + |B_i|) \leq C$), where C is a small constant. In that case, we can get $O(K)$ speed, indicating the CS algorithm can be very fast. This has been confirmed in our scalability experiment in §7.2.2.

4.2.3 Pre-Computation: Deciding Matcher Order

In general, putting more selective matchers upfront will improve the performance. Suppose M_j is not selective, *i.e.*, $|A_j + B_j| = |A_j| + |B_j|$ is large. Large B_j is worse than large A_j, since it enlarges S_j and produces large overhead for the next iteration. By arranging M_j later, more rules are covered by other matchers. Thus $|B_j|$ will be reduced.

Although matcher reordering can reduce $avg(|S_i|)$, it will bring buffering overhead and increase the memory usage per connection. If we match the matchers in the order decided by the field arriving order, we do not need to buffer any protocol field. In §5.3 we study eight popular protocols, and find that the protocol fields will arrive at an order decided by the protocol. A single field may correspond to multiple matchers; we can put the most selective ones first without additional buffering. In other cases, we have to buffer certain protocol fields because we want to match their matchers later. Since keeping per-flow states small is important, we need to balance between the reduction of $avg(|S_i|)$ and the buffer usage. Here, we assume a limited buffer size ($BufLen$), and try to minimize $avg(|S_i|)$. The buffer can be re-used for different fields with non-overlapping buffer-occupation time. However, This problem is NP-Hard (by reducing Knapsack problem to this problem, proven in our tech. report [19]).

Given the problem is NP-Hard, we propose a greedy algorithm (Algorithm 3) to improve the worst case performance as much as possible. We only reorder the matchers when necessary (the worst case $|B_i|$ larger than the predefined threshold $MaxB$ and when the buffer size allows). The function $estmaxB(M_i)$ returns the worst case (largest) $|B_i|$ when M_i is considered as the next matcher. $EstmaxB(M_i)$ can be calculated in pre-computation. For number fields, we can buffer them directly. However, for the string fields with unbounded length, currently we choose to not buffer them, because the overhead can be too costly.

4.3 Attack Resilience Analysis

We consider two possibilities—attacks specific to the NetShield system and attacks general to any stateful NIDS/NIPS. The performance of NetShield is determined by the signature ruleset and the complexity of protocol parsing. In fact, attackers have no control over either the ruleset or the protocol design, but with the ability to generate the worst case traffic to slow down the processing by introducing more candidates.

To show attack resilience, we demonstrate Algorithm 3 with $MaxB = 10$ and $BufLen = 10$. For WINRPC, we reorder two number-fields using a five-byte buffer. After reordering, we can prove, even under the worst case traffic, $avg(|S_i|)$ bounded by the maximum number of vulnerabilities with the same UUID, because the UUID field is required by all the signatures and very selective. In our evaluation WINRPC ruleset $avg(|S_i|) \leq 3$. For HTTP, we find no reordering is necessary. Since it is hard to prove a bound for HTTP, we generate the worst case traffic to evaluate it. It turns out that to generate the worst case traffic is a NP-Hard problem (proven in our tech. report [19]). We use a greedy approach to maximize the $|S_i|$ at each step. The result shows that the approximate worst case traffic can achieve about 68.4% of the throughput of normal traces, which demonstrates that the CS algorithm works reasonably well under the worst case traffic.

Algorithm 3 **MatcherOrdering**()

```
$OrderM \leftarrow \phi$;
$BUF \leftarrow \phi$;
For $M_i$ in $AllMatchers$
    While ($BUF$ is not empty)
        find $M_j$ in $BUF$ with minimum $estmaxB(M_j)$;
        If ($estmaxB(M_j) \leq MaxB$)
            remove $M_j$ from $BUF$, and append $M_j$ to $OrderM$;
        Else
            Break;
        Endif
    Endwhile
    If ($estmaxB(M_i) \leq MaxB$)
        append $M_i$ to $OrderM$;
    Else
        append $M_i$ in $BUF$;
        While ($len(BUF) > BufLen$)
            find $M_j$ in $BUF$ with minimum $estmaxB(M_j)$;
            remove $M_j$ from $BUF$, and append $M_j$ to $OrderM$;
        Endwhile
    Endif
EndFor
Return $OrderM$;
```

Stateful NIDSes/NIPSes all subject to state holding attacks. Then, the key metric is how many connections (states) can be sustained. Including NetShield, most payload inspection NIDS/NIPSes only create states for successful connections with application payloads. Thus, IP spoofing does not work here. Evaluation shows that our design needs on average 28 bytes/connection for HTTP and 32 bytes/connection for WINRPC. We believe our design is capable to handle millions of connections and thus is robust to attacks.

4.4 Algorithm Refinement

Allow a negative condition. A signature may require a negative condition on a matcher. For example, a signature requires that the regex ".*abc" cannot be matched on the HTTP filename field. For such cases, we can put the signature in S_i by default, and remove it if it is matched.

Handle a list of fields. In many protocols, protocol fields may form arrays or associative arrays. For these cases, the conditions in signatures may need to use "any" or "all" operators. For example, a condition requires that all the lengths of the directories in a URI be smaller than 100. Another example would be checking if any of the lengths of directories is larger than 100. "All" can always be expressed by "not" and "any". Thus, in our design, we just model "not" and "any". The "any" cases are quite common in the vulnerability signatures. For "any" cases, we check each of the elements of the array with the matcher to accumulate A_i and B_i. Some rules may require multiple "any" conditions on the same array to be met simultaneously (an AND relationship). In that case, we just treat each such condition as a separate matcher, and use the CS algorithm to merge them.

Handle the mutually exclusive fields. We treat the matchers related to mutually exclusive fields as a group. If one of the mutually exclusive fields is present, we know all the other fields will not appear. Thus, we can directly delete all the candidates belonging to them. Then, we skip the whole group of matchers and continue with the next matcher not in the group.

4.5 Extension to Multiple PDU Matching

Most simple multiple PDU protocols do not have transition loops in their protocol state machines. We can directly extend the single PDU CS algorithm to these protocols. Without transition loops, the fields arrive sequentially, which is similar to a single PDU case. For the protocols with transition loops, we need to make an enhancement. Basically for each transition loop, the protocol goes back to a previous state and resends certain PDUs. We use the concept of checkpoints to save the scenarios of the candidate selection process, so that when it jumps back to an old state, we can restore the checkpoint and start from there again. What we need to save in the checkpoints are the candidate set S_i of the last matcher of the previous PDU and the position of the buffer at that instant.

5. AUTOMATIC LIGHTWEIGHT PARSING

We need protocol parsing to recover the protocol fields for vulnerability signature matching. However, manually building protocol parsers is tedious and error-prone. We design and implement UltraPAC, an efficient automated protocol parser generator. UltraPAC generates C++ code for protocol parsing, given a program-like protocol description. The generated parser is specially designed for signature matching and is much faster than the state of the art. Meanwhile, it is general enough to handle all protocol and language features supported by BinPAC [20].

5.1 Stream Parser For Signature Matching

For the ease of description, we make some definitions as follows: each PDU corresponds to a protocol parse tree (concrete syntax tree), which is a hierarchy of protocol fields. The leaf nodes of the parse tree correspond to simple data objects, including numbers and strings. The inner nodes correspond to more complex data objects, such as arrays of numbers, strings or C-like “struct” or “union”. We define the leaf nodes as *basic fields* and the inner nodes as *compound fields*.

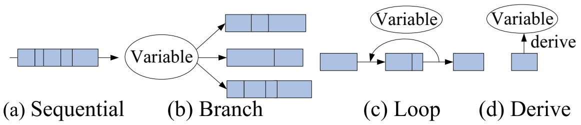
BinPAC [20] and GAPA [9] are two major efforts towards building yacc-like tools for *tree parser* generation, *i.e.*, the parser that reconstruct the protocol parse tree. Both use recursive descent parsers (top-down parsers).

However, we are targeting at the protocol parsing problem solely for signature matching. The *stream parser* is sufficient for this purpose, *i.e.*, the parser that recovers protocol fields consecutively from a input stream. We make three important distinctions.

First, a parsed field is used once by the matching engine and never used again. The parser does not have to preserve a copy of it anywhere. Second, we only need to parse the fields which are either directly required by the vulnerability signature matching (*Type-I fields*) or indirectly required for parsing Type-I fields (*Type-II fields*), *e.g.*, the `header.qdcount` field in DNS protocol that specifies how many question records the PDU has. Third, Type-I and Type-II fields are basic fields in dominant cases, as demonstrated in Figure 2. We further validate this observation by studying the vulnerabilities targeted by Snort and related literature [9, 10, 13, 29].

5.2 Limitation of Existing Work

The BinPAC parser is faster than the GAPA one, so we focus our analysis on BinPAC. We divide the major overhead into three parts: (a) buffer management, the cost to copy network traffic into the buffer and expand/shrink buffer at runtime, (b) parse tree traversal, the cost to construct and take down tree nodes and the recursive descent parsing function calling, and (c) protocol field extraction, the cost to compute the field length, record starting and ending position, update parsing state, *etc.*



(a) Sequential (b) Branch (c) Loop (d) Derive

Figure 5: The building blocks of parsing state machine.

BinPAC is not optimal for our protocol parsing purpose, *i.e.*, for signature matching. Ideally, overhead (a) should be removed, since the parser only needs to record and return the starting and ending position of the protocol field, even in the rare case where one field is separated in several packets. Overhead (b) is also unnecessary because the parser does not need to construct the complete view of the protocol parse tree. A much simpler approach to search along the traffic for the wanted basic field is sufficient in dominant cases. When a compound field is needed in signature matching, it can be constructed from the basic ones. However, it happens very infrequently and does not affect the overall performance significantly. For overhead (c), the extraction of all compound fields can be eliminated due to the same reason.

Unfortunately, these overheads are by design *inherent* to BinPAC parser, and thus cannot be removed by implementation optimization. Since there is no restriction on what the parsed fields are used for, BinPAC parser must handle the worst case where the whole protocol parse tree is required by other components in the system. As a result, it *must* preserve a copy of the parsed fields, traverse the protocol parse tree and parse all nodes.

5.3 Proposed Parsing State Machine

We devise the parsing state machine (called PSM later on) to achieve stream parsing and to eliminate the identified unnecessary overhead to the maximum extent. Please see § 7.2.1 for the experimental results.

We have studied eight popular protocols: HTTP, FTP, SMTP, eMule, BitTorrent, WINRPC, SNMP and DNS. We find three common relationships among fields: $sequential$, $branch$ and $loop$. Sequential fields appear in the PDU one after another in a fixed order. For fields with branch relationship, one and only one will appear. A condition called *branch variable* controls the branch. A loop field will appear repeatedly in the PDU until a *termination condition* is satisfied. In addition, Type-II fields might derive parsing variables which control the parsing process.

Based on these findings, we propose the PSM. A state is the basic field that is being parsed. The state transition marks the end of parsing the previous field and the start of parsing the next one in the PDU. Before quitting a state, it will derive the parsing variables, if any. We show the four basic building blocks of protocol parsing state machine in Figure 5. A PSM is a combination of these basic building blocks.

An Example. In Figure 6 we illustrate a simplified PSM for WINRPC protocol. We merge the fields which are not related to Type-I and Type-II fields as $merge_i$ fields to save space.

The parser continuously fetches the length of current field and moves the offset pointer in the input data segment accordingly. For example, to parse the WINRPC header, the offset pointer increases by 1, 1, 1, 1, 4, 2, 6 in each parsing step, respectively. Since the `ptype` and `frag_length` are needed as parsing variables, they are loaded into variables (R_i). If `ptype` $==$ `BIND_ACK`, we can directly jump over the remaining payload (field merge$_3$) in the PDU by `frag_length` $- 16$ bytes. If `ptype` $==$ `BIND`, we go through the parsing states in the lower right part of the graph.

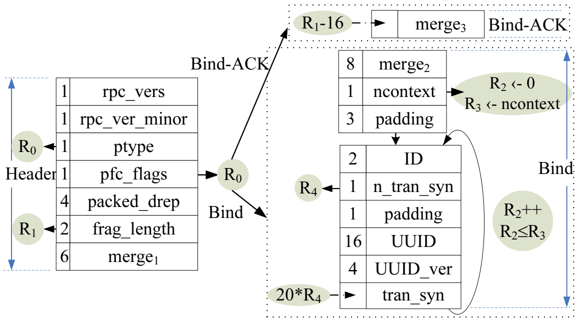


Figure 6: A simplified parsing state machine for the WINRPC protocol (not considering RPC reassembly).

5.4 Automatic Parsing State Machine Generation

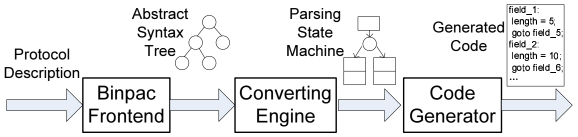


Figure 7: Workflow of UltraPAC.

We leverage on the frontend of BinPAC and reuse the BinPAC language. After that, our customized converting engine produces the parsing state machine (PSM) for the input protocol. Figure 7 shows the workflow of UltraPAC. The code generation step is straight forward, so we omit detailed description due to space limitation.

5.4.1 *Acquiring Abstract Syntax Tree*

The first component reads a program-like description of the protocol format, and constructs the abstract syntax tree (AST) out of it. An AST is a tree-like representation of abstracted protocol format. Note that it is different from a protocol parse tree. An AST gives out all the ways that a legal PDU can possibly be built. It can be determined by the protocol format description. On the contrary, a protocol parse tree states how a given PDU is assembled from basic fields. It can only be determined during the runtime of a parsing process.

We reuse BinPAC to acquire the AST. BinPAC constructs the AST of a protocol in memory before it generates C++ code of the parser. We keep the AST for further processing and discard the code generation part. Accordingly, we keep the BinPAC language for the protocol description.

5.4.2 *Converting into PSM*

A converting engine further converts the AST into the PSM. This process is illustrated in Algorithm 4.

In Algorithm 4, $root$ is the root node of the AST, which represents the whole PDU. S is the internal node space maintained by UltraPAC. It contains all nodes that are to be processed. $Record$, $Case$ and $Array$ are the three possible types of a parent node. They indicate sequential, branch and loop relationship among its children, respectively. The loop relation is handled as a special case of branch relation, where the next protocol field is either the current field itself or the subsequent field of the parent. At the end of each iteration, we add the logic of how to derive the parsing variables into the PSM, so that the actual value of the parsing variables can be determined during runtime using the logic. The iteration stops when the node space contains only leaf nodes in the AST.

UltraPAC can essentially handle arbitrarily complex protocol format, since it supports sequential, branch and loop relation among fields. One caveat is that attribute constraints, *e.g.,* field length, may be applied to compound fields, whereas the PSM produced by UltraPAC works directly on basic fields. We tackle this problem by breaking and distributing such attributes to the proper children basic fields. It's feasible because the set of children fields is determined in the AST. In addition, we have studied the BinPAC language and found that all the supported attributes can be properly distributed, while preserving the original functionality.

Algorithm 4 **ParsingStateMachineGeneration()**

```
$S \leftarrow \{root\}$
While $\exists n$, $n \in S$ and $n$ is inner node
    $children \leftarrow$ the set of $n$'s children
    $S \leftarrow S - n$
    $S \leftarrow S \cup children$
    If $n$ is of $RECORD$ type
        assign sequential ordering among $children$
    Elseif $n$ is of $CASE$ type
        assign branch ordering among $children$
        branch variable controls the branch
    Else
        assign branch ordering between $children$ and $next$
        array terminating condition controls the branch
    Endif
    add logic to derive parsing variable from $n$
Endwhile
```

5.5 Further Improvement

Multiple layer parsing. One application protocol may tunnel through another and use the latter one as a transport layer. One PDU body can be in multiple messages. Therefore it needs application layer reassembly. For example, by treating WINRPC as two sub-protocols and using two layers of PSM, we can solve the reassembly problem. Once the first layer parses the header, we call the second layer to parse the partial body and save the parsing states as well as the offset pointer. Then, after the next message arrives, we can continue parsing the remaining part of the PDU.

Combine the unnecessary fields. A data flow analysis can be adopted to combine the consecutive fields that are neither Type-I nor Type-II fields into one field whenever possible. This combination further simplifies the parsing process without affecting the signature matching.

6. IMPLEMENTATION

6.1 Core Engine Implementation

Parsing: We implement UltraPAC partially based on BinPAC. As shown in Figure 7, we reuse the BinPAC language and code to construct the abstract syntax tree. Accordingly, we use the protocol specification distributed with BinPAC with minor revision. We implement the converting engine and code generator with about 3,000 lines of C/C++ code.

Matching: We implement the three types of matchers and the CS algorithm with about 6,800 lines of C/C++ code. We implement a path-compressed trie for exact string matching and leverage binary search for integer range checking. In addition, we use `Ragel` for regex compilation and write our own code for regex matching.

6.2 Signature Rule Language

We design a language to describe the symbolic predicate signatures. We want to make it simple, intuitive and sufficient. To this end, we have studied the vulnerabilities that the Snort and Cisco rulesets target, as well as those studied in [9, 10, 12, 13, 29], and develop the language features which meet the real-world needs. Here, we briefly introduce the core features.

We support three types of Boolean operators: &&, `||`, and `!`and two basic data types: string and number. For the number type, we support the following relationship operations: $==,>,<,>=,<=$

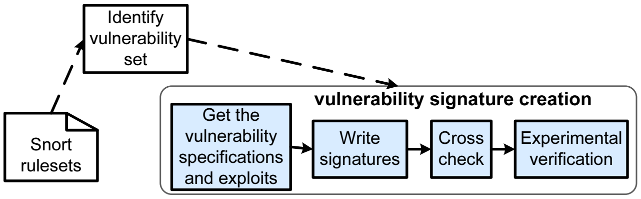

Identify vulnerability set
Snort rulesets
vulnerability signature creation
Get the vulnerability specifications and exploits
Write signatures
Cross check
Experimental verification

Figure 8: Manual vulnerability signature creation.

and ! =. For the string type, we support the `len()` and `match_re()` functions and the exact matching == comparison. In addition, we enable arrays and associative arrays. For instance, `dirs` is an array of directories in the URI. We use `any(dirs)` to represent any element of the array and `len()` function to get the array length. For associative arrays, we support the mapping operation. For example, `len(HTTP_Headers["Host"])>300` means that if the string length of the value corresponding to the key "host" in the HTTP_Headers associative array is larger than 300, the condition is true.

6.3 Vulnerability Signature Creation

It would be more objective to evaluate our approach with standardized vulnerability signature rulesets. Unfortunately, given no available existing vulnerability rulesets or open-source vulnerability signature generator, we have to manually create the vulnerability rulesets on our own. To figure out which vulnerabilities we should include in the ruleset, we target the vulnerabilities Snort tries to detect. We focus on HTTP and WINRPC, because they correspond to the two largest rule subsets of Snort ruleset. 794 HTTP and 45 WINRPC vulnerability signatures[3] are manually created based on vulnerability information of 973 HTTP and 3,519 WINRPC Snort rules (11/2007 version), following the workflow shown in Figure 8. We first identified the vulnerability CVE IDs of Snort rules. Each CVE ID corresponds to a vulnerability, so we wrote one rule for it. Next, we collected the vulnerability specifications and exploit samples from various online vulnerability database and hacker forums. We then wrote the signatures. After that, we cross checked the signatures written by different people and found 94.8% agreements. For the remaining 5.2% debatable ones, we set up a vulnerable host, modified the exploits with the similar idea in [13], and further refined the signatures.

6.4 Software Prototype and Deployment

We build a software NIDS prototype to demonstrate NetShield. It currently runs on Windows. We have deployed the prototype at a campus data center of Tsinghua university. We feed the live traffic from a Cisco router that manages the university-wide web servers and computer labs. The average and peak traffic rate is about 20Mbps and 106Mbps, respectively. We have continuously run our prototype online without any identified packet loss.

7. EVALUATION

To evaluate the performance of NetShield prototype, we measure the throughput on different traces across different protocols, networks and time. The results show that in all the traces NetShield can achieve high throughput. For 794 HTTP vulnerability signatures we can achieve 11+Gbps core engine throughput on an eight-core machine. The UltraPAC generated parsers are about 3 $\sim$ 12 times faster than those of BinPAC. The candidate selection based matching is 8.8 to 11.7 times faster than sequential matching for 794 signatures.

7.1 Evaluation Environment and Datasets

We evaluate the NetShield prototype using two platforms: a Pentium IV 3.8GHz single core PC (P4) with 4GB memory, and an eight-core Xeon E5520 2.2Ghz (XE) with 16GB memory. The latter is mainly for evaluating the performance of NetShield on multicore platforms. Because for vulnerability signature matching connections are independent from each other, we dispatch the connections in traffic roughly evenly to the different NetShield core engines running on different CPU cores. The experiment results suggest multi-core platforms can indeed boost the performance. The overall throughput when using the eight cores is nearly $5.5 \sim 7.1$ times of the single core throughput.

Network traces: We captured traces from the aforementioned router at Tsinghua university (TH) and the EECS departmental gateway of Northwestern University (NU). The MIT DARPA 1998 Intrusion Detection Data Sets [1] is also used. Table 3 shows some statistics of the traces. The NU HTTP trace exhibits much longer average flow length than the other two HTTP traces. We find it has less attack traffic and HTTP signalling or error replies (status code other than 200), which have usually short flow length.

Location	TH	TH	NU	TH	NU	DARPA
Protocol	DNS	WINRPC	WINRPC	HTTP	HTTP	HTTP
Start Time	11/2007	11/2007	10/2006	05/2008	10/2006	1998
Duration	159 hours	207 days	765 days	13 hours	32 hours	26 days
App layer size	1.33GB	598MB	1.33GB	15GB	3.96GB	3.89GB
Flow number	17M	681K	2.24M	2.35M	71.9K	1.83M
Avg flow len	77B	879B	596B	6.56KB	55KB	2.13KB

Table 3: The characteristics of the traces.

7.2 Core Engine Performance Analysis

Methodology: We evaluate the protocol parsing and signature matching throughput of the proposed core engine rather than a product-level NIDS/NIPS ,which takes much more engineering effort. Therefore, the throughputs reported are not what a NIDS/NIPS achieves when monitoring a network link online. However, if the core engine is fast, a well-engineered NIDS/NIPS can achieve high throughput, without vulnerability signature matching being the bottleneck. Given existing commercial regex-based NIDSes/NIPSes have already achieved high throughput, we believe it is also achievable for vulnerability signature based NIDSes/NIPSes.

In all the experiments, we pre-load the TCP streams after TCP reassembly as input. Moreover, we process the connections one after another to exclude the flow switching overhead. The reported throughput is application layer throughput, not including the TCP/IP or link layer headers.

7.2.1 Parsing Performance

We evaluate the parsing performance on both single core and multi-core implementation. *Original BinPAC* is the BinPAC distributed with Bro 1.3. We run it in standalone mode rather than combining with Bro. In *Opt. (optimized) BinPAC*, Instead of creating string objects and array objects fully, we only keep the current chunk of a string or the current element in an array in the memory, which reduce the memory copy/alloc/dealloc operations. In UltraPAC, we disable the "combine the unnecessary fields" optimization (in §5.5) for fair comparison and parse *every* basic field according to the protocol specification. The speedup ratio is calculated between UltraPAC and Opt BinPAC on the single core P4.

We evaluate three protocols: HTTP, WINRPC and DNS. HTTP traffic is one of the dominating traffic on the Internet. WINRPC is a multi-PDU protocol that has been heavily exploited. DNS has low throughput in the BinPAC paper [20]. The results are consistent across different network traces.

[3] sample signatures are available at the anonymized website www.nshield.org.

Trace	**TH DNS**	**TH WINRPC**	**NU WINRPC**	**TH HTTP**	**NU HTTP**	**DARPA HTTP**
Throughput(Gb/s)						
Original BinPAC(P4)	0.10	1.37	1.04	2.02	13.00	1.52
Opt BinPAC(P4)	0.31	1.41	1.11	2.10	14.21	1.69
UltraPAC(P4)	3.43	16.19	12.90	7.46	44.41	6.67
Speed Up Ratio(P4)	11.2	11.5	11.6	3.6	3.1	3.9
Throughput(Gb/s)						
UltraPAC(XE 1core)	3.63	19.88	12.78	7.86	42.22	6.64
UltraPAC(XE 8core)	23.75	123.18	91.09	48.67	295.09	42.31
Max. Memory Per Conn. (Bytes)	16	15	15	14	14	14

Table 4: Parsing results.

Throughput: Table 4 shows that on the single core P4 our parsers parse WINRPC at 13+ Gbps, DNS at 3.4Gbps and HTTP at 6.7+ Gbps. Using eight cores the throughput further increases by 6 ∼ 7 times. Comparing with the Opt BinPAC, we speed up binary protocols (DNS and WINRPC) about 12 times, and a text protocol, HTTP, about 3 ∼ 4 times.

The original BinPAC's throughput here is higher than that in the BinPAC paper [20], because the BinPAC paper measures the throughput with Bro together, which includes TCP reassembly and other lower layer overhead. Our measurement excludes such overhead.

Protocol Trace size (MB)	HTTP 200		DNS 140	
Parser	BinPAC	UltraPAC	BinPAC	UltraPAC
Func call # (K)	12,949	4,850	91,394	1,685
Mem copy/alloc/dealloc time	23%	6%	76%	≈0%

Table 5: Execution profiling results.

To further understand this performance boost, we profile the execution of both BinPAC and UltraPAC parser using the same sample traces. Table 5 shows: 1) UltraPAC parser heavily reduces the number of function calls; and 2) it spends much smaller portion of the execution time on memory copy, allocation and deallocation. Among the three major overheads of BinPAC parser (§5.2), *buffer management overhead* is already minimized in our experiment setting. Table 5 mainly confirms the elimination of BinPAC's large overhead on *parse tree traversal*. The elimination of *inner nodes extraction* in UltraPAC parser contributes to the remaining of throughput increase.

Due to the smaller protocol field size, the BinPAC DNS parser suffers from relatively larger overhead on memory operations. That is why UltraPAC gets higher speedup ratio on DNS protocol. The memory operation time of UltraPAC HTTP parser is incurred in the computation of field length.

With BinPAC design, it is not trivial to reduce this part of overhead to a similar level as UltraPAC. We further optimize the BinPAC DNS parser to remove the creation and deletion of parse tree nodes, by reusing preallocated nodes via a linked list. This implementation optimization reduces the execution time by almost 50%, but the performance is still not even close to the UltraPAC parser.

Memory Consumption: The UltraPAC parsers have to maintain the parsing variables required in the parsing state machine. In Table 4, we also report the maximum memory size required per connection. It is no more than 16 bytes for all the three protocols.

7.2.2 Parsing + Matching Performance

Next, we evaluate the matching performance.

Candidate Set Size: We validate the observation in §4.1 (the candidate sets usually are small). For all protocols and traces, the maximum size of the candidate sets is no more than 8. The average size is less than 1.5.

Throughput: We evaluate our candidate selection based matching on both single core and multi-core implementation. Table 6 shows that even on the single core P4 the CS algorithm can achieve about 11∼14Gbps for WINRPC (45 signatures) and about 1.9+Gb/s for HTTP (794 signatures). The throughput on NU HTTP trace is much higher, because it has much longer average flow length, and most of the bytes are contributed by the HTTP BODY field in the HTTP response message. The HTTP BODY field is not required by most signatures and thus involves little matching overhead. Fully using the eight cores can speed up matching 5.5 ∼ 7.1 times than only using a single core on the XE machine, and achieve 11+Gb/s for the 794 signatures. The throughput of WINPRC is higher because of the small number of vulnerabilities.

Trace	**TH WINRPC**	**NU WINRPC**	**TH HTTP**	**NU HTTP**	**DARPA HTTP**
Throughput (Gb/s)					
Sequential(P4)	10.68	9.23	0.34	2.37	0.28
CS(P4)	14.37	10.61	2.62	17.63	1.85
Matching Only Time					
Sequential(P4) (secs)	0.0048	0.33	344.28	12.68	106.74
CS(P4) (secs)	0.0012	0.18	30.46	1.08	12.16
Speed Up Ratio(P4)	4	1.8	11.3	11.7	8.8
Throughput (Gb/s)					
CS(XE 1core)	18.25	12.03	3.02	19.90	2.01
CS(XE 8core)	118.61	84.69	18.48	128.57	11.00
Avg Memory Usage Per Connection (Bytes)	32	32	28	28	28
Avg # of Candidates	1.16	1.48	0.033	0.038	0.0023

Table 6: Parsing+Matching results.

We implement the sequential matching with short-circuit evaluation, *i.e.*, a signature is skipped upon the first condition that does not meet. In Table 6, the matching only time is obtained by subtracting the parsing time from the parsing+matching time. The speedup ratio is computed as the matching only time of sequential matching over the matching only time of our scheme on the P4. For HTTP, we speed up the matching by 8.8 ∼ 11.7 times. For WINRPC, we speed up the matching by a factor of two to four although there are only 45 signatures.

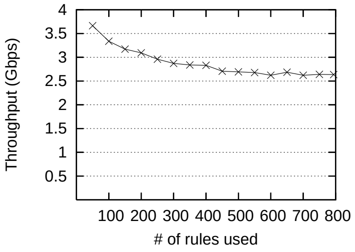


Figure 9: Scalability to the # of rules.

Scalability: Figure 9 shows the scalability of NetShield in terms of throughput under increasing number of rules, evaluated with an 1GB TH HTTP trace. The system throughput degrades gracefully when increasing the number of rules. This is because the throughput mainly depends on the number of matchers, *not* the number of rules. In the beginning, including more rules will add more matchers, and thus increases the overhead. After about 400 rules, all matchers have been included; thus, the throughput remains stable.

Memory Consumption and Breakdown: There are two types of memory consumption: the matching data structures for representing the ruleset shared by all the connections and the memory states maintained for each connection. Table 7 shows the memory usage of the matching data structure on 794 HTTP vulnerability signatures and the breakdown. We only need about

DFA	5.29GB
NetShield	2.3MB
NetShield Breakdown	
CS Matching	12.8KB
Trie	1.4MB
DFA	907KB
Integer Range Checking	0.3KB

Table 7: ***Size of matching data structures on 794 HTTP signatures.***

2.3MB memory. The small memory usage is because vulnerability signatures are defined on multiple protocol fields. Thus the corresponding matching data structure for each field becomes simpler and more memory efficient. For comparison, we also calculate the size of the combined minimized DFA on the same rule set derived from Snort. We employ the same methodology used in [25]. The result DFA size is 5.29GB.

Table 6 also shows the average memory usage per connection, which is dominated by the parsing variables during the parsing. For HTTP protocol, we need 14 bytes for parsing, 12 bytes for keeping three individual matchers' states (a field can map to several matchers, maximum three in the HTTP case), 2 bytes (on overage) for keeping S_i, and zero byte for buffering protocol fields. For WINRPC protocol, we need 15 bytes for parsing, 8 bytes for keeping two individual matchers' states, 4 bytes (on overage) for keeping S_i, and 5 bytes for buffering protocol fields.

Comparing With Existing Regex Approaches: While having better accuracy than regex-based approaches, NetShield is not slower. One of the state-of-the-art regex approaches, XFA [25, 26], reports that, for 863 Snort HTTP rules, it needs 36 bytes/connection and 1.08MB to store the XFA data structures. Their earlier work [25] shows XFA can achieve 75 seconds/GB (108Mb/s) for the Snort HTTP signatures on a 3GHz PC. Their later work [26] is about seven times faster (Figure 9 in [26]). Therefore, we estimate it can achieve about 756Mbps on the 3GHz PC. On the other hand, for 973 Snort HTTP rules, NetShield needs 28 bytes/connection and 2.3MB on shared data structures. It can achieve $1.9 \sim 17$ Gbps for HTTP on a 3.8GHz single core P4. Due to the lack of their code, we cannot make a direct comparison. Nevertheless, from the performance metrics above, we believe the performance of NetShield will be comparable to that of XFA. Moreover, by combining NetShield with XFA, we can possibly achieve even better performance with smaller memory requirement.

There are five reasons for NetShield to obtain similar or even better performance comparing with regex-based approaches: (i) we achieve fast protocol parsing; (ii) after parsing, the protocol fields not used in signatures can be directly skipped, but regex-based approaches need to match every byte; (iii) the CS algorithm introduces very little overhead; (iv) the matching operation for each protocol field is simple; (v) multiple regex rules can be converted to one vulnerability signature (especially for binary protocols) to reduce the ruleset size.

Comparing With Existing Vulnerability Signature Approaches: All existing approaches [9, 22, 29] use sequential matching. We show our scheme speeds up $8.8 \sim 11.7$ times over sequential matching for 794 signatures. We believe that the larger the ruleset, the bigger the speedup ratio.

Worst Case Traffic for the HTTP Ruleset: As mentioned in §4.3, for WINRPC, we prove the CS algorithm works well even in the worst case traffic. For the HTTP ruleset, we use the greedy algorithm to generate the worst case traffic given the problem is NP-Hard. We generate two synthetic traces with same traffic characteristics, except that one is the worst case traffic but the other one is normal. For the trace of the worst case traffic, our scheme can achieve 64.8% of the throughput of the normal one. It shows our scheme works reasonably well even in the worst case.

7.3 Accuracy Evaluation

Previous work [9, 13, 22, 29] has already demonstrated that vulnerability signature is much more accurate than existing regex-based approaches. The results shown here are mainly to confirm that NetShield is also able to achieve good accuracy.

First, we evaluate three WINRPC vulnerabilities Snort tries to detect: Bugtraq 8205, Bugtraq 6005 and MS08-067 (exploited by the recent Conficker worm). We find some of the bit patterns in Snort signature are not related to vulnerabilities. After we change the bit pattern in the exploit code, the exploit still works, but Snort cannot detect the attack and thus has false negatives. If we include the bit patterns in normal requests, Snort will report the requests as false positives. On the other hand, NetShield detects all the polymorphic variants we create from the real exploits accurately.

Furthermore, we evaluate a 10-minute "clean" HTTP trace from TH (1.2GB). In that period, Snort generates 42 alerts while NetSheild generates zero alert. We manually checked those alerts and found they are all false positives.

8. RELATED WORK

Intrusion Detection/Prevention Systems. Snort uses the PCRE library for regex matching guarded by a string matching based prefilter. However, the worst case performance is mainly decided by the PCRE library, which is a NFA based approach and quite slow [26]. Bro is another popular NIDS with a regex signature engine. It can also access semantic information with an expressive policy language, which is close to a general programming language. As the tradeoff, it is hard to optimize its speed for detecting a large number of vulnerabilities.

Regular Expression Matching Engines. The current research of regex matching focuses on improving the matching speed and the memory efficiency [7, 8, 17, 25, 26, 30]. However, as we mentioned, only relying on regexes is not enough. It is very hard to extend these approaches to handle vulnerability signatures.

Protocol Parsing. In §5, we compare NetShield parsing with BinPAC [20] and GAPA [9]. Recently, Schear *et al.* [22] propose the first high-speed parsing design by leveraging on string matching, *i.e.*, to locate the invariant string close to the required field. They demonstrate that their system works well when considering a small number of vulnerability signatures with sequential matching. On the other hand, our goal is to design an automated parser generator that can support a large number of vulnerability signatures.

Packet Classification Algorithms. The SPMSM problem we formulate is related to the classical packet classification problem but is more complex. Both problems are defined on a set of matching dimensions, and allow wildcards. In [28], Taylor classified the packet classification techniques into four categories: exhaustive search, decision tree, tuple space and decomposition. Unfortunately, none of them can be directly applied to the SPMSM problem due to the five characteristics of vulnerability signature matching (§ 4.1).

Ternary Content Addressable Memory (TCAM) uses brute-force hardware parallelism to achieve fast exhaustive search for packet classification. However, it remains unknown how to apply TCAM for the SPMSM problem. For example, currently there are no efficient ways to encode regexes used in vulnerability signatures into TCAM. Decision tree algorithms such as HiCuts [16] and HyperCuts [24] require interleaving different dimensions (*i.e.*, combining them as a big tree), which is quite impossible when dealing with vulnerability signatures since the fields arrive at different time. Moreover, they need huge amount of memory when being used with a large number of wildcards. Tuple Space based algorithms [27] exploit the fact that the five tuples in packet classification are all integers so that the tuple space is small. For vulnerability signatures with many long string fields, the tuple space can be very large. Also, it cannot handle regex matching which is required by vulnerability signatures. The decomposition based approaches are not suitable either. For example, the recursive flow classification (RFC) [15] partitions all fields into fixed-size chunks. It is remain unknown how to extend the scheme to work with variable-length string fields and regular expression matchers. Bit vector approaches [6, 18] need large memory ($O(N)$) per connection and high computation overhead.

9. DISCUSSIONS

When a vulnerability logic is deeply embedded in the application, it is hard to directly use the protocol fields in the symbolic predicates to describe the signature. In this case, we have to recover the internal state of the application as the state variables and use them in the symbolic predicates. We achieve this by inserting into the protocol parser a function that partially reproduces the application logic to compute and return the state variables. In this way, we bear the extra computational overhead, but can detect all possible vulnerabilities accurately. For example, Bugtraq ID 599 is a buffer overflow vulnerability in wu-ftpd 2.5. If a deep FTP path is created by making new directories recursively, the buffer will be overflowed. However, neither the path nor the path length is a protocol field. To solve this problem, we insert a customized function into the parser to calculate the path length, and use it in the symbolic predicates. In all the vulnerabilities we have studied including those mentioned in other papers [9, 10, 13, 29], there are only a few such cases.

Another problem is that, when applying vulnerability signatures at network level, ambiguities might arise if we do not know the software variances running on the hosts. It is possible that a flow can trigger a vulnerability on variance A but not B. One solution is to actively map the software variance and their versions on the enterprise network [23]. We argue that, even without knowing the exact version, vulnerability signatures of popular software will not cause false positives since normal traffic will not trigger the vulnerability; otherwise the software will crash often and cannot be popular. Furthermore, different software variances might interpret the protocol slightly differently, which might cause parsing ambiguity. The active mapping approach can help in this case as well.

10. CONCLUSIONS

In this paper, we present NetShield, the first systematic design of vulnerability signature based parsing and matching engine. Essentially, we propose the state machine based parsing execution model and the CS algorithm for fast matching on a large vulnerability ruleset. We also implement, deploy, and release the NetShield prototype. The real trace evaluation demonstrates that NetShield achieves similar speed to that of the current regular expression based NIDS/NIPS while offering much better accuracy.

11. ACKNOWLEDGEMENTS

We gratefully acknowledge our shepherd, Cristian Estan, and the anonymous reviewers for their valuable inputs on earlier versions of this paper. This work was supported in part by US NSF CNS-0831508, and China NSFC (60625201, 60873250), 973 project (2007CB310701), 863 high-tech project (2007AA01Z216) and Tsinghua University Initiative Scientific Research Program.

12. REFERENCES

[1] 1998 DARPA Intrusion Detection Evaluation Data Set. `www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html`.

[2] Conficker. `http://en.wikipedia.org/wiki/Conficker`.

[3] DAG card. `http://www.endace.com/dag-8.1sx.html`.

[4] NetShield Website. `http://www.nshield.org`.

[5] PRX Traffic Manager. `http://www.ipoque.com/products/prx-traffic-manager`.

[6] F. Baboescu and G. Varghese. Scalable packet classification. In *proc. of ACM SIGCOMM*, 2001.

[7] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. of ACM CoNEXT*, 2007.

[8] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proc. of IEEE/ACM ANCS*, 2008.

[9] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *proc. of NDSS*, 2007.

[10] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proc. of IEEE Security and Privacy Symposium*, 2006.

[11] B. Chazelle. Lower bounds for orthogonal range searching. ii: The arithmetic model. *Journal of the ACM*, 37(3):439–463, July 1990.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of ACM SOSP*, 2005.

[13] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. Shieldgen: Automated data patch generation for unknown vulnerabilities with informed probing. In *proc. of IEEE Security and Privacy*, 2007.

[14] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *Proc. USENIX Security Symposium*, 2005.

[15] P. Gupta and N. McKeown. Packet classification on multiple fields. In *proc. of ACM SIGCOMM*, 1999.

[16] P. Gupta and N. McKeown. Classification using hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, Jan 2000.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expression matching for deep packet inspection. In *Proc. of ACM SIGCOMM*, 2006.

[18] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *proc. of ACM SIGCOMM*, 1998.

[19] Z. Li, X. Gao, Y. Chen, and B. Liu. Netshield: Matching with a large vulnerability signature ruleset for high performance network defense. Technical Report NWU-EECS-08-07, Northwestern University, 2009.

[20] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *proc. of ACM IMC*, 2006.

[21] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31, 1999.

[22] N. Schear, D. Albrecht, and N. Borisov. High-speed matching of vulnerability signatures. In *Proc. of RAID*, 2008.

[23] U. Shankar and V. Paxson. Active mapping: Resisting nids evasion without altering traffic. In *Proc. of IEEE Security and Privacy*, 2003.

[24] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *proc. of ACM SIGCOMM*, 2003.

[25] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proc. of IEEE Security and Privacy*, 2008.

[26] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *Proc. of ACM SIGCOMM*, 2008.

[27] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *proc. of ACM SIGCOMM*, 1999.

[28] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[29] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of ACM SIGCOMM*, 2004.

[30] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. of ANCS*, 2006.