Backtracking Algorithmic Complexity Attacks Against a NIDS

Randy Smith
Cristian Estan
Somesh Jha

Technical Report #1561

September 2006

Backtracking Algorithmic Complexity Attacks Against a NIDS

Randy Smith   Cristian Estan   Somesh Jha
Computer Sciences Department
University of Wisconsin-Madison
{smithr,estan,jha}@cs.wisc.edu

Abstract

Network Intrusion Detection Systems (NIDS) have become crucial to securing modern networks. To be effective, a NIDS must be able to counter evasion attempts and operate at or near wire-speed. Failure to do so allows malicious packets to slip through a NIDS undetected. In this paper, we explore NIDS evasion through algorithmic complexity attacks. We present a highly effective attack against the Snort NIDS, and we provide a practical algorithmic solution that successfully thwarts the attack. This attack exploits the behavior of rule matching, yielding inspection times that are up to 1.5 million times slower than that of benign packets. Our analysis shows that this attack is applicable to many rules in Snort's ruleset, rendering vulnerable the thousands of networks protected by it. Our countermeasure confines the inspection time to within one order of magnitude of benign packets. Experimental results using a live system show that an attacker needs only 4.0 kbps of bandwidth to perpetually disable an unmodified NIDS, whereas all intrusions are detected when our countermeasure is used.

1. Introduction

Network Intrusion Detection Systems (NIDS) and Intrusion Prevention Systems (IPS) have become crucial to securing today's networks. Typically, a NIDS residing on the edge of a network performs deep packet inspection on every packet that enters the protected domain. When a packet is matched against a signature, an alert is raised, indicating an attempted intrusion or other misuse.

To be effective in an online environment, packet inspection must be performed at or near wire speed. The consequences of not doing so can be dire: an intrusion *detection* system that fails to perform packet inspection at the required rate will allow packets to enter the network undetected. Worse, an inline intrusion *prevention* system that fails to keep up can cause excessive packet loss.

A NIDS must also guard against evasion attempts which often succeed by exploiting ambiguities in a protocol definition itself. For example, attack mechanisms have relied on ambiguities in TCP to develop evasion techniques using overlapping IP fragments, TTL manipulation, and other transformations [10, 15, 18].

In this paper, we explore NIDS evasion through the use of algorithmic complexity attacks [9]. Given an algorithm whose worst-case performance is significantly worse than its average case performance, an algorithmic complexity attack occurs when an attacker is able to trigger worst-case or near worst-case behavior. To mount evasion attempts in NIDS, two attack vectors are required. The first is the true attack that targets a host inside the network. The second is aimed squarely at the NIDS and serves as a cover by slowing it down so that incoming packets (including the true attack) are able to slip through undetected. Evasion is most successful when the true attack enters the network, and neither it nor the second attack is detected by the NIDS.

We present an algorithmic complexity attack that exploits worst-case signature matching behavior in a NIDS. By carefully constructing packet payloads, our attack forces the signature matcher to repeatedly backtrack during inspection, yielding packet processing rates that are up to 1.5 million times slower than average. We term this type of algorithmic complexity attack a *backtracking attack*. Our experiments show that hundreds of intrusions can successfully enter the network undetected during the course of a backtracking attack against a NIDS. Further, the backtracking attack itself requires very little bandwidth; *i.e.*, a single attack packet sent once every three seconds is enough to perpetually disable a NIDS.

Our countermeasure to the backtracking attack is an algorithmic, semantics-preserving enhancement to signature matching based on the concept of memoization. The core idea is straightforward: whereas the backtracking attack exploits the need of a signature matcher to evaluate signatures at all successful string match offsets, a memoization table can be used to store intermediate state that must otherwise be recomputed. Our defense against the backtracking attack relies on the use of better algorithms that reduce the disparity between worst and average case without changing functionality. Empirical results show that this solution confines the processing times of attack packets to within one order of magnitude of benign packets.

Our result applies directly to Snort [17], a popular open-source package that provides both NIDS and IPS functionality and claims more than 150,000 active users. Snort uses a signature-based architecture in which each signature is composed of a sequence of operations, such as string or regular expression matching, that together identify a distinct misuse. In our experiments, we use Snort over both traces and live traffic. In addition, we provide a practical implementation of the defense by extending Snort's signature matching functionality directly.

In summary, our contributions are two-fold. First, we discuss NIDS evasion through algorithmic complexity attacks. We present a highly effective real attack, the backtracking attack, that yields slowdowns of up to six orders of magnitude and is feasible against the (estimated) tens of thousands of networks monitored by Snort. Second, we present an algorithmic defense, based on the principle of memoization, that confines the slowdown to less than one order of magnitude in general and to less than a factor of two in most cases. We provide a practical implementation of this solution and show its efficacy in a live setup.[1]

We organize the remainder of this report as follows: Section 2 provides a summary of related work, and Section 3 describes Snort's rule-matching architecture. Sections 4 and 5 present the backtracking attack and the countermeasure, respectively. Section 6 details our experimental results, and Section 7 considers other types of complexity attacks. Section 8 concludes. Appendix A contains additional experimental results, and Appendix B briefly describes another class of algorithmic complexity attacks.

2. Related work

To our knowledge, Crosby and Wallach [8, 9] were the first to provide an algorithmic basis for denial of service attacks. They exploit weaknesses in hash function implementations and backtracking behavior in common regular expression libraries to produce worst-case behavior that is significantly more expensive than the average case. The result is denial of service in general and evasion in our context. For their examples, the authors observe that algorithmic attacks against hash tables and regular expressions can be thwarted by better algorithm and data structure selections. Our defense also relies on algorithmic improvements.

The backtracking attack we present falls within the general family of algorithmic attacks, although to the best of our knowledge our method of achieving evasion through backtracking is novel.

In a systems-oriented approach to addressing resource consumption and other attacks, Lee *et al.* [12] dynamically divide the workload among multiple modules, making adjustments as necessary to maintain performance. Load-shedding is performed as necessary to distribute the load to different modules, or to lower its priority. Alternatively, Kruegel *et al.* [11] have proposed achieving high speed intrusion detection by distributing the load across several sensors, using a scatterer to distribute the load and slicers and reassemblers to provide stateful detection. Still other approaches seek to provide better performance by splitting up (and possibly replicating) a sensor onto multiple cores or processors [6, 25]. These approaches show that allocating more hardware can better protect large networks with large amounts of traffic, but they are not a cost effective way of dealing with algorithmic complexity attacks.

The use of custom hardware has also been proposed for performing high-speed matching [3,5,20,23,24]. The backtracking attack is probably not applicable to these solutions. As our focus is on software-based systems, we do not consider hardware solutions further in this paper.

Both [12] and [13] propose the use of monitors to track the resource usage and performance history of a NIDS. In [12], if a monitor discovers abnormally long processing times, the current operations are aborted and optionally transferred to a lower priority process. For [13], on the other hand, the monitor simply triggers a restart of the NIDS. In the general case, such techniques may provide a useful mechanism for ensuring guaranteed minimum performance rates at the cost of decreased detection accuracy. However, such mechanisms result in periodic lapses in detection capability. Our solution is semantics-preserving, in the sense that it does not sacrifice detection to maintain performance.

Finally, NIDS evasion has been extensively studied in the literature. The earliest work describing evasion was presented by Paxson [13] and Ptacek and Newsham [15]. Handley *et al.* [10] show that normalization combined with stateful analysis to remove protocol ambiguities can foil evasion attempts, although it may affect stream semantics. Shankar and Paxson [19] address semantics by providing an online database of network attributes, such as the hop count from the NIDS to a protected host, that provides the same benefits as normalization without the risk of changing stream semantics. These solutions are orthogonal to the problem discussed in this paper.

3. Rule matching in Snort

Our work is performed in the context of the Snort NIDS. Snort employs a signature-based approach to intrusion detection, defining distinct signatures, or rules, for each mis-

[1]We have presented our findings to the Snort developers, who have confirmed the efficacy of the evasion attack and have integrated the solution into their NIDS.

Predicate	Description	Type
`content`:$< str >$	Searches for occurrence of $< str >$ in payload	multiple-match
`pcre`:$/regex/$	Matches regular expression $/regex/$ against payload	multiple-match
`byte_test`	Performs bitwise or logical tests on specified payload bytes	single-match
`byte_jump`	Jumps to an offset specified by given payload bytes	single-match

Table 1. Subset of Snort predicates used for packet inspection. Multiple-match predicates may need to be applied to a packet several times.

```
alert tcp $EXT_NET any -> $HOME_NET 99
 (msg:"AudioPlayer jukebox exploit";
  content:"fmt=";                 //P1
  pcre:"/^(mp3|ogg)/",relative;  //P2
  content:"player=";              //P3
  pcre:"/.exe|.com/",relative;   //P4
  content:"overflow",relative;   //P5
  sid:5678)
```

Figure 1. Rule with simplified Snort syntax describing a fictional vulnerability.

use to be searched for. Each signature is in turn composed of a sequence of *predicates*, that describe the operations that the signature must perform. Section 3.1 gives an overview of the language used to specify these rules. Section 3.2 describes the algorithm used to match rules against packets.

3.1. Expressing rules in Snort

Snort's rules are composed of a header and a body. The header specifies the ports and IP addresses to which the rule should apply and is used during the classification stage. The body has a sequence of predicates that express conditions that need to succeed for the rule to match. A rule matches a packet only if all predicates evaluated in sequence succeed. Of the predicates that are part of Snort's rule language, we focus on those used to analyze the packet payloads. Table 1 summarizes the relevant rules.

Figure 1 depicts a signature using a simplified version of Snort's rule language. The header of the rule instructs Snort to match this signature against all TCP traffic from external sources to servers in the home network running on port 99. The body of the rule contains three `content` predicates, two `pcre` [14] predicates, and two terms, `msg` and `sid`, used for notification and bookkeeping. The rule matches packets that contain the string `fmt=` followed immediately by `mp3` or `ogg`, and also contain the string `player=`, followed by `.exe` or `.com`, followed by `overflow`.

Predicates have one important side effect: during rule matching a predicate records the position in the payload at which it succeeded. Further, when a predicate contains a `relative` modifier, that predicate inspects the packet beginning at the position at which the previous predicate succeeded, rather than the start of the payload. For example, if predicate `P3` from Figure 1 finds the string `player=` at offset i in the payload, the subsequent `pcre` predicate (`P4`) succeeds only if it matches the packet payload after position i.

3.2. Matching signatures

When matching a rule against a packet, Snort evaluates the predicates in the order they are presented in the rule, and concludes that the packet does not match the rule when it reaches a predicate that fails. To ensure correctness, Snort potentially needs to consider all payload offsets at which `content` or `pcre` predicates can succeed. We term these *multiple-match* predicates. In contrast, predicates `byte_test` and `byte_jump` are *single-match*, meaning that any distinct predicate invocation evaluates the payload once.

In the presence of a multiple-match predicate `P`, Snort must also retry all subsequent predicates that either directly or indirectly depend on the match position of `P`. For example, consider matching the rule in Figure 1 against the payload in Figure 2. The caret (ˆ) in `P2` indicates that `P2` must find a match in the payload immediately after the previous predicate's match position. If Snort considers only `P1`'s first match at offset 4, then `P2` will fail since `P2` is looking for `mp3` or `ogg` but finds `aac` instead. However, if Snort also considers `P1`'s second match at offset 28, `P2` will succeed and further predicates from the rule will be evaluated. Snort explores possible matches by backtracking until either it finds a set of matches for all predicates or it determines that such a set does not exist.

Figure 3 presents a simplified version of the algorithm used by Snort to match rules against packets.[2] All predicates support three operations. When a predicate is evaluated, the algorithm calls `getNewInstance` to do the required initializations. The previous match's offset is passed to this function. The `getNextMatch` function checks whether the predicate can be satisfied, and it sets the offset of the match returned by calls to the `getMatchOffset` predicate. Further invocations of `getNextMatch` return true as long as more matches are found. For each of these matches, all subsequent predicates are re-evaluated, because their outcome can depend on the offset of the match.

[2]The Snort implementation uses tail calls and loops to link predicate functions together and to perform the functionality described in Figure 3. The algorithm presented here describes the behavior that is distributed throughout these functions.

```
Payload  fmt=aac player=play 000 fmt=mp3 rate=14kbps player=cmd.exe?overflow
Offset   01234567890123456789012345678901234567890123456789012345678901234567
                   1         2         3         4         5         6

                                                                                                          (P5,59,67)
                                                                                             (P4,51,59)   (P4,51,59)
                                                                                (P3,31,51)   (P3,31,51)   (P3,31,51)
              (P2, 4, f)                                  (P2,28,31)   (P2,28,31)   (P2,28,31)   (P2,28,31)
(P1, 0, 4)    (P1, 0, 4)    (P1, 0,28)    (P1, 0,28)    (P1, 0,28)    (P1, 0,28)    (P1, 0,28)
```

Figure 2. Packet payload matching the rule in Figure 1 and corresponding stack trace after each call to `getNextMatch` on line 3 of Figure 3.

```
MatchRule($Preds$):
1 $Stack \leftarrow (Preds[0]$.getNewInstance$(0))$;
2 while $Stack.size > 0$ do
3     if $Stack.top$.getNextMatch() then
4         if $Stack.size == Preds.size$ then return $True$;
5         $ofst \leftarrow Stack.top$.getMatchOffset();
6         Push($Stack, Preds[Stack.size]$.getNewInstance$(ofst)$);
7     else Pop($Stack$);
8 return $False$;
```

Figure 3. Rule matching in Snort. The algorithm returns $True$ only if all predicates succeed.

The rule matching stops when the last predicate succeeds, or when all possible matches of the predicates have been explored. Figure 2 shows the stack at each stage of the algorithm. Each stack record contains three elements: the predicate identifier, the offset passed to `getNewInstance` at record creation, and the offset of the match found by `getNextMatch` (`f` if no match is found). In this example, the algorithm concludes that the rule matches.

4. NIDS evasion via backtracking

The use of backtracking to cover all possible string or regular expression matches exposes a matching algorithm to severe denial of service attacks. By carefully crafting packets sent to a host on a network that the NIDS is monitoring, an attacker can trigger worst-case backtracking behavior that forces a NIDS to spend seconds trying to match the targeted rule against the packet before eventually concluding that the packet does not match. For the rule from Figure 1, `P2` will be evaluated for every occurrence of the string `fmt=` in the packet payload. Furthermore, whenever this string is followed by `mp3`, `P2` will succeed and the matcher will evaluate `P3`, and if `P3` succeeds it will evaluate `P4`. If `fmt=mp3` appears n_1 times, `P3` is evaluated n_1 times. If there are n_2 occurrences of `player=`, `P4` will be evaluated n_2 times for each evaluation of `P3`, which gives us a total of $n_1 \cdot n_2$ evaluations for `P4`. Similarly, if these occurrences are followed by n_3 repetitions of `.exe` or `.com`, `P5` is evaluated $n_1 \cdot n_2 \cdot n_3$ times. Figure 4 shows a packet that has $n_1 = n_2 = n_3 = 3$ repetitions. Figure 5 shows the evaluation tree representing the predicates evaluated by the algorithm as it explores all possible matches when matching Figure 1 against the payloads in Figure 2 and in Figure 4. Our experiments show that with packets constructed in this manner, it is possible to force the algorithm to evaluate some predicates hundreds of millions of times while matching a single rule against a single packet.

The amount of processing a backtracking attack can cause depends strongly on the rule. Let n be the size of a packet in bytes. If the rule has k unconstrained multiple-match predicates that perform $O(n)$ work in the worst case, an attacker can force a rule-matching algorithm to perform $O(n^k)$ work. Thus the following three factors determine the power of a backtracking attack against a rule.

1. *The number of backtracking-causing multiple-match `content` and `pcre` predicates k.* The rule from Figure 1 has $k = 4$ because it has 4 backtracking-causing multiple-match predicates (including `P5` which does not match the attack packet, but still needs to traverse the packet before failing). Note that not all `content`s and `pcre`s can be used to trigger excessive backtracking. Often, predicates that have constraints on the positions they match cannot be used by an attacker to cause backtracking. An example of such a predicate is the first `pcre` from Figure 1, predicate `P2`, which has to match immediately after the first `content`.
2. *The size of the attack packets n.* We can use Snort's reassembly module to amplify the effect of backtracking attacks beyond that of a single maximum sized packet. The rule from Figure 1 is open to attacks of complexity $O(n^4)$. When Snort combines two attack packets into a virtual packet and feeds it to the rule-matching engine, n doubles, and the rule-matcher does 16 times more work than for either packet alone.
3. *The total length of the strings needed to match the k predicates.* If these strings are short, the attacker can repeat them many times in a single packet. This influences the constants hidden by the O-notation. Let $s_1, \ldots, s_k$ be the lengths of the strings that can cause matches for the

```
Payload  fmt=mp3fmt=mp3fmt=mp3player=player=player=.exe.exe.exe
Offset   0123456789012345678901234567890123456789012345678901234
                   1         2         3         4         5
```

Figure 4. A packet payload that causes rule matching to backtrack excessively.

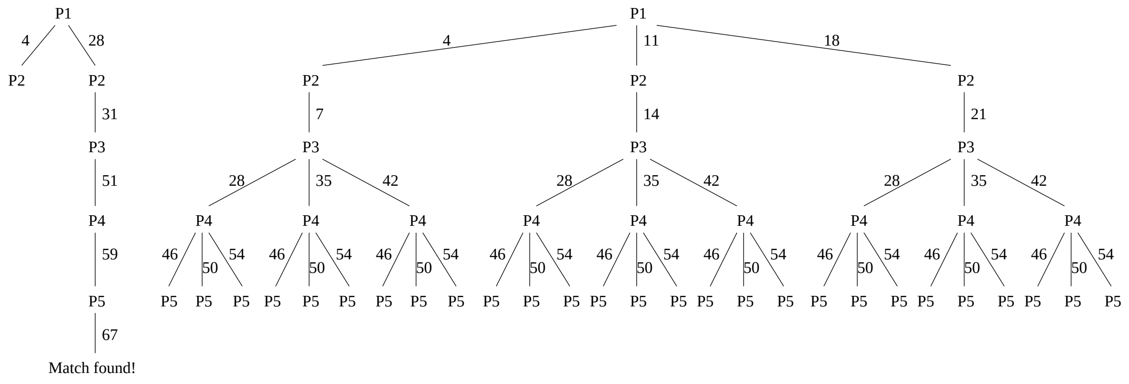

Figure 5. Predicate evaluation trees in Snort. The left tree represents the 6 predicate evaluations performed on the payload in Figure 2, and the right tree shows the 43 evaluations performed for the payload in Figure 4. Numbers on edges indicate payload offsets where a predicate matched.

k predicates. If we make their contribution to the processing time explicit we can compute for each string the exact number of repetitions. If we divide the packet into k equal-sized portions, each filled with repetitions of one of these strings, we obtain $n_i = \lfloor \lfloor n/k \rfloor / s_i \rfloor$. The cost of the attack is $O(\prod_{i=1}^{k} n_i) = O(n^k/(k^k \prod_{i=1}^{k} s_i))$. Other factors such as the amount of overlap between these strings, the length of the strings needed to match predicates that do not cause backtracking, and the details of the processing costs of the predicates also influence the processing cost. These factors remain hidden by the constants inside the O-notation.

Approximately 8% of the 3800+ rules in our ruleset were susceptible to backtracking attacks to some degree. Our focus is on the most egregious attacks, which typically yielded slowdowns ranging from three to five orders of magnitude. We quantify the strength of these attacks experimentally in Section 6.

5. Memoization, a remedy for backtracking

As illustrated above, rule-matching engines are open to backtracking attacks if they retain no memory of intermediate results, which for Snort are predicate evaluations that have already been determined to fail. Thus, matching engines can be forced to unnecessarily evaluate the same doomed-for-failure predicates over and over again, as Figure 5 indicates.

Figure 6 shows our revised algorithm for rule matching that uses memoization [7,16]. It is based on the observation that the outcome of evaluating a sequence of predicates depends only on the payload and the offset at which processing starts. The memoization table holds $(predicate, offset)$ pairs indicating for all predicates, except the first, the offsets at which they have been evaluated thus far. Before evaluating a predicate, the algorithm checks whether it has already been evaluated at the given offset (line 7). If the predicate has been evaluated before, it must have ultimately led to failure, so it is not evaluated again unnecessarily. Otherwise, the $(predicate, offset)$ pair is added to the memoization table (line 8) and the predicate is evaluated (line 9). Note that memoization ensures that no predicate is evaluated more than n times. Thus, if a rule has k' predicates performing work at most linear in the packet size n, memoization ensures that the amount of work performed by the rule matching algorithm is at most $O(k' \cdot n \cdot n) = O(k'n^2)$. Figure 7 updates Figure 5 to reflect the effects of memoization.

```
   MemoizedMatchRule($Preds$):
1  $Stack \leftarrow (Preds[0]$.getNewInstance$(0))$;
2  $MemoizationTable \leftarrow \emptyset$;
3  while $Stack.size > 0$ do
4      if $Stack.top$.getNextMatch() then
5          if $Stack.size == Preds.size$ then return $True$;
6          $ofst \leftarrow Stack.top.getMatchOffset()$;
7          if $(Stack.top, ofst) \notin MemoizationTable$ then
8              $MemoizationTable \leftarrow MemoizationTable \cup \{(Stack.top, ofst)\}$;
9              Push$(Stack, Preds[Stack.size]$.getNewInstance$(ofst))$;
10     else Pop$(Stack)$;
11 return $False$;
```

Figure 6. The memoization-enhanced rule-matching algorithm. Lines 2, 7, and 8 have been added.

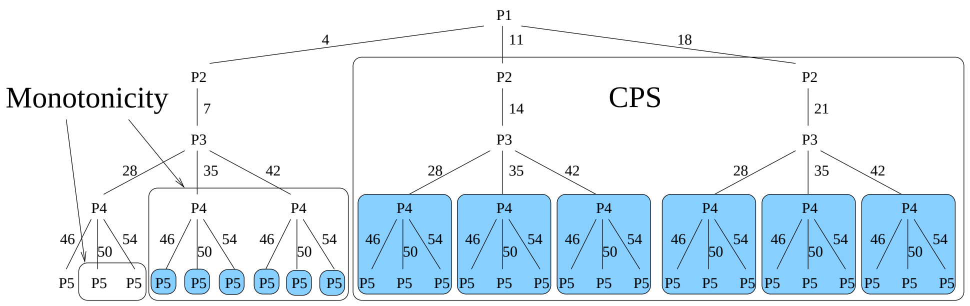

Figure 7. The memoization algorithm performs only 13 predicate evaluations instead of 43 as it avoids the grayed-out nodes. The CPS optimization reduces the number of predicate evaluations to 9, and the monotonicity optimization further reduces the evaluations to 5.

The greyed out nodes in the large tree from Figure 7 correspond to the predicates that would not be re-evaluated when using memoization. For the most damaging backtracking attacks against rules in Snort's default rule set, *memoization can reduce the time spent matching a rule against the packet by more than four orders of magnitude* (with the optimizations from Section 5.1, more than five orders of magnitude).

To implement memoization, we used pre-allocated bitmaps for the memoization table, with a separate bitmap for each predicate except the first. The size of the bitmaps (in bits) is the same as the size v (in bytes) of the largest virtual packet. Thus if the largest number of predicates in a rule is m, the memory cost of memoization is $v(m-1)/8$ bytes. In our experiments, memoization increases the amount of memory used in Snort by less than 0.1%.

A naive implementation of memoization would need to initialize these bitmaps for every rule evaluated. We avoid this cost by creating a small array that holds up to 5 offsets and an index into the array. When a rule is to be evaluated, only the index into the array needs to be initialized to 0. If the number of offsets a predicate is evaluated at exceeds 5, we switch to a bitmap (and pay the cost of initializing it). It is extremely rare that packets not specifically constructed to trigger backtracking incur the cost of initializing the bitmap.

5.1. Further optimizations

We present three optimizations to the basic memoization algorithm: detecting constrained predicate sequences, monotonicity-aware memoization, and avoiding unnecessary memoization after single-match predicates. The first two of these significantly reduce worst case processing time, and all optimizations we use reduce the memory required to perform memoization. Most importantly, all three optimizations are sound when appropriately applied; none of them changes the semantics of rule matching.

Constrained predicate sequences: We use the name *marker* for predicates that ignore the value of the offset parameter. The outcome of a marker and of all predicates subsequent to the marker are independent of where predicates preceding the marker matched. As a result, markers break a rule into sequences of predicates that are independent of each other. We use the name *constrained predicate sequence* (CPS) for a sequence of predicates beginning at one marker and ending just before the next marker. For example, `P3` in Figure 1 looks for the string `player=` in the entire payload, not just after the offset where the previous predicate matches because `P3` does not have the `relative` modifier. Thus the rule can be broken into two CPSes: `P1`-`P2` and `P3`-`P4`-`P5`.

Instead of invoking the rule-matching algorithm on the entire rule, we invoke it separately for individual CPSes and fail whenever we find a CPS that cannot be matched against the packet. The algorithm does not need to backtrack across CPS boundaries. Less backtracking is performed because the first predicate in each CPS is invoked at most once. For the example in Figure 7, detecting CPSes causes the algorithm not to revisit `P1` and `P2` once `P2` has matched, thus reducing the number of predicate invocations from 13 to 9.

Monotone predicates: Some expensive multiple-match predicates used by Snort have the monotonicity property which we define below. For these predicates we use the more aggressive *lowest-offset memoization*. In this optimization, we skip calls to a monotone predicate if it has previously been evaluated at an offset smaller than the offset for the current instance. For example, say we first evaluate a monotone `content` predicate starting at offset 100 that does not lead to a match of the entire rule. Later we evaluate the same predicate starting at offset 200. The second instance is guaranteed to find only matches that have already been explored by the first instance. With basic memoization, after each of these matches of the second instance we check the memoization table and do not evaluate the next predicate because we know it will lead to failure. But, the `content` predicate itself is evaluated unnecessarily. With monotonicity-aware memoization, we do not even evaluate the `content` predicate at offset 200.

The monotonicity property generalizes to some regular expressions too, and it can be defined formally as follows: let S_1 be the set of matches obtained when predicate p is evaluated at offset o_1, and S_2 the matches for starting offset o_2. If for all packets and $\forall o_1 \leq o_2$ we have $S_2 \subset S_1$, then p is monotone. In our example from Figure 1, all `contents` and `pcres` are monotone with the exception of the first `pcre`, `P2`, because it matches at most once *immediately after* the position where the previous predicate matched.

Lowest-offset memoization helps reduce worst case processing because for some predicates the number of worst-case invocations is reduced from $O(n)$ to 1. For the example in Figure 7, this optimization would have eliminated the second and third evaluations for predicates `P4`, and `P5` (and for `P3` also if CPSes are not detected). This further reduces the number of predicate instances evaluated from 9 to 5.

Unnecessary memoization: Basic memoization guarantees that no predicate is evaluated more than n times. For some rules with single-match predicates we can provide the same guarantee even if we omit memoizing some predicates. If we employ memoization before evaluating a single-match predicate, but not before evaluating its successor, we can still guarantee that the successor will not be evaluated more than n times (at most once for every evaluation of our single-match predicate). Also, if we have chains of single-match predicates it is enough to memoize only before the first one to ensure that none is evaluated more than n times. Thus, our third optimization is not to perform memoization after single-match predicates, such as `byte_test` and `byte_jump` (see Table 1), except when they are followed by a monotone predicate. For our rule set, this optimization reduces by a factor of two the amount of memory used for memoization.

6. Experimental results

We performed empirical evaluations with traces and in a live setting. In Section 6.1, we present measurements comparing backtracking attack packets with traces of typical network traffic. Our results show that three to six orders of magnitude slowdowns achieved with the backtracking attack are reduced to less than one order of magnitude slowdown under memoization. In Section 6.2, we show actual evasion using a non-memoized implementation, and the resulting recovery with the memoized version.

For our experiments we used the Snort NIDS, version 2.4.3, configured to use the Aho-Corasick [2] string matching algorithm. Snort is run on a 2.0 GHz Pentium 4 processor and is loaded with a total of 3812 rules. We instrumented Snort using cycle-accurate Pentium performance counters. When enabled, instrumentation introduced less than 2% overhead to the observed quantities of interest. We found that our measured observations were consistent with the instrumentation results collected in [4].

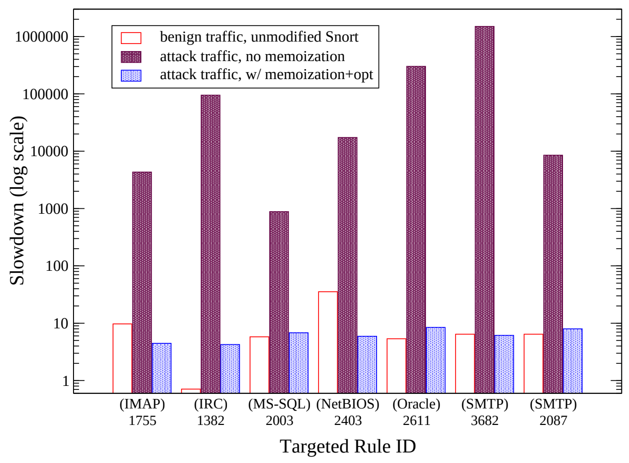

Figure 8. Relative processing times for benign and attack traffic, and attack traffic with memoization. Memoization confines the slowdown to less than one order of magnitude.

6.1. Trace-based results

For benign traffic, we obtained two groups of three traces each captured on different days at distinct times. The first group of traces were captured on the link between a university campus and a departmental network with 1,200 desktop and laptop computers, a number of high-traffic servers (web, ftp, ntp), and scientific computing clusters generating high volumes of traffic. These traces are 7 minutes long and range in size from 3.1 GB to just over 8 GB. The second group of traces were captured in front of a few instructional laboratories totaling 150 desktop clients. They are also 7 minutes long and range in size from 816 MB to 2.6 GB.

We created attack traffic by generating flows corresponding to several protocols and supplying payloads that are constructed in a similar manner to the payload construction outlined in Section 4.

In the trace-based experiments, we fed the benign traffic and attack traffic traces into Snort and observed the performance. We performed these experiments with and without memoization enabled. Figure 8 shows the slowdowns experienced due to backtracking attacks targeting several rules and the corresponding defense rates. It summarizes the information in Table 2. In each group, the leftmost bar represents the cost of packet processing for the specified protocol relative to 20.6 s/GB, the combined average packet processing rate in all our traces. For Rule 1382 (IRC), the rate is less than 1, reflecting the fact that the average traffic processing time for IRC traffic is less than the baseline.

The central bar in each group shows the slowdown observed by packets crafted to target the specific rules indicated at the base of each group. The attacks result in processing times that are typically several orders of magnitude

Protocol	Rule ID	Processing time (seconds/gigabyte)				Slowdown w.r.t. avg traffic		Slowdown w.r.t. same protocol	
		Trace traffic	Backtracking attack						
			Original	Basic Memo.	Memo+Opt.	Original	Memo+Opt	Original	Memo+Opt
IMAP	1755	200.6	89,181	1,802	91.9	4,329×	4.46×	444×	0.46×
IRC	1382	14.6	1,956,858	1,170	87.6	94,993×	4.25×	134,031×	6.00×
MS-SQL	2003	119.3	18,206	715	140.4	884×	6.82×	152×	1.17×
NetBIOS	2403	729.7	357,777	57,173	122.0	17,368×	5.92×	490×	0.17×
Oracle	2611	110.5	6,220,768	3,666	174.0	301,979×	8.45×	56,296×	1.57×
SMTP	3682	132.8	30,933,874	2,192	126.4	1,501,644×	6.14×	232,936×	0.95×
SMTP 3682, w/o reassembly			1,986,624	903	103.1	96,438×	5.00×	14,960×	0.78×
SMTP	2087	132.8	175,657	5,123	164.5	8,527×	7.99×	1,323×	1.24×

Table 2. Strength of the backtracking attack and feasibility of the memoization defense. Columns 7-8 shows the overall slowdown under attack when memoization is not and is used. Columns 9-10 shows similar slowdowns with respect to the same protocol.

slower than the baseline, with the most egregious attack coming in at a factor of 1.5 million times slower. Finally, in the rightmost bar of each group we see the result of each attack repeated with the memoization defense deployed. In most cases, Snort performance when under attack is comparable if not better than when not under attack.

Table 2 details the attacks and the defenses quantitatively for several different protocols. For each attack, Columns 1 and 2 give the protocol and the targeted Rule ID to which the attack belongs, respectively. Column 3 shows the average processing time for each protocol. Columns 4 through 6 show the raw processing times for attack packets under an unmodified Snort, Snort with basic memoization, and Snort with fully optimized memoization. Columns 7-8 give overall slowdowns and Columns 9-10 supply the slowdowns on a per-protocol basis. The backtracking attack achieves slowdowns between 3 and 5 orders of magnitude for rules from many protocols. When memoization is employed, the overall slowdown is confined to within one order of magnitude. Per-protocol, memoization confines most attacks to within a factor of two of their normal processing time.

Rows 7 and 8 highlight the impact that reassembly has on the processing time. In this experiment, when reassembly is performed the size of the virtual packet fed to the rule-matching engine is only twice the size of a non-reassembled packet, but the processing time is almost 16× longer.

The effects of the three memoization optimizations can be seen by comparing Columns 5 and 6 in Table 2. The strength of the optimizations varies by protocol, ranging from just under a factor of 10 to just over a factor of 30, excluding the NetBIOS outlier. In the Snort rule set, NetBIOS rules contain many predicates that can be decomposed into constrained predicate sequences. These rules benefit considerably from the optimizations. Section A.2 contains the individual contributions of each optimization to the reduction in processing time.

Recall that the attacks applied are all low-bandwidth attacks. Even though the overall slowdown rate using memoization is up to an order of magnitude slower, these rates apply *only* to the attack packets (which are few in number) and not to the overall performance of Snort. Under memoization, processing times for attack packets fall within the normal variation exhibited by benign packets.

In the rightmost column, slowdowns less than 1.0 indicate that with all the optimizations included, Snort was able to process backtracking attack packets more quickly than it could process legitimate traffic. In other words, our optimizations allowed Snort to reject these attack packets more quickly than it otherwise was able since fewer overall predicate evaluations are performed.

6.2. Evading a live Snort

In this section we demonstrate the efficacy of the backtracking attack by applying it to a live Snort installation. We first show successful evasion by applying the attack under a variety of conditions. We then show that with memoization, all the formerly undetected attacks are observed.

Figure 9 shows the topology used for testing evasion for this experiment. To induce denial of service in Snort, we use an SMTP backtracking attack that connects to a Sendmail SMTP server in the protected network. We are using this attack to mask a Nimda [1] exploit normally recognized by Snort. Both the Nimda exploit and its SMTP cover are sent from the same attacking computer. Each Nimda exploit is sent one byte at a time in packets spaced 1 second apart. To simulate real world conditions, we used the Harpoon traffic generator [22] to continuously generate background traffic at 10 Mbps during the experiments.

We measure the effectiveness of the backtracking attack by the number of malicious exploits that can slip by Snort undetected over various time frames. We initiated a new Nimda exploit attempt every second for 5 minutes, yielding 300 overlapping intrusion attempts. Table 3 shows the results. Test 1 is the control: when the backtracking exploit is not performed, Snort recognizes and reports all 300 exploits despite our fragmenting them. In Test 2, we sent two backtracking attack packets every 60 seconds for the

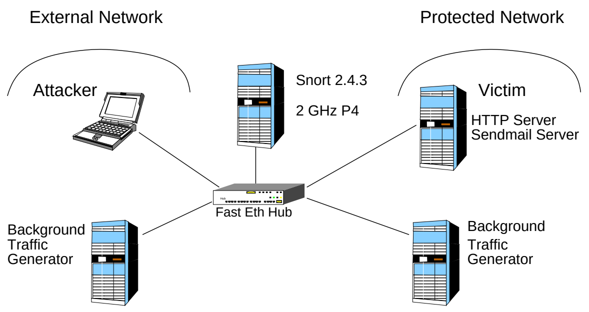

Figure 9. Live Snort evasion environment. Snort monitors a network composed of web and mail servers.

Test	Description of backtrack attack	Exploits detected	Required rate (kbps)
1	Control; no attack	300/300	N/A
2	*two* packets every 60 sec.	220/300	0.4
3	*two* packets every 15 sec.	6/300	1.6
4	*one* packet every 5 sec.	4/300	2.4
5	*one* packet every 3 sec.	0/300	4.0
6	*twenty* packets initially	0/300	0.8
7	*one* packet every 3 sec. (memoization enabled)	300/300	N/A
8	*twenty* packets initially (memoization enabled)	300/300	N/A

Table 3. Summary of live Snort experiments. Without memoization, 300 intrusions pass into the network undetected.

duration of the experiment. Snort missed only one-third of the attacks, detecting 222 out of 300 intrusion attempts. In Test 3, we increased the frequency of the backtracking attacks to 2 packets every 15 seconds, dropping the detection rate to just 2% of the transmitted exploits. Test 4 decreased the detection rate even further, and in Tests 5 and 6 the attacker successfully transmitted all 300 exploits without detection. Aside from high CPU utilization during the attacks and an occasional, sporadic port scan warning directed at the SMTP attack, Snort gave no indication of any abnormal activity or intrusion attempt.

These experiments show that the transmission rate needed to successfully penetrate a network undetected is quite low, with both tests 5 and 6 requiring no more than 4.0 kbps of bandwidth. Test 5, in particular, suggests that perpetual evasion can be achieved through regular, repeated transmissions of backtracking attack packets.

Tests 7 and 8 demonstrate the effectiveness of memoization. These tests repeat Tests 5 and 6 with memoization enabled (including all optimizations). With memoization, Snort successfully detected all intrusions in both tests.

In summary, these experiments validate the results of our trace-based experiments and illustrate the real-world applicability of the backtracking attack. Using carefully crafted and timed packets, we can perpetually disable an IPS without triggering any alarms, using at most 4 kilobits per second of traffic. Correspondingly, the memoization defense can effectively be used to counter such attacks.

7. Discussion

Often, algorithmic complexity attacks and their solutions seem obvious once they have been properly described. Nevertheless, software is still written that is vulnerable to such attacks, which begs the question–how can a NIDS or IPS designer defend against complexity attacks that she has not yet seen? A possible first step is to explicitly consider worst-case performance in critical algorithms and to look at whether it is significantly slower than average case and can be exploited. For example, [9] has shown that in the Bro NIDS, failure to consider worst-case time complexity of hash functions leads to denial of service. With this mindset, we briefly consider mechanisms employed by existing NIDS with an eye towards triggering the worst case.

- Deterministic finite automata (DFA) systems can experience exponential memory requirements when DFA's corresponding to individual rules are combined. In some cases, automata are built incrementally [21] to reduce the footprint of a DFA that cannot otherwise fit in memory. Because each byte of traffic is examined exactly once in a DFA, backtracking does not occur. However, it may be possible for an adversary to construct packets that trigger incremental state creation on each byte of payload, resulting in consistently increased computation costs and potentially leading to memory exhaustion.
- Nondeterministic finite atomata (NFA) systems reduce the memory requirement costs of DFA systems by allowing the matcher to be in multiple states concurrently. In practice, this is achieved either through backtracking or by explicitly maintaining and updating multiple states. In the first case, algorithmic complexity attacks are achieved by triggering excessive backtracking. In the second, the attacker tries to force the NIDS to update several states for each byte processed.
- Predicate-based systems such as Snort can be slowed down if the attacker can cause more predicates to be evaluated than in the average case. We have presented an attack that forces the repeated evaluation of a few predicates many times. In contrast, attacks can be devised that seek to evaluate many predicates a few times. For example, Snort employs a multi-pattern string matcher [2] as a pre-filter to pare down the rules to be matched for each packet. Constructing payloads that trigger large numbers of rules can lead to excessive predicate evaluations.

We have performed preliminary work that combines the second and third observations above to yield packet processing times in Snort that are up to 1000 times slower than average. These results, combined with those of this paper, suggest that left unaddressed, algorithmic complexity attacks can pose significant security risks to NIDS.

8. Conclusions and future work

Algorithmic complexity attacks are effective when they trigger worst-case behavior that far exceeds average-case behavior. We have described a new algorithmic complexity attack, the backtracking attack, that exploits rule matching algorithms of NIDS to achieve slowdowns of up to six orders of magnitude. When faced with these attacks, a real-time NIDS becomes unable to keep up with incoming traffic, and evasion ensues. We tested this attack on a live Snort installation and showed that the protected network is vulnerable under this attack, along with the tens of thousands of other networks protected by Snort.

To counter this attack, we have developed a semantics-preserving defense based on the principle of memoization that brings Snort performance on attack packets to within an order of magnitude of benign packets. Our solution continues the trend of providing algorithmic solutions to algorithmic complexity attacks.

In general, it is not clear how to find and root out all sources of algorithmic complexity attacks. To do so requires knowledge of average- and worst-case processing costs. Without a formal model of computation, such knowledge is difficult to obtain and is often acquired in an ad-hoc manner. Mechanisms for formally characterizing and identifying algorithms and data structures that are subject to complexity attacks can serve as useful analysis tools for developers of critical systems, such as NIDS. We are currently exploring these issues.

References

[1] Cert advisory ca-2001-26 nimda worm, 2001. `http://www.cert.org /advisories /CA-2001-26.html`.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, June 1975.

[3] M. Attig and J. W. Lockwood. SIFT: Snort intrusion filter for TCP. In *Hot Interconnects*, Aug. 2005.

[4] J. B. Cabrera, J. Gosar, W. Lee, and R. K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *43rd IEEE Conference on Decision and Control*, Dec. 2004.

[5] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, Napa, California, Apr. 2004.

[6] The Snort network intrusion detection system on the intel ixp2400 network processor. Consystant White Paper, 2003.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[8] S. Crosby. Denial of service through regular expressions. In *Usenix Security work in progress report*, Aug. 2003.

[9] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, Aug. 2003.

[10] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Usenix Security*, Aug. 2001.

[11] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, Oakland, CA, May 2002. IEEE Press.

[12] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *RAID*, Zurich, Switzerland, Oct. 2002.

[13] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks (Amsterdam, Netherlands: 1999)*, volume 31, pages 2435–2463, 1999.

[14] PCRE: The perl compatible regular expression library. `http://www.pcre.org`.

[15] T. H. Ptacek and T. N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, Jan. 1998.

[16] T. Reps. "Maximal-munch" tokenization in linear time. *ACM Transactions on Programming Languages and Systems*, 20(2):259–273, 1998.

[17] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.

[18] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *ACSAC '04*, pages 28–38, Washington, DC, USA, Dec. 2004. IEEE Computer Society.

[19] U. Shankar and V. Paxson. Active mapping: resisting NIDS evasion without altering traffic. In *IEEE Symposium on Security and Privacy*, pages 44–61, May 2003.

[20] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs, 2001.

[21] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, Washington, DC, Oct. 2003.

[22] J. Sommers and P. Barford. Self-configuring network traffic generation. In *Internet Measurement Conference*, pages 68–81, 2004.

[23] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system. In *International Conference on Field Programmable Logic and Applications*, Sept. 2003.

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

[25] T. Vermeiren, E. Borghs, and B. Haagdorens. Evaluation of software techniques for parallel packet processing on multi-core processors. In *IEEE Consumer Communications and Networking Conference*, Jan. 2004.

```
alert tcp $EXT_NET any -> $SMTP_SERVERS 25
 (msg:"SMTP From comment overflow attempt";
  content:"From|3A|";
  content:"<><><><><><><><><><><><><><><><><><><><><><>",relative;
  content:"|28|",relative;
  content:"|29|",relative;
  sid:2087)
```

Figure 10. A backtracking-susceptible rule, slightly simplified, from Snort's rule database. As long as the last content string (a right parenthesis) is not found, no alert will be generated.

Stuffing Strategy	Slowdown Factor	
	Snort	Snort+Memo
blocked	1,500.4×	2.3×
tiled	25.2×	1.3×
clear	1.0×	1.0×

Table 4. Comparison of blocked and tiled payload-stuffing schemes with regard to slowdowns. The blocked schemes provides significantly higher slowdowns. Fortunately, slowdowns in both schemes are tempered using memoization.

A Backtracking Attacks Revisited

A.1 Building Attack Packets

The first-order effectiveness of a backtracking attack is measured by the number of distinct `content` or `pcre` predicates in an attack payload and the number of their repetitions. A second-order effect is the arrangement of the strings in the attack payload, especially when there is overlap in the content strings. Consider Snort Rule 2087, described in slightly simplified form in Figure 10. To perform a backtracking attack against this rule, we repeatedly stuff the packet payload with the first three content strings, and we omit the fourth string entirely.

We describe two payload stuffing techniques for this rule. In the first, we repeatedly tile each of the first three content strings in order. That is, the payload contains the string `From:`, followed by 22 occurrences of `<>`, followed by a left parenthesis. For a 1448-byte payload, 30 repetitions of this sequence can be tiled.

In the second scheme, we draw on the repetitions of the `<>` characters in the second `content` predicate to increase the number of repetitions. In this scheme, we partition the payload into equal-sized chunks, one chunk per content string. Each partition is then assigned a content string and is filled with repetitions of that string. We consider the first 21 occurrences of `<>` to be overhead and count them only once. Thus, the first block of 465 bytes contains 93 occurrences of `From:`, the second block contains 255 occurrences of `<>` (21 occurrences for overhead, and 234 occurrences to fill the partition), and the third block contains 473 left parentheses, yielding a 1448-byte payload.

We quantified the effects of these approaches by building payloads using both techniques and comparing their runtimes to a payload without any of the content strings. Results of this test are provided in Table 4. Compared to the tiling approach, blocking is almost 60× more effective. Fortunately, in both cases memoization limits the slowdown.

A.2 Additional Experimental Results

Table 5 extends Table 2 to show the effects of the three optimizations. Columns *M+CPS*, *M+Mtn*, *M+SM* show the strength of constrained predicate sequences, monotone memoization, and single-match memoization in that order. Constrained Predicate Sequences and Monotone memoization provide the most benefit, but their relative strength depends strongly on the protocol and rule. For example, in SMTP 3682, CPS and monotonicity perform equally well, bringing the processing cost down to 127.1 and 187.1 s/GB, respectively, from 2,192. For SMTP 2087, CPS provides only a small improvement whereas monotonicity gives an 8× cost reduction. On the other hand, for IRC and NetBIOS rules, CPS provides slightly improved performance.

Interestingly, while all three optimizations together do provide improvements over basic optimization, the best performance results do not always result from the use of all three optimizations. Slightly better performance for IRC 1382 is obtained using CPS only, and Oracle 2611 performs best using only monotonicity only.

B Partial Match Attacks

Partial Match Attacks comprise another class of algorithmic complexity attacks to which Snort is vulnerable. We describe partial match attacks briefly, give some experimental results, and explain the techniques we used to construct partial match attack packets.

In order to minimize processing of legitimate traffic that makes up most of its workload, Snort adopts the principle of stopping the processing of a packet as soon as it becomes clear that it is not a malicious packet using one of the known

Protocol	Rule ID	Slowdown for original Snort	Processing time (seconds/gigabyte of traffic)							Slowdown for modified Snort
			Trace traffic	Backtracking attack						
				Original	Memo.	M+CPS	M+Mtn	M+SM	M+all	
IMAP	1755	444×	200.6	89,181	1,802	96.7	123.0	1,786	91.9	0.46×
IRC	1382	134,031×	14.6	1,956,858	1,170	84.3	629.2	1,230	87.6	6.00×
MS-SQL	2003	152×	119.3	18,206	715	143.6	153.8	724	140.4	1.17×
NetBIOS	2403	490×	729.7	357,777	57,173	118.8	429.7	57,160	122.0	0.17×
Oracle	2611	56,296×	110.5	6,220,768	3,666	367.2	171.6	3,650	174.0	1.57×
SMTP	3682	14,960×		1,986,624	903	104.3	138.1	921	103.1	0.78×
SMTP 3682, w/o reas.		232,936×	132.8	30,933,874	2,192	127.1	187.1	2,343	126.4	0.95×
SMTP	2087	1,323×		175,657	5,123	4,694.0	619.2	5,224	164.5	1.24×

Table 5. Extension to Table 2 showing the effects of each optimization on backtracking attack processing time.

exploits. An example of this principle is the algorithm for matching one rule against a packet (see Figure 3) which stops after the first predicate that cannot be satisfied. Another example of this principle is the quick rejection heuristic that creates a "fast path" for processing packets. We describe this heuristic below.

Matching strings is easier than matching rules. Aho and Corasick proposed [2] a memory-efficient single pass algorithm based on a deterministic finite automaton that determines what subset of a given set of strings occurs in a given packet. Most Snort rules have `content` predicates, and the rule cannot match if the strings these `content` predicates look for are not in the payload. Within every class of traffic, Snort collects from each rule the longest `content` string, and it builds an Aho-Corasick state machine. Packets pass through this filter first, and only those rules whose longest strings were found are actually matched against the packet.

An adversary can mount a *partial-match attack* to slow down Snort. In this scheme, the attacker incorporates into packet payloads the longest strings from corresponding rules. By crafting the packets in a way that forces Snort to match many predicates in the rules, the processing load can be increased even further. Our measurements show that for some traffic classes, such attacks can increase the processing cost of Snort to three orders of magnitude beyond the cost of processing normal traffic. Maximizing the number (and cost) of predicates that Snort will need to evaluate is not a simple process, and we do not claim that in our experiments we found the absolute worst cases for various protocols (traffic classes). Determined hackers might be able to force larger slowdowns, so our numbers might underestimate the power of partial-match attacks.

Algorithmic improvements might reduce somewhat the magnitude of partial-match attacks one can mount against Snort, but we believe that this problem cannot be eliminated without changing the semantics of the rule matching operation because it reflects some fundamental underlying issues: quick tests can establish that many legitimate packets are not malicious, rules describing exploits are complex in order to keep false positives low, and some protocols have more rules associated with them than the average.

B.1 Building Attack Packets

Whereas the target of backtracking attacks is a specific rule, the goal of partial-match attacks is to trigger the evaluation of as many rules as possible. In addition, for each triggered rule we seek to evaluate as many predicates as possible without triggering an alarm.

A Partial Match attack for a specific protocol class P is built by stuffing packet payloads with the longest content strings from as many rules corresponding to P as possible. We built partial match packets as follows: first, for each protocol class P, the longest content string from each rule in P is extracted, and the strings are sorted in increasing order according to length. Second, a packet is tiled with longest content strings starting with the shortest longest-content strings first (the beginning of the list). The tiling for a packet stops when the sequence of longest content strings exceeds the payload size of an Ethernet frame (1460 bytes), or when the list of content strings is exhausted, whichever comes first.

During detection, each of these tiled longest content strings will be matched in the Aho-Corasick prefilter, maximizing the number of rules per packet that are examined (note that Snort maintains a per-protocol bitmap of rules that have been examined, so that examining a rule multiple times by repeating its longest content string is not possible).

In many cases, the content strings from a rule are required to appear at well-defined offsets in the packet payload. The Aho-Corasick state machine finds matches in a position-independent manner, however. In these cases, tiling content strings in the payload independent of their position allows the packet to pass through the filter only to be rejected immediately by the rules themselves. Greater slowdowns can be achieved by combining partial match attacks along with hardest protocol attacks; i.e., construct packet

Traffic class	Processing (s/GB)		Slowdown	
	Trace traffic	Part.-m. attack	wrt. same trf. class	wrt. entire traffic mix
Oracle	110.5	113,658	1,028 ×	5,517 ×
ftp-ctrl cli.	117.6	1,930	16.4 ×	93.7 ×
smtp cli.	128.4	1,642	12.8 ×	79.7 ×
pop3 cli.	20.1	618	30.7 ×	30.0 ×
ssh cli.	23.3	611	26.2 ×	29.7 ×
namesrvr cli.	72.1	447	6.2 ×	21.7 ×
imaps cli.	27.7	135	4.9 ×	6.6 ×
web srv.	5.1	34.9	6.8 ×	1.7 ×
All traffic	20.6	NA	NA	NA

Table 6. Processing cost of partial-match attacks (column 3) against processing paths corresponding to some of the traffic classes recognized by Snort. We compute the slowdown of the partial-match attack (column 4) as the ratio to the cost of processing normal traffic from the same traffic class (column 2) and as the ratio to the cost of processing all traffic (column 5).

payloads such that many rules are invoked and many (but not all) predicates in each rule are evaluated.

In this work, most partial-match attacks were built either by hand or using simple tools that stuff payloads without regard to protocol fields. Mechanisms for automatically constructing attacks that maximize the processing time as a function of the the number of rules examined and the number of predicates evaluated per rule is the subject of future work.

B.2 Experimental Results

Table 6 shows the effect of partial-match attacks against certain protocols. The partial-match attacks consisted of packets of various sizes that force the evaluation of all rules for the given protocol (or just the most expensive ones), and in some cases we crafted payloads that forced certain rules to evaluate multiple predicates, but without matching any of the rules. In the case of Oracle database rules, the attack traffic took extremely long to process because it hit many rules with complex regular expressions.

One of the goals of the partial-match attack is to defeat the processing-time advantage gained through the Aho-Corasick pre-filter. Table 7 shows preliminary results that quantify the degree to which the pre-filter is circumvented.

We measured the processing time and rule counts for Snort under three scenarios and reported these results in Table 7. The first scenario captures the processing time and the average number of examined rules per packet for several protocols using memoized Snort with all optimizations enabled. The second scenario repeats the first except that the Aho-Corasick filter is disabled. Finally, the third scenario is identical to the first scenario except that distinct traces composed only of partial match attacks are used in place of the traces.

These results do not correspond exactly to the results in Table 6 because these results are drawn from a single trace, whereas Table 6 contains results averaged from all traces.

The Rule/Packet columns show the average number of rules examined per packet for each of the scenarios. The role of the Aho-Corasick filter is apparent, reducing the average number of examined rules per packet significantly and decreasing the processing time by approximately a factor of 5 for most protocols. In all cases, partial match attacks result in larger numbers of rules being examined, leading to increased processing times.

The *oracle* entry provides some insight into the combined effects of partial-match and hardest-protocol attacks. Many of the oracle-specific rules are composed of a `content` predicate followed immediately by a complex `pcre` predicate. The `content` predicate serves as a guard against the regular expression. During matching, if the content string is not found then the `pcre` processing cost is avoided. On the other hand, if the content string is found then the `pcre` must be matched. A partial match attack composed of content strings from many rules triggers `pcre` predicate evaluation for each rule, yielding a processing time significantly larger than that obtained simply by eliminating the pre-filter.

Protocol Class	Aho-Corasick		Aho-Corasick Disabled		Partial Match Attacks	
	s/GB	Rules/packet	s/GB	Rules/packet	s/GB	Rules/packet
oracle	110.5	2.6	622.3	294.8	113356.5	77.3
smtp	115.8	3.3	536.8	123.8	1642.2	230.0
ssh	30.9	0.0060	153.0	84.9	610.7	43.3
imaps	62.3	1.2	141.8	86.7	134.3	46.0
mysql	27.7	0.2	45.1	89.7	261.0	70.0
dns	41.1	0.1	256.4	62.0	156.5	37.5

Table 7. Comparison of partial match attacks to Aho-Corasick filtering performance. The average processing time and average number of examined rules per packet are reported for three scenarios, showing the extent to which partial match attacks invalidate Aho-Corasick filtering. Note that these results measure only one direction in the corresponding flows of the traces and therefore do not correspond exactly to the results in Table 6.