

GoldenEye: stream-based network packet inspection using GPUs

Qian Gong, Wenji Wu, Phil DeMar
Fermi National Accelerator Laboratory
Batavia, IL 60510, USA
Email: {qgong, wenji, demar}@fnal.gov

Abstract—High-performance packet analysis systems have attracted great interest as tools to deal with security concerns in high-speed networks. Recently, researchers have utilized GPUs to improve packet processing performance. However, most existing work has been targeted at per-packet analysis level. Flow-centric operations have been challenging for GPUs because they require sequential operations and large buffers for flow reassembly. In this work, we present the GoldenEye GPU Packet Processing System (GoldenEye), a deep packet inspection (DPI) system that tracks out-of-order TCP packets and provides stream-based signature matching. When a batch of packets arrives, GoldenEye sorts packets into flow-reassembled streams and normalizes retransmission through a GPU-implemented *reordering* module. For signatures that straddle batch boundaries, GoldenEye couples a small set of metadata with a functionally-equivalent minimal regular expression retrieval algorithm to connect the partial matches. Results show that GoldenEye can reassemble tens of millions of packets/sec and conduct stateful DPI operations on TCP streams at multi-ten Gbit/sec rates.

I. INTRODUCTION

Network packet analysis systems serve critical roles in enterprise network environments, particularly with regard to cyber security [1]–[3]. Based on the style of detection applied, network security applications can be broadly classified as either anomaly or misuse detection, where the latter is mostly deployed in the form of signature systems that employ DPI to scan network traffic for known malicious behaviors. The byte-wise packet inspection is a computational and I/O throughput-intensive task. Recently, as 40/100 GE network technologies have come into play, keeping up with full packet inspection has become quite difficult.

Researchers have been working to accelerate packet processing with different types of computing platforms. GPU packet processing is drawing increasing attention, due to GPU's ample memory bandwidth and native data-parallel execution mode. Since more than 90% [4] of internet traffic is TCP, and most intrusion detection systems (IDS) are triggered by flow events, DPI typically needs to track flow states, so that packets from the same flow connections are processed in the order the receiver would see them. However, despite the abundance of GPU work with network security applications [5]–[7], most of those applications assume that packets transferred into the GPU domain have been prepared in the form of flow-reassembled byte-streams. This is because the critical path operations, such as flow state management and stream reassembly, require threads to serialize when they attempt to

update the state of the same TCP flow. This serialization can cause large overheads for GPU implementations.

In order to amortize the performance cost of PCIe data transfer, GPU-accelerated network applications are normally designed in the form of batch-processing. Input packets are buffered, and only transferred to GPU after the batch reaches a certain fixed size, or a timeout occurs. This batch-processing mechanism also introduces challenges for stream-based packet processing by requiring detection of signatures across batch boundaries. One approach widely adopted by middlebox packet analysis systems is to buffer the entire flow payload in memory. Pattern matching is then performed on the fully reassembled data streams. This solution, however, has resource and scaling limitations, since there may be hundreds of thousands of concurrent flows, including very long-lived ones. Since GPU memory is normally limited to several gigabytes, leaving these incomplete streams un-delivered may rapidly exhaust the packet buffer. Although several distributed signature detection algorithms [8]–[10] have recently been proposed, those algorithms are mainly applicable to simple patterns, and may require large amounts of post-process to connect patterns that are split across different batches.

In this work, we present GoldenEye, a GPU-based traffic analysis system that provides TCP flow tracking and stream-based DPI. For packets within a batch, GoldenEye organizes them into a flow-reassembled payload, using GPU sorting. The reassembled data streams are then passed to a parallel regular expression (regex) engine to detect occurrences of pattern matches. For streams spanning batch boundaries, GoldenEye tracks stream sequences and detects matches that straddle batch boundaries, using a distributed regex-matching algorithm. GoldenEye addresses the case of inter-stream out-of-order delivery. Specifically, it detects and records any bytes that can potentially be part of a signature pattern match across the batch boundaries, from both forward (for prefix-matches) and backward (for suffix-matches) directions. These partial-match states, as well as records of stream connections, are searched, recovered, and combined with the data in subsequent batches for possible full pattern matches. Note that while our buffer-free design and distributed signature detection algorithm were designed with GPU memory constraints in mind, they can be implemented on any platform needing to improve memory efficiency with stream-based packet processing.

To summarize our key contributions: First, we present a

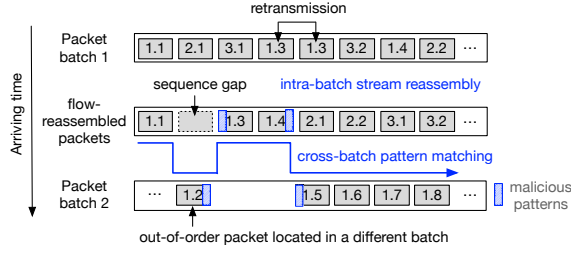


Fig. 1. As GPU loads and processes packets in batches, pattern matching needs to track TCP flows and reassemble the partial matches that straddle batch boundaries, reordering where necessary.

high-throughput, GPU-implemented flow processing system. This system unifies the low-level flow management and customized packet processing logics on a single GPU platform. Second, we build a new DPI engine capable of performing regex matching over TCP streams that straddle GPU batch boundaries. Finally, we demonstrate that the performance of GoldenEye is resistant to packet mis-ordering and requires less buffer space than conventional buffer-and-reassembly solutions.

The remainder of the paper is organized as follows. In Section II, we provide a comprehensive overview of the state-of-the-art. Section III presents the high-level system design. Section IV discusses our stream-based packet inspection mechanism. Section V evaluates the performance of GoldenEye, and Section VI concludes the paper.

II. BACKGROUND AND RELATED WORKS

A. Finite automaton and regex matching

Regex has been increasingly used to describe threat signatures for network security systems, due to its extraordinary expressive power. Current regex engines are mostly automaton-based, with matching operations equivalent to traversing the input characters through a finite automaton (FA) of the pattern sets. Typically, the design of an automaton makes a trade-off between automaton size and worst-case performance. Deterministic (DFA) and non-deterministic finite automata (NFA) are the two extreme cases. DFAs have predictable bandwidth and processing cost, requiring only one state traversal per character. However, the well-known state-explosion problem limits DFA's ability to represent large and complex sets of regexes. NFAs require much less storage space, but at the cost of worst-case performance issues, since each input character potentially triggers multiple state transitions in parallel. New studies have been seeking the middle ground between DFA and NFA, with notable examples such as hybrid-FAs [11], XFAs [12], and extended-FAs [13]. They reduce the chance of DFA state explosion by re-organizing the regexes or by introducing auxiliary variables. Nevertheless, all these solutions must operate on fully-reassembled data streams.

B. Challenges in stream-based packet processing

Stream-based packet measurements are important for network monitoring and traffic control [14], [15]. Applications

such as intrusion detection and surveillance are also required to scan packets on a per-flow basis to prevent fragmentation evasion type of attacks. Although studies [16], [17] have suggested that packet processing acceleration needs be implemented in conjunction with fast flow tracking and packet reassembly as a whole to improve the overall system performance, flow management operations are still largely conducted on CPUs, even with some well-known GPU-assisted IDS [18]. One notable GPU implementation of TCP stream reassembly is GASPP [19]. It hashes TCP 4-tuples (src ip, dst ip, src port, dst port), packet lengths, and sequence numbers, such that consecutive packets from the same stream can be paired in parallel. However, this work is unable to identify overlapped packets, and doesn't provide any solutions for hash collisions on stream assembly.

Cross-batch signature detection is another critical challenge for GPU implementations. Previous works have suggested to either buffer all packets that make up a message [20], or drop the out-of-sequence packets and force the host to retransmit them [19]. Since GPU does not have sufficient memory to hold the entire stream, it makes more sense for the GPU to store partial-matches and link them into a full result when applied. Shown in Fig. 1, pattern vectors may arrive in two orders: (1) the pattern-prefix arrives first, when packets are received in sequence (e.g., packet 1.4 → packet 1.5); (2) the pattern-suffix arrives first, when packets are received out-of-sequence (e.g., packet 1.3 → packet 1.2). To ensure detection with both cases, pattern matching engine must scan stream chunks from both a forward (checking for the pattern-prefix) and a backward (checking for the pattern-suffix) direction.

Connecting partial-matches across *in-sequence* packets is relatively simple, since the matching process of the next stream chunk will continue from the final state of its predecessor. The main challenges are to find the suffix vectors in out-of-order packets and reconstruct the full patterns when subsequent packets arrive to fill the sequence gap (e.g., a pattern that spans packets 1.2 and 1.3 in Figure 1). A previous work [21] proposes to buffer the first n bytes of the payload when receiving out-of-order packets, where n is the maximum size of the signatures. However, since regex can represent arbitrary long strings in the presence of unbounded repetitions, the needed buffer size is difficult to calculate. Johnson et al. [8] proposes an exhaustive DFA-based search of all potential traversals through any out-of-order stream chunks. These traversals would then be validated by comparing against the beginning and end states of stream predecessor and successor. While this scheme may be viable for small rulesets, it does not offer a predictive worst-case bound, since the choice of possible initial DFA-states of a packet could be prohibitively large. Chen et al. [9] proposes an AC-suffix-tree for explicit-string detection. This algorithm enumerates all suffix-patterns of an explicit-string and builds support-FA to detect them. Yu et al. [10] develops a similar scheme, named O³FA, for regex matching. However, because of its expressive power, a regex could potentially generate unbounded size of sub-strings, in the presence of counters, alternatives, and large character sets.

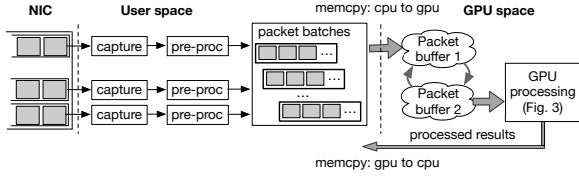


Fig. 2. GoldenEye packet analysis pipeline. The inter-device data transfer and GPU kernel functions are pipelined and implemented asynchronously to alleviate the cost of PCIe transfer.

This constrains O³FA's use with complex regex sets.

III. GOLDENEYE SYSTEM DESIGN

This section describes the framework of the GoldenEye packet analysis system. Cross-packet inspection is a dominant feature of GoldenEye. Packets are reassembled into flows, and patterns are matched across an arbitrary number of packets. To support cross-batch inspection, GoldenEye detects and records partial-matches that reach to the boundary of a stream and rescans them for possible full-patterns when subsequent packets arrive.

Figure 2 shows the collaboration of CPU and GPU in packet processing. The CPU is responsible for compiling the regex into automata, loading packets from the NIC, pre-processing them as needed, and transferring them into GPU space for analysis. Our design exploits multithreading to fetch packets from the NIC ring buffer into user space. Each capture thread has its own buffer to collect packets. Completely-filled batches are queued, waiting for a memory-copy thread to transfer them into GPU memory. In the GPU domain, a dual-buffer scheme is utilized to pipeline the CPU-to-GPU data movement, in order to optimize GPU kernel execution. When GPU finishes processing packets in the first buffer, it swaps the two buffers, and then continues to process packets in the new one. The old buffer is recycled to receive the next batch of packets from the CPU. This scheme reduces PCIe data transfer cost by overlapping part of the kernel function execution with the inter-device data movement.

The packet processing pipeline in the GPU domain is shown in Figure 3. GoldenEye contains five GPU packet processing units: (i) a packet filter; (ii) a stream processor; (iii) a regex engine; (iv) a functionally equivalent minimal regex (FEM-regex) reconstructor; and (v) a state buffer.

The **packet filter** decides which packets will be processed. We have implemented the well-known Berkeley Packet Filter [22], tailoring it for the GPU platform. By default, the packet filter separates TCP packets from other stateless protocols and forwards them to the stream processor for flow management.

The **stream processor** tracks TCP flow states, reassembles and normalizes packets into streams. It re-organizes the indices of the in-sequence packets and registers packet-wise overlaps, without physically moving the data.

The **pattern matching engine** processes packets against the FAs of regex sets, returns accepting states, and stores the metadata of partial-matches back into the state buffer.

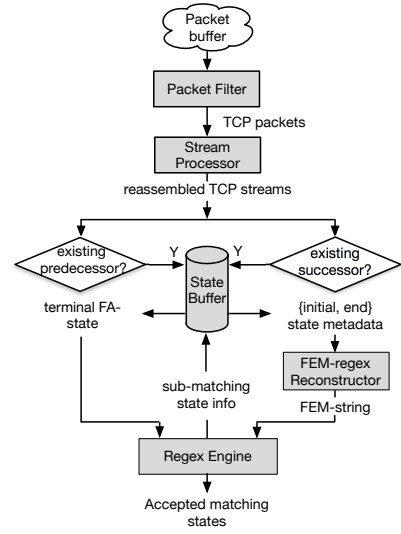


Fig. 3. Packet processing units in the GPU domain. Matched patterns are transferred back to CPU for further actions.

The **FEM-regex reconstructor** is an auxiliary component that assists the pattern matching. It reconstructs functionally equivalent strings of the partial-matches, from the metadata stored in the state buffer.

The **state buffer** is a hash-table that stores the states of incomplete TCP connections and their metadata of partial-matches. IP addresses and TCP ports are hashed into a 4-byte identifier, to designate the position of flow entries. Unlike the end-host TCP stack, GoldenEye tracks partially-reassembled stream fragments resulting from out-of-order packets, updating them every time the next in-order fragment arrives, or clearing them if the connection is closed or expired. The end state of regular FA traversal and the {initial, end} state metadata of suffix-FA traversal are stored. For a stream chunk whose predecessor has been previously received, its matching process starts from the last FA-state of its predecessor. Later, when the matching engine scans toward the end of a stream, and that stream has an out-of-order successor, GoldenEye calls the FEM-regex reconstructor to recover the string of previous matches, and concatenates it to the end of the current stream for possible full matches.

IV. GOLDENEYE STREAM-BASED PACKET INSPECTION MECHANISMS

Stream-based DPI in GPU domain faces two challenges: (1) per-stream synchronization in flow reassembly; and (2) cross-batch signature matching. In this section, we present our solutions to each.

A. GPU flow tracking and TCP stream reassembly

Our solution to flow tracking and stream reassembly is sorting the flow connections and the sequence numbers of packets. Studies on GPU primitive libraries [23] show that the Nvidia Tesla P100 GPU can perform a 32-bit prefix-scan of over 64 billion elements and a 64-bit radix-sort over 1.7 billion

elements per second. This remarkable throughput inspired our strategy for parallel packet reassembly.

Algorithm 1 Parallel TCP stream reassembly

input : packet headers p
output: sorted packets' keys & indices $\{k_{1s}, k_{2s}\}, v_s$

```

1: for each packet  $i$  do
2:    $v[i] \leftarrow p[i].index$ 
3:    $k_1[i] \leftarrow (p[i].saddr \ll 32 + p[i].daddr)$ 
4:    $k_2[i] \leftarrow (p[i].sport \ll 48 + p[i].dport \ll 32 + p[i].seq)$ 
5:  $(\{k_{1s}, k_{2s}\}, v_s) \leftarrow \text{MultiKeySort}(\{k_1, k_2\}, v)$ 

```

The flow management component of GoldenEye performs three operations: stream reassembly, flow tracking, and stream normalization. Stream reassembly sorts packets by their TCP 4-tuples and sequence numbers (Algorithm 1). Each thread computes a {key, value} set for a packet. It uses source and destination IPs, TCP ports, and TCP sequence number to build a reassembly key, and the packet's index in the batch to build a value. The program next performs a multi-key sort (since two 64-bit sorts are faster than one 128-bit sort) so that packets are grouped by their flow connections and aligned in the order of their TCP sequences. To avoid hash collisions in stream reassembly, the key is not compressed.

Algorithm 2 Parallel flow tracking

input : $\{k_{1s}, k_{2s}\}, v_s, p$
output: flow records f_r , flow labels of sorted packets f_i , $expt_next_seq$

```

1: for each packet  $i$  do
2:    $k_{port}[i] = k_{2s}[i] \gg 32$ 
3:    $seq[i] = k_{2s}[i] \& 0xffffffff$ 
4:    $enxt\_seq[i] = seq[i] + p[v_s[i]].len$ 
5:   if  $(\{k_{1s}[i], k_{port}[i]\} \neq \{k_{1s}[i+1], k_{port}[i+1]\})$  OR
6:    $(enxt\_seq[i] < seq[i+1])$  then  $\triangleright$  flow edge or sequence gap
7:      $f_i[i] = 1$ 
8:  $f_i \leftarrow \text{PrefixScan}(f_i)$ 
9: for each packet  $i$  do
10:  if  $f_i[i] \neq f_i[i-1]$  then
11:     $f_r[f_i[i]].init\_seq = seq[i]$ 
12:     $f_r[f_i[i-1]].last\_seq = seq[i-1]$ 
13: for each flow-reassembled stream fragment  $k$  do
14:  Update_Connec_Table( $f_r[k]$ )

```

The flow tracking function identifies the TCP flow that a packet belongs to and updates the connection status (Algorithm 2). For each packet, a thread compares its TCP 4-tuple and sequence range to its neighbor's, and tags the packet if it is the last one in the flow. A prefix-sum operation is then applied to the resulting tagged array, converting it into flow indices of the sorted packets. Next, threads scan through the flow-labeled packets, aggregating per-flow statistics, such as connection status and sequence ranges. Lastly, each thread takes one stream record, queries the state buffer, and updates the states of corresponding flow entries. Here, we keep the collided stream identifiers in the form of linked list. In addition, we set a keep-alive timer and periodically remove the expired connection entries to avoid stack overflow.

The stream normalization function rescans the sorted packets and eliminates retransmission. Given the policy of the end-host operation systems [24], GoldenEye supports flexible

Algorithm 3 Parallel TCP stream normalization

input : $v_s, f_i, p, expt_next_seq$
output: next_packet and a packet-wise overlap_byte

```

1: for each packet  $i$  do
2:   if  $f_i[i] \neq f_i[i+1]$  then
3:     next_packet[i] = END
4:   else if  $(f_i[i] == f_i[i-1])$  AND  $(enxt\_seq[i] \leq enx\_seq[i-1])$  then
5:     next_packet[i] = REXMIT  $\triangleright$  packet  $i$  itself is a REXMIT
6:   else
7:     next =  $i + 1$ 
8:     while  $(f_i[i] == f_i[next])$  AND  $(enxt\_seq[i] \geq enx\_seq[next])$ 
9:       do next = next + 1
10:    if  $f_i[i] == f_i[next]$  then
11:      next_packet[i] = next
12:      over_byte[next] =  $enxt\_seq[i] - p[v_s[next]].seq$ 

```

normalization mechanisms (e.g., overwriting contents with its retransmitted payload or not), which can be configured on a per-flow basis. Algorithm 3 shows the implementation of a *keep-original* normalizer. For each sorted packet, a thread first compares the sequence range against its predecessor, checking for retransmission. After that, it scans subsequent packets and updates the next_packet array with the index of the next in-order and non-retransmitted packet (or REXMIT to indicate the packet is a retransmission and END to indicate it marks the end of a stream). In case of partial retransmission, the bytes of overlaps are saved in an over_byte array.

B. GPU stream-based signature matching

Signature patterns may span multiple packets of a stream within or across batch boundaries. DPI systems need to detect and reconstruct partial matches on a per-flow basis. Here, we present two algorithms: (1) a general GPU implementation of regex matching for intra-batch, flow-reassembled packets; (2) a novel solution to detect regex patterns across GPU batches.

1) Intra-batch signature matching: As a trade-off between memory consumption and pattern-match performance, we adopt the hybrid-FA (HFA) proposed by Bechhi et al. [11]. HFA starts from a DFA head and keeps any subset whose expansion would cause state explosion in NFA form. The NFA-parts of HFA remains inactive until a search reaches a *border-state*. Fig. 4 is an example of HFA adapted from [11] representing a regex set: 1) $ab.\{3\}cd$, 2) $cefc$, 3) efb . The tail after DFA-state 2 is kept in NFA form, to avoid state blowup. This hybrid data structure of HFA combines benefits of NFA and DFA, resulting in a modest state transition table size and a similar average case memory bandwidth compared to full DFA. It thus allows GPU to hold the FA table in memory and match multiple regexes in one pass.

GoldenEye executes pattern matching in a per-packet parallel fashion. When the processing reaches the end of a packet, it jumps to the position of the next in-order packet, and continues until an accepting or a failure state (e.g., states of gray color in Fig. 4) is reached. GoldenEye returns the identities of matched patterns as well as the associated packets, and stores the last matching state of a stream in the state buffer.

Regex matching is known to be a computationally expensive

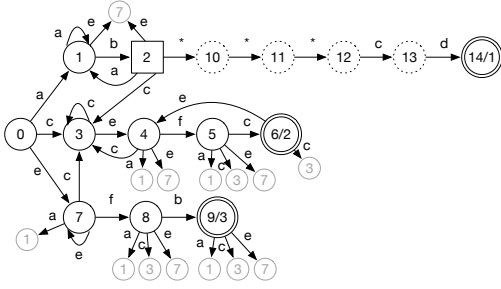


Fig. 4. An example of the hybrid-FA representing the regex set: 1) $ab.\{3\}cd$, 2) $cefc$, 3) efb . The DFA-state 2 is a border state, whose tail $\{3\}cd$ is retained in an NFA form to avoid state-explosion.

task. To reduce the number of packets needing to be matched against, we used a string matching engine as a first-level filter. The filter employs a parallel Aho-Corasick algorithm [7], searching explicit-strings of a regex. It may look for multiple strings contained in a regex. The states of filtered packets are synchronized on a per-flow basis. If all sub-strings of a regex are found, associated packets of the stream will be sent to regex engine for further processing.

2) *Cross-batch signature matching*: Cross-batch detection searches signature patterns that straddle batch boundaries. Regex segments may arrive in-sequence or out-of-order. In the first case, the matching process of subsequent stream fragments will continue from the last FA-states of the previous streams. In the second case, for any out-of-order stream, GoldenEye detects the pattern-suffix, then stores and rescans it for possible full-patterns upon arrival of its predecessors. Specifically, we introduce an NFA-oriented regex-suffix matching algorithm and a FEM-regex reconstructor in support of cross-batch, out-of-order signature matching.

The regex-suffix detection engine employs two data structures—an NFA representation of the regex-suffix set and a table of state transition vectors. The regex-suffix is created by replacing the first regex symbol with an anchor expression, indicating the match shall ignore full patterns and start from the beginning of a stream. The table of state transition vectors is built by sorting the transition paths of NFA by their triggering symbols. An example is shown in Fig. 5b. Multiple transition paths can be triggered from one symbol. For each out-of-order stream, the regex-suffix engine loops through all possible transition paths given the first symbol of a stream, traverses subsequent payload, and terminates when reaching an accepting state, or when no valid transition states exist. Since the performance of suffix-NFA traversals is proportional to the number of possible initial states, we break the regex of the form $A.B$ into two sub-patterns, to avoid the arbitrary number of transition paths caused by the wildcard symbol. The sub-patterns of A and B will be searched individually and combined on a per-flow basis in the end. Currently, our workload peaks at around 2000-9000 NFA-states and several hundred state transition vectors per triggering symbol. This compares to tens of millions of equivalent DFA-states

Some high-performance regex engines avoid NFA, since it

has exponential computational complexity in the worst case. Nevertheless, because the intra-batch packet reassembly fills up most of the sequence holes created by out-of-order packets, the NFA-implementation of GoldenEye only needs to perform on the small number of streams that arrive without an existing predecessor. As a result, the throughput of regex matching with GoldenEye is still dominated by the speed of sequential intra-batch signature matching.

Since NFA contains backtracking, the number of traversals that could be reconstructed from a pair of beginning and end states potentially becomes prohibitively large. To solve the concerns with suffix reconstruction, we introduce the concept of *depth*. Because the NFA traversal is essentially a depth-first search of the FA tree, we envision each regex as a branch of the tree, and represent the layer of a state in that tree using a depth variable (Figure 5a). The regex-suffix engine returns a 12-byte metadata of the form $(\{s_1, s_2\}, \{c_1, c_2\}, d)$, denoting $\{\text{initial}, \text{end}\}$ states, the pair of triggering symbols, and the transited depth. Later, when a subsequent packet arrives to fill the sequence gap, the FEM-regex reconstructor links $\{s_1, s_2\}$ back to their original regex and extracts the suffix segment by looking at the depths of the states in the tree. $\{c_1, c_2\}$ and d are auxiliary parameters used to narrow the choice when alternative traversal paths exist.

Algorithm 4 NFA-based pattern-suffix matching

input : $\text{transit_vector}[c, \{s_{1k}, s_{2k}\}]$, state transition table NFA, flow-reassembled payload input
output: the longest suffix-regex sm_0

```

1: for each  $\{s_{1k}, s_{2k}\}$  in  $\text{transit\_vector}[\text{input}[0]]$  do
2:    $\text{sm}[k].\text{init\_state} = s_{1k}$ 
3:    $\text{sm}[k].\text{depth} = 1$ 
4:    $\text{sm}[k].\text{init\_cha} = \text{input}[0]$ 
5:    $\text{Insert}(Q_1, \{s_{1k}, 1\})$ 
6:    $i \leftarrow 0$ 
7:   while  $!Q_1.\text{empty}$  do
8:      $c \leftarrow \text{input}[+ + i]$ 
9:      $Q_2 \leftarrow Q_1$ 
10:     $\text{Empty}(Q_1)$   $\triangleright$  empty  $Q_1$  to receive the next transition states
11:    while  $!Q_2.\text{empty}$  do
12:       $(s, d) \leftarrow \text{Pop}(Q_2)$ 
13:      if  $(\text{NFA}[s] == \text{ACPT}) \ \&\& \ (\text{sm}[k].\text{depth} < d)$  then
14:         $\text{sm}[k].\text{last\_state} = s$ 
15:         $\text{sm}[k].\text{depth} = d$ 
16:         $\text{sm}[k].\text{last\_cha} = c$ 
17:      for each non-zero transition state  $s_j$  in  $\text{NFA}[s][c]$  do
18:         $d_j = d + (s_j \neq s)$   $\triangleright$  self-transition does not change the
        depth of search
19:         $\text{Insert}(Q_1, \{s_j, d_j\})$ 
20: return  $\text{sm}_0 \leftarrow \text{sm}[k]$  w/ the  $\text{MAX}(\text{sm}[k].\text{depth})$ 
```

Algorithm 4 shows the execution of regex-suffix matching. Parallelism is exploited not only on a per-stream basis but also on a per-initial-state basis in the stream (i.e., line 1 in Algorithm 4). Specifically, each CUDA thread-block processes one out-of-order stream fragment. Threads within a thread-block loop through the initial states of the triggering symbol, traversing the subsequent payload individually, and synchronizing at the end of their search. In the case where threads of a block return multiple valid traversals, the program reports the longest one in depth, since it should include others.

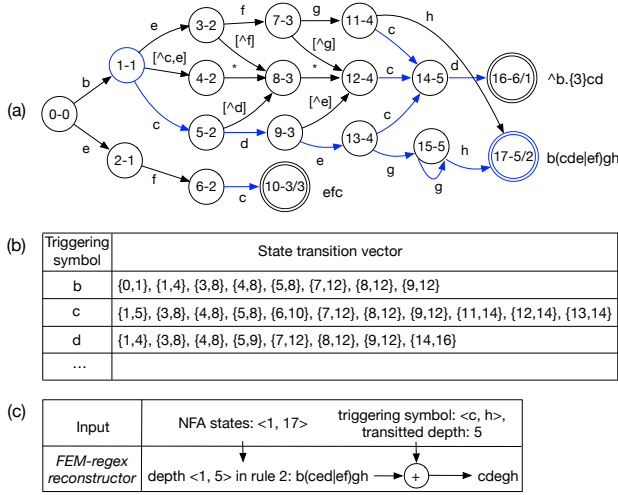


Fig. 5. a) an NFA representing 1) $b.\{3\}cd$, 2) $b(cde|ef)g+h$, and 3) efc ; $x-y$ denotes NFA state x , with a depth variable y ; the blue lines denote the suffix traversals of $cdeggh$. b) list of transition vectors grouped by triggering symbols. c) retrieval of FEM-string from the metadata of the matches.

To illustrate this algorithm, consider matching a stream $cdeggh$ against any subset of 1) $b.\{3\}cd$, 2) $b(cde|ef)g+h$, and 3) dfc . Shown in Fig. 5, the suffix-regex engine finds five valid traversals ($cdeggh$ for state $1 \rightarrow 17$, c for state $6 \rightarrow 10$, cd for state $11 \rightarrow 16$, state $12 \rightarrow 16$, and state $13 \rightarrow 16$). The longest one is stored in the buffer. Next, given the state vector $\{1, 17\}$, the regex reconstructor queries their depths—1 and 5, and links them back to the regex $b(cde|ef)g+h$. Since depths 1→5 can represent either $cdeggh$ or $efgh$ in the regex, we then use the pair of triggering symbols $\{c, h\}$ and the transited depth 5 to narrow down and obtain the final answer— $cdeggh$. We have named the obtained expression as the functionally-equivalent-minimal-string since it represents the shortest traversal in depth between the pair of beginning and end states (e.g., the match of $g+$ is registered as the g).

V. EVALUATION

This section evaluates the following aspects of the GoldenEye: (i) performance of two component applications—packet reassembly and regex-matching; (ii) savings in buffer size compared to buffer-and-reassembly schemes; and (iii) end-to-end DPI performance in streaming mode.

A. Experiment setup and datasets

1) *System setup*: We evaluate GoldenEye applications and conduct the comparison on a machine with dual Intel E5-2650 v4 CPUs (12 cores per socket), 128 GB of RAM, and an NVIDIA K40 GPU. Applications throughput are gauged by pre-loading an offline traffic trace into RAM space, then transferring it into GPU space in batches, at the desired rate.

2) *Traffic trace*: We measured the performance of GoldenEye with real network packet traces obtained from two sources: i) an intrusion detection dataset created by Canadian Institute for Cybersecurity (CICIDS_2012) [4]; ii) science data flows mirrored from the border router at Fermi National

TABLE I
PERFORMANCE OF STREAM REASSEMBLY UNDER REAL TRAFFIC

Dataset	Fermilab	CICIDS_2012
Software		
GoldenEye w/ PCIe transfer	623.299 Gbit/s	335.727 Gbit/s
GoldenEye w/ PCIe transfer	552.229 Gbit/s	232.487 Gbit/s
Libnids (12 CPU-cores)	186.649 Gbit/s	31.102 Gbit/s

Accelerator Laboratory (Fermilab). The CICIDS_2012 dataset contains both normal and malicious activities, including Brute Force SSH, HTTP DoS, DDos, and inside infiltration. The size of the dataset is 17.1 GB. It has 44.34 million TCP packets (average packet-size 386 bytes) and 1.10 million TCP connections. The Fermilab dataset has a much higher percentage of long-lived, large-volume TCP flows that are characteristic of science data traffic. The size of the dataset is 9.8 GB. It records 8.7 million TCP packets (average packet-size 1118 bytes) and 54.78k TCP connections. These two datasets allow us to evaluate the performance of GoldenEye over a wide spectrum of packet and flow characteristics.

3) *Regex datasets*: We used two regex sets obtained from the widely-used Snort IDS (snapshot from Snort 2.9.7.2). The regex sets contains 104 and 192 regular expressions of *spyware* and *malware* signatures, respectively. We eliminated the *lookahead* and *lookbehind* expressions (approximately 1%-2%) from both datasets, since the current regex-to-FA generator of GoldenEye has not implemented those features.

B. Performance of flow reassembly

In this experiment, we measured the overall throughput of flow tracking, TCP stream reassembly and normalization, and compared it to an equivalent hash-based TCP reassembly library from Libnids [25]. We chose Libnids as our comparison, rather than the packet reassembly module of GASPP [19], because Libnids is open-source and has similar features to GoldenEye. We implemented the Libnids code in a lock-free multithreading mode to gauge its parallel performance. Table I shows GoldenEye flow operations running 3.3-10.4 times faster than the 12-threading Libnids implementation. When we include the cost of PCIe data transfer, GoldenEye is still 3.0-7.1 times faster than equivalent 12-threading Libnids operations.

We next evaluated the performance of GoldenEye over the number of concurrent connections. We simulated the flow connections with 7 synthesized packet traces, each containing 5K, 25K, 50K, 75K, 100K, 125K, and 150K concurrent connections. Results are plotted in Fig. 6. We note the throughput of TCP reassembly drops as the number of concurrent connections increases for both Libnids and GoldenEye implementations. Nevertheless, GoldenEye scales much better than the traditional hash-based implementation on CPUs. As Fig. 6 shows, while the throughput of Libnids drops to 12.29% of its peak when processing 150K concurrent connections, the throughput of GoldenEye only drops to 79.75% of its peak.

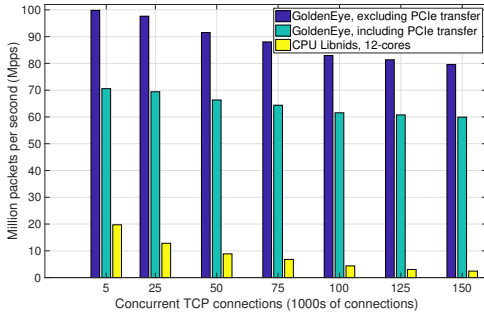


Fig. 6. The throughput of GoldenEye flow tracking and TCP reassembly over varied number of concurrent TCP connections.

C. Buffer size saving

One advantage of GoldenEye is the ability to match signatures across packets or data batches, without requiring the system to retain the old payload data. Instead, GoldenEye keeps a *stream state*, consisting of partial-matching metadata, flow connection status, and the range of sequence numbers, for each incomplete or out-of-order TCP stream fragment. The partial-matching metadata has a fixed memory footprint over patterns of different length (1-byte of the last HFA-state and 12-byte of suffix-matching states). The total number of stream states in processing depends on the number of incomplete and out-of-order streams that happen to span batch boundaries.

To evaluate the impact of packet re-ordering on the memory buffer size, we generated 8 synthetic traffic traces, by replaying the Fermilab and CICIDS_2012 traffic traces with different reordering patterns. The replay process was guided by two parameters: 1) the out-of-order rate p , and 2) the time of delay t . Collectively, they indicated each packet had a probability p to be delayed by t ms (assuming a 10GE connection). In our experiments, we chose p to be 0.01 and 0.05, since previous work reported that around 2%-5% of Internet packets are affected by sequence reordering [20].

Table II compares the *peak* buffer size required by GoldenEye to a conventional buffer-and-reassembly scheme. Because the stream reassembly logic of Libnids is not mature enough to handle out-of-order TCP fragments [17], we built a customized flow reassembly function derived from the mechanism presented in [26]. We customized the reassembly function with an 8 KB starting buffer-size for each flow connection, archived partially-reassembled packets using linked-list, and flushed flow entries when their payloads were fully received.

Overall, GoldenEye required 2698x–6964x less buffer to hold the stream states than what the buffer-and-reassembly scheme needed. When packets were received out-of-order, we observed that: (i) the packet reordering only increased buffer memory slightly with GoldenEye, since most sequence gaps got filled during intra-batch packet reassembly; and (ii) the *extra* buffer size needed by cross-batch detection was proportional to p and t , but independent of the number of concurrent connections. The trends showed no major differences between the Fermilab and CICIDS datasets.

TABLE II
MEMORY FOOTPRINT ANALYSIS

dataset	mis-ordered rate p delay t (ms)	buffer & reassembly scheme	GoldenEye <i>stream state</i> size
Fermilab	$p=0, t=0$	307.81 MB	44.20 KB
	$p=0.01, t=1$	extra 8.19 KB	extra 0.480 KB
	$p=0.05, t=1$	extra 67.68 KB	extra 4.08 KB
	$p=0.01, t=10$	extra 167.28 KB	extra 5.93 KB
	$p=0.05, t=10$	extra 775.08 KB	extra 30.88 KB
CICIDS	$p=0, t=0$	186.32 MB	48.05 KB
	$p=0.01, t=1$	extra 78.21 KB	extra 0.24 KB
	$p=0.05, t=1$	extra 305.77 KB	extra 3.74 KB
	$p=0.01, t=10$	extra 573.23 KB	extra 5.08 KB
	$p=0.05, t=10$	extra 2.76 MB	extra 22.02 KB

D. Performance of stream-based pattern matching

We measured the pattern-matching speed of GoldenEye, and compared it to a high-performance pattern-matching library—Hyperscan [27]. Hyperscan is known as the fastest CPU regex library, and can match patterns across streams of data. However, Hyperscan does not track the order of TCP sequences. Consequently, its streaming mode is equivalent to the sequential matching of GoldenEye, but only by assuming that packets would be received and processed in-sequence.

Figure 7 shows GoldenEye throughputs over the two regex sets and two traffic traces. The matching rate of *malware* is 5 matches/megabyte on CICIDS_2012 dataset, and below 0.5 match/megabyte on the other three combinations. In sequential matching mode, GoldenEye achieved comparable performance to Hyperscan running on 12 CPU-cores. GoldenEye was actually slightly faster than Hyperscan on traffic traces composed of large and relatively equal-sized packets (i.e., the Fermilab dataset). This is because Hyperscan employs a compiler in its pre-processing, arranging packets into stream of payloads, so that memory access will be performed in a coalesced fashion for pattern matching. This pre-processing overhead, however, was not considered in our comparisons. By contrast, GoldenEye uses a *next_packet* array to guide cross-packet traverses, which does incur latency and load-imbalance when the search is implemented in a per-packet parallelism fashion. We designed GoldenEye this way to avoid the overhead of data arrangement, considering that the system will ultimately be used to process packets captured on high-speed networks in real-time.

In the experiment of Figure 7, we noticed there was still a considerable number of packets going through the regex-suffix search, even though we did not deliberately reorder the packet traces. These non-contiguous streams are mainly caused by either a ‘cold start’ (i.e., traffic captured from already-established connections) or packet loss during the capture. Suffix-matching operations on those packets added about 9.5-13.1% computational overhead, while generating ‘cold’ connection and matching states that would never be re-retrieved. This observation suggests us that we need to develop a cognitive filter for *fake* out-of-order streams.

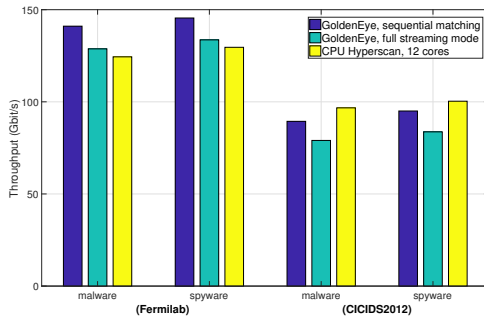


Fig. 7. Throughputs of GoldenEye stream-based regex matching. The benchmarks were obtained without counting PCIe data transfer.

TABLE III
PERFORMANCE OF CONSOLIDATED DPI APPLICATION

Regex \ Dataset	Dataset	
	Fermilab	CICIDS_2012
malware	105.55 Gbit/s	62.13 Gbit/s
spyware	108.69 Gbit/s	64.95 Gbit/s

E. Performance of consolidated stream-based DPI application

Finally, we measured the overall throughput of stream-based DPI, including flow management, cross-batch state connection, and regex matching. The benchmarks in Table III show that raw DPI with GoldenEye reaches up to 100 Gbit/s when fed long-lived TCP flows with large packets, while dropping to around 65 Gbit/s with small packet, short-lived streams. Note that the benchmarks in Table III excluded the inter-device data transfer time. CPU↔GPU transfer time largely depends on PCIe bandwidth, which is a more limiting factor than GPU computing capacity. However, NVLink technology [28] has recently been reported to achieve a 2.5-3x speedup over PCIe Gen3, with about 275 Gbit/s throughput in practice. We plan to evaluate the end-to-end, on-line performance of GoldenEye on systems equipped with NVLink technology in the future.

VI. CONCLUSION

In this paper, we have presented a fast, stream-based packet inspection framework running on GPUs. The novelty of our design is in tracking and reassembling TCP streams with GPUs, and matching patterns across any number of packets that span any number of GPU batches, all without retaining old traffic data. Experiments show that the TCP reassembly of GoldenEye requires significantly less memory for packet buffering, even when the packets are severely mis-ordered. GoldenEye runs considerably faster than equivalent CPU implementations, with speedups ranging 40x–125x. For the stream-based regex matching, GoldenEye edges Intel Hyperscan running on 11-13.5 cores, and offers the capability to return matches across packets that arrive out-of-order.

The main goals of this paper have been to verify the effectiveness and feasibility of implementing stream-based packet processing on GPUs, and introduction of GoldenEye’s framework design. So far, we have only implemented a straightforward regex engine for GoldenEye. We anticipate GoldenEye’s

DPI performance will improve significantly by incorporating advanced regex compilers and optimization chains.

REFERENCES

- [1] C. Sanders, *Practical packet analysis: Using Wireshark to solve real-world network problems*. No Starch Press, 2017.
- [2] T. Xing *et al.*, “Snortflow: A openflow-based intrusion prevention system in cloud environment,” in *Research and Educational Experiment Workshop (GREE), 2013 Second GENI on*, pp. 89–92, 2013.
- [3] J. Sherry *et al.*, “Blindbox: Deep packet inspection over encrypted traffic,” in *ACM SIGCOMM Computer Communication Review* 45.4, vol. 45, pp. 213–226, 2015.
- [4] A. Shiravi *et al.*, “Toward developing a systematic approach to generate benchmark datasets for intrusion detection,” *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.
- [5] W. Wu *et al.*, “G-netmon: a gpu-accelerated network performance monitoring system,” in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pp. 76–79, IEEE, 2011.
- [6] M. A. Jamshed *et al.*, “Kargus: a highly-scalable software-based intrusion detection system,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 317–328, 2012.
- [7] C.-H. Lin *et al.*, “Accelerating string matching using multi-threaded algorithm on gpu,” in *Global Telecommunications Conference (GLOBE-COM 2010)*, pp. 1–5, IEEE, 2010.
- [8] T. Johnson *et al.*, “Monitoring regular expressions on out-of-order streams,” in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 1315–1319, IEEE, 2007.
- [9] X. Chen *et al.*, “Ac-suffix-tree: Buffer free string matching on out-of-sequence packets,” in *Architectures for Networking and Communications Systems, Seventh ACM/IEEE Symposium on*, pp. 36–44, IEEE, 2011.
- [10] X. Yu *et al.*, “O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection,” in *Symposium on Architectures for Networking and Communications Systems*, 2016.
- [11] M. Becchi *et al.*, “A hybrid finite automaton for practical deep packet inspection,” in *Proceedings of the 2007 ACM CoNEXT conference*.
- [12] R. Smith *et al.*, “Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,” in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 207–218, ACM, 2008.
- [13] M. Becchi and P. Crowley, “Extending finite automata to efficiently match perl-compatible regular expressions,” in *Proceedings of the 2008 ACM CoNEXT Conference*, 2008.
- [14] Z. Liu *et al.*, “One sketch to rule them all: Rethinking network flow monitoring with univmon,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ACM, 2016.
- [15] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in *NSDI*, vol. 13, pp. 29–42, 2013.
- [16] P.-C. Lin *et al.*, “Re-examining the performance bottleneck in a nids with detailed profiling,” *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 768–780, 2013.
- [17] M. A. Jamshed *et al.*, “mos: A reusable networking stack for flow monitoring middleboxes,” in *NSDI*, pp. 113–129, 2017.
- [18] “Suricata open source ids/ips/nsm engine. <https://suricata-ids.org>.”
- [19] G. Vasiliadis *et al.*, “Gaspp: A gpu-accelerated stateful packet processing framework,” in *USENIX Annual Technical Conference*, 2014.
- [20] S. Dharmapurikar *et al.*, “Robust tcp stream reassembly in the presence of adversaries,” in *USENIX Security Symposium*, pp. 65–80, 2005.
- [21] J. Ma *et al.*, “Detecting distributed signature-based intrusion: The case of multi-path routing attacks,” in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 558–566, 2015.
- [22] S. McCanne *et al.*, “The bsd packet filter: A new architecture for user-level packet capture,” in *USENIX winter*, vol. 93, 1993.
- [23] “Cuda unbound (cub). <https://nvlabs.github.io/cub/index.html>.”
- [24] J. Novak, “Target-based tcp stream reassembly,” Sourcefire Inc., 2007.
- [25] R. Wojtczuk, “Libnids. <http://libnids.sourceforge.net>.”
- [26] A. V. Rana *et al.*, “Queue engine for reassembling and reordering data packets in a network,” US Patent 6,781,992, 2004.
- [27] “Hyperscan: a high-performance multiple regex matching library. <https://01.org/hyperscan>.”
- [28] “Nvidia nvlink. <https://developer.nvidia.com/openpower>.”