Enhancing Network Intrusion Detection With Integrated Sampling and Filtering

Jose M. Gonzalez and Vern Paxson

International Computer Science Institute
Berkeley, California, USA

chema@icsi.berkeley.edu, vern@icir.org / vern@ee.lbl.gov

Abstract The structure of many standalone network intrusion detection systems (NIDSs) centers around a chain of analysis that begins with packets captured by a packet filter, where the filter describes the protocols (TCP/UDP port numbers) and sometimes hosts or subnets to include or exclude from the analysis. In this work we argue for augmenting such analysis with an additional, separately filtered stream of packets. This "Secondary Path" supplements the "Main Path" by integrating sampling and richer forms of filtering into a NIDS's analysis.
We discuss an implementation of a secondary path for the Bro intrusion detection system and enhancements we developed to the Berkeley Packet Filter to work in concert with the secondary path. Such an additional packet stream provides benefits in terms of both efficiency and ease of expression, which we illustrate by applying it to three forms of NIDS analysis: tracking very large individual connections, finding "heavy hitter" traffic streams, and implementing backdoor detectors (developed in previous work) with particular ease.

1 Introduction

The structure of many standalone network intrusion detection systems (NIDSs) centers around a chain of analysis that begins with packets captured by a packet filter [21,18]. The filter reduces the volume of the stream of packets to analyze by specifying which protocols (TCP/UDP port numbers) and sometimes hosts or subnets to include or exclude from the analysis. Such filtering can prove vital for operating a NIDS effectively in a high-volume environment [5].

In addition, modern NIDS do not analyze isolated packets but instead perform inspection at different layers (network/transport/application), which requires maintaining large quantities of state in order to reassemble packet streams into byte streams and then parse these into the corresponding Application Data Units (ADUs). A NIDS may therefore need to store an indefinite amount of per-connection data for an indefinite amount of time, including both ADU contents and also network- and transport-layer contents for resilience to evasions based on stack or topology ambiguities [19,18].

In this work we propose augmenting a NIDS's analysis with an additional, separately filtered stream of packets. This "Secondary Path" supplements the "Main Path" by integrating sampling and richer forms of filtering into a NIDS's analysis. We argue that while many forms of NIDS analysis require the traditional deep-and-stateful processing path, for other forms of analysis we can trade off isolated packet processing in exchange for significant efficiency gains. The Secondary Path complements a NIDS's main analysis by providing a lightweight, stateless, packet-capture processing path.

The power of Secondary Path processing depends critically on the power of the filtering mechanism that drives it. To this end, we develop two enhancements to the popular Berkeley Packet Filter (BPF; [16]) that allow analyzers to "cherry-pick" the packets they are interested in. We can use the first enhancement, introducing randomness as a first-class object in BPF, for in-kernel random sampling of packets, connections, hosts, host pairs, or such. The second enhancement provides richer in-kernel filter control mechanisms, including a lightweight form of persistent state. We do so by adding to BPF fixed-size associative tables plus a set of hash functions to index them.

After presenting these enhancements, we then present three examples of additional analysis enabled by the Secondary Path: tracking large connections, identifying "heavy hitter" flows, and incorporating backdoor detection algorithms developed in previous work. While we can easily implement each of these by themselves in a standalone fashion, the Secondary Path allows us to unify their expression using a single mechanism, one that also incorporates the analysis they provide into the broader context of a NIDS's full analysis.

Section 2 introduces related work. In Section 3 we discuss our enhancements to BPF and in Section 4 our implementation of a Secondary Path for the Bro intrusion detection system [18]. Section 5 presents the example applications mentioned above, and Section 6 concludes.

2 Related Work

The field of network intrusion detection has an extensive literature. In particular, numerous signature-based, packet-oriented approaches such as provided by Snort [21] are based in essence upon various forms of packet filtering. Here, we confine ourselves to the subset closely related to the notion of incorporating *additional* packet processing into a NIDS, or extending packet filters for enhanced performance.

Earlier work has discussed the central role which packet filters can play in high-performance network intrusion detection [18,21]. More recent work has also explored precompiling a set of filters that a NIDS can then switch among depending on its workload [15,5] or upon detecting floods. To our knowledge, however, supplementing a NIDS's primary filter with an additional, quite different filter, has not been previously explored in the literature.

Related to our packet filter extensions, MPF [25] explored adding state to BPF [16] in order to process IP fragments. xPF added persistent memory to BPF in the form of an additional memory bank that BPF filters can switch to and from [12]. This work also removed BPF's prohibition of backward jumps, with an intent to enabling packet filters to perform in-kernel analysis (such as computing connection round-trip times) as opposed to simply filtering. xPF's persistent state is similar in spirit to what we have added to BPF, though implemented at a lower level of abstraction, which can provide greater flexibility but at a cost of requiring many more BPF instruction executions, and permitting arbitrary looping in BPF programs. mmdump introduces a method to construct dynamic filters in order efficiently to support capture of multimedia sessions for which some of the connections use dynamic ports [24]. Finally, some firewall packet filters (Linux Netfilter, BSD pf) offer similar functionality to that of the packet filter extensions, namely randomness and some state control.

The example applications described in Section 5 have roots in previous work. The problem of detecting large connections is similar in spirit to previous work on "sample and hold" [9], though our approach exploits the transport sequencing structure of TCP rather than enhancing random sampling. (We note that we can combine our random-number and associative table enhancements to BPF to implement sample-and-hold.) Our "heavy hitters" detector, which aims to capture the quantitative importance of different granularities of traffic, was inspired by *Autofocus*, a tool that automatically characterizes network traffic based on address/port/protocol five-tuples [8]. Finally, we take our backdoor detectors from [26]. We use them as examples of the ease-of-expression that the Secondary Path can provide.

3 New Packet Filter Mechanisms

In this section we introduce two extensions to BPF that bolster the expressive power of the Secondary Path while minimizing the performance overhead of the additions. For details and more discussion, including performance experiments, see [11].

3.1 Random Number Generation

When dealing with large volumes of network traffic, we can often derive significant benefit while minimizing the processing cost by employing sampling. Generally, this is done on either a per-packet or per-connection basis. BPF does not provide access to pseudo-random numbers, so applications have had to rely on proxies for randomness in terms of network header fields with some semblance of entropy across packets (checksum and IP fragment identifier fields) or connections (ephemeral ports). These sometimes provide acceptable approximations to random sampling, but can also suffer from significant irregularities due to lack of entropy or aliasing; see [11] for an analysis.

To address these problems, we added pseudo-random number generation to BPF. We do so by providing a new instruction that returns a pseudo-random number in a user-provided range. We also provide high-level access to these numbers via a new "random" keyword for tcpdump's expression syntax. The semantics of the new term are straightforward: "random(x)" yields a random number between 0 and $x - 1$, so, for example, the expression "random(3) = 0" returns true with probability 1 in 3.

Our implementation provides two different PRNGs, a fast-but-not-strong Linear Congruential Generator [17], and a slower-but-stronger random number generator based on RC4 [22]. We also permit the user to seed the PRNG directly to enforce deterministic behavior, useful for debugging purposes.

The main implementation difficulties relate to BPF's optimizer, which considers itself free to arbitrarily reorder terms. Doing so can change the expression semantics when using "random". This problem also arises when using persistent state (see next section), as an insert may affect a later retrieve. Moreover, BPF is keen to collapse two equivalent subexpressions with no dependencies, which would cause two calls to "random" with the same value of x to produce the same result. We avoid these problems by modifying the optimizer to forbid reordering around "random" terms or hash table accesses, and by marking all "random" instructions differently so none are viewed as equivalent [11].

3.2 Persistent State

The second modification to BPF consists of the introduction of persistent state, i.e., a mechanism for storing and recovering information across packets. Our implementation does so by providing multiple fixed-size associative arrays, which can be indexed using a subset of packet header fields as hash keys, or, more generally, any values we can compute using BPF expressions. For each associative array, the user can specify the key length, value (yield) length, and table size. Access is via functions to insert, retrieve, and delete entries.

Associative arrays permit efficient, dynamic, fine-grained control of the filter program. For example, we can configure an associative array to keep one bit per connection to indicate whether to filter packets from the connection in or out (essentially a Bloom filter [1]). Testing this for the presence of a given packet's connection is $O(1)$ (efficiency), and adding or deleting elements in the table requires only an insert or a delete operation (dynamic access).

A key issue, however, is sizing the arrays. We need to limit the size of each array lest they grow to consume too much kernel memory; particularly problematic if an attacker can cause the filter to continually add new entries. One possibility would be to allow dynamic expansion of arrays up to a given point, using incremental resizing as discussed in [5] to avoid processing spikes within the kernel as we expand an array.

This introduces considerable implementation complexity, however, so currently we keep the arrays fixed-size. Doing so exacerbates a different problem, however: when inserting a new entry, a collision in the hash table may require eviction of an existing tuple without the BPF program explicitly requesting it, violating the consistency of the state used by the program. We diminish this effect by providing pseudo-random hash functions (to resist adversaries) and by introducing set-associativity in the tables, as described below. However, these do not provide a complete solution, so for now we must restrict ourselves to those applications for which we can tolerate such evictions.

Associative tables require hash functions to index them, and different applications call for different tradeoffs in the properties of these functions. Our implementation provides three function types: (a) LCG [17], a simple, fast function, but prone to worst-case behavior with either degenerated workloads or algorithm complexity attacks [3]; (b) MD5, slow but with cryptographic strength [20]; and (c) UHASH, a universal hash function that provides less strong guarantees than cryptographic hash functions, but runs much faster [2].

In addition, the user can specify for each table its set-associativity, i.e., how many different keys reside at each hash location in the table. The higher the set-associativity, the fewer forced evictions, but also the more processing required per lookup.

We provide two types of access to the associative arrays: from within BPF programs, which lets us maintain filtering decisions across packets (such as for random sampling on a per-connection basis, in order to remember which connections we previously selected), and directly from user-level (via *ioctl*, though the implementation of this is not complete yet). This latter allows us to flexibly and quickly tailor packet capture in response to changing conditions. For example, we can use a filter that consults a table indexed by connection 5-tuples (addresses, ports, transport protocol) to capture packets corresponding to specific connections of interest, and might update this dynamically when our user-level analysis parses an FTP control channel to find the dynamic port negotiated for a pending FTP data connection.

User-level control also facilitates downloading very large tables; for example, a list of 1000s of botnet addresses for which we wish to capture traffic involving any of them. This application is infeasible using unmodified BPF. Even if the in-line BPF code to check so many addresses fit within the space allowed for BPF programs, the $O(N)$ processing for BPF to scan such a list would be prohibitive. Similarly, for unmodified BPF, if an application needs to make any change to its filter (e.g., add a new connection or delete an existing one), it must create the new filter from scratch, write the tcpdump expression, compile and optimize it, and then send it to the kernel for the latter to check and install.

Here is an example[1] of a tcpdump filter that checks whether the connection associated with a given packet is in table #2 (using the LCG hash function), and, if not and the packet represents an initial SYN (no ACK), randomly samples the packet with probability 1% by adding it in that case to the table (with a yield value of 1):

```
(lookup(2, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]))) or
(lookup(2, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2]))) or
( (tcp[13] & 0x12 = 0x2) and
  (random(100) = 1) and
  (insert(2, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]), 1)) )
```

Note that this code is imperfect: if the sender retransmits the initial SYN, we will generate a fresh random number, increasing the probability that we sample the connection. We could avoid this problem by always inserting connections into the table and using different yield values to indicate whether or not to subsequently sample packets belonging to the connection. The code will always be imperfect, however, since the "insert" might cause eviction of a previous connection due to a collision. In general, we cannot use our associative tables for bullet-proof analysis, but only for often-correct-is-good-enough analysis (with which our example applications below conform).

4 Introducing a Secondary Path for Packet Processing

The structure of a stateful NIDS typically consists of (a) capturing traffic from one or several packet-capture devices, (b) checking network- and transport-layer headers, (c) reassembling application-layer contents (ADUs), and (d) dispatching the contents to an application-specific analyzer. We call this mechanism the "Main Path." The

[1] The expression begins with two "lookup"'s to test both directions of the connection for presence in the table. Clearly, it would be useful to introduce some tcpdump idioms for some of the common constructions.

connection-oriented nature of the Main Path permits hiding the details of header verification and reassembly from the application-layer analyzers.

The main drawback of designing for full, application-oriented analysis is that the traffic processed by the Main Path must correspond to full connections. This limits substantially the use of input-volume control techniques (sampling or filtering)—which may be highly desirable for performance reasons—to those that we can express on a per-connection basis (such as filtering on elements of the connection 5-tuple).

While we view full-payload analysis as a must for sound, deep, stateful analysis, for some forms of analysis we can obtain complementary information much more efficiently by the analysis of isolated packets. In our architecture, we obtain this information in a fashion independent from the Main Path, and use it to supplement or disambiguate the analysis produced by the latter.

The "Secondary Path" provides an alternate channel for acquiring packets. It works by capturing packets from one or several packet-capture devices in addition to those used by the Main Path, and dispatching the packets to corresponding analyzers without any previous analysis.

It is very important to note that the Secondary Path is an alternate channel: it provides a stateful NIDS with a means to obtain information about the monitored traffic whose generation using the Main Path is either inefficient or ambiguous. It does not aim to substitute for the Main Path, but to complement it.

Our main contribution regards not the analysis by a NIDS of isolated (e.g., sampled) packets, but rather the integration of the results from such analysis with a NIDS's regular, full-payload analysis. In our case, this integration is facilitated by the flexible and powerful state capabilities of Bro. We use the Secondary Path to distill information that when solely employing Primary Path processing would be expensive (due to volume) or difficult to obtain (due to the Primary Path's initial filtering not capturing the necessary information). For example, we can use the Secondary Path to spot flooding sources or victims via random sampling, which can then inform load-shedding decisions made by the Primary Path [5]. For a number of types of analysis, Secondary Path processing can be quite cheap because we can perform it at a much lower rate than Primary Path processing, such as illustrated in the example applications discussed in § 5.

It is important to stress that the information distilled from the Secondary Path is typically limited to identifying subsets of traffic that are either large enough to ensure they can be detected by sampling, or distinctive enough to ensure they can be spotted using static filtering. The Secondary Path is therefore not a tool to detect specific attacks (unless their signature is distinctive enough as to permit detection by packet filtering), but a means for gathering additional information or context.

One significant feature of the Secondary Path is its simplicity. It serves analyzers isolated packets instead of full connections. Because it does not carry out reassembly, its can operate in a stateless fashion, unless the analyzer itself chooses to maintain state. However, an important, negative consequence of this stateless operation is that analysis through the Secondary Path is often susceptible to evasion due to the inability to detect or resolve traffic ambiguities [19,18]. Similarly, Secondary Path analyzers must exercise care when using transport- or application-layer contents, as these may be only partially present, or arrive out of order or even duplicated.

Table 1 summarizes the main differences between the Main Path and the Secondary Path.

	Main Path	*Secondary Path*
Processing performed	L3, L4 analysis	none
Objects provided	L7 ADUs	L3 packets
L4 reassemble	yes	no
Memory	stateful	stateless
Filtering flexibility	port-, address-oriented	rich when coupled with stateful BPF (see § 3.2)
Sampling	connection-oriented only	rich when coupled with randomness in BPF (see § 3.1)

Table 1. List of Differences between the Main and Secondary Paths

4.1 Filtering

A major benefit of the Secondary Path is its potential efficiency, with its key application being to tasks for which only a low volume of traffic will match the filters it employs. Such filters can be in terms of network- and/or transport-layer headers, which are readily supported by packet capture mechanisms such as BPF. Note however that transport-layer based filtering is less reliable, as TCP headers can be divided across multiple IP packets. On the other hand, in the absence of adversary evasion, such fragmentation is generally rare [23].

The filter can also include application-layer contents. While BPF limits filtering to matching bytes at essentially fixed positions, modern application-layer protocols sometimes use headers with distinctive contents in specific locations [26]. For example, HTTP request headers start with one of seven different method strings (“`GET`”, “`POST`”, etc.), and HTTP response headers start always with the string “`HTTP/`” [10]. We could thus filter on the first 5 bytes of TCP payload being “`HTTP/`” to capture with high probability exactly one packet per HTTP transaction, since HTTP entity headers are typically sent in a different packet than the previous entity body. Such an analyzer can also access HTTP responses seen using non-standard ports.

Due to the fixed-location limitation of packet filtering, and the stateless condition of the Secondary Path, application-layer contents provide less leverage than network- or transport-layer contents, and more vulnerability to attacker manipulation. For example, if an attacker wants to avoid detection of an HTTP connection, they can split the first 5 bytes across two TCP packets; if they want burden a NIDS trying to detect HTTP traffic, they can cheaply forge faked packets with those 5 bytes at the beginning.

4.2 Sampling

A particularly handy form of of filtering in terms of thinning the volume of traffic the NIDS must process for some types of analysis concerns sampling. Using our extensions to BPF presented in the previous section, we can do this on (for example) either a per-packet or per-connection basis. When deciding which to use, it is important to bear in mind that packet-based sampling generates a completely unstructured traffic stream, but for which many properties remain related to those of the original stream [6,7].

An example of the utility that sampling can provide is in efficiently detecting "heavy hitters," i.e., connections, hosts, protocols, or host pairs that account for large subsets of all the traffic, or that have peculiarly large properties (such as very high fan-out). Given unbiased sampling (which our BPF "random" operator provides, unlike previous approaches based on masking out header bits), a heavy hitter in the full traffic stream is very likely also a heavy hitter in a sampled traffic stream. We explore this further as an example application in Section 5.2.

4.3 Operation

The operation of the Secondary Path is fairly simple: analyzers provide a packet filter expression that defines the traffic subset for which they wish to perform isolated packet analysis. The Secondary Path creates a filter resulting from the union of all the analyzer filters (Secondary Filter), and opens a packet filter device with it. When a packet matches the common filter, the Secondary Path runs each particular analyzer filter against the packet, demultiplexing the packet to all analyzers whose filters match the packet.

One subtlety arises, however, due to the fact that during Secondary Path operation we actually run each analyzer filter twice (first as a part of the full Secondary Filter, second to see whether the analyzer's particular filter matched). This "re-filtering" does not present problems for stock BPF filters, since they are idempotent—running a filter F over a set of packets already filtered by F does not cause the rejection of any packet. However, when using our BPF extensions for randomness and maintaining state, filters are no longer idempotent.

This generally will not present a problem for filters that maintain state, since two copies of the state exist, one in the kernel used for the initial filtering (i.e., the matching of the entire Secondary Filter), and the other at user-level used for the demultiplexing. The latter will be brought into sync with the former when we rerun the filter.

However, the random operator remains problematic. Our current implementation maintains a separate packet filter device for each filter that uses "random", so that we do not require re-filtering to demultiplex what the filter captures. A drawback of doing so is that the BPF optimizer can no longer factor out common elements of filters that use "random", which may significantly degrade performance if we have multiple such filters. A second drawback is that the OS often limits the number of packet filter devices available.

An alternate approach would be to modify BPF to track which elements of a filter have been matched and to return this set when a packet is accepted. Designed correctly, this would allow optimization across all packet filters (including the one used by the Main Path), but is a significant undertaking given that the notion of "element of a filter" becomes blurred as BPF's optimizer rearranges and collapses terms within a filter.

4.4 Implementation

We have implemented the Secondary Path in Bro, a stateful, event-oriented NIDS [18]. Bro's analyzers are structured around a Main Path such as we have outlined in this paper. We added a new script-accessible table, `secondary_filters`, which is indexed by a packet filter (expressed as a string) and yields a Bro event handler for packets the filter matches.

We open the interface(s) being monitored twice, once for the Main Path and once for the Secondary Path. The Secondary Filter is the OR'ed juxtaposition of all the filter indices specified for `secondary_filters`. Figure 1 shows an example Bro script. It uses the secondary filter to invoke the `SFR_flag_event` event handler for every packet matching the expression "tcp[13] & 7= 0!", i.e., any TCP packet with any of the SYN, FIN, or RST flags set. `pkt_hdr` is a Bro record type representing the network- and transport-layer headers of a packet.

This particular filter can be used to track connection start and stop times, and hence duration, participating hosts, ports, and (using differences in sequence numbers) bytes transferred in each direction. The few lines shown are all that is required to then further analyze these packets using Bro's domain-specific scripting language.

```
redef secondary_filters += { ["tcp[13] & 7 != 0"] = SFR_flag_event };

event SFR_flag_event(filter: string, pkt: pkt_hdr)
  {
  # Perform analysis on the packet header fields given in "pkt" here.
  }
```

Figure 1. Secondary Path Use Example

4.5 Performance

In this section we briefly assess the performance of our Secondary Path implementation.[2] Our goal is to compare the cost within a NIDS implementation of the infrastructure required to implement the Secondary Path (dispatching plus internal piping) versus the cost of the packet filter processing. To do so, we use the Secondary Filter to trigger a null event handler, i.e., an event that does not carry out any work and returns as soon as it is invoked.

The processing cost depends not only on the number of packets that raise the Secondary Path event, but also on the number of packets than do not raise the Secondary Path event but still must be read by the kernel and eventually discarded by the Secondary Filter.

[2] Unless otherwise noted, all experiments described in this paper were carried out using an idle single-processor Intel Xeon (Pentium) CPU running at 3.4 GHz, with 512 KB cache and 2 GB of total memory, under FreeBSD 4.10. All times reported are the sum of user and system times as reported by the OS. We ran each experiment 100 times, finding the standard deviation in timings negligible compared to the average times.

Figure 2 shows the corresponding performance for different volumes of traffic and different capture ratios (proportion of packets that match the filter). Note that both axes are logarithmic.

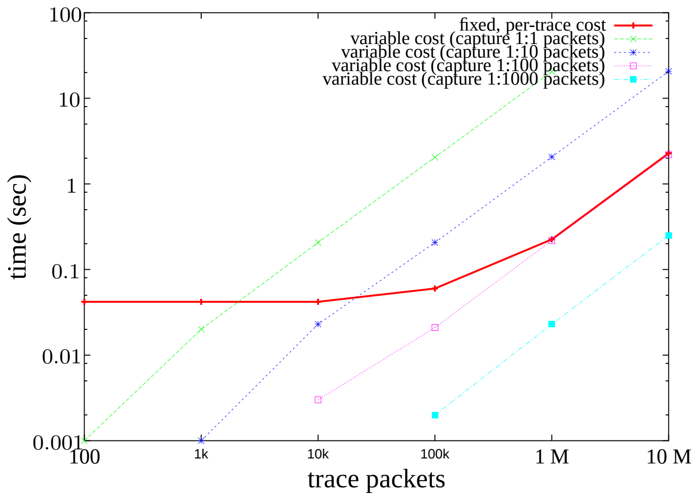

Figure 2. Performance of the Secondary Path with an Empty Event

The thick line represents the cost of rejecting all packets with the Secondary Filter. We call this cost “fixed”, as it is independent of the number of packets accepted by the Secondary Filter. It is the sum of two effects, namely (a) the fixed cost of running Bro, and (b) the cost of accessing all the packets in the stream and running the Secondary Filter over them. It is clear that the first effect is more important for small traces (the flat part to the left of the 10K packet mark), while the second effect dominates with large traces.

The dashed and dotted lines show the additional cost of empty event handlers when a given ratio of the packets match the filter. Not surprisingly, we see that this variable cost is proportional to the ratio of packets matching the filter: the variable cost of sampling, say, 1 in 10 packets is about 10 times larger than the variable cost of sampling 1 in 100 packets. We also see that the fixed cost of running the Secondary Path is similar to the variable cost of capturing 1 in 100 packets. This means that provided the analysis performed on captured secondary packets is not too expensive, whether the detector’s filter matches say 1 in 1,000 packets, or 1 in 10,000 packets, does not affect the Secondary Path overhead. When the ratio approaches 1 in 100 packets, however, the Secondary Path cost starts becoming appreciable.

5 Applications

In this section we present three examples of analyzers we implemented that take advantage of the Secondary Path: disambiguating the size of large TCP connections (§ 5.1), finding dominant traffic elements (§ 5.2), and easily integrating into Bro previous work on detecting backdoors (§ 5.3; [26]). The first of these provides only a modest enhancement to the NIDS's analysis, but illustrates the use of a fairly non-traditional style of filter. The second provides a more substantive analysis capability that a NIDS has difficulty achieving efficiently using traditional main-path filtering. The third shows how the Secondary Path opens up NIDS analysis to forms of detection that we can readily express using some sort of packet-level signature.

Unless otherwise stated, we assess these using a trace (named *tcp-1*) of all TCP traffic sent for a 2-hour period during a weekday working hour at the Gbps Internet access link of the Lawrence Berkeley National Laboratory (LBNL). The trace consists of 127 M packets, 1.2 M connections, and 113 GB of data (averaging 126 Mbps and 892 bytes/packet).

5.1 Large Connection Detection

A cheap mechanism often used to calculate the amount of traffic in a stateful (TCP) connection consists of computing the difference between the sequence numbers at the beginning and at the end of a connection. While this often works well, it can fail for (a) connections that do not terminate during the observation period, or for which the NIDS misses their establishment, (b) very large (greater than 4 GB) connections that wrap around the TCP sequence number (note that TCP's operation allows this), or (c) broken TCP stacks that emit incorrect sequence numbers, especially within RST segments.

As we develop in this section, we can correct for these deficiencies using a secondary filter. In doing so, the aim is to augment the main path's analysis by providing a more reliable source of connection length, which also illustrates how the Secondary Path can work in conjunction with, and complement, existing functionality.

Implementation Our large-connection detector works by filtering for several thin, equidistant, randomly-located stripes in the sequence number space. A truly large flow will pass through these stripes in an orderly fashion, perhaps several times. The detector tracks all packets that pass through any of the stripes, counting the number of times a packet from a given flow passes through consecutive regions (K).

Figure 3 shows an example. The 4 horizontal stripes (s_A, s_B, s_C, and s_D) represent the parts of the TCP sequence number space where the detector "listens" for packets. As the TCP sequence number range is 4 GB long, each stripe is separated 1 GB from the next one.

The thick diagonal lines depict the time and TCP sequence number of the packets of a given TCP connection. The dotted, vertical lines represent events in the Secondary Path. Note that we could use a different number of lines, and lines with different width (see below). If the detector sees a connection passing through 2 consecutive stripes ($K = 1$), it knows that the connection has likely accounted for at least 1 GB.

We locate the first stripe randomly to prevent an adversary from predicting the sections of monitored sequence space, which would enable them to overwhelm the detector by sending a large volume of packets that fall in the stripes. The remaining stripes then

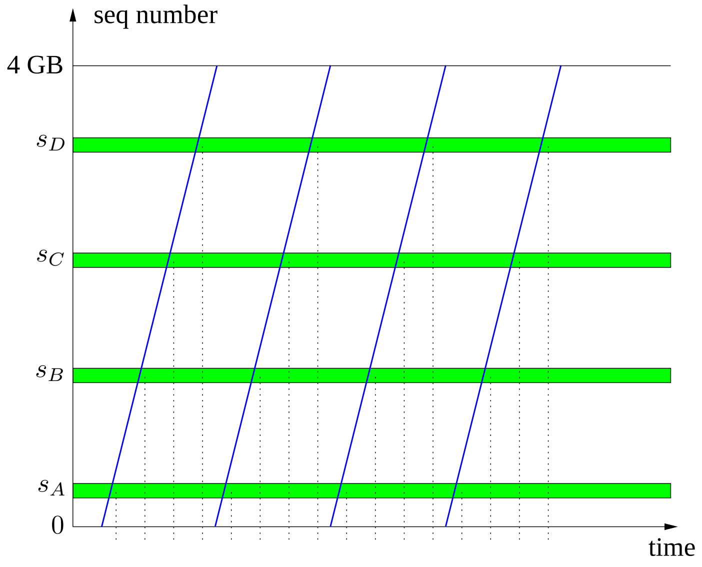

Figure 3. Large Connection Detector Example

come at fixed increments from the first, dividing the sequence space into equidistant zones.

Our detector always returns two estimates, a lower and an upper limit. If a connection has been seen in two consecutive stripes, the estimated size may be as large as the distance between 4 consecutive stripes, or as small as the distance between 2 consecutive stripes. In the previous example, we know that the connection has accounted for at least 1 GB and at most 3 GB of traffic.

We then use these estimates to annotate the connection record that Bro's main connection analyzer constructs and logs. This allows us to readily integrate the extra information provided by the detector into Bro's mainstream analysis.

One issue that arises in implementing the detector is constructing the tcpdump expression, given that we want to parameterize it in both the number of stripes and the width of the stripes. See [11] for details on doing so, and the current Bro distribution (from *bro-ids.org*) for code in the file *policy/large-conns.bro*. Note that the number of stripes does not affect the complexity of the tcpdump filter, just the computation of the bitmask used in the filter to detect a sequence number the falls within some stripe.

A final problem that arises concerns connections for which the sampled packets do not progress sequentially through the stripes, but either skip a stripe or revisit a previous stripe. These "incoherencies" can arise due to network reordering or packet capture drops. Due to limited space, we defer discussion of dealing with them to [11].

Evaluation We ran the Large Connection Detector on the *tcp-1* trace, varying the number S of stripes. We used a fixed stripe-size of 2 KB; stripe size only plays a significant

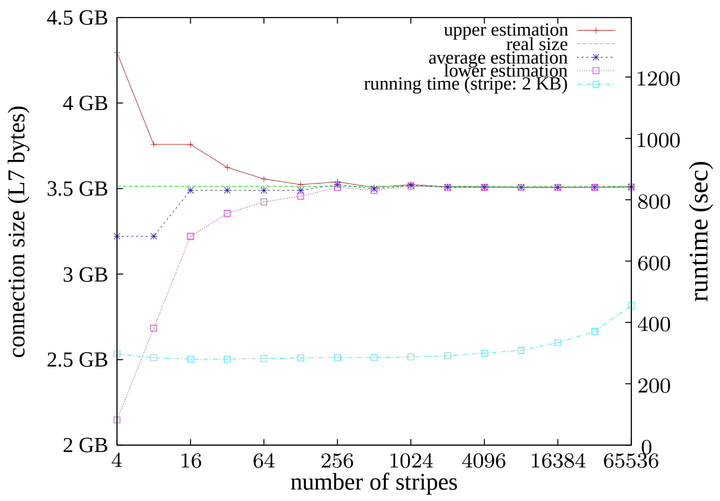

Figure 4. Detector Estimation for a Large Connection

role in the presence of packet filter drops (see [11] for analysis), but for this trace there were very few drops.

Figure 4 shows for the largest connection in the trace (3.5 GB application-layer payload), its real size, the upper and lower estimations reported by the detector, and the average of the last two (the *average estimation*), as we vary S. The lower line shows the running time of the large connection detector. (Rerunning the experiment with wide stripes, up to 16 KB, reported very similar results.) All experiments ran with the Main Path disabled, but we separately measured its time (with no application-layer analysis enabled) to be 890 sec. Thus, the running time is basically constant up to S = 8192 stripes, and a fraction of the Main Path time. Finally, we verified that as we increase the number of stripes, our precision nominally increases, but at a certain point it actually degrades because of the presence of incoherences (non-sequential stripes); again, see [11] for discussion.

5.2 Heavy Hitters

The goal of the “heavy hitters” (*HH*) detector is to discover heavy traffic *macroflows* using a low-bandwidth, pseudo-random sampling filter on the Secondary Path, where we define a macroflow as a set of packets that share some subset of the 5-tuple fields (IP source and destination addresses, transport-layer source and destination ports, and transport protocol). This definition includes the high-volume connections (sharing all 5 fields), but also other cases such as a host undergoing a flood (all packets sharing the same IP destination address field) or a busy server (all packets sharing a common IP address and port value). The inspiration behind assessing along different levels of granularity comes from the *AutoFocus* tool of Estan et al [8].

As indicated above, macroflows can indicate security problems (inbound or outbound floods), or simply inform the operator of facets of the “health” of the network in terms of the traffic it carries. However, if a NIDS uses filtering on its Main Path to reduce its processing load, it likely has little visibility into the elements comprising significant macroflows, since the whole point of the Main Path filtering is to *avoid* capturing the traffic of large macroflows in order to reduce the processing loads on the NIDS. Hence the Secondary Path opens up a new form of analysis difficult for a NIDS to otherwise efficiently achieve.

The HH detector starts accounting for a traffic stream using the most specific granuarity, i.e., each sampled packet's full 5-tuple, and then widens the granularity to a set of other, more generic, categories. For example, a host scanning a network may not have any large connection, but the aggregate of its connection attempts aggregated to just source address will show significant activity.

Note that HH differs from the large connection detector discussed in Section 5.1 in that it finds large macroflows even if none of the individual connections comprising the macroflow is particularly large. It also can detect macroflows comprised of non-TCP traffic, such as UDP or ICMP.

table name	specificity	description
`saspdadp`	4	connection (traditional 5-tuple definition)
`saspda__`	3	traffic between a host and a host-port pair
`sa__da__`	2	traffic between two hosts
`sasp____`	2	traffic to or from a host-port pair
`sa____dp`	2	traffic between a host and a remote port
`sa______`	1	traffic to or from a host
`__sp____`	1	traffic to or from a port

Table 2. Tables Used by the Heavy Hitters Detector

Operation HH works by clustering each pseudo-random sample of the traffic it obtains at several granularities, maintaining counts for each corresponding macroflow. Whenever a macroflow exceeds a user-defined threshold (e.g., number of packets, connections, or bytes), HH generates a Bro event reporting this fact and removes the corresponding traffic from the coarser-grained table entries. Note that more specific tables generally use lower thresholds than more generic ones.

Table 2 shows the tables maintained by HH. The *specificity* field orders the tables from more specific (higher numbers) to more general. The mnemonics `sa` stands for "source address," `dp` for "destination port," etc. We use Bro's state management capabilities to automatically remove table entries after a period of inactivity (no read or write).

Output Table 3 shows an example of a report generated by HH (with anonymized network addresses). The first 5 lines were produced in real-time at the given timestamp. The remaining lines are produced upon termination The *flags* field states whether the

Time	Macroflow Description	Pkts	Bytes	Event	Flags
1130965527	164.254.132.227:* <-> *:*	986 K	823 MB	large src	internal
1130969123	*:* <-> 164.254.133.198:80/tcp	1.07 M	654 MB	large dst	internal
1130990210	*:* <-> 164.254.133.194:*	1.12 M	357 MB	large dst	internal
1130992153	54.75.124.72:19150/tcp <-> 164.254.133.146:*	977 K	79 MB	large flow	
1130999627	164.254.132.247:80/tcp <-> *:*	1.02 M	781 MB	large src	internal
	164.254.132.227:* <-> *:*	1.90 M	1.47 GB	large src	internal
	164.254.133.198:80/tcp <-> *:*	1.84 M	1.22 GB	large src	internal
	164.254.132.247:80/tcp <-> *:*	1.21 M	968 MB	large src	internal
	71.213.72.252:80/tcp <-> 164.254.133.56:*	498 K	522 MB	large flow	
	:80/tcp <-> 164.254.132.88:	459 K	479 MB	large dst	internal
	: <-> 164.254.133.194:*	1.35 M	427 MB	large dst	internal

Table 3. Example Report From Heavy Hitters Detector

reported host belongs to the list of hosts belonging to the internal network being monitored (a user-configurable parameter); it is omitted for macroflows whose granularity includes both an internal and an external host.

Finally, we note that we can extend this sort of analysis using additional macroflow attributes, such as packet symmetry [14] or the ratio of control segments to data segments. Due to limited space, we defer discussion of these to [11].

5.3 Backdoor Detection

Another example of analysis enabled by the Secondary Path is our implementation of previous work on using packet filters to efficiently detect backdoors [26]. That work defines a backdoor as an application not running on its standard, well-known port, and proposes two different mechanisms to detect these.

The first mechanism consists of looking for indications of interactive traffic by analyzing the timing characteristics of small (less than 20 bytes of payload) packets. This approach comes from the intuition that interactive connections will manifest by the presence of short keystrokes (large proportion of small packets) caused by human responses (frequent delays between consecutive small packets).

The second mechanism consists of extracting signatures of particular protocols (SSH, FTP, Gnutella, etc.) and looking for instances of these on ports other than the protocol's usual one.

We implemented both approaches in Bro using our Secondary Path mechanism. Doing so is quite simple, and provides an operational capability of considerable value for integrating into Bro's mainstream analysis.

Keystroke-based Backdoor Detection Bro already includes an implementation of the "generic algorithm" for detecting interactive backdoors. In creating an implementation based on the Secondary Path, our goals were increased ease-of-expression and performance.

See [11] for details regarding our implementation. We verified its correctness by comparing its results with that of the original detector. As our evaluation trace, *tcp-1*, had almost no backdoor-like interactive traffic (just some AOL Instant Messenger), we checked how well each detector performed for discovering the trace's well-known interactive connections, namely SSH traffic. (The site no longer allows Telnet or Rlogin traffic over the Internet.) We did so by removing 22/tcp from the list of well-known ports where the detector does not carry any processing. We also had to adjust the original algorithm's notion of "small" packet upwards from 20 bytes to 50 bytes due to how SSH pads packets with small payloads.

Approach	Run Time
Main Path, no analyzers	890 sec
Main Path-based generic backdoor analyzer	+406 sec
Main Path, SP-based generic backdoor analyzer	+289 sec
SP-based generic backdoor analyzer, no Main Path	284 sec

Table 4. Performance of Generic Backdoor Detector, Main Path vs. Secondary Path

We measured four different configurations on the *tcp-1* trace, as shown in Table 4. The extra time incurred by the original detector is 406 seconds, while the extra time incurred by the SP-based version is 289 seconds.

Signature-Based Backdoor Detection We also implemented the signature-based backdoor detectors developed in [26], except we discarded the Rlogin and Telnet ones because we have found from subsequent experience (running the detectors 24x7 for several years at LBNL) they are too broad. For example, in *tcp-1*, 50 K packets match the Rlogin signature, and 92 match the Telnet one.

Again, we gain both ease-of-implementation and performance by using the Secondary Path. Regarding the former, Figure 5 shows full code for a Secondary Path implementation to detect SSH backdoors.

Approach	time
Main Path, no analyzers	890 sec
Main Path-based backdoor analyzer	+769 sec
Main Path, Secondary Path-based backdoor analyzer	+174 sec
Secondary Path-based backdoor analyzer only	327 sec

Table 5. Performance of Signature-Based Backdoor Detector

Regarding the latter, we ran four experiments using the *tcp-1* trace, for which Table 5 shows the corresponding performance. The extra cost caused by the original, Bro-event-based backdoor detector implementation is 769 sec. In comparison, the Secondary Path implementation (which is basically several pieces of the form depicted in Figure 5) adds only 174 sec. The final row shows that the analyzer by itself requires more time than just the 174 sec, since it must also read the entire (very large) traffic stream into user memory prior to filtering it, which for the third row has already been done by the Main Path.

```
# The following gobbledygook comes from Zhang's paper:
const ssh_sig_filter = "
  tcp[(tcp[12]>>2):4] = 0x5353482D and
  (tcp[((tcp[12]>>2)+4):2] = 0x312e or tcp[((tcp[12]>>2)+4):2] = 0x322e)";

# Don't report backdoors seen on these ports.
const ignore_ssh_backdoor_ports = { 22/tcp, 2222/tcp } &redef;

event backdoor_ssh_sig(filter: string, pkt: pkt_hdr)
  {
  # Discard traffic using well-known ports.
  if ( ["ssh-sig", pkt$tcp$sport] in ignore_ssh_backdoor_ports ||
       ["ssh-sig", pkt$tcp$dport] in ignore_ssh_backdoor_ports )
    return;

  print fmt("%s SSH backdoor seen, %s:%s -> %s:%s", network_time(),
      pkt$ip$src, pkt$tcp$sport, pkt$ip$dst, pkt$tcp$dport);
  }

# Associate the event handler with the filter.
redef secondary_filters += { [ssh_sig_filter] = backdoor_ssh_sig };
```

Figure 5. SSH Backdoor Detector Example

We might also consider coupling this detector with BPF state tables (Section 3.2) to activate the Main Path when a backdoor uses a protocol that the NIDS knows how to analyze. For example, if the analyzer detects an SSH connection on a non-standard port, it could add a new entry to a BPF table that captures packets for particular connections, and label the traffic accordingly so that the Main Path knows it must use its SSH analyzer to process traffic from that connection. A significant challenge with doing so, however, is the race condition in changing the filter's operation, and the NIDS's application analyzer missing the beginning of the connection. Concurrent work by Dreger et al pursues this functionality using a different approach [4].

Finally, we have explored extending this approach further to implement the P2P Traffic Profiling scheme proposed by Karagiannis et al [13]. See [11] for discussion.

6 Conclusions

We have described the Secondary Path, an alternate packet-capture channel for supplementing the analysis performed by a network intrusion detection system. The Secondary Path supports analyzers oriented towards analyzing individual, isolated packets, rather than stateful, connection-oriented analysis.

The power of the Secondary Path depends critically on the richness of packet capture that we can use it to express. To this end, we presented enhancements to the standard BPF packet-capture framework [16] to support random sampling, and retention of state between packets (similar in spirit to that of xPF [12]) and in response to user-level control.

Our implementation within the Bro intrusion detection system exhibits good performance, with a rule-of-thumb being that the Secondary Path does not significantly impair Bro's overall performance provided that we keep the volume of traffic captured with it below 1% of the total traffic stream.

We illustrated the additional power that Secondary Path processing provides with three examples: disambiguating the size of large TCP connections, finding dominant traffic elements ("heavy hitters"), and integrating into Bro previous work on detecting backdoors [26]. While none of these by itself constitutes a "killer application," the variety of types of analysis they aid in addressing bodes well for the additional flexibility that we gain using Secondary Path processing.

7 Acknowledgments

This work was made possible by the U.S. National Science Foundation grant STI-0334088, for which we are grateful.

References

1. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. J.L. Carter and M.N Wegman. Universal classes of hash functions. In *Journal of Computer and Systems Sciences*, volume 18, Apr 1979.
3. S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, Aug 2003.
4. H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. Technical report, in submission, 2006.
5. H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of CCS*, 2004.
6. N. Duffield, C. Lund, and M. Thorup. Properties and prediction of flow statistics from sampled packet streams. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 159–171. ACM Press, 2002.
7. N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 325–336. ACM Press, 2003.
8. C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 137–148. ACM Press, 2003.

9. C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 323–336. ACM Press, 2002.

10. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999. Status: INFORMATIONAL.

11. J.M. Gonzalez. *Efficient Filtering Support for High-Speed Network Intrusion Detection*. PhD thesis, University of California, Berkeley, 2005.

12. S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. Keromytis. xpf: packet filtering for lowcost network monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, 2002.

13. T. Karagiannis, A. Broido, M. Faloutsos, and K.C. Claffy. Transport layer identification of p2p traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, 2004.

14. C. Kreibich, A. Warfield, J. Crowcroft, S. Hand, and I. Pratt. Using packet symmetry to curtail malicious traffic. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets-IV) (to appear)*. ACM SIGCOMM, 2005.

15. 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*, pages 252–273, 2002.

16. S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

17. S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

18. V. Paxson. Bro: A system for detecting network intruders in real-time. *Proceedings of the 7th USENIX Security Symposium*, 1998.

19. T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Calgary, Alberta, Canada, 1998.

20. R. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. Status: INFORMATIONAL.

21. M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238. USENIX Association, 1999.

22. B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

23. C. Shannon, D. Moore, and K. C. Claffy. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6):709–720, 2002.

24. J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan. mmdump: a tool for monitoring internet multimedia traffic. In *SIGCOMM Computer Communications Review*, volume 30, pages 48–59, 2000.

25. M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, pages 153–165, 1994.

26. Y. Zhang and V. Paxson. Detecting backdoors. In *Proceedings of the 9th USENIX Security Symposium*, pages 157–170, August 2000.