Detecting Evasion Attacks at High Speeds without Reassembly

George Varghese
Cisco Systems, UCSD

J. Andrew Fingerhut
Cisco Systems

Flavio Bonomi
Cisco Systems

ABSTRACT

Ptacek and Newsham [14] showed how to evade signature detection at Intrusion Prevention Systems (IPS) using TCP and IP Fragmentation. These attacks are implemented in tools like FragRoute, and are institutionalized in IPS product tests. The classic defense is for the IPS to reassemble TCP and IP packets, and to consistently normalize the output stream. Current IPS standards require keeping state for 1 million connections. Both the state and processing requirements of reassembly and normalization are barriers to scalability for an IPS at speeds higher than 10 Gbps.

In this paper, we suggest breaking with this paradigm using an approach we call Split-Detect. We focus on the simplest form of signature, an exact string match, and start by splitting the signature into pieces. By doing so the attacker is either forced to include at least one piece completely in a packet, or to display potentially abnormal behavior (e.g., several small TCP fragments or out-of-order packets) that cause the attacker's flow to be diverted to a slow path. We prove that under certain assumptions this scheme can detect all byte-string evasions. We also show using real traces that the processing and storage requirements of this scheme can be 10% of that required by a conventional IPS, allowing reasonable cost implementations at 20 Gbps. While the changes required by Split-Detect may be a barrier to adoption, this paper exposes the assumptions that must be changed to avoid normalization and reassembly in the fast path.

Categories and Subject Descriptors: C.2.6 Internetworking : Routers

General Terms: Algorithms, Intrusion Prevention Systems, Design.

Keywords: Normalization, TCP reassembly, evasion attacks.

1. INTRODUCTION

We argue that the need to normalize and reassemble TCP streams has become a router/IPS folk theorem; we follow this with a quick introduction to Intrusion Detection and Prevention Systems (IDS/IPS) and a paper outline.

1.1 Router Folk Theorems

When a field first develops, a set of rules based on experience eventually harden into folk theorems. In the initial stages, these guidelines are very helpful as rules of thumb for dealing with complexity and are not expensive to implement. As time goes on, however, these rules gravitate towards more dogmatic status. At the same time, it becomes so expensive to implement the original rules, as systems scale to higher speeds, that the rules are eventually challenged. Examples of such folk theorems include strict layering in protocol implementation (challenged in [4] using upcalls), wire speed forwarding for routers (challenged in [5] using randomized algorithms for forwarding), and the need for routers to have 200 msec of buffering (challenged in [3] by showing that the overall buffer can be reduced by the square root of the number of flows).

In this paper, we challenge an important folk theorem in Intrusion Prevention systems: that packet reassembly and normalization are *necessary* to deal with evasions [14] by attackers. Briefly, an Intrusion Prevention System or IPS is a device that sits on a network and drops all traffic indicative of an attack. The seminal paper in this area [8] shows that reassembly and normalization are *sufficient* to detect all evasions. However, the IPS industry has gone further and assumed that reassembly and normalization are necessary.

Besides the scientific interest in critically re-examining assumptions, the need for normalization and reassembly has enormous engineering consequences. As speeds get higher, reassembly and normalization in the network requires an increasing amount of resources in terms of memory and processing. Looked at from a pure implementation viewpoint, reassembly at an IPS appears wasteful. For the sake of a few bad flows using fragmented attacks, a router with a packaged IPS must reassemble all flows.

Looked at another way, reassembly (which is done at every receiving endnode) must be duplicated at every IPS in the path. If we simply multiply the memory required for reassembling 1 million connections by the number of IDS/IPS boxes, the total cost (even ignoring the cost of processing) is very large.

A recent paper [6] shows that the processing cost of TCP reassembly can greatly be reduced by optimizing for the expected case when most TCP segments are in order. However, this optimization does not reduce the *state* required for TCP reassembly. Further, we will see that normalization can

SIGCOMM'06, September 11–15, 2006, Pisa, Italy.

potentially increase the costs of memory and processing by an order of magnitude beyond that required by reassembly.

1.2 Intrusion Prevention Systems

In a perfect world, where all endnodes detect and prevent attacks, Intrusion Detection Systems (IDS) would be useless. Unfortunately, network administrators cannot rely on endnode software (often controlled by a different organization) being up to date in terms of Anti-Virus updates and patches. Thus, just as in the case of firewalls, the use of an IDS is a popular retrofit strategy. Almost every major organization runs an IDS of some sort, and many organizations, motivated by the threat of internal attacks, deploy an IDS in several parts of the internal network. Thus, the IDS market is a billion dollar market, and continues to grow.

We will focus in this paper on signature based IDS. Such an IDS consists of a database of rules. Each rule specifies a predicate on packet headers, optionally contains a content string, and has an associated action. In classical IDS systems such as the open source tool Snort [15] the associated action is usually an alert to the administrator. Signature based IDS are very popular and are supported by every major IDS vendor. The bane of most IDS users is the potentially large number of false positives in alerts.

By the time an IDS can raise an alert and a human administrator respond, a fast-moving attack (such as a worm or a DDOS attack) can have done considerable damage. Thus in recent years, the IDS market has morphed into the so-called IPS (Intrusion Prevention System) market. Somewhat cavalierly, an IPS can be described as an IDS where a subset of rules (which the IDS implementers are confident can cause almost no false positives) are enabled with the corresponding action to drop any packet that matches this rule. An IPS must be inline to drop packets, while an IDS can simply tap the data to generate alerts.

Both IDS and IPS systems are required to reassemble TCP flows and IP fragments. This ensures that a content string in a rule that is fragmented across packets can be detected. IPS systems are further required to normalize [8, 13] TCP flows. Roughly speaking, normalization seeks to normalize the data sent in a flow to avoid inconsistencies that can be exploited by an attacker. As the market for IDS and IPS systems has matured, there are now well-established tests that check for conformance. For example, the NSS report [11] tests vendors for resilience to evasion attacks by running *fragroute* [17], and *whisker* or *nikto* [10]. All the major vendors appear to have demonstrated [11] their ability to detect evasion attempts.

As the speed of enterprise networks moves from 1Gbps to 10 Gbps, IPS devices have been attempting to scale up in speed as well. In terms of speed, some vendors are already deploying IPS systems at 8 Gbps. Further, as a reaction to the number of ad hoc network devices (e.g., load balancers, content accelerators, and routers) in networks, there is an increasing trend towards consolidating devices within a single enterprise switch. As many switches have 10 Gbps and even 20 Gbps ports, it is desirable to scale a single IDS chip or device to 20 Gbps.

What are the main bottlenecks for an IPS? One major bottleneck is scanning a stream of bytes for a content string or even a regular expression. Searching for content strings is fairly well understood [5]. In recent years, many IPS devices have allowed the specification of regular expressions for content strings but hardware algorithms for even these are well understood [5]. However, a second major bottleneck is the effort required to reassemble TCP flows and to normalize them if needed. Many IPS vendors advertise support for up to 1 million concurrent TCP flows; the number of flows may seem surprisingly large for an enterprise. However, recall that in a security context, a TCP flow cannot be timed out quickly in case a fragment of the attack is sent much later.

It is this second bottleneck, and especially in the context of a packaged IPS/router, that we focus on in this paper.

1.3 Paper Outline and Contributions

The rest of this paper is organized as follows. Section 2 contains an implementation model for a packaged IPS in a router or switch, and describes the main assumptions as well and measures. Section 3 provides a brief overview of the possible evasions made possible by fragmentation. Section 4 begins the solution description by first dealing with two complicating issues: overlapping TCP segments and IP fragmentation. Then Section 5 describes the Split-Detect solution that is scalable and yet able to detect damage caused by out-of-order fragments and chaff. Section 6 contains a proof that Split-Detect is correct; a proof is needed as several of our initial attempts had flaws. Section 7 describes a trace-driven analysis of the performance improvement resulting from Split-Detect. Section 8 suggests clean slate approaches to the problem of simplifying IPS devices based on the lessons learned in this paper. Finally, Section 9 states conclusions.

Contributions: The main contribution of this paper is critically examining the need for reassembly and normalization. As part of this examination, we propose an alternative (Split-Detect) to full reassembly and data normalization for all flows passing through an IPS by efficiently identifying a small subset of traffic in the fast path that requires normalization/reassembly. Note that Split-Detect only avoids reassembly and normalization in the *fast path*.

Unfortunately, Split Detect requires three assumptions: a small modification to TCP receivers to check for inconsistent retransmissions, a change in the definition of signature detection to allow the start and end of a signature to be missed, and a restriction to exact signatures or regular expressions with a fixed exact length. The first assumption seems to be fundamental, the second can be removed by an implementation or protocol change, and the third assumption may be relaxed by future work.

Given the difficulties with these assumptions, the main contribution of this paper is exposing the assumptions that need to be changed to avoid reassembly and normalization in the fast path. We hope that our initial study will stimulate further work.

A second contribution is the formalization of several concepts (such as critical packets and possible reassembly) that seem fundamental to the theoretical modeling of evasion attacks.

2. MODEL

> *Between what matters and what seems to matter, how should the world we know judge wisely?* —
> E.C. Bentley, Trent's Last Case

We capture with a model the salient aspects and parameters for IPS implementations. Figure 1 shows a model of

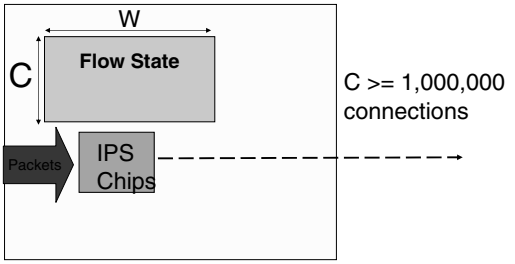

Figure 1: A model of a standard IPS integrated into a switch

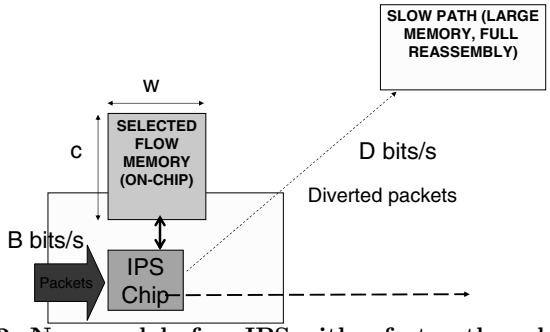

Figure 2: New model of an IPS with a fast path and a slow path

a classical IDS/IPS implemented at speeds greater than 5 Gbps. Packets are inspected by some set of chips, often ASICs. Many products use two or three such chips. The TCP and IP flow state is stored in a large state table with memory for C connections (often required to be at least 1 million) with W bits per connection. Even if all the packets in a connection are in-order the minimum state for a connection is at least the TCP 5-tuple and the sequence number, which is at least 128 bits. Typical implementations especially for IPS devices probably keep at least 10 times this much state, given that full data normalization [8] appears to require keeping an RTT's worth of TCP stream data.

Thus the overall memory required for the flow table is at least 128 Mbits, and more likely to be closer to 1280 Mbits, which is sufficiently large to require external DRAM. In practice, several DRAM chips are required. The net result is that the IPS implementation, counting processing chips and external memory, requires several chips and supporting processors, which makes it expensive and hard to package cheaply into every line card.

A natural alternate IPS model is shown in Figure 2. The idea is that the IPS complex in Figure 1 is replaced by a simpler (and potentially single) IPS chip that handles the common case, but also detects exceptions by keeping track of a much smaller number of connections. When an exception is detected, the remainder of the TCP flow is diverted to a second processor that handles the exception case using the full connection state, reassembly, and normalization. However, the idea is that the slow path processor only handles the exception flows.

One particularly attractive packaging of this model is to place the fast path IPS chip in every line card of a switch, and to keep the slow path processor(s) in a separate card shared by all other line cards of the switch. For this packaging to make sense, the amount of memory required by the fast path processor should ideally be sufficiently small to fit into on-chip memory or a small CAM, making the per line-card cost very small. At the same time, the amount of traffic diverted to the slow path must be sufficiently small to allow the slow path processor to be shared by several fast path processors.

Thus referring to Figure 1 and Figure 2, two relevant performance measures of the leverage of the new model (Figure 2 versus the classical model (Figure 2) are:

- **Speedup:** Speedup can be defined as $S = B/D$. In other words, this is the reciprocal of the fraction of bytes diverted to the slow path processor. A potential target is a speedup of 10 for $B = 20$ Gbps and $D = 2$ Gbps. This allows either a cheap slow path processor or the sharing of the slow path processor across 10 line cards.

- **Memory Compression:** Memory compression is the ratio of the connection memory required by the classical IPS to the memory required by the fast path processor in the new model (Figure 2). From the figure, this is $M = CW/(cw)$. A potential target is a memory compression of 25 to yield a fast path memory $cw = 5$ Mbits (which should fit into on-chip memory) assuming $C = 1$ million and $W = 128$ bits.

3. THE GENTLE ART OF EVASION

Know thyself, Know thy Enemy

— Sun Tzu, circa 500 BC

In this section we briefly review some of the power available to an attacker using fragmentation. The power arises from the combination of TCP and/or IP fragmentation with out-of-order, redundant, and overlapping segments. We illustrate these by the following example attacks. These examples are by no means a comprehensive list of attacks. In all the examples, the intruder is attempting to send a string "ATTACK" that is passed by the IPS but is received by the receiver. In this section, we restrict ourselves to TCP fragments.

3.1 The Case of the Misordered Fragments

In this warm-up example, the intruder breaks the string "ATTACK" into two fragments "ATT" and "ACK". The attacker then sends the content string "ACK" in the first segment to physically pass the IPS (time flows from left to right in all examples) with TCP sequence number 13. Later, the first part of the attack string "ATT" is sent in a second TCP packet with sequence number 10. Although these fragments pass the IPS out-of-order (and potentially with a long time between fragments), the receiver to which these packets are destined will reassemble the string correctly by first placing the string "ATT" (because it has start byte sequence number 10) and then attaching the string "ACK"

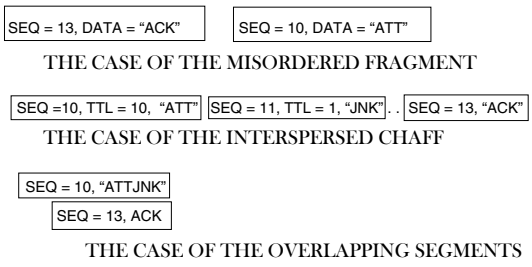

Figure 3: Pictorial representation of 3 powerful evasion techniques

(because it has start byte sequence number 13). The top row of Figure 3 shows this attack pictorially.

Clearly, an IPS that does reassembly can catch this case because it duplicates receiver reassembly before checking for strings such as ATTACK. in the reassembled byte stream.

3.2 The Case of the Interspersed Chaff

In this example, the attacker breaks the string into two fragments "ATT" and "ACK" as before but now adds some "noise" or chaff to the attack to confuse the IPS without damaging correct reassembly at the receiver. There are many ways to do this; one technique is to send the good packets with a large enough TTL (Time to Live) to reach the receiver, while sending the chaff with a small TTL that causes it to be dropped before reaching the receiver.

In the example in the second row of Figure 3, the fragments "ATT" and "ACK" are sent with a large TTL of 10, while the interspersed chaff "JNK" is sent with a TTL of 1. Assuming that the IPS has no knowledge of network topology, the IPS cannot tell which fragments will make it to the receiver. Thus the IPS does not know whether the receiver will receive "ATTJNK" or "ATTACK".

It is easy to construct cases with P pieces of overlapping chaff starting at several positions within the attack signature such that there are 2^P possible reorderings. Since it is computationally hard for the IPS to compute exponential numbers of reorderings, a more elegant solution is data normalization [8]: the IPS picks a canonical reordering (in this example, say ATTJNK), realizes that it does not match a valid attack string, and so lets it pass without an alert. However, when the packet with the string "ACK" goes by the IPS, the IPS rewrites the string "ACK" to "JNK" to be consistent with data sent in the past. Once again, this example plausibly argues for the need to both reassemble and normalize.

3.3 The Case of the Overlapping Fragments

A more pernicious form of attack using overlapping sequence numbers is shown pictorially in the third row of Figure 3. The first TCP packet carries sequence number 10 and the string "ATTJNK". The second TCP packet carries the sequence number 13 and the string "ACK". . One convention at a receiver when faced with overlapping bytes is to deliver the most recently received bytes. With such a convention, the receiver will deliver the string "ATTACK" and the intruder will succeed. Clearly, normalization avoids this problem by either sending "ATTJNK" consistently or dropping data that reassembles to "ATTACK".

While overlapping fragments abstractly looks similar to interspersed chaff and they have the same cure (normalization), they have a subtle difference. In the case of interspersed chaff, a packet is either completely chaff or completely good data. In the case of overlapping fragments, a packet can partially contain chaff and good data. In particular, after cutting a signature into pieces, any attack that only contains interspersed chaff will be forced to send small fragments, a behavior that can be detected.

On the other hand, using overlapping fragments, the sender can send arbitrarily large packets while still (effectively) fragmenting the signature into pieces of size as small as 1 byte. For example, imagine a sequence of large packets whose sequence numbers are X, $X+1$, $X+2$, etc., and where the new byte of the i-th packet is the i-th byte of the signature, and the rest of the data bytes are chaff. In summary, overlapping fragments are deadly because they allow signatures to be segmented virtually into as small pieces as desired without any accompanying physical manifestations in terms of small packet sizes.

4. CLEARING THE UNDERBRUSH

Before we move to our final solution, we simplify the problem by addressing two attack mechanisms: IP fragmentation and overlapping fragments.

4.1 IP Fragmentation

Clearly, many of the same attacks described in Section 3 can be duplicated with IP fragments with the IP fragment offset fields playing the part of TCP sequence numbers. Combinations of TCP and IP fragmentation in the same attack can complicate the mechanisms and proofs. For example, not every IP fragment of a TCP packet contains a TCP header. Because IP fragmentation is so rare in practice (a fraction of a percent in our traces and in previous reports [16]), we use a conservative solution: divert any IP fragments and any connection whose connection ID is in an IP fragment to the slow path.

4.2 Overlapping TCP Fragments

Detecting overlapping TCP fragments appears very hard without keeping state for every connection. While overlapping TCP fragments do result in out-of-order TCP segments, benign out-of-order segments occur because of route changes, load balancing, and retransmission (consider sending packets P1, P2, P3, and then resending P1). It appears hard to distinguish benign out-of-order packets from segments with overlapping sequence numbers without keeping a record of all past sequence numbers, which is no better than keeping TCP state for all connections.

It appears possible to prove that detecting overlapping sequence numbers requires a large amount of space[1] using a reduction to the set disjointness problem, as pioneered in [2] and applied in [9] to show similar hardness results for a number of other security problems. However, one must be cautious about such results. First, the hardness of the set disjointness problem is based on assumptions which may be relaxed in practice. Second, as pointed out in [9] the same

[1]We are grateful to Yossi Mattias for this observation

phenomenon (overlapping TCP fragments) may have several manifestations (e.g., overlapping sequence numbers, overlapping content). Proving that one manifestation is hard to detect does not imply that scalable detection of some other manifestation is impossible. Despite these caveats, it does seem that detecting overlapping fragments is fundamentally hard.

Instead, we will rely on changes to endnodes to satisfy the following atomicity property. In the following we use "delivered" to mean "delivered to the application", not merely "the segment is delivered to the target host".

Weak Atomicity Property: None of the bytes in a TCP segment that are *delivered* will be inconsistent with bytes of another TCP segment that are delivered.

Note, that this restriction cannot cause any difficulty to good sender stacks because the TCP protocol does not allow inconsistent data transmission. The implementation of the weak atomicity property is fairly easy. Maintain a buffer, the Overlap Detect buffer, of up to an MSS size worth of the bytes last delivered to the socket buffer. When a new packet becomes in-order and is a candidate for delivery, compare any overlapping bytes with the bytes in the overlap buffer. If there is inconsistency, do not deliver the segment and reset the connection.

Note that this implementation takes more space (1 MSS) and more processing (byte-by-byte comparison in case of overlap) than a standard TCP implementation. However, it is very likely that most socket buffers will need storage for up to 10 MTUs or more. Thus the additional storage cost should be a small percentage of the existing storage cost. Also, the processing cost can be mainly a matter of writing to the circular overlap buffer and the actual byte-by-byte check is incurred only in the rare case of an overlapping segment.

Weak atomicity also appears to introduce a new Denial-of-Service attack wherein an attacker could inject inconsistent data and cause the connection to be reset. However, the alternative is to allow the attacker to inject arbitrary wrong data, which is worse. Note that SSH also resets a TCP connection on detecting a possible TCP injection attack. We argue that such an endnode change is actually good because:

- It prevents bad behavior (*delivered* inconsistent data) from harming an endnode.
- It does not require implementing a complete IPS (no signatures are required at endnodes in this proposal) or normalizer at the endnode.
- It can easily be implemented. If deployed by Windows and Linux, then the two most common targets of attacks can be protected while allowing IPS systems to scale.

In summary, we believe that an easily-implementable change in endnodes to implement an obvious consistency check (which should have been required in the past) can greatly improve the scalability of IPS systems. Further, it appears possible to prove [2] that without this change, IPS devices will have to keep memory for all connections in the fast path.

4.3 What Still Remains

It may appear that with the finessing of this section that we have trivialized the problem and "defined the problem away". However, note that the attacker still has great power:

- The attacker can still break up an attack signature across several small TCP Fragments. Compounding the difficulty is the fact that small TCP segments are common in innocent traffic.
- The attacker can still send out-of-order Fragments. Compounding the difficulty is that fact that out-of-order traffic is common in real traffic because of retransmissions.
- The attacker can still send redundant packets/segments that never get to the receiver (e.g., Chaff with low TTLs) in attack traffic but hard to detect at the IPS. The attacker can still use chaff to create an exponential number of possible reassemblies at the IPS and thus normalization at the IPS is still required.

5. THE SOLUTION: SPLIT-DETECT

> *Strategy without tactics is like the empty sound before defeat . . .* —- Sun Tzu

We describe the basic idea, and provide a quick example of cutting a signature into pieces. We proceed with a detailed statement of the state variables, and the fast and slow path processing algorithms.

5.1 Basic Framework

We call the algorithm *Split-Detect* because our major tactic is to split a signature into K equal pieces. The detection of any piece will cause the line card to divert the connection to the slow path. The fast-path algorithm consists of:

- **Split:** Break a signature into K equal pieces and arm the fast path to detect any piece.
- **Divert**: Divert a TCP flow to the slow path if —
  - Fast path chip detects any piece
  - Fast path chip detects small packet or out-of-order behavior.

As K increases, the IPS has more pieces to detect but the speedup increases because a smaller amount of traffic will be diverted. More pieces do not necessarily mean K times more storage in the fast path. For example, DFA based string matchers such as Aho-Corasick [1] require space linear in the total number of bytes and do not increase significantly in time or storage with the number of pieces.

However, if the pieces are too small, there will be false positives detected in innocent traffic. We pick 4 bytes as the smallest acceptable piece size because the resulting extra diversion caused by false positives of 4 byte random data is sufficiently small for TCP flows that send less than 2^{32} bytes. The random model of false positives is insufficient. Even using piece sizes greater than 3, care must be taken to ensure that a piece is not part of a common application string. For example,"HELO" is a 4-byte string used in the SMTP handshake; use of it as a piece would cause every SMTP

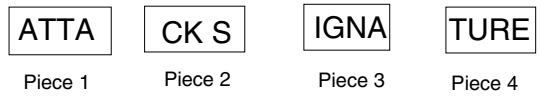

Figure 4: Example of Cutting up a Signature into 4 pieces

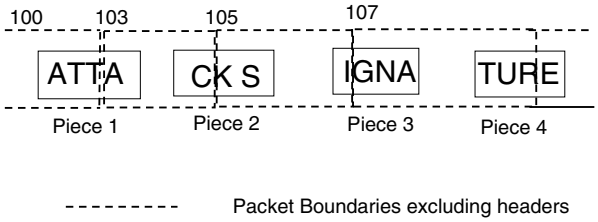

Figure 5: Example of Splitting each piece (packet boundaries are shown using dashed lines) into packets such that no complete piece is detected in a packet passing though the IPS

connection to be diverted. Note that if a signature is long enough, one could discard some initial bytes to change the alignment of pieces, so that a string like "HELO" by itself does not form a complete piece. Similarly, for a signature that starts with "http . . .", it is best to discard the first few bytes. Generally, the longer the piece size, the less likely is it for this to be a problem.

5.2 Example of Cutting a Signature into Pieces

Figure 4 shows an attack signature "ATTACK SIGNATURE" broken up into 4 pieces of 4 bytes each.

Breaking up a signature into pieces and looking for each piece individually has the following intuitive consequences (we will prove them formally in the next section):

- If a packet contains a piece, it will be detected.
- Thus all K pieces must be split to evade
- If endnode atomicity (Section 4) is enforced, a packet containing a split piece cannot contain non-signature data that conflicts with the signature, or the entire signature will not be delivered.
- All but the first and last splits will create "small packets" with payload size $< 2PieceSize-1$, where $PieceSize = \lfloor S/K \rfloor$ and S is the signature length and K the number of pieces.

Figure 5 shows that an attacker can cut the pieces (of size 4 bytes in this example) to evade detection into several pieces of size at most 6 bytes $(2 * 4 - 2)$. The attacker's packet boundaries are shown using dashed lines. Note that the first and last packets can be large but the middle three must be at most 6 bytes. Notice that the middle 3 pieces are consecutive in sequence number space.

This seems to imply that one can detect such an evasion attempt by looking for a certain number of consecutive small packets, where "small" means the packet payload size is strictly smaller than $2PieceSize - 1$. Unfortunately, the attacker has more power using out-of-order packets and "chaff" that does not reach the receiver.

5.3 Motivating the Algorithm

Clearly, looking for small packets in sequence cannot suffice because packets can be sent out-of-order. A first attempt at a state machine to detect evasions would be to look for either K small packets in order, or K out-of-order small packets

Even this does not suffice. Suppose the attacker cuts up the signature into fragments such that no fragment contains a piece. The attacker divides the fragments into odd and even fragments. The attacker then sends the even fragments (Frag 0, Frag 2, Frag 4 etc.) in-order by interspersing the even fragments with appropriately numbered chaff. More precisely, the attacker sends Frag 0, $J1$, Frag 2, $J2$, Frag 4, $J3$, etc.), where $J1$, $J2$, $J3$ . . . are large chaff packets with small TTL that will not reach the receiver but with starting sequence numbers that match Frag 1, Frag 3, $Frag5$, etc.

Once the attacker has sent the even fragments, he can go ahead and send the odd fragments in the same way. Thus there will be at most $K/2 + 1$ out-of-sequence transition (one after each chaff packet and one at the boundary between even and odd fragments) and there will never be the case of two consecutive small packets. Note than even in this example there are K small packets where K is the number of fragments the attacker is forced to cut the signature into. While we could detect this, we would like a stronger predicate because there are many innocuous connections that will send a number of small packets over the lifetime of the connection. The example above shows that such connections cannot be distinguished from a deliberate attack as in the example, because the gap between the sending of the train of even fragments and the train of odd fragments can be made arbitrarily long by filling it in with chaff.

To create a stronger diversion predicate, examine the example more carefully, and notice that the small packets in the even train and in the odd train *must be spaced apart by the length of the signature*. Otherwise, they will not be assembled together at the receiver to be part of a complete signature.

To define anomalous behavior we introduce the following terminology. *Consecutive small packets* are two packets received by the IPS that are small and in between the receipt of which the IPS does not receive any other small packets. Thus if the IPS received $P1, P2, P3, P4, P5$ in order where packets $P1, P4$, and $P5$ are small and $P2$ and $P3$ are large, then $P1$ and $P4$ are consecutive small packets, and so are $P4$ and $P5$. However, $P1$ and $P5$ are not consecutive small packets because of the presence of $P4$ in between.

Thus to detect an anomalous connection, intuitively we look for K anomalous events in a connection, where an anomalous event is defined as either:

1. **Condition 1, Closely Spaced Small Packets:** Condition 1 is triggered if the IPS receives two consecutive small packets whose sequence numbers differ by at most the signature length [2]
2. **Condition 2, Out-of-Order:** Condition 2 is triggered if the IPS receives two consecutive small packets between which there is at least one out-of-order transition (the out-of-order packet could be any packet in the middle of the two up to and including the second small packet).

Thus between two consecutive small packets, either the sequence is completely in-order in which case only Condition 1 can occur or Condition 2 occurs. The intent is that an *innocuous* connection that sends small packets that are sufficiently spaced apart will not be diverted to the slow path. Similarly, an innocuous connection that sends very few out-of-order small packets will not be diverted.

5.4 Fast Path State Machine

The fast path algorithm can be compactly described by a state machine that can easily be implemented in hardware. The IPS system first picks the number of pieces K. Note that the piece length *PieceSize* is $\lfloor SignatureLength/K \rfloor$. A packet is defined to be small if its payload size is in the range $[1, 2 \cdot PieceSize - 2]$. Note that ACKs with 0 data bytes are not considered to be small packets. We use the term TCP flow and connection interchangeably in what follows.

State Instantiation: The fast path keeps state for a flow only after it sends its first small packet

State Variables: When the IPS decides to keep track of a flow, it keeps the following variables (all indexed by the TCP connection 5-tuple, using say a CAM),

- *NES*(Next Expected Sequence Number, 32 bits)
- *OOO*(Out Of Order since last small packet, Boolean)
- *length*(Length in bytes since last small packet, 7 bits can support signatures 127 bytes or shorter)
- *count*(count of anomalies, 4 bits can support values of K up to 16, $K-1$ strikes and the flow is out)
- LUT(Last Update Time, 3 bits can store a coarse time value sufficient for aging out old unused table entries)

In summary, the IPS fast path maintains a flow table for every active TCP flow that has ever sent a small packet, where each flow entry contains a small amount of state (*NES*, *OOO*, *length*, *count*, LUT) for a total of 48 bits of state per flow that is kept track of (plus 96 bits for IPv4 source and destination address, and TCP source and destination port). This reduces memory compared to standard IPS implementations that need to keep track of *all* active flows (not only the ones that ever sent a small packet) and appear to keep track of a round trip time's worth of packet data for normalization. Note that our slow path is no worse in terms of state or processing than a traditional IPS. The state machine processing is as follows:

To update *count*:

- *count* is initialized to 1 when the flow is first placed in the flow table.
- *count* is subsequently incremented on receiving a small packet for a flow if:
  - the packet's sequence number is not equal to *NES*, or
  - *OOO* is true (i.e., some out-of-order since last small packet), or
  - $length \leq SignatureLength$

Note that *count* is never updated for large packets, and is never incremented past $K-1$ (i.e. it "sticks" at that value).

To update *OOO*, *NES*, and *length*:

- *OOO* is set to true if the current packet sequence number is not equal to *NES* and the packet is large; *OOO* is reset to false if the current packet is small (this reflects the intuition that *OOO* is a flag that detects out-of-order reception between small packets; hence it is reset when a small packet is received.)
- *NES* is set equal to $s+l$, where s is the current packet sequence number and l is the TCP payload length of the current packet, (*NES* is set to reflect the sequence number of the next expected in-order TCP segment in this flow.)
- *length* is incremented by the payload length if the current packet is large and reset to zero if the current packet is small (*length* measures the length in bytes received for this flow since the last small packet was received.)

As a special case, TCP packets with no data cause no change to any of *count*, *OOO*, *NES*, or *length*. All packets, including those with no data, cause LUT to be updated to the current time.

After state update, the entire flow (including the current packet causing the update) is diverted to the slow path if either of the following two conditions are true:

1. The packet is found to contain a piece of some signature (by some string matching algorithm). In this case the fast path can simply set $count = K-1$ in order to divert the flow.
2. The anomaly count *count* is equal to $K-1$ (one less than the number of pieces)

If the flow is not diverted, the packet is forwarded normally but, in addition, a copy of the packet is sent to the slow path if and only if the packet is small (i.e., payload size is in the range $[1, 2PieceSize - 2]$). In other words, if a packet contains plausible evidence (i.e., packet is small or contains a piece), then a copy of the packet is sent to the slow path for examination. However, if the anomaly count is too high or a piece is detected, the entire flow is diverted to the slow path.

[2]Since we are looking for multiple signatures, this should really be the maximum length across all signatures being detected.

5.5 Slow Path Algorithm

Every packet sent from the fast path to the slow path is sent with additional information indicating whether it is a copy of a forwarded packet, or if the packet has been diverted and thus has not been forwarded normally by the fast path. When the slow path receives a packet marked as a copy, it stores it in a table indexed by the packet's 5-tuple. These packets may be needed in the future for detecting an occurrence of an attack signature, but do not require any other immediate action.

If the slow path receives a packet for a flow F that is marked as diverted, then it becomes responsible for deciding whether to forward the packet on to the receiver. The slow path tries to paste together the fragments received for flow F; if it gets close to forming a signature (the first and last pieces may be missing) then the packets of the flow are dropped. The more precise specification is as follows.

For every flow (diverted flows as well as flows for which it receives copies), the slow path maintains *a single version of the reassembled TCP stream up to this point.* Clearly, if segments overlap and have inconsistent data, one can create an exponential number of possible reassemblies [8]. Since we wish not to burden the slow path, we set a flag at the first sign of inconsistent data for the flow and drop the later segment that is inconsistent. Note that this is like data normalization [8] except that in data normalization, the later segment is modified to be consistent with previous segments. If data is inconsistent, then if the flow is diverted, we simply drop all further packets of the flow. This is a Draconian stance, but one which is consistent with end-node weak atomicity enforcement.

Finally, if a flow is diverted, the slow path looks for the concatenation of pieces 2 through $K-1$ (of any signature in the database) in the reassembled stream. If such a "near match" is found, further packets of the TCP flow are dropped and the TCP connection is reset. Note that while looking for such a near match appears to worsen the false positive rate (because we are not looking for 2 out of K pieces), one can argue that if the signature is fragmented across packets, the probability of that happening on innocuous data is very unlikely. However, a careful argument requires a model of how random data splits across packets. Overall, we do not feel that the false positive rate will increase at all in practice; even if it does it can be combated by making the signature longer.

It is possible, but not strictly necessary, for the slow path to do standard data normalization after diversion. In either case, the state and processing requirements (per flow) of the slow path are similar to that of a standard IPS doing data normalization but working only with a small number of diverted flows.

We assume that header normalization (i.e., setting header values to canonical values to avoid information leakage or to prevent attacks) is done both in the fast path and in the slow path. Such header normalization is neither state nor computation intensive [8]. Finally, there are many other details such as when state can be safely released for which the extensive techniques in [8] can be used.

6. PROOF

In this section, we will establish correctness of Split-Detect algorithm. More precisely, the combination of the fast path state machine and the slow path processing will never let a flow containing a signature to be sent to a receiver, despite the use of evasion techniques by the sender. We assume that the receiver terminates any TCP flow that attempts to violate weak atomicity before delivery of inconsistent data.

Assume the IPS splits the signature into K pieces, 1 through K. We need preliminary definitions.

Definition 1: Consider an IPS that has received a sequence S of packets for a TCP flow. A reassembled flow for sequence S is a possible reassembly of sequence S (including cases where S is reordered or arbitrary subsets are dropped) at any endnode enforcing atomicity.

Note that the sequence S is received at the *IPS* but we consider the reassembly done at *an endnode enforcing atomicity.* Thus even if the sequence S contains chaff that may not reach the endnode we still apply the endnode operation to the sequence. The definition of possible reassembled TCP streams provides a formal definition of an evasion. An evasion is a TCP connection in which some possible reassembly of the connection contains a forbidden string S. The next definition formalizes the notion that the slow path looks for a near miss of string S.

Definition 2: For any string S, we define the string *Almost*(S) to be the string containing Pieces 2 through $K-1$ of S in sequence.

We now formalize the notion of the critical packet, the forwarding of which can cause the string S to be delivered to the endnode, and the game to be lost. Since it is hard to guarantee that such a packet will be detected (it could be a large packet containing the last piece) we relax the definition to say that the critical packet is one that can cause *Almost*(S) to be delivered. Clearly, preventing *Almost*(S) will prevent S from being delivered. It is essential that the fast and slow path conspire together to drop the critical packet. Thus:

Definition 3: The critical packet for a TCP connection containing string S with respect to an IPS is the *first* packet from the TCP connection received at the IPS such that *Almost*(S) is contained in some reassembled TCP stream for this connection up to and including this packet, but such that that the string *Almost*(S) is not contained in any reassembled TCP stream not including this packet. The *collaborators* of a critical packet are any prior packets in the TCP connection that are used in some reassembled TCP stream containing *Almost*(S) up to and including the critical packet.

Observe that merely containing a byte of *Almost*(S) does not qualify a packet to be a collaborator; it must have sequence numbers for this byte that qualify the byte to be part of a reassembly of *Almost*(S). We use "some" reassembled stream because there can be more than one possible way to reassemble a TCP stream in case there is more than one packet for the same sequence number or overlapping segments with inconsistent data.

Example: We use (s, P) to denote a TCP packet with sequence number s and payload P. With respect to *Almost*(S) = $ABCDE$, suppose a TCP connection sends the first packet containing $(0, \text{AB})$, the second containing $(3, \text{XY})$, the third containing $(5, E)$, the fourth containing $(3, \text{CD})$, and the fifth containing $(0, \text{ABC})$. Then the critical packet is the fourth packet because while there are two possible reassembled streams up to and including this packet (ABXYE and ABCDE), there is one reassembled stream containing the

string. Note that while this is also true after the fifth packet, this is first true after the fourth packet received by the IPS. The collaborators of the critical packet are the first, third, and fourth packets.

The following lemma states, intuitively, that the critical path and its collaborators must either contain a piece or be sufficiently small to warrant being copied to the slow path. See Figure 5.

LEMMA 6.1. *For any TCP connection and string S, the critical packet for string S and all collaborators of the critical packet will either contain a piece of S in its entirety or have payload length $< 2PieceSize - 1$.*

PROOF. Consider a packet P that is either the critical packet or a collaborator of the critical packet. Such a packet must contain some byte of string S that contributes to some reassembly of string *Almost*(S) at an endnode satisfying atomicity. Thus it contains a portion of some Piece $I, 1 < I < K$, that is used in this reassembly.

Case 1: If packet P contains any piece in its entirety that is used in the reassembly of *Almost*(S) we are done.

Case 2: Packet P does not contain a piece in its entirety that is used in the reassembly. Thus, it must either contain a portion of Piece I only, or contain a beginning portion of Piece I and a portion of Piece $I-1$, or a trailing portion of Piece I and a portion of Piece $I+1$. It cannot contain a complete piece of either Piece $I-1$ or Piece $I+1$ by assumption. Thus the part of packet P containing a portion of String *Almost*(S) must (in all three cases) be of length $< 2PieceSize - 1$.

Now we want to show that P can only contain this portion of Signature S (i.e., it can have no more bytes that makes this packet "large" and hence undetectable by the slow path). If it has at least $2PieceSize - 1$ bytes and these bytes are consistent with string *Almost*(S), we would have a complete piece which contradicts the assumption of Case 2. Otherwise, if it has more bytes and these bytes are inconsistent with *Almost*(S), then there must be another packet that contains the correct bytes for *Almost*(S) at the corresponding sequence numbers that are part of the reassembly of *Almost*(S) at some endnode, and these two packets would violate the weakatomicity delivery assumption at the endnode. Thus P can only contain portions of *Almost*(S) of length $< 2PieceSize - 1$ and no further bytes, and thus must itself be of length $< 2PieceSize - 1$. □

The main theorem formalizes the role of the fast path.

THEOREM 1. *(***Fast Path Diversion***) A TCP connection containing string S in some reassembled stream will be diverted to the slow path before or while processing the critical packet in the fast path. Further, if prior to diversion the fast path processed a collaborator of the critical packet, then a copy of the collaborator was sent to the slow path.*

PROOF. We prove the second part of the theorem first. Consider any collaborator packet processed by the fast path before diversion. By Lemma 6.1, such a packet will either contain a piece in its entirety or be of payload size $< 2PieceSize - 1$. In either case, the forwarding rules will ensure that a copy will be sent to the slow path. For the first part of the theorem, consider 2 cases:

Case 1: A complete piece is sent in its entirety before or including the critical packet. In that case, by the forwarding rules, after (and including) this packet, the connection will be diverted to the slow path.

Case 2: A complete piece is not sent before or including the critical packet. Then we know that for each piece $I, 1 < I < K$, a portion of the piece must be sent in a separate packet up to and including the critical packet. By Lemma 6.1 and by the fact that we have excluded **Case 1**, the packet containing the portion of Piece I must be of size $< 2PieceSize - 1$. Order these fragments by the time at which they first arrive at the IPS. Thus fragment 1 is the first packet containing a portion of the signature that is processed by the fast path, fragment 2 is the next packet, and so on.

We claim that between the arrival of Fragment J and Fragment $J+1$, *count* for the connection must increment by 1. Suppose not. We know that Fragment J and Fragment $J+1$ are small because their payload size is $< 2PieceSize-1$. If *count* does not increment, then the *OOO* bit must be false when $J+1$ is received, and so the sequence numbers must increase in order from Fragment J to Fragment $J+1$. But since Fragment J and $J+1$ both contain portions of the signature, then the difference in sequence numbers from Fragment J to Fragment $J+1$ must be less than the signature length L. But in this case (see state machine), *count* must have incremented, a contradiction.

But if between any 2 consecutive fragments, *count* increases by 1, and there are $K-1$ fragments, then *count* must have reached $K-1$ before the critical packet. But in that case, by the diversion rules, the connection must have been diverted after the critical packet is processed by the fast path. □

The final theorem formalizes the role of the slow path.

THEOREM 2. **(Slow Path Blocking)***: A TCP connection containing string S in some reassembled stream will have its critical packet dropped in the slow path* **(Safety)**. *Conversely, a TCP connection that does not contain Almost(S) in some reassembly of the connection and has no inconsistent data will not have any packets dropped at the IPS* **(Liveness)**.

PROOF. The safety part of Theorem 2 follows from Theorem 1. If Theorem 1 is true, then it is clear that after and including the critical packet, the flow is being processed by the slow path. Also, any collaborators of the critical packet forwarded by the fast path are already at the slow path processor. Since the flow is now being handled by the slow path, all remaining (if any) portions of Pieces 2 through $K-1$ of S (by the definition of collaborators and *Almost*(S)) will also be received by the slow path. If there is more than possible reassembly of the packets that the slow path has received, then we know that the slow path will be configured to drop further packets of this flow and we are done. If, on the other hand, there is only one reassembly of the packets received so far, and the slow path has received Pieces 2 through $K-1$, it must reassemble these pieces to put them in sequence. Since the slow path algorithm is configured to drop all subsequent packets if it finds Pieces 2 through $K-1$ in sequence, then at least the critical packet will not have been forwarded before the Slow Path begins dropping. Hence, by the definition of the critical packet, string *Almost*(S) (and hence string S) cannot reassembled at the endnode.

The liveness part follows trivially from the fact that the slow path only drops packets from a connection when it

either finds more than one possible TCP reassembly (which can only happen if the connection has inconsistent data), or if the slow path finds Pieces 2 through $K-1$ of some signature in sequence in its reassembly of the connection. □

7. RESULTS

Beyond correctness, the motivation for Split-Detect is performance. In this section, we describe preliminary results that indicate that Split-Detect can achieve a speedup of 10, and a memory compression of between 10 and 100, making it possible to implement it on-chip at 20 Gbps. We also show robustness of the result across *time* (same packet capture point, different times) and *space* (different packet capture points, different networks).

In the trace-driven simulations, flow states are created when the first packet whose payload contains at most $2 * PieceSize - 2$ bytes is encountered for the flow, as described earlier. Flow states are aged out if no packet is received for the connection for at least 2 minutes. Figure 6 provides statistics about the traces we analyzed. All of them except "A Large enterprise" are publicly available.

The results that follow are described in terms of tables with the following headings for columns. Packets with between 1 and $Small\_Thresh$ bytes (inclusive) in their TCP payloads are considered small, where $Small\_Thresh$ is equal to $2PieceSize - 2$. The count at which redirection occurs) is equal to $K-1$, where $Num.Pieces = K$. *Max flows* is the maximum number of flows in the fast path's flow table at any time during the simulation over the packet trace. "% flows" is equal to Max flows divided by the total number of flows in the trace. To calculate the total number of flows in the trace, we ran a separate program that simply created a flow entry the first time a packet was seen for a new flow (regardless of its size), and aged out entries when no packets had been seen for the connection for 2 minutes. This is intended to represent the number of flow states that would be required by a traditional IPS system for the same traffic.

In the results we report separately the fraction of packets/bytes copied to the slow path, and the total fraction of packets/bytes that were diverted (either copied or redirected). This latter set of numbers represents the total load on the slow path. Our simulations are effectively performed with no pieces installed. A more complete simulation depending upon packet content and signatures is difficult to perform using public sources due to privacy concerns.

Our first experiment examines the effect of varying the number of pieces on the two metrics of interest: the diversion ratio (fraction of traffic shunted to slow path) and the amount of state kept in the fast path (fraction of connections fast path keeps state for). We used a single OC-48 trace and varied both the Signature length and the number of pieces: we stay within the range of Signature lengths used in practice, and never decrease the piece size below 4 bytes.

In Figure 7 we vary the signature length and the number of pieces. All other parameters (e.g., small packet threshold, which is $2PieceSize - 2$) can be derived from these two parameters. The overall message is that using a reasonable small packet threshold of 8 to 16 bytes for the common case of 40 byte signatures with 4 to 8 pieces results in keeping state for only 5% of the flows and diverts 8% to 12% of the traffic in either bytes or packets, providing a factor of 10 improvement in throughput. This implies that the slow path can run at 2 Gbps, which is easily achievable today.

Similarly, keeping 5% of 1 million flows (not the numbers in the traces but the numbers aimed for by an IPS today) results in keeping track of 50,000 flows which at 150 bits per flow (100 bits for flow ID and 48 bits for state) for the state machine described in Section 5.4 results in 7.5 Mbits of memory. 7.5 Mbits is easily achievable using on-chip memory, allowing a single chip implementation of the fast path state machine. For graceful degradation, if the fast path exhausts its on chip memory, all subsequent flows that contain a small packet have to be diverted to the slow path.

igure 8 compares one of the previous results against one that we consider to be a poor choice of parameters. In general, breaking a large signature into only a few large pieces is bad for two reasons. First, it leads to a larger $Small\_Thresh$ value, and thus more packets are considered small by the fast path. Second, it leads to a smaller value that count must reach before the rest of the flow is redirected to the slow path.

For all of the remaining results, we report only the results for $PieceSize = 6$ (thus $Small\_Thresh = 10$) and $K = 5$, which is possible for signatures containing at least 30 bytes. Although the table is not shown, we compared results using the same parameters, but for traces collected from the same link (an OC-48 link) at three different times, each taken several months apart. The intent was to determine whether there was any obvious trend in the traffic characteristics indicating that the performance of Split-Detect changes over time. The results for the August 14, 2002 and January 15, 2003 are similar. The fraction of packets sent to the slow path for the April 24, 2003 trace is noticeably lower. We attribute this to the fact that it was taken at a different time of day, when the total traffic load was lower. The results show that in all cases the fraction of diverted traffic stays under 10 %, as needed for our speedup arguments, but can be lower (as low as 3 %) during some periods.

So far the results have been for a single wide-area trace. Figure 9 uses the same parameter values and reports the results for all the traces shown in Figure 6. Figure 9 shows that the results seem fairly invariant to the type of trace used. The University and large enterprise traces we used, in particular, are more representative of enterprises where an IPS is more likely to be deployed.

Finally, we note that with DFA implementations of string matching (such as Aho-Corasick [1]), the cost of string matching increases linearly with the bytes in a string. Thus increasing the number of pieces (without changing the overall bytes matched) should not greatly increase complexity.

8. CLEAN SLATE APPROACHES

While much of this paper deals with existing TCP endnodes, we have argued for a small change in TCP endnodes (weak atomicity). Given that there is a general dissatisfaction with the status quo as evinced by proposals such as FIND to rearchitect the Internet [7], it is worth posing the question: what other changes in transport protocols could make the job of detecting signatures easier in the network?

Even with the assumption of weak atomicity in endnodes and exact signatures, the solution described in this paper had the disadvantage of only detecting $Almost(S)$ instead of the exact signature S. Recall that $Almost(S)$ is S with the first and last pieces missing. While one can argue that if S is long enough, this does not change the false positive rate appreciably, this is difficult to sell to security analysts.

Trace	Duration (min)	Avg. pkts/sec	Avg. bits/sec	% of TCP packets
CAIDA 2002-08-14 09:00	5	75 K	344 M	93%
CAIDA 2003-01-15 09:59	5	59 K	326 M	88%
CAIDA 2003-04-24 00:55	5	23 K	92 M	91%
A large enterprise	2.7	6 K	25 M	90%
Univ. Florida	1.5	39 K	223 M	91%
Lawrence-Berkeley National Lab	60	2 K	7 M	97%

Figure 6: Summary data for the packet traces analyzed

Sig Length	*PieceSize*	Num. Pieces	Small_Thresh	Max flows	% flows	% pkts/bytes copied	% pkts/bytes diverted
40	10	4	18	34494	6.76%	0.51% / 0.04%	10.85% / 9.98%
40	8	5	14	30482	5.97%	0.51% / 0.04%	9.06% / 8.31%
40	5	8	8	22645	4.44%	0.46% / 0.04%	6.02% / 5.29%
40	4	10	6	18830	3.69%	0.38% / 0.03%	3.43% / 4.04%
30	10	3	18	34494	6.76%	0.38% / 0.03%	12.66% / 12.01%
30	6	5	10	24215	4.75%	0.42% / 0.03%	8.03% / 7.20%
30	5	6	8	22645	4.44%	0.42% / 0.03%	6.88% / 6.24%
30	3	10	4	7837	1.54%	0.15% / 0.01%	2.00% / 2.41%
20	5	4	8	22645	4.44%	0.34% / 0.03%	8.39% / 7.95%
20	4	5	6	18830	3.69%	0.31% / 0.02%	5.48% / 6.38%

Figure 7: Effect of varying parameters on the same packet trace

However, the following radical change in the transport protocol can remedy this. Imagine that the transport protocol repeats the last X bytes of each packet in the first X bytes of the next packet. Then it follows that any string S of length no more than X will be contained in its entirety in some packet. If string S happens to split across packet J, it will be found in its entirety in packet $J+1$. If X is as large as the piece size (say 10 bytes), then a very simple fast path state machine can divert a flow to the slow path if any piece is detected.

The Clean Slate approach also suggests the following *implementation* alternative to architectural revolution. Send the last X bytes and the first X bytes of every packet to the slow path. If the average packet size is say 200 bytes and X is 10, this will add a further 10% overhead to the diverted traffic. Finally, if the Slow Path detects $Almost(S)$, then these additional bytes can be examined to confirm that S was sent before dropping packets in the connection.

9. CONCLUSION

This paper is a gentle first volley suggesting an alternative to full state reassembly and normalization at high speeds using the ideas of cutting signatures into pieces that are looked for as well as looking for unusual small packet activity indicative of attempts to cut between pieces. While much remains to be done:

- The experimental data seems to support a speedup of 10, and a state compression of 10 and 20.
- More compression appears possible via compact data structures (e.g., Bloom Filters) in return for diverting slightly more than the required number of flows.

The endnode atomicity required by Split-Detect may seem too high a price to pay. However, we believe it is possible to prove a lower bound to show that detecting overlapping fragments in the network requires almost as much memory as data normalization. If this is true, then endnode atomicity may be required for high performance reassembly and normalization. From our preliminary investigation, weak atomicity appears easy to implement after adding one MTU worth of extra buffering and a small amount of extra processing in rare cases. Note that the assumption that $Almost(S)$ is detected and not S can be remedied by diverting some of the first and last bytes of every packet.

Finally, our paper has dealt only with exact matching but most IPS vendors support regular expressions. First, note that while regular expressions are commonly used to abstractly describe *vulnerabilities*, exact strings suffice to describe *exploits* which may suffice for blocking at switches in the immediate aftermath of an attack such as a worm.

Further, we have already identified a class of regular expressions that can be handled by Split-Detect techniques. These are regular expressions that use the OR operator (i.e., $A|B|C$) and the . operator (any single character). Thus we can do regular expressions of the form $((A|B)..(C|D)..(E|F)$, etc.). These can be broken into pieces; all we need is the fast path to be able to match these expressions using standard DFA techniques. In particular, we can handle upper and lower case, which is very common in the IPS rules.

Also, if the regular expression has a form such as $X * Y$, the fast path can send a copy of string X or Y (whichever is detected first) without diverting the flow. When the other string is seen, the flow can diverted to the slow path. Our preliminary study of a commercial IPS database as well as the Clam AV database shows that 60 to 80% of the rules fit these categories. We are working on further extensions to this, and on techniques to rewrite regular expressions to make them fit the classes we can handle.

Changes to endnodes and to the use of regular expres-

Sig. Length	*PieceSize*	Num. Pieces	Small_Thresh	Max flows	% flows	% pkts/bytes diverted
30	6	5	10	24215	4.75%	8.03% / 7.20%
48	16	3	30	48026	9.41%	15.18% / 13.76%

Figure 8: Effect of poorly chosen parameters on the fraction of diverted traffic

Trace	Max flows	% of flows	% pkts / bytes copied	% of pkts / bytes copied/redirected
CAIDA OC-48 2002-08-14	24215	4.75%	0.42% / 0.03%	8.03% / 7.20%
A Large enterprise	1743	5.00%	0.38% / 0.03%	8.81% / 11.79%
Univ. Florida	6657	7.66%	0.54% / 0.04%	5.68% / 5.56%
Lawrence-Berkeley National Lab	760	20.92%	0.21% / 0.02%	1.52% / 3.07%

Figure 9: Variation in the diverted traffic across traces taken from different networks

sions are clearly difficult to popularize. But when considered against the backdrop of an even more difficult problem, that of detecting attack signature at very high speeds, perhaps such changes seem more reasonable.

10. ACKNOWLEDGEMENTS

We are grateful to Jonathan Chang, Tom Edsall, Mike Hall, Pere Monclus, Sushil Singh, and Sumeet Singh of Cisco Systems for fruitful discussions. George Varghese would also like to acknowledge NSF Grant ANI 0137102 and a grant from NIST that helped stimulate the direction of the current research, which was done entirely at Cisco.

11. REFERENCES

[1] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching: An aid to bibliographic search." *Communications of the ACM* 18(6):333-340, June 1975.

[2] N. Alon, Y. Matias, and M. Szegedy. "The space complexity of approximating the frequency moments". *Proceedings 28th ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[3] G. Appenzeller, I. Keslassy, and N. McKeown "Sizing Router Buffers". *Proceedings of ACM SIGCOMM*, 2004.

[4] D. Clark, "The Structuring of Systems Using Upcalls". Proceedings of the *10th ACM Symposium on Operating Systems Principles*, pp. 171–180, December 1–4 1985.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters. *Hot Interconnects*, Aug. 2003.

[6] S. Dharmapurikar, V. Paxson, "Robust TCP stream reassembly in the presence of adversaries". *Proceedings of the 14th USENIXSecurity Symposium*, Baltimore, 2005.

[7] "The Future of the Internet". *Red Herring*, April 10th, 2006.

[8] M. Handley, C. Kreibich, and V. Paxson. "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics". *Proc. USENIX Security Symposium*, May 2001.

[9] K. Levchenko, R. Paturi, and G. Varghese. "On the Difficulty of Scalably Detecting Network Attacks". *Proc. of the Eleventh ACM Conference on Computer and Communication Security*, October 2004.

[10] Nikto, *http://www.cirt.net/code/nikto.shtml*

[11] NSS Group. Intrusion Prevention Systems (IPS) Group Test (Edition 3), NSS Group, August 2005, *http://www.nss.co.uk*

[12] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time". *Computer Networks*, 31(23-24), pp. 2435-2463, 14 Dec 1999

[13] V. Paxson and M. Handley, "Defending Against NIDS Evasion using Traffic Normalizers". *Second International Workshop on the Recent Advances in Intrusion Detection*, September 1999.

[14] T. Ptacek and T. Newsham. "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection", Secure Networks, Inc., Jan. 1998.

[15] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks", *LISA 99*.

[16] C. Shannon, D. Moore, k. claffy, "Characteristics of Fragmented IP Traffic on Internet Links", *Workshop on Passive and Active Measurement*, 2001.

[17] Dug Song, 2002, Fragroute, *http://www.monkey.org/ dugsong/fragroute/*