Title

Real-Time GPU-based Timing Channel Detection using Entropy

Permalink

https://escholarship.org/uc/item/1vf9f3t6

Authors

Gegan, Ross
Ahuja, Vishal
Owens, John D
et al.

Publication Date

2016-07-28

Data Availability

The data associated with this publication are available at: http://www.caida.org/home/

Real-Time GPU-based Timing Channel Detection using Entropy

Ross K. Gegan, Vishal Ahuja, John D. Owens, and Dipak Ghosal
University of California, Davis
{rkgegan, vahuja, jowens, dghosal}@ucdavis.edu

***Abstract*—As line rates continue to grow, network security applications such as covert timing channel (CTC) detection must utilize new techniques for processing network flows in order to protect critical enterprise networks. GPU-based packet processing provides one means of scaling the detection of CTCs and other anomalies in network flows. In this paper, we implement a GPU-based detection tool, capable of detecting model-based covert timing channels (MBCTCs). The GPU's ability to process a large number of packets in parallel enables more complex detection tests, such as the corrected conditional entropy (CCE) test—a modified version of the conditional entropy measurement, which has a variety of applications outside of covert channel detection. In our experiments, we evaluate the CCE test's true and false positive detection rates, as well as the time required to perform the test on the GPU. Our results demonstrate that GPU packet processing can be applied successfully to perform real-time CTC detection at near 10 Gbps with high accuracy.**

I. Introduction

Network security applications must utilize new techniques in order to process packets at network line rates of 10 Gbps, and eventually 40 and 100 Gbps. For scaling these tasks, multi-core CPUs and other parallel systems such as Massively Parallel Processing Array (MPPA) or Field Programmable Gate Arrays (FPGA) architectures can be applied. However, these approaches each have their own limitations in terms of programmability, performance, and flexibility. Containing thousands of cores, Graphics Processing Units (GPUs) possess much greater thread-level parallelism and memory bandwidth compared to CPUs [1]. Therefore, using the GPU presents another possible way to scale real-time software-based packet processing, and several papers have already demonstrated its effectiveness for tasks such as software routing [2] and firewall packet classification [3]. Covert timing channel (CTC) detection is one possible network security application which can benefit from GPU packet processing. CTCs exploit the timing of the inter-packet delays (IPDs) in authorized network flows to embed hidden data transfers. Detecting CTCs in real-time requires performing statistical tests on a large number of incoming flows and comparing the results with those expected for legitimate traffic. Since the test scores for each flow are independent, a GPU can process many flows in parallel. Furthermore, the GPU can also perform the tests on individual flows in parallel, allowing real-time usage of more complex and effective detection tests such as the corrected conditional entropy of the IPD sequence, which could not be included in our previous CTC detection experiments [4].

Although previous work shows that the first-order entropy alone can be somewhat effective for detection, the false positive rate is still high [4]. A more effective detection method requires calculating the corrected conditional entropy (CCE), which is the conditional entropy calculation plus a corrective term accounting for the number of unique subsequences in the sample. The CCE test has proven effective for detecting a variety of CTCs with minimal false positives [5]. Outside of CTC detection, the CCE test has a variety of applications, particularly in medical imaging applications, such as analyzing heart rate variability data and other biological processes [6]. However, calculating the CCE score for a large sequence is an expensive calculation computationally, requiring the construction of a tree for each individual flow [5]. For higher traffic rates with a large number of flows arriving each second, we need to calculate the CCE score for each flow more efficiently. In order to perform the calculation quickly, we propose that packets be processed on the GPU. The corrected conditional entropy formula and our GPU algorithm will be explained in detail in section 4.

To evaluate the large number of incoming flows without packet loss, we implement a detection tool that performs the CCE test using a NVIDIA Tesla K20C GPU. Our tool sniffs incoming traffic and gathers packet data into large batches, which are then tested on the GPU. The GPU will use the CCE calculation to report which flows are likely to contain covert channels. In our work, we consider a well-known CTC variety known as model-based covert timing channels (MBCTCs), which avoid detection by fitting the CTC's packet timings to a statistical model based on natural traffic [7]. By testing our tool against a traffic sample injected with MBCTCs, we confirm the CCE test's effectiveness as a classifier established in previous results [5]. We also evaluated the maximum performance of our tool, establishing that it can handle close to a full 10 Gbps line rate assuming average sized packets.

Implementing real-time CTC detection at high data rates requires capturing and storing large amounts of packet data by flow, and then performing the detection test in a small amount of time (less than 67.2 ns per packet for minimum-sized packets) to avoid packet loss. The limited time available to process each flow makes it especially difficult to perform more complex calculations such as the CCE test. While GPU packet processing offers good potential performance, mapping the problem is particularly difficult because the CCE test requires us to construct and process a k-ary tree for every flow, every few seconds. Since tree construction is a difficult task for GPUs [8], [9], we need an alternative method of calculating CCE scores. Since the CCE calculation uses k-ary trees, each tree can be interpreted as an array. Therefore, our solution involves transforming the tree structures into arrays, a form well-suited for GPU processing.

This work presents multiple new contributions. The most significant are as follows:

- Our detection tool marks a significant improvement over previous CTC detection work. In our experiments, we manage to detect nearly 100% of our sample's CTCs.
- In addition to confirming the CCE test's effectiveness, we achieve higher rates compared to previous real-time detection experiments [4]. We achieve close to 10 Gbps using medium-sized packets.
- By performing our tree transformation and completing the calculation using arrays, we compute the CCE scores in less than 1 ms per flow, an order of magnitude faster than previous results [5].
- Our work also demonstrates how GPU packet processing can efficiently calculate complex individual flow statistics using packet batches.

The rest of the paper will provide further background on the CCE test and CTC detection, as well as our experimental results and discussion. In Section II-B we discuss our motivations for this project. In Section IV we discuss our tool design in-depth, the CCE algorithm, and the experimental setup. In Section V we discuss the experimental results. Section VI describes some related work on GPU-based packet processing and CTC detection. Finally, in Section VII we give the conclusions and describe the potential for future work.

II. Timing Channels and Detection

A covert channel allows unauthorized data transfers through authorized channels. This can allow harmful exploits such as controlling botnets or leaking private data [4]. In this work, we will focus on detecting network covert timing channels, which function by encoding data inside the inter-packet delays (IPDs) of a network flow. A very basic channel type would be Cabuk's IP covert timing channel (IPCTC) [10], a simple on/off channel, where a packet transmission during a set time interval will be interpreted by the receiver as a 1, while no transmission during that interval will be interpreted as a 0 [10]. This encoding scheme, although functional, creates traffic where the shape and regularity differs greatly from the original, overt traffic, making detection simple [5]. More advanced timing channels attempt to mimic real traffic statistics to bypass detection. For example, Time-Replay Covert Timing Channels (TRCTC) uses two legitimate IPD sets, producing a 0-bit or a 1-bit by replaying according to one of the two sets. This encoding scheme makes the new traffic appear similar to overt traffic, complicating detection [5].

In this work, we focus on Model-Based Covert Timing Channels (MBCTC). MBCTCs evade detection by mimicking legitimate traffic's shape and regularity. To achieve this, the channel first analyzes outgoing flows to determine an appropriate statistical model, then encodes the message inside the flow's IPDs using that model's inverse distribution function. The channel can then be decoded by the receiver using the cumulative distribution function [5]. Figure 1 illustrates the process. While this encoding prevents detection by common detection methods such as Kullback-Liebler Divergence and the Kolmogorov-Smirnov Test, CCE detects MBCTCs with near-perfect accuracy [5].

A. Detection Methods

Ideally, we would want to eliminate all possible CTCs. Existing methods such as the network pump [11] and fuzzy time [12] introduce noise that alters packet timings, reducing a covert channel's reliability and capacity. However, applying these techniques to all incoming flows will harm legitimate traffic performance as well. Therefore, we want to first detect likely CTC flows, then disrupt them selectively. Basic channels—such as IPCTCs—are relatively simple to detect, but other channels can closely resemble the unaltered traffic. CTCs can be detected by measuring the shape and regularity of network traffic and comparing it to the expected legitimate statistics [10].

CTC detection tests can be grouped into regularity tests and shape tests. Shape tests represent first-order traffic statistics, including the Shannon entropy or Kolmogorov-Smirnov score, which measures the greatest difference between two IPD distributions. Regularity tests represent higher order statistics, such as conditional entropy [5]. Different detection tests will be effective depending on the channel type [4]. IPCTCs alter both traffic shape and regularity, allowing detection by a variety of tests. TRCTCs closely resemble the natural traffic's shape, but since the packet timings do not correlate naturally, the regularity scores will be significantly different than normal traffic. MBCTCs closely resemble both the regularity and shape of unaltered traffic, making most detection tests ineffective [4]. However, the CCE test, which is a modified conditional entropy measurement, effectively identifies MBCTCs [5]. Our work focuses on efficiently implementing the CCE test for real-time detection.

B. Motivation

Improved Real-Time Detection: Although many papers have been written on CTC detection, few consider real-time detection in streaming data. Our previous real-time CTC detection work used an MPPA architecture [4] (Tilera TilePro64 NIC). The incoming flows are equally assigned to the different cores. Each core, acting independently, uses sample-and-hold [13] to identify large flows. After a flow is identified as large, and potentially harboring a CTC, a histogram will be constructed representing that flow's inter-packet delays (IPDs). Once enough packets have been gathered for a given flow (typically 1,000), a detection test such as the first-order entropy test is performed using the histogram. If the score exceeds a certain threshold, the flow is reported as a CTC flow.

The MPPA detection tool could detect real-time CTCs with some success at line rates around 2 Gbps. However, there

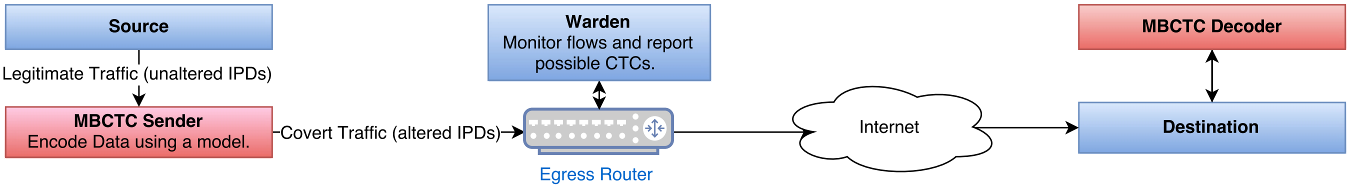
Source
Legitimate Traffic (unaltered IPDs)
MBCTC Sender
Encode Data using a model.
Covert Traffic (altered IPDs)
Warden
Monitor flows and report possible CTCs.
Egress Router
Internet
MBCTC Decoder
Destination

Fig. 1. Model-based covert timing channel detection model. A warden placed at a router monitors traffic and reports unusual flows.

were multiple limitations. Since the memory on a core is fairly small, only a relatively small amount of packet data could be kept on each one. In addition, by having each core handle different sets of flows, the tests themselves could not be parallelized across cores. Both of these factors limited the types of possible detection tests to simple ones such as first-order entropy, Kullback-Liebler divergence, and the Kolmogorov-Smirnov test. While the results show that the first-order entropy test is a decent classifier for MBCTCs, it is outclassed by the corrected-conditional entropy (CCE) test, which could not be implemented on the MPPA tool. Previous results have shown that the CCE test is an excellent CTC classifier, capable of identifying multiple covert channel types with low false positive rates [5].

Compared with specialized hardware like FPGAs or MPPAs, GPUs are more commonly available in existing systems. Therefore, GPU-based packet processing would be more valuable for implementing real-time CTC detection. Some existing GPU packet processing tasks include pattern matching and packet routing [1]. In addition, GPUs have previously been used to accelerate mutual information calculations, a measurement that shares similarities with conditional entropy [14]. Calculating the CCE scores for a large numbers of incoming flows in real-time should also demonstrate more complex GPU calculations on streaming packet data are plausible. The primary motivation behind this project was to determine whether or not CCE-based detection could be effectively implemented using GPU packet processing.

Efficient Entropy Calculation: In addition to detecting covert channels, entropy measurements have a variety of applications. Corrected conditional entropy in particular is useful for evaluating heart rate data and other bioinformatics [6]. Conditional entropy alone is useful for a variety of applications, including analyzing financial time series data [15]. Transfer entropy has applications in data mining and neuroscience, among other uses. For example, it can be used to measure a person's influence on social media such as Twitter, or to measure the connectivity in brain regions [15]. However, for a large value series, entropy calculations can become very time-consuming [15]. Therefore, improving the efficiency of entropy measurements using GPU processing is a worthwhile pursuit beyond its applications for covert channel detection.

III. Corrected Conditional Entropy

The corrected conditional entropy (CCE) test is simply the conditional entropy with a corrective term consisting of the Shannon entropy multiplied by the percentage of unique subsequences added. The Shannon entropy, or first-order entropy, measures the amount of randomness of a random variable. The formula is as follows:

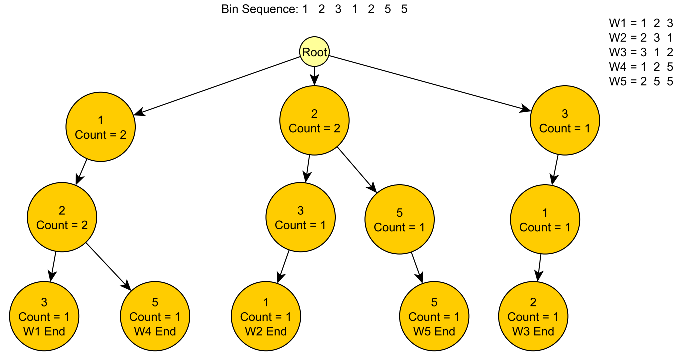

Fig. 2. CCE k-ary tree [7]. k = 5 bins, window size = 3. The bin sequence is divided into 5 windows, each representing a path through the tree. Each node maintains a count of how many windows have passed through it. The counts are then used to calculate the CCE score. Algorithm 1 describes how this same structure can be represented using arrays.

$$H = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{1}$$

with $P(x_i)$ referring to the probability of selecting the value x_i. The conditional entropy (CE) test, which measures the randomness of a variable given the value of another variable, can be calculated as follows for a sequence of random values:

$$H(X_i|X_1..X_{i-1}) = H(X_1..X_i) - H(X_1..X_{i-1}) \tag{2}$$

The CCE formula is as follows:

$$CCE(X_m|X_1..X_{m-1}) = H(X_m|X_1..X_{m-1}) + P \times H \tag{3}$$

with P representing the percentage of uniquely occurring sequences and H being the Shannon Entropy [5]. This corrective term is necessary, because with finite sequences the conditional entropy tends towards zero as the sequence grows longer, while the corrective term will increase [5]. The reason for this is that the conditional entropy calculation requires finding the Shannon entropy values for each subsequence. As the sequence windows get longer, the more likely we will have no repeating subsequences, giving us a final entropy value of 0. With CCE, the corrective term ensures that the score will not continue to decrease towards 0. The maximum CCE score possible is equal to the first-order entropy score [5]. In the context of CTC detection, the sequence we use to calculate the CCE is the sequence of bin numbers determined by the inter-packet delays (IPDs). Each IPD is compared to a range of values based on training data and assigned a bin between 0 and 4 inclusive. The minimum CCE scores can be used to reliably detect common varieties of CTCs, including IPCTC, TRCTC, and MBCTC.

To calculate the minimum CE score, we calculate the Shannon Entropy for each subsequence length from 1 through N, where N is the maximum subsequence length. Then, after calculating all the entropy values, we find the minimum entropy score. The typical way to calculate the CE score is to create a perfect k-ary tree, where k is the total number of bins. The IPD bin sequence for a flow is divided into subsequence windows, which have sizes equal to the tree height. Figure 2 demonstrates the tree used to calculate the CE for a small sequence of seven IPD bin values. Each node of the tree contains a count representing how many times a subsequence has passed through that node. For example, the leaf nodes represent a full window sequence, while the root's children represent the first value in a sequence. These counts are used to calculate the Shannon entropy for each level of the tree, and the minimum value gives the CE score for that network flow. The tree-based minimum CCE calculation functions the same way, but the corrective term is added to the entropy score for each tree level.

A. GPU Entropy Calculation

Algorithm 1 Steps for calculating the CCE for the tree shown in Figure 2 using arrays. The window size is 3, equal to the height of the tree.

```
input: IPD sequence array $A = [2, 3, 5, 3, 5, 4]$.
output: IPD sequence's CCE.
1. Convert the sequence into windows of size 3. $[[2, 3, 5], [3, 5, 3], [5, 3, 3], [3, 5, 4]]$
2. Convert each sub-sequence to a single base 4 value by combining the value in that sub-sequence. For example, in the first window, the sub-sequence of length 3, $[1, 2, 3]$ will become $2 + 3 * 4 + 5 * 16 = 94$. The windows will now be $[[2, 14, 94], [3, 23, 71], [5, 17, 65], [3, 23, 87]]$
3. Arrange the values such that tuples of equal length are together. $[[2, 3, 5, 3], [14, 23, 17, 23], [94, 71, 65, 87]]$
4. Sort the values. $[[2, 3, 3, 5], [14, 17, 23, 23], [65, 71, 87, 94]]$
5. Count how many times a number appears in the same group. $[[1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]$
6. Now, each group corresponds to the counts at a level of the tree. Using these counts, we can calculate the CCE at each level as we would using the tree.
7. Finally, take the minimum of these values to obtain the final CCE score.
```

Although using trees to calculate entropy works, it requires too much time to calculate the CCE score for long sequences of packets. After dividing the sequence of values into windows, the standard k-ary tree-based CCE calculation requires updating each node's count as it is visited while moving in a path from the root to the leaf nodes, repeating for each window in the sequence. Once the counts have been calculated, the CCE score is calculated for each level of the tree and the minimum value is selected as the final score. A previous paper showed this method requires 16 ms to calculate the CCE score for a single flow using a 3.4 GHz Intel Pentium D [5]. For high data rates, the need to process a large number of new flows constantly arriving necessitates a more efficient means of calculating the CCE score for each flow.

For this reason, we chose to perform the CCE calculation using the GPU. However, constructing thousands of large trees in real time on a GPU is difficult to perform efficiently. Rather than dynamically constructing a tree for each flow on the GPU, we instead represent each flow's tree within a single large array. Since the number of bins and window size for the trees is predetermined, the CCE calculation can be performed by first dividing each flow's portion of the array into windows of size N, then counting the number of matching sub-arrays of different lengths from 1 to N. Each sub-array represents a partial path through the tree, and is stored as a single base-4 value for comparison. Although some different sub-arrays will have the same base-4 translation, it is less likely for longer sub-arrays and simplifies the matching process. The conversion of a CCE tree into an array representation is further explained in Algorithm 1, which calculates the CCE score using the same IPD bin sequence used to create the tree shown in Figure 2.

This approach simplifies performing the CCE test on the GPU and has multiple advantages. Since the calculation can be performed entirely using arrays, we could implement the CCE calculation using NVIDIA's CUDA Thrust template library. Thrust is more manageable than raw CUDA kernels and easily portable with multicore CPUs using OpenMP or TBB [16]. In its current state, the algorithm expresses the nodes and counts for a k-ary tree in array form. However, with modifications, other types of data can be stored at each node. Other calculations that require binary or k-ary trees could benefit from using similar methods. Assuming we already know each node's maximum number of children, a tree could similarly be flattened into an array.

IV. System Design and Experimental Setup

Our detection tool is a heterogeneous system, using both the CPU and GPU to calculate flow statistics for an incoming packet stream. Similar to PacketShader and other GPU-based packet processing systems [2], [17], our detection tool first gathers packets into large batches on the CPU, then sends those batches to the GPU for processing. The tool uses two threads to allow an overlap between CPU and GPU operations. One thread receives packets, timestamping them and placing them into a lockfree queue, while the other removes packets from the queue and stores them into batches. Once enough packets have been obtained to form a complete batch, that batch is then copied to the GPU for processing. The processing consists of two steps. First, the GPU gathers converts the batch array into a smaller arrays consisting of only packets belonging to flows with enough packets to accurately perform the CCE test.

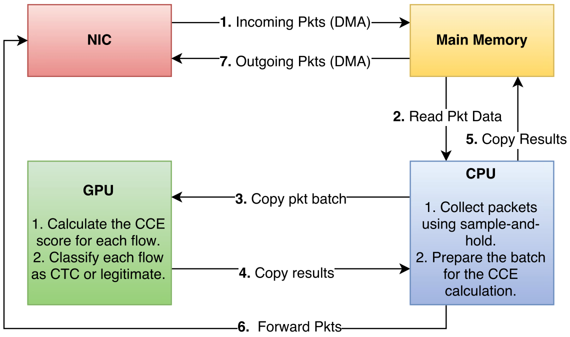
NIC
1. Incoming Pkts (DMA)
7. Outgoing Pkts (DMA)
Main Memory
2. Read Pkt Data
5. Copy Results
GPU
1. Calculate the CCE score for each flow.
2. Classify each flow as CTC or legitimate.
3. Copy pkt batch
4. Copy results
CPU
1. Collect packets using sample-and-hold.
2. Prepare the batch for the CCE calculation.
6. Forward Pkts

Fig. 3. Basic GPU-based packet processing model, adapted from Mukerjee et al. [17]. The GPU receives packet batches from the CPU, processes them and returns a result.

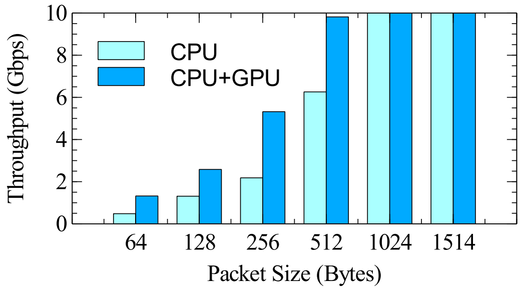

Fig. 4. CPU vs. GPU maximum throughput achieved for different packet sizes.

Second, the GPU takes this modified array and calculates the CCE score for each flow it contains. If a flow's reported CCE score is under a certain predetermined threshold, then that flow will be reported as containing a model-based covert timing channel (MBCTC). Figure 4 compares the performance with the OpenMP CPU-only version of the tool. Figure 3 gives an overview of how the system processes packet data.

A. PF_RING Packet Capture

Our CTC detection tool prototype uses PF_RING ZC (zero-copy), a NUMA-aware packet processing framework developed by ntop to receive packets at line rate on a 10 Gbps ethernet link [18]. Similar to Intel's DPDK [19] or netmap [20], PF_RING ZC bypasses the standard network stack, accelerating packet processing. Using DMA, the NIC copies packet data directly to memory, rather than copying between the kernel and user space. When beginning an application, PF_RING ZC establishes packet buffers to avoid any memory allocation. Rather than receiving packets through interrupts, PF_RING ZC polls for packets [21]. According to ntop [18], PF_RING ZC performs better than DPDK for smaller packets. Our packet sniffing code builds on ntop's zcount example program, which receives and counts packets at 10 Gbps line rates regardless of packet size.

B. Batch Processing

The packet I/O code is based on a modified version of pf_ring's zcount example. One thread is dedicated to collecting and timestamping incoming packets. The thread puts the raw packet pointers and timestamps into a lockfree queue. The processing thread continuously reads packet data from this queue, obtains the four-tuple identifying the flow (source ip and port, destination ip and port), and stores them in a buffer. Since we assume only large flows contain CTCs, and 80 percent of flows contain no greater than 20 packets [22], we use ”sample-and-hold” [13] to reduce the amount of packet data stored. For each incoming packet from a new flow, there is a small chance (about 0.5% in our case) that it and all further packets in that flow will be stored. Therefore, only large “Elephant” flows are likely to be stored in the buffer [23], reducing memory usage and creating a buffer containing mostly flows capable of carrying a high capacity CTC. Since the CTC flows we can detect will be very large (thousands of packets or more), we can afford to use a very low sample-and-hold probability. This processes continues until enough packets are gathered in the buffer to send a large batch to the GPU for testing. By default, the batch size is set to 12,500,000 packets. Larger batch sizes will increase bandwidth, but also latency.

Once enough new packets are stored, the packet data in the buffer array is prepared for the CCE calculation. First, the array containing the IPDs is converted into a new array containing only flows with enough data to obtain an accurate CCE score, typically between 500 and 2,000 packets. Then, we calculate the adjacent difference for the packet timestamps to obtain the inter-packet delays (IPDs). Using equiprobable binning, the IPDs are converted to a bin value between 1 and 5, with 5 representing the largest IPDs. If there are enough eligible flows, this modified batch array is copied to the GPU to perform the CCE calculation using Thrust. Although our approach will accurately detect CTCs in a batch, one issue is that processing packets in batches will inevitably capture only a fraction of the large flows. For example, assume we set the CCE test to process flows with 2000 packets or more, and the batch only contains the first 1000 packets of that flow. If the flow is only slightly larger than 2000 packets, the flow will not be reported, because not enough of its packets were present in any given batch. Therefore, our current approach can only sample a fraction of the overall large flows. Using our trace file with a 500 IPD threshold, we captured around 98.4% of the large flows, and 91% when replaying it together with near 10 Gbps traffic. However, we assume CTC flows are large and long-lived [4], and therefore a more significant portion of potential CTC flows are likely to be sampled in their lifetime. CUDA Thrust allows GPU code to be updated and tested quickly, while also being portable with multi-core CPUs.

C. Experimental Setup

For our setup, two PowerEdge T630 machines (a sender and a receiver) were connected by a 10 Gbps Ethernet connection. Both the sender and receiver contained two Intel Xeon E5-2637 v3 3.5GHz processors. The receiver contains a PowerEdge T630 GPU along with a NVIDIA Tesla K20C GPU accelerator, which is used for our experiments. Designed for general purpose computing, the Tesla K20C has 2496 CUDA cores,

TABLE I
MBCTC TRACE FILE STATISTICS. LARGE FLOWS CONTAIN 1000 PACKETS OR MORE.

Total Packets	Total Flows	Large Flows	Legitimate	CTCs
33581932	631089	3377	3056	321

208 GB/s memory bandwidth, and 5 GB GDDR5 memory [24]. The sender will transmit flows to the receiver, which sniffs incoming traffic and processes the packets in batches using the CCE test. By running our tool in this way, we performed a variety of tests, evaluating the CCE test's effectiveness of as a classifier, measuring the packet processing time, and the various trade-offs in our implementation between memory usage, latency, and throughput on the GPU. From a CAIDA repository [25], we used a real traffic trace containing one minute of traffic from a 10 Gbps San Jose OC-192 link. The trace file has been anonymized, meaning it only contains packet headers and therefore minimum-sized packets. This same trace was used in our previous real-time detection experiments [4]. We modify the pcap file by replacing roughly 10% of the flows containing greater than 1000 packets with MBCTC flows. The threshold choice of 1000 IPDs was chosen based on previous detection experiments [4], [5]. Table I describes the CTC-containing trace file we used for our experiments.

For sending packets, we used two different tools—tcpreplay and zsend. Depending on the experiment, we send packets using either tcpreplay, zsend, or both. Tcpreplay replays pcap files while maintaining accurate timing information, meaning we can use it for our detection rate tests. PF_RING ZC's zsend tool allows us to blast random packets at 10 Gbps line rate for testing the maximum data rates our system can handle without packet loss. By setting the packet size, zsend lets us test the maximum data rate at varying packet sizes. The two tools can be combined by replaying with tcpreplay while running zsend. By combining the two, we embedded our trace file while sending larger amounts of traffic to test detection at high data rates.

V. Results and Discussions

We performed tests on the tool's ability to identify CTCs using the CCE test, as well as the performance. Table II defines the important system and experimental parameters.

A. Classification Results

To ensure our tool functions properly, we measured the true and false positive rates for the CTCs reported after receiving all the packets from our trace file. In this case, a true positive refers to a flow being correctly reported as containing a CTC. As Figure 5 demonstrates, the corrected conditional entropy test score performs well at classifying MBCTCs. Even while sending our trace file embedded within 10 Gbps traffic, the CCE scores remain accurate. The percentages nearly match previous results [5], which reported a 95% true positive rate with a 1% false positive rate when testing a sample of 2000 packets. By tweaking the score threshold for reporting CTCs to reduce the false positive rate to the same value, we also obtain a true positive rate around 95%. For our results, we report a possible CTC if the CCE score is less than 0.4, ignoring outliers with CCE scores near zero. However, this threshold can be altered depending on how many false positives can be accepted. The response to a reported false positive could be to add noise to that flow through fuzzy time or other techniques [11], [12]. Since that flow's packets will still arrive, albeit at a reduced rate, some false positive may be allowed depending on the application. However, a stricter false positive rate could be necessary for some applications such as VoIP.

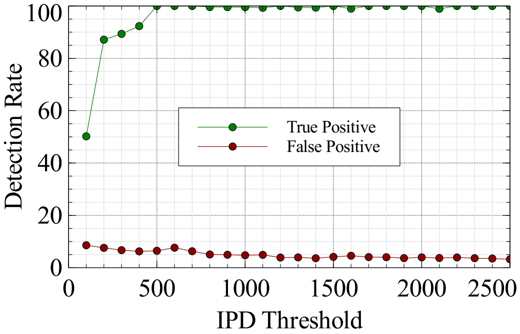

Fig. 5. MBCTC true and false positive rates vs. the amount of IPDs tested per flow. Flows with CCE scores < 0.4 were reported as CTCs.

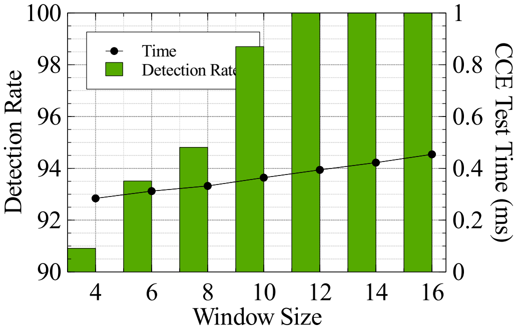

Fig. 6. CCE test time per flow and MBCTC true positive rates vs. the window size used to calculate the CCE score. The false positive rate for these values was roughly 2.5% and the IPD threshold was set to 2000.

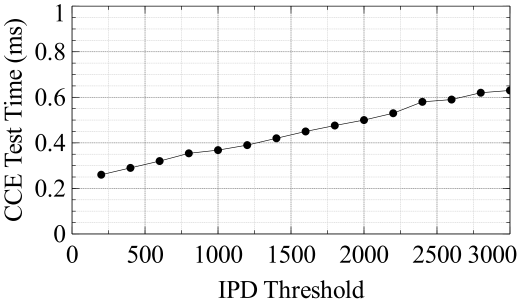

Fig. 7. CCE test time per flow vs. the IPD threshold used to calculate the CCE score. Window Size was set to 10.

TABLE II
EXPERIMENTAL PARAMETERS

Parameter	Definition
True Positive Rate	Percentage of tested CTC flows correctly classified as CTCs.
False Positive Rate	Percentage of tested legitimate flows incorrectly classified as CTCs.
Window Size	The IPD bin sequence window length. Equivalent to the CCE tree height.
IPD Threshold	The number of IPD bin values used per flow for calculating the CCE scores.
Batch Latency	The time spent gathering packets before processing a batch.
Maximum Throughput	The maximum data rates achievable without dropping packets.
Batch Threshold	The number of new packets required before testing a batch.
Batch Setup Time	The time spent preparing a packet batch for the CCE test passing the batch threshold.
CCE Test Time	The time required to calculate CCE scores and report flows as CTCs or not after preparing a batch.

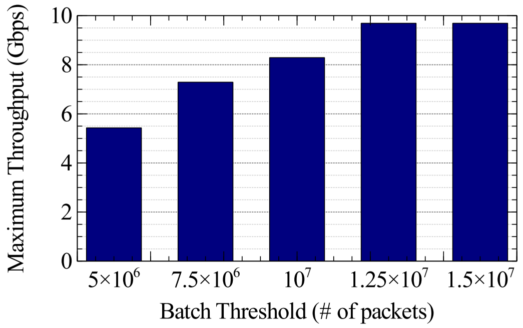

Fig. 8. Maximum throughput achieved with the GPU using different batch testing thresholds.

There are trade-offs to consider when selecting the window size and IPD threshold used for calculating the CCE score. Using a larger window size and testing more IPDs will give more accurate detection (Figure 6). However, that will also increase the time required to process the flows, and require more memory per flow. Figure 6 shows that the time per flow can increase significantly. Ideally, the smallest possible sample and window size should be used to handle higher data rates without packet loss. The detection rate for our sample stopped improving significantly at 2000 IPDs and window size 10. With those parameters, it takes around 0.4 ms per flow to calculate the CCE score for a batch.

B. GPU Packet Processing Results

In order to test for CTCs, we gather packets into large batches. Once enough packets are gathered, the packet batch is copied to GPU memory, and each flow in the batch is tested for CTCs. Before copying to GPU memory, the batch must be setup for the CCE calculation. This involves culling all flows below the IPD threshold, and ensuring that all remaining flows have an equal number of IPD bin values. The batch threshold affects the test result latency and throughput. A larger batch threshold increases the throughput by processing more flows per batch (Figure 8). Larger batch thresholds also increase the latency between receiving enough packets to perform the CCE test on a flow and reporting whether or not it contains a CTC (Figure 10). Assuming we test 2,500 IPDs per flow, we can test 3,000 large flows per batch without running out of memory during the Thrust CCE calculations. This translates to a maximum batch size of 7,500,000 packets. However, it is improbable that all the flows in a batch will each contain exactly 2,500 IPDs. Therefore, batch sizes larger than 7,500,000 are possible if larger throughput is desired, provided the number of flows to be tested is limited to 3,000 or less. We obtained the best results using a threshold of 12,500,000 or more, as shown in Figure 8.

In Figure 4, we show the highest possible rates we could achieve with different packet sizes on a CPU and GPU implementation of our detection tool. To measure this, we have our sender machine blast packets to the receiver using PF_RING's zsend application for five minutes, then report whether or not any packets were dropped. Since zsend cannot specify the number of packets per flow, we assign a random flow id between 0 and 14999 to each incoming packet, ensuring that flows are large enough to pass the sample-and-hold test and be stored in a batch. In order to simulate a heavy workload with hundreds of large flows being processed every batch, each batch is randomly assigned between 0 and 1,500 flows over the IPD threshold per batch for which to arrange and calculate the CCE score. On average, around 750 flows containing over 2500 IPDs will be tested per batch using this method. Assuming average 512 byte packets, our tool can handle 10 Gbps traffic at near line rate (about 9.69 Gbps, or 2,200,000 pps). However, higher rates should be possible when testing with real traffic samples, depending on how many flows pass the sample-and-hold test. Since code written in Thrust is portable between CUDA and OpenMP, comparing the performance is simple [16]. Figure 4 shows that the GPU version performs significantly better than the parallel OpenMP CPU version running on eight Intel Xeon E5-2637 cores for packet sizes up to 512B.

The total time to complete the two batch processing steps depends primarily on a few factors—the number of eligible flows, the window size, and the IPD threshold value. We obtained our best results by having Thrust perform the batch setup on the CPU and the CCE calculation on the GPU. If the batch contains the maximum number of flows possible (3000), the calculation will require slightly over 2 seconds to classify the flows (Figure 9). However, since a batch will almost never contain only eligible flows, packet loss is unlikely. The CCE

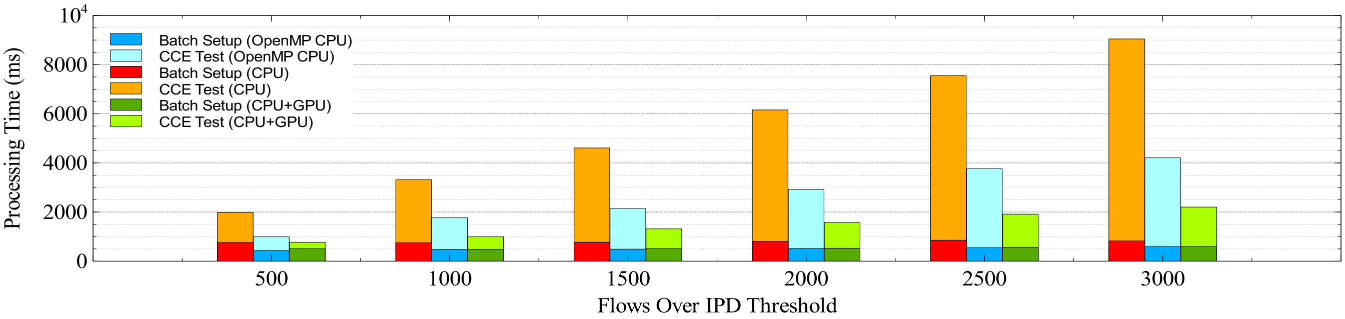

Fig. 9. CPU Batch processing time vs. the numbers of flows tested.

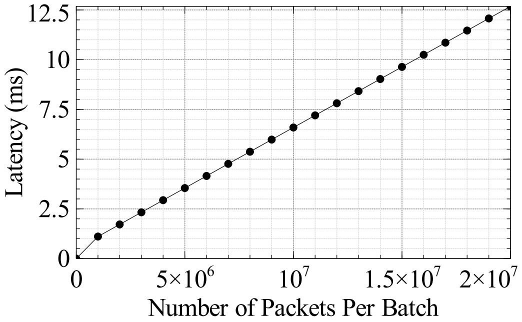

Fig. 10. Batch Latency vs. Batch Threshold.

calculation time per flow remains below 0.5 ms regardless of how many flows are being tested. Even including the batch setup time, this is significantly faster than the 16 ms per flow CCE calculation described in previous results [5].

VI. Related Work

Covert Timing Channel Detection: Although many papers describe techniques for either limiting channel capacity or eliminating network covert timing channels completely [11], [12], these techniques usually hurt legitimate traffic performance as well, making covert timing channel detection the more appealing choice [4]. Cabuk et al. [10] introduced two covert timing channel detection techniques—the regularity test and the ϵ-similarity test. Gianvecchio and Wang [5] provide background on a variety of covert timing channel detection techniques, and also introduce entropy-based detection using measurements of first-order entropy and the corrected-conditional entropy to identify covert traffic. Many other measurements have been used for detection, such as mean-max ratio [5]. Commonly, detection techniques are created to counter a particular covert timing channel, but have limited effectiveness in detecting other channel types [5]. Examples include the ϵ-similarity test (effective for detecting IPCTC traffic), and measuring the data and acknowledgement packet timing intervals (effective for detecting the Cloak CTC) [5]. After new detection techniques are introduced, new covert timing channel types designed to counter those techniques tend to follow [5], [7].

GPU Packet Processing: There have been many papers showing GPU packet processing as an effective means of scaling network packet processing applications using commodity hardware [1]. Given their higher memory bandwidth, GPUs have been shown to perform high data rate software packet processing more efficiently than CPUs alone [2]. PacketShader is a GPU-based processing framework that can perform OpenFlow flow matching and ipv4/ipv6 packet forwarding at multi-10Gbps rates [17]. Similarly, Snap is a framework built on top of Click, a modular software router. Snap performs SDN forwarding and other processing tasks at 30 Gbps with minimum-sized packets, and can reach 40 Gbps with packet sizes starting at 128-bytes [26]. In addition to packet forwarding, GPU-based processing has been used to quickly perform pattern matching for intrusion detection systems, such as Kargus and Gnort [27], [28]. GPU-based processing has also been applied to software-defined networks (SDNs). For example, GSwitch is a recent system that performs packet classification using the GPU to improve packet searches. Their Bloom search algorithm outperforms a CPU-based equivalent by a factor of 12, processing 64-byte packets at 10 Gbps [29].

Depending on the application, much of the benefits of GPU processing come from the advantages of writing algorithms in languages such as OpenCL or CUDA, which has inherent advantages such as vectorization and hiding memory latency [30]. By optimizing memory latency in CPU software packet processing applications, the authors significantly closed the gap between CPU and GPU performance [30]. Therefore, GPU-based packet processing could be more worthwhile for tasks that benefit more from vectorization than reducing memory latency, since programming for CPU-based vectorization is difficult [30], although compilers such as Intel's ispc provide extensions for single-instruction multiple data (SIMD) programming [30]. For this reason, CTC detection could be expected to benefit significantly from GPU processing, since certain detection tests—including the CCE entropy test—require processing a large number of flows, each with corresponding vectors of inter-packet delays (IPDs).

VII. Conclusion and Future Work

Our results confirm that covert timing channel detection can be performed efficiently in real-time by calculating the corrected conditional entropy scores for network flows on a GPU. Our tool manages to detect CTCs more accurately and at higher data rates when compared to previous results [4], even while running purely on the CPU. As predicted by earlier results [5], the CCE test is a reliable classifier for model-based covert channels, identifying suspicious flows with a low false positive rate. By converting the conditional entropy tree structures into arrays, batches of packet data can be converted into a format that is easily parallelized and translated into GPU code. This technique could potentially be applied to other entropy measurements, such as those used to evaluate financial transfers or connectivity in the brain [15]. Our results demonstrate that, in addition to tasks such as firewall rule lookups and packet forwarding [2], GPU packet processing can be applied for improving statistical analysis of individual network flows at the packet level.

There are multiple ways in which our tool could be expanded upon and improved. One obvious way would be to include additional CTC detection tests that can identify channel types that evade CCE detection, such as including Welch's t-test for Jitterbug channels [31]. Although our detection tool could handle traffic at 10 Gbps line rates with average sized packets, 40 Gbps line rates are becoming more common. Although using raw CUDA kernels might increase the maximum throughput, using CUDA Thrust provides multiple advantages, notably the ease of coding that allows new detection tests to be added quickly, as well as portability between CPUs and GPUs. One potential direction to expand this work would be to integrate it with the Bro intrusion detection system by writing policy scripts that respond to reported CTCs by disrupting the flow's packet timing to reduce or eliminate the channel's capacity. Finally, although our implementation uses only a single GPU, the CCE computation should scale well for handling higher data rates. The more memory available, the more batches can be created. On a system with multiple GPUs, multiple batches can be processed in parallel, allowing our tool to handle much higher rates of traffic.

Acknowledgment

This work was supported by NSF grants CNS-1528087 and CNS-1018886.

References

[1] K. Tsiamoura, "A survey of trends in fast packet processing," *Network*, vol. 41, 2014.

[2] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[3] J. Zheng, D. Zhang, Y. Li, and G. Li, "Accelerate packet classification using GPU: A case study on HiCuts," in *Computer Science and its Applications*. Springer, 2015, pp. 231–238.

[4] R. K. Gegan, R. Archibald, M. K. Farrens, and D. Ghosal, "Performance analysis of real-time covert timing channel detection using a parallel system," in *Network and System Security*. Springer, 2015, pp. 519–530.

[5] S. Gianvecchio and H. Wang, "An entropy-based approach to detecting covert timing channels," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 785–797, 2011.

[6] A. Porta, G. Baselli, D. Liberati, N. Montano, C. Cogliati, T. Gnecchi-Ruscone, A. Malliani, and S. Cerutti, "Measuring regularity by means of a corrected conditional entropy in sympathetic outflow," *Biological Cybernetics*, vol. 78, no. 1, pp. 71–78, 1998.

[7] K. Kothari and M. Wright, "Mimic: An active covert channel that evades regularity-based detection," *Computer Networks*, vol. 57, no. 3, pp. 647–657, 2013.

[8] R. Strzodka, M. Doggett, and A. Kolb, "Scientific computation for simulations on programmable graphics hardware," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 667–680, 2005.

[9] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP routing processing with graphic processors," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 93–98.

[10] S. Cabuk, "Network covert channels: Design, analysis, detection, and elimination," Ph.D. dissertation, Dept. Computer Science, Purdue University, West Lafayette, Indiana, 2006.

[11] M. H. Kang, I. S. Moskowitz, and S. Chincheck, "The pump: A decade of covert fun," in *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE, 2005, p. 7.

[12] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3–4, pp. 233–254, 1992.

[13] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "ElephantTrap: A low cost device for identifying large flows," in *15th Annual IEEE Symposium on High-Performance Interconnects*, ser. HOTI 2007. IEEE, 2007, pp. 99–108.

[14] Y. Lin and G. Medioni, "Mutual information computation and maximization using GPU," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, ser. CVPRW'08. IEEE, 2008.

[15] S. Shao, C. Guo, W. Luk, and S. Weston, "Accelerating transfer entropy computation," in *International Conference on Field-Programmable Technology*. IEEE, 2014, pp. 60–67.

[16] L. Lo, C. Sewell, and J. P. Ahrens, "PISTON: A portable cross-platform framework for data-parallel visualization operators," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2012, pp. 11–20.

[17] M. Mukerjee, D. Naylor, and B. Vavala, "Packet processing on the GPU." Unpublished manuscript. Department of Computer Science, Carnegie Mellon University.

[18] "Ntop," http://www.ntop.org.

[19] "Impressive packet processing performance enables greater workload consolidation," Intel Solution Brief http://www.intel.com, 2013, accessed: Whitepaper.

[20] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[21] S. Gallenmuller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE, 2015, pp. 29–38.

[22] I. Matta and L. Guo, "Differentiated predictive fair service for TCP flows," in *Proceedings of the International Conference on Network Protocols*. IEEE, 2000, pp. 49–58.

[23] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang, "SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws," in *43rd Allerton Conference on Communication, Control and Computing*, 2005.

[24] "NVIDIA," http://www.nvidia.com/object/tesla-workstations.html, accessed: 2015-03-28.

[25] "CAIDA data." http://www.caida.org/data/overview/.

[26] W. Sun and R. Ricci, "Fast and flexible: parallel packet processing with GPUs and click," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 2013, pp. 25–36.

[27] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 317–328.

[28] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 116–134.

[29] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multi-layer packet classification with graphics processing units," in *Proceedings of the 10th*

ACM International on Conference on Emerging Networking Experiments and Technologies. ACM, 2014, pp. 109–120.

[30] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, “Raising the bar for using GPUs in software packet processing,” in *12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI 15, 2015, pp. 409–423.

[31] R. Archibald and D. Ghosal, “A comparative analysis of detection metrics for covert timing channels,” *Comput. Secur.*, vol. 45, pp. 284–292, Sep. 2014.