CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs

(article starts on next page)

CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs⋆

Charalampos Stylianopoulos[0000−0002−6845−9163] ✉, Linus Johansson, Oskar Olsson, and Magnus Almgren[0000−0002−3383−9617]

Chalmers University of Technology, Gothenburg, Sweden
{chasty, magnus.almgren}@chalmers.se

Abstract. While IoT is becoming widespread, cyber security of its devices is still a limiting factor where recent attacks (e.g., the Mirai botnet) underline the need for countermeasures. One commonly-used security mechanism is a Network Intrusion Detection System (NIDS), but the processing need of NIDS has been a significant bottleneck for large dedicated machines, and a show-stopper for resource-constrained IoT devices. However, the topologies of IoT are evolving, adding intermediate nodes between the weak devices on the edges and the powerful cloud in the center. Also, the hardware of the devices is maturing, with new CPU instruction sets, caches as well as co-processors. As an example, modern single board computers, such as the Odroid XU4, come with integrated Graphics Processing Units (GPUs) that support general purpose computing. Even though using all available hardware efficiently is still an open issue, it has the promise to run NIDS more efficiently.

In this work we introduce *CLort*, an extension to the well-known NIDS Snort that a) is designed for IoT devices b) alleviates the burden of pattern matching for intrusion detection by offloading it to the GPU. We thoroughly explain how our design is used as part of the latest release of Snort and suggest various optimizations to enable processing on the GPU. We evaluate *CLort* in regards to throughput, packet drops in Snort, and power consumption using publicly available traffic traces. *CLort* achieves up to 52% faster processing throughput than its CPU counterpart. *CLort* can also analyze up to 12% more packets than its CPU counterpart when sniffing a network. Finally, the experimental evaluation shows that *CLort* consumes up to 32% less energy than the CPU counterpart, an important consideration for IoT devices.

Keywords: IoT · NIDS · GPU · pattern matching · high throughput.

1 Introduction

Even though Internet of Things (IoT) technologies have become widespread and mature, cyber security is still a problem. Several attacks, across very different

---

environments, demonstrate in painstaking detail that the community needs to build security mechanisms suitable for IoT, or else deployment may slow down. A recent example is the series of attacks against the electricity network in both the distribution and transmission grid in Ukraine by controlling the devices found in substations.

Challenges to improve security in IoT stem from different factors. For a long time, an IoT system was designed with very limited edge devices that communicated with a powerful cloud. Even though the cloud could handle many security mechanisms, the attacks happen at the edges of the network, targeting devices that need to be cheap, conserve power and are too limited to run their own security mechanisms. Fortunately, modern IoT systems have become more heterogeneous with different types of devices. The previously limited edge is becoming slightly more powerful with new processors and architectures, and the powerful cloud has been complemented by a range of devices, the so-called fog in-between the edge and the cloud, with devices that offer more computational power and, for some applications, a much faster response rate than sending the data to the cloud. These intermediate IoT devices promise to also improve the security of the system as a whole.

In this paper, we take advantage of the recent maturity of IoT devices and investigate how a network intrusion detection system, one of the cornerstones of regular IT security, can run efficiently in the IoT. More specifically, as recently released devices come with integrated co-processors or graphics processing units, we investigate how to use the full hardware of a dedicated "security node" to improve the speed (throughput) of the analysis, while using less energy to do so. Moreover, as one challenge of IoT is the distributed nature of the system, it may not be possible to define a single choke point for network analysis. As we demonstrate that our solution processes packets faster, it may be possible to run the intrusion detection system on existing nodes in the network while still leaving enough CPU cycles for the nodes' primary function.

The outline of the paper is the following. In Section 2, we outline background concepts related to this work, namely Snort, the Aho-Corasick algorithm and a high-level description of general purpose computing on GPUs. In Section 3, we explain the design of our system followed by the evaluation in Section 4. Section 5 describes related work and we conclude the paper in Section 6.

2 Background

Given the prominence of Snort as a network intrusion detection system, we start with an introduction to such systems in general and Snort in particular. We then describe the pattern matching algorithm in Snort (Aho-Corasick). Finally, we give a brief background on general purpose computing on GPU devices.

2.1 Network Intrusion Detection Systems and Snort

The purpose of a Network Intrusion Detection System (NIDS) is to inspect all incoming and outgoing network traffic and alert for any malicious behaviour.

Many NIDS are *signature-based*, meaning that they rely on a set of patterns that are part of known attacks or vulnerabilities. One of the benefits of NIDS, over for example a firewall, is that they inspect not only the packet headers but also the packet payload (a.k.a. *deep packet inspection*) in order to detect a wide range of malicious attacks.

Nowadays, Snort is one of the most commonly deployed signature-based NIDS. Originally developed in the late 90s, Snort has been in active development ever since and has become the de facto NIDS. Its most recent version (Snort 3, in alpha version when this paper is written), offers many new features, such as a modular architecture, cross-platform support and multi-threaded processing of traffic from different interfaces.

Snort relies on *rules* that determine what kind of malicious behaviour it should look for in a packet. Rules usually contain a fixed string pattern, as well as other options that need to be true to flag a packet as malicious (e.g., traffic towards specific ports). A very brief outline of Snort's processing pipeline is the following: (i) Snort *captures* packets from a network interface or a capture file, (ii) a *decode* module creates common metadata for this packet, such as source and destination ports and encapsulated protocols, (iii) packets that belong to a TCP stream are reassembled, (iv) a *search engine* performs pattern matching on the packets, where the payload data are compared against the malicious patterns, and (v) if a match is found, a *validation* step is invoked to ensure that the rest of the rule options are also true for the packet containing the match. Finally, (vi) Snort outputs a verdict for the packet (whether or not it is malicious).

The pattern matching in step (iv) is an expensive bottleneck and therefore the focus of this paper. Snort uses the Aho-Corasick pattern matching algorithm, as described below.

2.2   The Aho-Corasick patten matching algorithm

Aho-Corasick [1] is a popular, *state machine* based algorithm that allows Snort to match the payload against multiple patterns at the same time. The first step of Aho-Corasick is to build a state machine out of all the patterns, where the individual characters in the patterns become the transitions to new states. The state machine is usually implemented as a two-dimensional *state transition array*, with a row for each state and a column for every possible transition from that state to the next one. An extra bit in the array is reserved for final states, i.e. states that indicate that a full pattern has been matched.

After building the state machine at setup time, performing pattern matching on the packet payload is relatively straightforward: starting from the initial state, the algorithm examines one character and uses it to determine the next state. The algorithm keeps jumping from state to state, based on the information found in the state transition array. If the execution reaches one of the final states, a pattern has been found in the payload and Snort will then check other parameters of the full rule before sending out an alert.

We have chosen to use Aho-Corasick as a cornerstone for the work in this paper because: (i) it is what Snort actually uses and (ii) it can be parallelized, making it a good match for the GPU.

2.3 General Purpose GPU Computing

Originally designed for graphics processing tasks, in the last decade GPUs have been proven increasingly successful for offloading computation from the CPU [5]. Hence, General Purpose Computing on the GPU (GPGPU computing) is a term used for the use of GPUs to perform tasks that would be usually performed on the CPU.

The internal architecture of GPUs involves thousands of threads (orders of magnitudes more than on a standard CPU) that have a very simple pipeline and generally operate on a lower frequency. As such, the GPU is an appealing platform for computing tasks that benefit from a high degree of parallelization.

There are two main frameworks that make general purpose computing possible on GPUs: CUDA [10], developed by NVIDIA and OpenCL [8], an open-source library developed by the Khronos Group. Although high-end desktop GPUs have been extensively used for various projects using these two frameworks, embedded GPUs, such as the one we use in this project, have only recently gained support for GPGPU computing. The platform used in this work offers OpenCL 1.2 support, so we use this framework in this paper.

3 Design of *CLort*

As one of the most expensive operations of the NIDS for the CPU is the pattern matching engine, we describe the design of *CLort* and the way it extends Snort by offloading the pattern matching to the GPU. We start with the general, high level design of *CLort*. Then, we discuss issues related to several steps of this design, namely the transferring of data to and from the GPU and the parallelization of pattern matching on the GPU. Finally, we show how optimizations, such as the double buffering technique, are incorporated into our design to get the most speedup.

3.1 *CLort*'s general design

The general design of *CLort* is described in the left part of Figure 1 (where the right part is described later in Section 3.4). Incoming packets enter *CLort*'s pipeline after being processed by the first, pre-processing stages of Snort (see Section 2.1). The payload of each packet is sent to the GPU, to be checked against the state machine created by the patterns that are relevant to that packet. After that, the GPU executes the kernel that implements the Aho-Corasick pattern-matching algorithm. The CPU waits until the execution of the GPU is finished and the results are available. After that, execution continues with the rest of Snort's pipeline that includes validating the matches and logging the verdict for the packet (i.e. logging whether it is malicious).
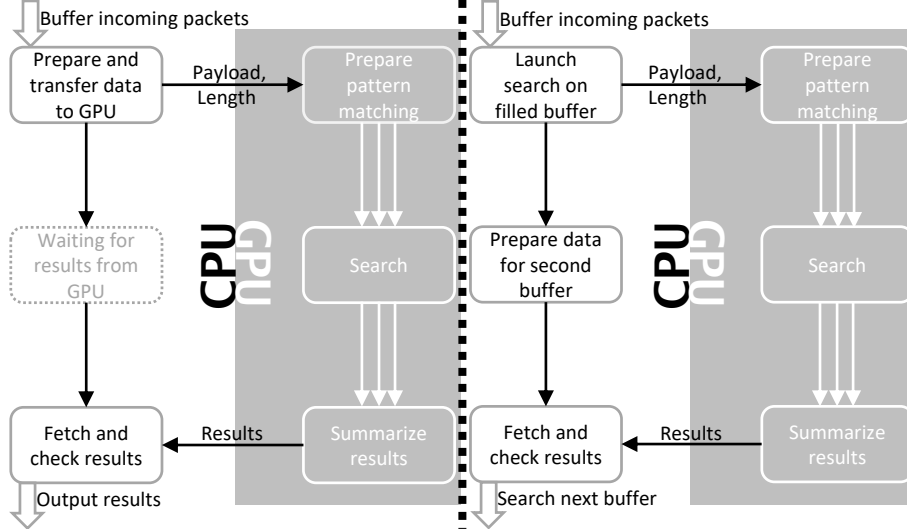
Fig. 1. The left part shows the high level design of *CLort*, with the different steps involved in offloading the pattern matching of Snort to the GPU. The right part depicts an optimization with double buffering to increase the utilization of the CPU.

3.2 Data transfers between the CPU and the GPU

Performing the pattern matching on the GPU requires that relevant data is transferred to the memory of the GPU, and then that the result is transferred back to the CPU. In general, data transfers to and from the GPU's device memory can be a significant bottleneck. However, for our hardware (further described in Section 4.1), the particular characteristics of the GPU offer an interesting way to alleviate that bottleneck. The Mali GPU of the Odroid XU4 does not have a separate device memory but shares the physical memory with the CPU. Thus, we can avoid unnecessary data transfers by mapping the memory region (using OpenCL's interface) of the data that we should send. The memory region is then directly accessible to the GPU. To allow the CPU to read the results, we map the region back to the CPU address space.

Related to data transfers, it is worth mentioning some details on the data structures that are transferred (or, in our case, mapped) to the GPU, specifically the state machine of Aho-Corasick (described in Section 2.1). Originally, the state machine is a two-dimensional array, with a row for each state and a column for each possible transition from that state to a next one. Here we note that: (i) in order to be mapped to the GPU, the state machines need to be serialized as a one-dimensional array (a simple transformation). The serialization and the corresponding mapping of the memory only happen once per state machine during setup, as the state machines are read-only data structures known at the start of Snort. (ii) Snort creates multiple state machines based on traffic characteristics (protocols, ports, etc.) and packets are matched against a state

machine that is relevant to their traffic which also our implementation respects: when a packet is mapped to the GPU for processing, the correct state machine is used as an argument to the kernel that will process that packet.

3.3   Search on the GPU: Parallel Aho-Corasick

When state machines and the packet payloads are available to the GPU, pattern matching is performed using the Aho-Corasick algorithm (Section 2.2).

We parallelize Aho-Corasick in the following way: we split the payload data into a number of chunks, equal to the number of available GPU threads. Each thread is able to process its own chunk, in parallel, without the need for inter-thread communication. The input is divided evenly, so that every thread has equal amount of work to do, compared to the other threads. This avoids the problem of some threads terminating early and stalling, which exists in other parallelization methods for Aho-Corasick [9].

However, splitting the payload into chunks might result in a malicious pattern being split across more than one chunk, with no single thread being able to detect the full pattern in "their" part. In order to detect such patterns, we let each thread process a fixed number of characters also from the chunk of the next thread (equal to the length of the longest pattern). This way, at least one thread will detect every malicious pattern. The disadvantage, however, is that short patterns that exist at the beginning of the chunks will be reported by two threads. We compensate by keeping an auxiliary data structure that holds the length of every pattern that is associated with a final state (a state indicating that a full pattern has been found). When we have a match in a thread, we use this data structure to determine the starting position of the match. If the start is within the chunk of the thread that found the match, it will be reported otherwise it will be ignored (as the next thread "owning" that chunk will find the same pattern and report it).

3.4   Packet Buffering: the double-buffering technique

As mentioned in other work [18,7], launching a kernel for every single packet is not efficient for two main reasons. Firstly, there is significant overhead associated with launching a GPU kernel and it is good to amortize this cost over several packets. Secondly, with a single packet, especially if the packet is small, there might not be enough parallelism to fully exploit the GPU. There will not be enough data to distribute to all available GPU threads or each thread will only process a very small amount of data before exiting. For that reason, we buffer packets on the CPU to submit in batches to the GPU. When a new packet arrives in the Snort pipeline, it will be copied into a buffer. The processing of that packet is postponed at this point and Snort can continue acquiring new packets. When the buffer is full, we launch the GPU kernel to process all packets at once. Having more data to process allows us to make the most of the parallelism the GPU has to offer. Even though we introduce a small amount of latency before a packet is being processed, it is not a problem on regular networks as the buffer

is significantly smaller than the traffic received during a short period of time. However, as we describe later in Section 4, our current implementation that uses buffers cannot make use of the final parts of Snort's pipeline (validation and verdict).

We have investigated two different designs in our work (Figure 1). In the basic design (to the left in the figure), when a kernel is being executed on the GPU, the CPU waits until the end of the execution to get the results. While this is a straightforward design, it does not optimize throughput for a node dedicated for monitoring the network but may work well if there are other tasks needing cycles on the CPU.

In the *double buffering* design (shown to the right), both the CPU and the GPU perform work in parallel and, as will be shown in our evaluations, this increases the utilization of the CPU. In short, in the double buffering technique, as proposed by [19], two buffers are used to store packets on the CPU. When the first buffer is full and the GPU starts processing packets, the CPU can keep buffering packets in the second buffer. When the second buffer is also full, the CPU will first collect the results from the GPU execution, before launching another kernel to process data in the second buffer.

In Section 4.2 we measure the effect of the double buffering technique and show that it successfully reduces the overall processing time.

4 Evaluation

We implemented *CLort* using the OpenCL framework. This section presents the results from the experimental evaluation of *CLort*, using a wide range of experiments to measure and evaluate the benefits that *CLort* brings in intrusion detection for IoT. The experiments are performed on four versions of Snort: Snort original, Snort modified (CPU), *CLort* single buffer (GPU), and *CLort* double buffer (GPU). The *Snort modified (CPU)* is included to make the comparisons as fair as possible. This version of Snort behaves just like *CLort* (buffers packets and does not perform the validation and verdict steps from Section 2.1), but runs the search on the CPU. All comparisons and relative speedups reported use Snort modified (CPU) as a baseline.

4.1 Experimental Methodology

Hardware: We use the Odroid XU4 platform [12], a single board computer with a big.LITTLE architecture (ARM Cortex-A15 and ARM Cortex-A7). The reason for choosing this hardware platform is that it supports an integrated GPU (ARM Mali-T628, 6 shader cores) that is compatible with OpenCL 1.2, allowing us to perform General Purpose Computing on its GPU. The GPU offers many interesting differences compared to standard high-end GPUs, such as individual program counters for each thread, the lack of local memory, as well as a shared device memory between the GPU and CPU (2GB). The device also supports a high speed Ethernet port, making it a good candidate for a high speed NIDS.

Name	Details
SmallFlows	Appneta sample, 9.4 MB data, 1209 flows over a 5 minute duration.
BigFlows	Appneta sample, 368 MB data, 40686 flows over a 5 minute duration.
ISCX12 131	The first 1 million packets from ISCX2012 on 13 of June, 634 MB of data from a data set that includes activity from network infiltration.
ISCX12 121	The first 1.5 million packets from ISCX2012 on 12 of June, 1.01 GB of data from a data set without malicious activity.
ISCX12 12 Full	The entire file from ISCX2012 on 12 of June, 4.22 GB of data from a data set without malicious activity.

Table 1. The data sets used throughout the evaluation section.

For a subset of the experiments (c.f. Section 4.4) an almost identical platform is used (Odroid XU3), that, contrary to the XU4, is equipped with energy sensors but with a slower network card.

The Odroid would most likely be counted as quite powerful for consumer IoT in the home, but its cost could be motivated for professional settings for industrial IoT, especially if the node can run several functions for the network. Moreover, accounting for the recent trends of development of the hardware (i.e. Raspberry Pie 3), it is likely that these devices will also be common in the consumer space.

Realistic Traffic Traces: We use publicly available data sets that capture a realistic behaviour of network traffic for the experiments in this paper. Five different capture files are used, as shown in Table 1. The first two traffic traces (hereby named *SmallFlows* and *BigFlows*) come from Appneta [2], the current developers of Tcpreplay. SmallFlows is a synthetic capture representing a combination of different applications and BigFlows is a capture of real traffic from a busy private network.

The other capture files come from ISCXIDS2012 [16,15]. These data sets are specifically designed to simulate real traffic in order to test and evaluate IDSs. These capture files are larger, ranging from just a few up to several gigabytes. As all capture files are publicly available, they form a repeatable baseline.

Rule sets: Unless otherwise stated, we use the 829 rules (each rule containing at least one pattern) that are enabled by default in Snort's community distribution. In Section 4.2, we experiment with bigger sets of rules.

Metrics: First, we measure the *throughput*: how much traffic is processed per unit of time (Section 4.2). We then measure the *percentage of received packets that are analyzed* by the NIDS (either Snort or *CLort*), when capturing live traffic from the network interface (Section 4.3). We also measure the *power consumption* (important consideration for IoT devices): what is the power consumption of different hardware components when processing incoming traffic (Section 4.4).
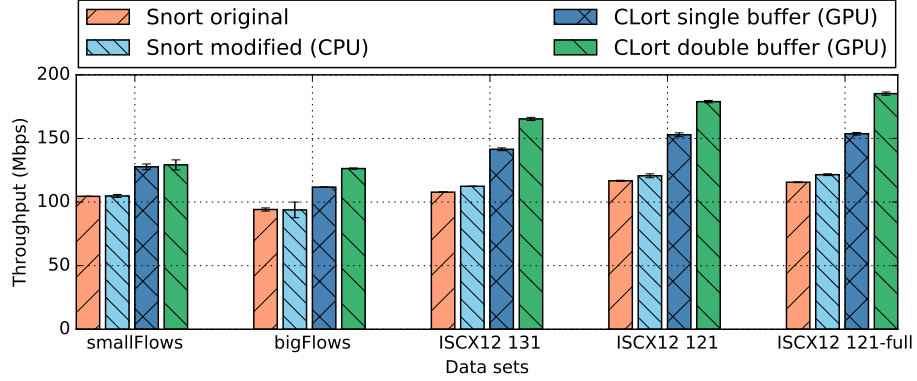
(a) The overall throughput of *CLort*, across different data sets.
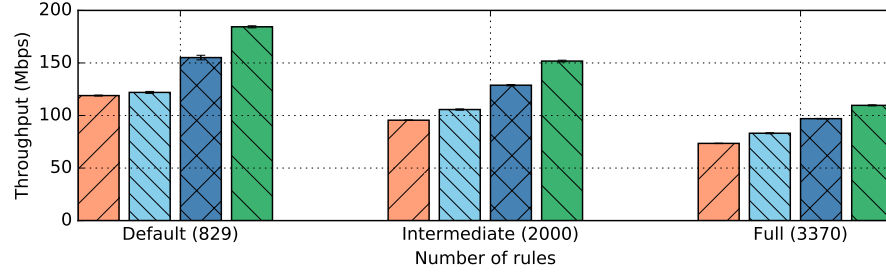
(b) The overall throughput of *CLort*, for different numbers of rules

Fig. 2. Throughput evaluation of *CLort* across different (a) data sets and (b) number of rules.

4.2 Evaluating Throughput

The first set of experiments focus on *throughput*, by varying the traffic to be analyzed as well as the number of rules in Snort.

Overall throughput: Figure 2a presents the processing throughput across different data sets, where we measure the complete execution of Snort (Section 2.1) by reading the pcap files from disk. In these experiments, we use the default number of rules (829 rules). The experiments were repeated 5 times and we report the average and the standard deviation of the measured throughput across all 5 runs.

First, both *CLort* versions that use the GPU consistently outperform the CPU versions across all data sets in our experiments, suggesting that the GPU is capable of accelerating the task of pattern matching. We achieve up to 52% higher throughput compared to the CPU (modified) version of Snort, which is significant, considering that: (i) we only offload pattern matching (step iv) from

Section 2.1, while the other steps of Snort's processing (steps i-iii) are still part of the measured time and (ii) we achieve it using resources (the embedded GPU) that are already available on the platform.

Second, in almost all cases, the double buffering technique provides a performance boost (up to 20%) compared to the single buffer approach. This means that the double buffer optimization successfully overlaps the CPU and GPU execution, keeping both processing units busy with useful work.

Varying the number of rules: By changing the number of rules, we can determine how it affects the Snort runtime performance for scenarios with more rules than the default community rule set (baseline, 829 rules). We enable all available rules that contain fixed string patterns (3370 rules) and also create an intermediate set with 2000 randomly chosen rules.

We run the experiments with several pcap files from Table 1, but only include the ISCX12 121 data set as the results were similar across all runs. In Figure 2b, both *CLort* versions that utilize the GPU continue to outperform the CPU versions of Snort. Increasing the number rules reduces the raw throughput of all versions as expected since the state machines grow larger and there is extra processing work for the rest of Snort's pipeline. In the case of the full rule set, we see that the relative speedup achieved by *CLort* is smaller. This is because many of the extra rules introduce processing that is not related to the search engine that we parallelize (e.g. many of the rules involve regular expression matching).

4.3   Sniffing the network

The experiments in Section 4.2 show that *CLort* has a higher processing throughput when reading packets from a capture file. In this section, we test the performance of *CLort* in a setting much closer to the way a NIDS is deployed in practice by capturing traffic directly from the network.

The experimental setup is the following. We connect the Odroid XU4, running *CLort*, to the span port of a switch (HP V1910-24G). As such, it sees all traffic on the network segment handled by the switch. We then use a laptop (MacBook Pro '14) to replay the pcap files from the ISCX12 131 data set using *tcpreplay* at different speeds. Also, versions of Snort and *CLort* use the default set of 829 rules. The network segment also contains a *dhcp* server, so there is spurious minimal traffic in addition to the traffic being replayed by the laptop.

There are several potential bottlenecks in the system: the hardware replaying the pcap file, the switch handling the span port, the network card of the Odroid in promiscuous mode, the kernel processing before handing the packets to the NIDS, and finally the NIDS's pipeline. To exclude problems beyond our improvements of Snort, we measure the ratio between the packets that are received by the NIDS and the ones that the NIDS successfully analyzes.

Figure 3 shows the percentage of the received packets that *CLort* and Snort manage to analyze at various traffic rates. After approximately 70Mbps, all versions start dropping packets. However, both versions of *CLort* are able to process
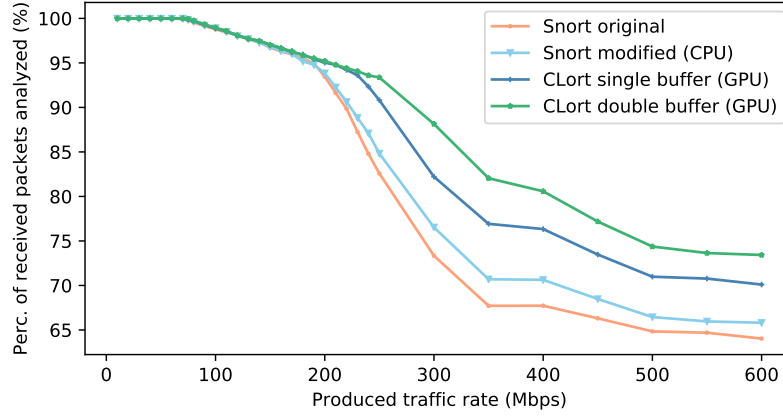
Fig. 3. Percentage of the received packets that *CLort* managed to analyze, as we increase the rate at which we replay traffic.

a larger portion of the received packets, up to 12% more than the modified CPU version of Snort. These results show that the throughput gained from using the GPU translates to *CLort* being able to handle more packets than its CPU counterpart.

4.4 Evaluating Energy Consumption

The final part of the evaluation studies the energy consumption. The ODROID-XU4 is unfortunately not equipped with power measuring sensors. For this reason, we use an older version (ODROID-XU3 [11]) for the energy consumption experiments. The ODROID-XU3 is equipped with the same processor setup as well as the *same GPU and CPU* as the ODROID-XU4. The only significant difference (for the power consumption tests) between these two hardware systems is that the network card is slower for the XU3 (100 Mbps instead of 1Gbps), but the RAM speed and the memory bandwidth is faster. The RAM speed of the ODROID-XU3 is 933Mhz and the memory bandwidth is 14.9GB/s, whereas the RAM speed of the ODROID-XU4 is 750Mhz and the memory bandwidth is 12GB/s.

We measured the power consumption of the following three components: CPU (A15), GPU and RAM memory with a sample rate of 100 samples/second running the ISC12 121 data set using the default number of rules. Figure 4 summarizes the results, with one graph each for the CPU, GPU, RAM, along with the total power consumption. Note that each sub-figure uses its own scale on the y-axis.

As expected, looking at Figure 4a (the power consumption of the GPU in isolation), we can see that only the GPU versions consume any power on the GPU, while the CPU versions consume little to no power on the GPU. The
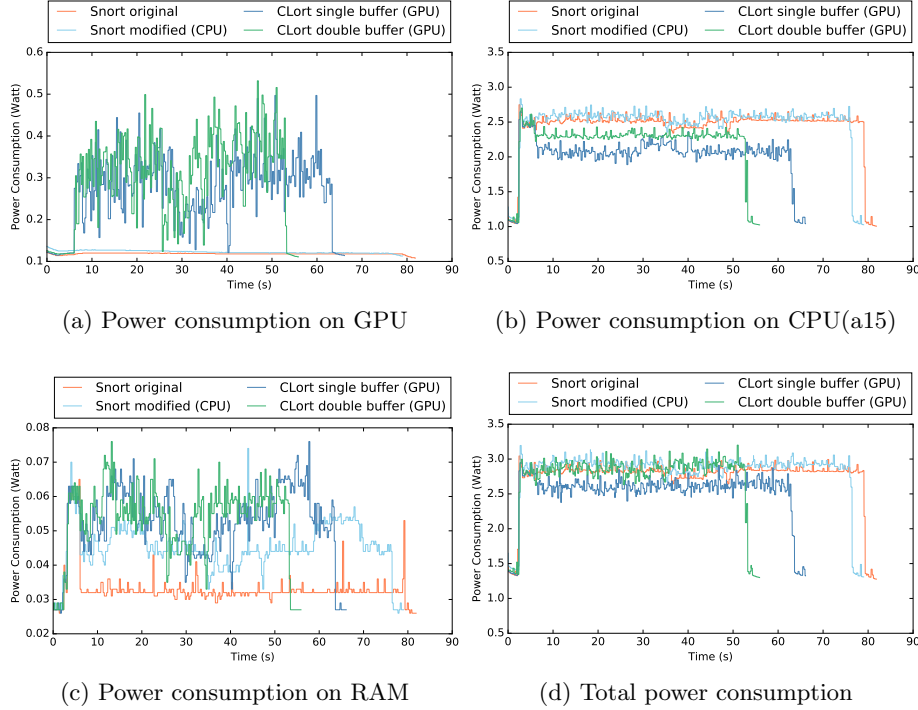
(a) Power consumption on GPU

(b) Power consumption on CPU(a15)

(c) Power consumption on RAM

(d) Total power consumption

Fig. 4. Power consumption measurements of the CPUs, GPU and RAM.

double buffer version of *CLort* consumes slightly more power than the single buffer, but for a shorter period of time.

The power consumption of the CPU (A15) in Figure 4b shows that the CPU versions are almost equal in their execution time and they consume the most power. The GPU versions utilize the CPU less; since the pattern matching has been offloaded to the GPU, it leads to lower power consumption on the CPU. The single buffer version consumes the least CPU power on average between the different versions (close to 2 Watt) but runs longer than the double buffer version.

Figure 4c shows the power consumed by the memory, where the range on the y-axis is very small compared to the other components. In general, the memory is responsible for only a small part of the power draw in all versions, never more than 0.08 W. Notice that the original version of Snort consumes the least amount of power on average. This is because all other versions include extra memory operations to read and write packet data to the buffers.

Figure 4d shows the total, aggregated power consumption from the different components. Overall, the CPU versions of Snort and the double buffer version of *CLort* have almost the same average power draw, though the double buffer ver-

Version	Average Power (W)	Total Energy Consumed (Joule)
CLort GPU (double)	2.87	**145.7**
CLort GPU (single)	**2.63**	159.4
Snort CPU (modified)	2.91	215.6
Snort CPU (original)	2.83	217.5

Table 2. Average power draw and total energy consumed for each version.

sion has a much shorter execution time. The single buffer GPU version consumes the least amount of power (2.63 Watt on average).

Table 2 summarizes the average power consumption, along with the total energy consumed during the execution time of each version. The single buffer version of *CLort* consumes 9.8% less power on average than the CPU version making it a better fit for scenarios where the power envelope is limited. On the contrary, the double buffer version of *CLort* consumes less energy in total (32.4% less than the CPU), since it is able to process traffic faster. This, and in conjunction with the results from Section 4.3 makes it an appealing alternative for scenarios where the traffic load is high and the total consumed energy must be minimized.

5 Related Work

Below we discuss related work, divided into two lines of work: NIDS on high-end systems with GPUs and then NIDS on devices typical of IoT.

5.1 NIDS on GPUs

Over the years, significant efforts have focused on accelerating the functions of a NIDS using high-end, desktop GPUs. The seminal work by Jacob et al. [6] was the first to offload the pattern matching on the GPU. Due to the lack of general-purpose GPU programming APIs at the time, they used graphics libraries (OpenGL). Their prototype, PixelSnort, achieved at best a 40% increase in performance when the CPU was under high load, but with no noticeable performance gain under normal load. Moreover, their pattern matching algorithm is based on the Boyer-Moore algorithm [3], which evaluates each pattern individually, making it hard to scale for a large number of patterns.

More recent work takes advantage of the ease of programming and performance offered by general purpose APIs such as OpenCL and CUDA. Vasiliadis et al. [18] use CUDA and implement the Aho-Corasick algorithm to offload pattern matching and Xie et al. [20] use OpenCL to implement a modified version of Aho-Corasick (PFAC [9]). Apart from differences with our design, both of these works target high-end GPUs, while we focus on resource-constrained, embedded GPUs that share resources with the CPU (memory).

Another, interesting line of work focuses on how to make efficient use of all the computing devices in the system and orchestrate the processing between the

CPU and the GPU. Vasiliadis et al. [19] present Midea, a system based on Snort that makes use of highly parallel CPUs, multiple GPU devices and networks cards. They also describe different optimization techniques to alleviate bottlenecks, due to data transfers and synchronization. Jamshed et al. [7] present Kargus, a similar, highly parallel system based on their own, custom IDS. Recently, Papadogiannaki et al. [13] presented a scheduler that dynamically distributes the packet processing workload across a system with heterogeneous hardware resources (including both discrete and integrated GPUs). Finally, Go et al. [4] also show that integrated GPUs are a cost-effective alternative for packet processing. All the above-mentioned work achieve very high processing throughput using high-end CPUs and GPUs and target large-scale networks or even backbone traffic. Contrary, we focus on resource-constrained devices that better fit the area of IoT networks.

5.2   NIDS on IoT related devices

Security for IoT and resource constrained devices is an active research topic. A project that examines the feasibility of using Snort for resource-constrained devices, similar to the spirit of this work, is RPiDS by Sforzin et al. [14]. In this work, a Raspberry Pi 2 running Snort to function as a portable IDS was thoroughly tested to evaluate the capacity of modern single-board-computers. The measurements showed that the Raspberry Pi could run Snort without ever filling its entire memory capacity. These results strengthen the argument that single-board-computers are a reasonable choice for security in future IoT networks, especially since it is expected that hardware improves with time. However, when the authors experimented with live traffic they reported that there are packet losses, even at low rates, which we also confirm in our experiments (Section 4.3). This raises interesting questions on the bottlenecks involved in the system that cause these losses. In this work, we take one step further and show how more hardware feature available at these devices (e.g. the GPU) can be used to improve the feasibility of a NIDS on resource-constrained devices and reduce the above-mentioned packet losses.

Moving to even more low-end devices and cyber-physical systems, a large body of work focuses on custom IDS that are tailored to the functionality of such devices. One such example is Tabrizi et al. [17] that present a software tool, which produces a customized IDS based on the memory capacity of the targeted device. Given the user-defined security coverage functions, the security properties of the system and memory requirements, the tool can produce an IDS customized to operate on the specified system. The authors were able to produce an IDS, tailored for an electrical smart meter, that operated on 4MB of memory. However, different from this work, they propose an anomaly-based IDS and their main focus is on minimizing memory consumption for low-end devices.

6 Conclusion

In this paper, we consider the security of the Internet-of-Things and address the processing challenges that are part of Network Intrusion Detection Systems. Specifically, we propose *CLort*, a system based on the latest release of Snort (version 3.0) that is designed to tackle the processing needs of NIDS for high-end IoT devices by offloading pattern matching to a GPU. We describe the system design and the effects of various optimizations, such as a double-buffering technique.

We thoroughly evaluate the performance of *CLort* under realistic traffic and show that by using the GPU: (i) *CLort* achieves up to 52% faster processing throughput than Snort (ii) is able to process up to 12% more packets from the network interface under high load and, (iii) achieves the above while consuming 32% less energy than its CPU counterpart.

The work in this paper suggests that using the GPU capabilities offered by modern, high-end IoT devices is an appealing alternative that strengthens security by alleviating the processing bottlenecks of security countermeasures, such as network intrusion detection. The source code of *CLort* is available at `https://github.com/Arklights/Master`

Acknowledgements

References

1. Aho, A.V., Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search. Commun. ACM **18**(6), 333–340 (Jun 1975). https://doi.org/10.1145/360825.360855
2. Appneta: Sample captures, `http://tcpreplay.appneta.com/wiki/captures.html/` [Accessed: 2018-09-18]
3. Boyer, R.S., Moore, J.S.: A Fast String Searching Algorithm. Commun. ACM **20**(10), 762–772 (Oct 1977). https://doi.org/10.1145/359842.359859
4. Go, Y., Jamshed, M.A., Moon, Y., Hwang, C., Park, K.: Apunet: Revitalizing GPU as packet processing accelerator. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). pp. 83–96. USENIX Association, Boston, MA (2017)
5. GPGPU: General-Purpose Computation on Graphics Hardware, `http://gpgpu.org` [Accessed: 2018-07-19]
6. Jacob, N., Brodley, C.: Offloading IDS Computation to the GPU. In: 22nd Annual Computer Security Applications Conference (ACSAC'06). pp. 371–380 (Dec 2006). https://doi.org/10.1109/ACSAC.2006.35

7. Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K.: Kargus: A Highly-scalable Software-based Intrusion Detection System. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 317–328. CCS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2382196.2382232

8. Khronos group: OpenCL Overview, https://www.khronos.org/opencl/ [Accessed: 2018-07-19]

9. Lin, C.H., Liu, C.H., Chien, L.S., Chang, S.C.: Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. IEEE Transactions on Computers **62**(10), 1906–1916 (Oct 2013). https://doi.org/10.1109/TC.2012.254

10. NVIDIA: About CUDA, https://developer.nvidia.com/about-cuda [Accessed: 2018-07-19]

11. ODROID-XU3: ODROID-XU3, http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127 [Accessed: 2018-06-08]

12. ODROID-XU4: ODROID-XU4 User Manual, https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf [Accessed: 2018-03-28]

13. Papadogiannaki, E., Koromilas, L., Vasiliadis, G., Ioannidis, S.: Efficient software packet processing on heterogeneous and asymmetric hardware architectures. IEEE/ACM Transactions on Networking **25**(3), 1593–1606 (June 2017). https://doi.org/10.1109/TNET.2016.2642338

14. Sforzin, A., Mármol, F.G., Conti, M., Bohli, J.: RPiDS: Raspberry Pi IDS - A Fruitful Intrusion Detection System for IoT. In: 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016. pp. 440–448 (2016). https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0080

15. Shiravi, A., Shiravi, H., Tavallaee, M., Ghorbani, A.A.: Intrusion detection evaluation dataset (ISCXIDS2012), http://www.unb.ca/cic/datasets/ids.html [Accessed: 2018-05-08]

16. Shiravi, A., Shiravi, H., Tavallaee, M., Ghorbani, A.A.: Toward developing a systematic approach to generate benchmark datasets for intrusion detection. Computers & Security **31**(3), pp. 357–374 (2012). https://doi.org/https://doi.org/10.1016/j.cose.2011.12.012

17. Tabrizi, F.M., Pattabiraman, K.: Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems. In: 2015 11th European Dependable Computing Conference (EDCC). pp. 1–12. IEEE (Sept 2015). https://doi.org/10.1109/EDCC.2015.17

18. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In: Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings. pp. 116–134. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87403-4_7

19. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Midea: A multi-parallel intrusion detection architecture. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS '11, ACM, New York, NY, USA (2011)

20. Xie, H., Xiang, Y., Chen, C.: Parallel Design and Performance Optimization based on OpenCL Snort. In: Proceedings of the 2017 2nd Joint International Information Technology, Mechanical and Electronic Engineering Conference, JIMEC (2017)