THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Parallel and Distributed Processing in the Context of Fog Computing

High Throughput Pattern Matching and Distributed Monitoring

CHARALAMPOS STYLIANOPOULOS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

Parallel and Distributed Processing in the Context of Fog Computing

High Throughput Pattern Matching and Distributed Monitoring

Charalampos Stylianopoulos

Technical report 179L
ISSN 1652-876X
Department of Computer Science and Engineering

Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: chasty@chalmers.se

Printed by Chalmers Reproservice
Göteborg, Sweden 2019

Parallel and Distributed Processing in the Context of Fog Computing

High Throughput Pattern Matching and Distributed Monitoring

Charalampos Stylianopoulos

Department of Computer Science and Engineering, Chalmers University of Technology

ABSTRACT

With the introduction of the Internet of Things (IoT), physical objects now have cyber counterparts that create and communicate data. Extracting valuable information from that data requires timely and accurate processing, which calls for more efficient, distributed approaches. In order to address this challenge, the fog computing approach has been suggested as an extension to cloud processing. Fog builds on the opportunity to distribute computation to a wider range of possible platforms: data processing can happen at high-end servers in the cloud, at intermediate nodes where the data is aggregated, as well as at the resource-constrained devices that produce the data in the first place.

In this work, we focus on efficient utilization of the diverse hardware resources found in the fog and identify and address challenges in computation and communication. To this end, we target two applications that are representative examples of the processing involved across a wide spectrum of computing platforms. First, we address the need for high throughput processing of the increasing network traffic produced by IoT networks. Specifically, we target the processing involved in security applications and develop a new, data parallel algorithm for pattern matching at high rates. We target the vectorization capabilities found in modern, high-end architectures and show how cache locality and data parallelism can achieve up to *three* times higher processing throughput than the state of the art. Second, we focus on the processing involved close to the sources of data. We target the problem of continuously monitoring sensor streams —a basic building block for many IoT applications. We show how distributed and communication-efficient monitoring algorithms can fit in real IoT devices and give insights of their behavior in conjunction with the underlying network stack.

Keywords: fog computing, resource-constrained devices, high throughput, pattern matching, vectorization, distributed processing, distributed monitoring

Preface

Parts of the contributions presented in this thesis have led to the following manuscripts.

- **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou,
  "Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization",
  appeared in the *46th International Conference on Parallel Processing (ICPP)*, Bristol, United Kingdom, 14-17 August, 2017
- **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou,
  "Geometric Monitoring in Action: a Systems Perspective for the Internet of Things",
  under submission, 2018

Acknowledgments

First, I would like to express my gratitude to my supervisors Magnus Almgren, Olaf Landsiedel and Marina Papatriantafilou. I feel privileged to have the continuous support and guidance of three excellent mentors. Without them, this thesis would not be possible.

I am also grateful to my past and current colleagues in the Division of Networks and Systems, for making these first years of my PhD studies a fun and rewarding experience. Many thanks to Adones, Ali, Aljoscha, Amir, Aras, Bapi, Bastian, Bei, Beshr, Boel, Carlo, Christos, Dimitris, Elad, Elena, Fazeleh, Georgia, Hannah, Yiannis N., Iosif, Ivan, Joris, Karl, Katerina, Nasser, Oliver, Paul, Philippas, Romaric, Thomas R., Thomas P., Tomas, Valentin T., Valentin P., Vincenzo, Vladimir and Wissam. Also, thanks to my collaborators Oskar, Linus and Simon.

Special thanks go to my friends in Göteborg that made it a lovely place to live in. Thank you Petros, Vaggelis, Stavros, Yiannis S., Angelos, Chloe, Maria, Vasiliki, Manos and Suvi for enduring my endless talking. Also, many thanks to old friends, especially Nikos, Kostas, Efi, Elias and Dinos, for all the great times we had and will keep having.

My thanks go to my family, my parents and my siblings for their unconditional love and trust in me. I owe it all to them. Finally, I want to thank Kelly for being there for me and for all the moments we share together.

Charalampos Stylianopoulos
Göteborg, June 2019

Contents

List of Figures

Part I

INTRODUCTION

1

Introduction

The last decade has been marked with many disruptive technologies where the Internet of Things (IoT) is one of the most well known. An increasing number of objects are now becoming “smart” by getting sensing and networking capabilities. The number of devices that are expected to be connected to the internet in the coming years is impressive [1], including everyday objects, cars, as well as devices that are part of the electricity grid and the industry [2]. However, what is even more impressive is the amount of data that they will produce. The upcoming challenge associated with all that data is now this: how, when and where to process the high volumes of data, in order to extract value [3] [4]?

1.1 Challenges in processing IoT data: the need for scalability on a range of new architectures

The usefulness of IoT computing comes from the fact that small devices and sensors are able to continuously provide readings about their state and the state of their environment. That data then needs to be processed to provide valuable information about the system, and often to create control commands that are fed back to the devices. IoT devices are typically resource-constrained nodes, sometimes equipped with only enough computational power to acquire readings and send them onwards. In order to accommodate the heavy processing required for many IoT applications, IoT networks rely on a connection with the cloud, where the hardware resources are abundant. This computing model, however, has scalability problems and fails to meet the requirements of many IoT applications. Primarily, the vast number of connected devices, in conjunction with the increasing volume of data generated by each device, means that it is impossible to send all this data to the cloud without exhausting the available bandwidth. Moreover, IoT applications often have tight latency requirements, which means that aggregating data to the cloud, processing it and sending back the control commands adds an unacceptably high overhead.

As a way to address these limitations, fog computing[1] has been proposed as an alternative extension to cloud computing described above [5]. The core idea of fog computing is to move the processing to where it is most needed, closer to the data origin. In this paradigm, the processing and control logic that would typically be found on cloud servers is now pushed down to intermediate nodes, closer to the sources of data. Base stations and gateways will thus be enhanced with processing capabilities and storage, enough to quickly act on aggregated data coming from IoT networks. Moreover, the networks themselves will take over some of the processing and control logic, so that a significant portion of data do not have to be forwarded upwards. It is estimated that this way, more

[1] The term fog computing is often used interchangeably with edge computing, with the later focusing more on processing at the devices that produce the data. We will use the term fog computing from now on, to emphasize the need for distributed processing.

than 40 percent of IoT data [6] will be processed on devices physically close to the data sources, by 2019. Hence, the up-link network bandwidth will no longer a limiting factor and there is opportunity to significantly reduce the overall processing latency.

The introduction of fog computing opens new, interesting research questions in several ways. On one hand, fog computing comes with a new set of challenges, that mainly revolve around the problems of (i) how to distribute computational tasks on the layers of fog [7], (ii) how to move them there, (iii) in what ways the different components of fog interact and connect with each other [3] and (iv) how to maintain Quality of Service [8]. On the other hand—and this is important in the context of this thesis—fog computing brings together processing applications with different requirements that target different platforms, under the same computing approach. Processing methods originally designed for servers in the cloud now also become relevant at the intermediate layer of fog and must adapt to the hardware found there. At the same time, computational tasks designed to operate on aggregated data on a single node can benefit if the processing logic is made distributed and handled *close to*, or even *by* the nodes that produce that data. This increases the design space of existing solutions and poses interesting research questions along the whole spectrum of processing involved in fog computing.

1.2 Background

This section outlines background topics that are relevant in the context of this thesis. We start by discussing relevant advances in hardware, followed by background information on wireless sensor communication.

1.2.1 Advances in processing hardware

Commodity hardware is constantly changing and evolving. Every new generation of platforms is enriched with new hardware features, designed to better serve the needs of applications. The introduction of such features enables new

techniques that applications can use to gain performance.

A characteristic example of such important techniques is vectorization. It utilizes processing units that operate on a vector of elements simultaneously, instead of separate elements at a time, in a Single Instruction Multiple Data (SIMD) manner [9, 10]. SIMD vectorization is traditionally used in computationally intensive, number-crunching applications, where computation is performed on independent data, *sequentially* stored in memory.

Vectorization has been available on commodity hardware for many years. Recent advances in hardware platforms have made it relevant again for the following reasons: (i) new vector instruction sets have introduced the *gather instruction* [11] that allows accessing data from *non-contiguous* memory locations and (ii) modern processor designs are shifting towards new architectures with more emphasis on vectorization. As an example Intel's Xeon Phi [12] supports 512 bit vector registers. On those platforms, vectorization is not just an option but a must, in order to achieve high performance [13].

Along with the introduction of new hardware techniques, existing features become more widespread and are adopted by a wider range of platforms. As an example, even embedded devices [14] are now massively parallel and support programmable Graphics Processor Units (GPUs), allowing some of the techniques found on high-end servers to be used on those devices as well.

1.2.2 Wireless sensors networks

Considering the lower layer of fog computing, the hardware found there has substantially different characteristics compared to the high-end servers described in the previous section. Typical devices at this layer are small, battery powered sensors. The main hardware components on these nodes are usually: (i) a set of simple sensors that periodically collect data from the environment (ii) a resource constrained microcontroller unit (MCU) for simple data and packet processing, and (iii) a radio transceiver for communication with other nodes that is used to form networks (either structured or mesh).

In wireless sensors networks (WSN), the battery lifetime is the most valu-

able resource. Thus the design of hardware and software for wireless sensors emphasizes on minimizing the energy footprint. As a consequence, applications on wireless sensors need to deal with the fact that processing components are very simple and resource constrained.

In addition to computation, communication in WSN is also expensive. In fact, the radio is the most energy-hungry component, often consuming up to 10 times more energy than the MCU. For this reason, the goal of most communication protocols is *radio duty cycling* (RDC), where the radio is kept off as much as possible and is turned on for only a small fraction of the total time. A simple and commonly used RDC policy is to turn the radio on a fixed number of times per second, called the *channel check rate* (CCR) [15]. A node that wants to transmit, will keep transmitting for a duration of at least $1/CCR$ seconds to ensure that all neighbouring nodes had a chance to turn their radio on and receive. The channel check rate is a tunable parameter of the protocol that directly affects the battery lifetime of the nodes.

In addition to energy reservation, latency and reliability are other important considerations in WSNs. Sensors usually form multi-hop networks over unreliable and lossy links that are constantly subject to interference. For this reason, a large body of WSN research has focused on how to design reliable and low-latency network protocols, with interesting new advances over the last few years [16–18].

1.3 Scope of the thesis

This thesis covers aspects of processing and communication in fog, through representative applications that pose challenges stemming from two key problems: (i) *hardware diversity* and (ii) *data diversity*.

Hardware diversity is important since processing can happen on a wide and heterogeneous range of platforms, from resource-constrained embedded devices to high-end servers. On each of these platforms, understanding and making efficient use of the new architectures' features enable new possibilities. Part of the work in this thesis focuses on how to make efficient use of new

architectures and features found on these platforms to enable new processing methods.

Data diversity ranges from high-rate data aggregated to a single point, to lower-rate data from many sources, where processing needs to happen in a distributed manner. Applications that operate on accumulated data usually require high throughput processing rates, while distributed applications require minimal, energy efficient communication and distributed control.

We use two applications that instantiate the problems just described and highlight challenges and solutions. Those applications span across the two ends of the spectrum of fog computing: processing for security applications close to the cloud and distributed monitoring of sensor readings inside IoT networks. These applications target different hardware and address different challenges and they become relevant in the context of fog computing, as part of the same processing pipeline. We return to these applications in the next section.

Figure 1.1 shows a conceptual representation of a fog computing architecture. The two problem domains that this thesis focuses on are shown in this figure.

1.4 Representative problems in the context of fog computing

This section introduces the two problem domains that are used as a basis for the work in this thesis. We summarize the challenges in this section and elaborate further on the contributions in each domain in Section 1.5.

1.4.1 Data processing in the context of cyber-security

In the first problem domain, we consider the problems and challenges involved in pattern matching, with a focus on its application for Network Intrusion Detection Systems (NIDS).

Motivation. NIDSs are typically found at the entry point of networks and their purpose is to analyze the incoming and outgoing network traffic to detect

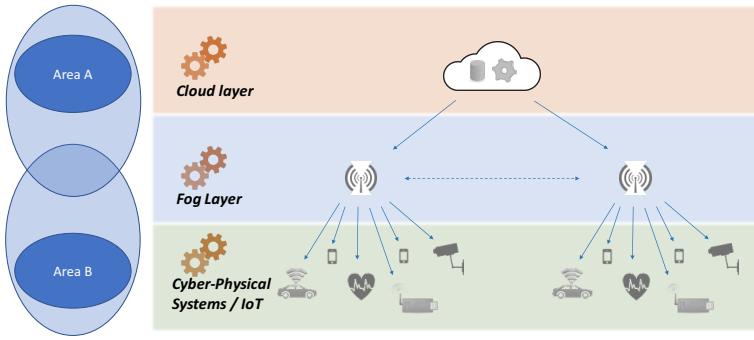


Figure 1.1: *An example fog computing architecture that includes a cloud layer with high-end servers, an intermediate layer close to the gateways and the IoT layer with resource-constrained devices. The work in this thesis targets representative problems on Area A (centralized, high throughput processing) and Area B (distributed processing and communication).*

any malicious behavior, ranging from unauthorized access, malware that exploit software vulnerabilities, data exfiltration etc. They typically employ sophisticated analysis that considers not only the packet headers but also the contents of each packet (deep packet inspection [19]). There are many available NIDSs, with Snort [20] and Bro [21] being some of the most popular and mature in the open source community.

Network Intrusion Detection Systems gain new significance in the context of fog computing. IoT networks are connected to the Internet, sending sensor readings upwards towards the cloud and receiving back control traffic. The end-devices producing sensitive data are potential attack targets. However, since they are typically resource-constrained, traditional security mechanisms cannot be employed on them. Hence, it is important to add protection mechanisms, both at the entry point of the network and along the data path towards the cloud.

Challenges. An essential building block of many such systems is *pattern matching*, i.e., to discover if any of many predefined patterns exist in an input stream (*multiple* pattern matching), for whitelisting or blacklisting. [2] Consid-

[2] Apart from its role in intrusion detection, pattern matching is also a core function in many other

ering the processing involved in NIDSs, pattern matching is the most computationally intensive part and represents a major performance bottleneck. More than 70% of the running time of a NIDS is spent on pattern matching [25, 26]. This fact, in conjunction with the ever increasing rates of traffic that needs to be processed, pushes the performance of NIDSs to their limits. Achieving high pattern matching throughput is challenging yet crucial for these systems: if the processing throughput cannot match the incoming traffic rate, the system will have to start dropping packets and maybe miss significant attacks.

Related work. Pattern matching has been an active field of research for many years and there are many proposed approaches. The algorithm proposed by Aho and Corasick [27] is one of the most well known and the one currently used by Snort. The first step of Aho-Corasick is to create a finite-state automaton from the malicious patterns. Then, the algorithm scans the input traffic byte-by-byte to traverse the automaton, until it arrives at a final state that indicates the detection of an attack. Even though Aho-Corasick performs only a small number of operations per byte, it fails to perform well in practice, due to poor cache locality.

State of the art approaches have been proposed to address the limitations of Aho-Corasick. A family of algorithms in the literature replace the state machine of Aho-Corasick with filters. Choi et al. [22] use a series of succinct filters, created using a small part of each malicious pattern. In this way, most of the benign input traffic is quickly filtered out, using cache-resident data structures. The part of the input that matches the information in the filters is further examined in a later verification phase that involves lookups in hash tables that contain the actual patterns. Similarly, Moraru et al. [28] use a modification of Bloom Filters [29] to scan both the input and the subset of patterns that are relevant.

Open problems. Even though the state of the art approaches have substantially increased the achieved throughput, they perform sub-optimally in modern architectures, because they fail to make use of the new characteristics and features available. As an example, most pattern matching algorithms do not make

tasks, such as virus detection [22], text search [23] and genome analysis [24].

use of the vector execution units and leave them underutilized. In this thesis we present techniques that allow us to make the most out of the available hardware and achieve considerable speedups in throughput. A summary of this work follows in Section 1.5.2.

1.4.2 Distributed monitoring of sensor readings

In the second problem domain, considering the computation involved in resource-constrained IoT networks, we focus on the important problem of distributed monitoring of sensor readings.

Motivation. We address the issue of continuously monitoring a distributed set of sensor values and keeping track of a function of interest, defined over the network-wide aggregate of these values. Often, the goal is to always be able to detect whether the value of the monitored function has exceeded a predefined threshold. Keeping track of such a function is a basic building block for many IoT applications and control loops, e.g. for detecting outliers [30], hot-spots [31] or denial-of-service attacks [32].

Monitoring sensor values is a prime example of the applications that fog computing is designed for. The need for timely monitoring and low latency detection of a threshold violation calls for local processing, close to the sources of data. Ideally, the monitoring logic can even be placed inside the IoT network and distributed to the sensor nodes themselves.

Challenges. Keeping track of a function defined over a network-wide aggregate is a challenging task in practice. A simple solution is to aggregate every reading from every node in the network to a central entity and compute the aggregate there. Such an approach is impractical in networks with battery-constrained devices: using the radio for transmission or reception is the single most expensive operation in terms of energy [33]. Thus, the challenge associated with this problem is to reduce the number of sensor readings that need to be transmitted, by letting all nodes locally determine whether a reading should be transmitted. However, finding such local criteria is challenging when the function to monitor is non-linear (e.g. the variance of the readings), yet it is

non-linear functions that are particularly interesting for many real-world applications (e.g. detecting a denial-of-service attack).

Related work. Sharfman et al. [34] proposed a general method called *geometric monitoring (GM)* that can monitor any function (linear or not) defined over the average of network readings and keep track of its value with respect to a threshold. Every node in this algorithm is capable of deriving constraints on its local values and avoid communication as long as those constraints are not violated. The GM method has been extended with sketches [35] and prediction models [36] and has been applied to outlier detection [30] and data stream queries [37].

Open problems. Apart from the existing general analysis of algorithms, such as the one described above, the applicability to a real IoT deployment is still unclear, from a practical perspective. Up until now, there are no insights in how the system aspects of IoT networks interact with such algorithms. Specifically, the underlying network stack can have a significant impact on the efficiency of the algorithms, in terms of energy consumption on the nodes, as well as latency and reliability of communication. Moreover, the resource constrained nature of the sensor nodes makes the processing required by the algorithm challenging in practice.

In this thesis we take a step beyond the existing analysis and consider, not just the algorithm in isolation, but also the interplay with system aspects, such as the network stack. A summary of this works follows in Section 1.5.3.

1.5 Research questions and contributions

Based on the challenges discussed above, in this section, we introduce general research questions that have driven the work in this thesis, as well as a summary of the papers included in this thesis that contribute to these questions.

1.5.1 Research questions

This thesis addresses the following research questions that emerge in the context of fog computing:

- **RQ1:** How can new hardware support be used to improve the processing throughput of data, across a wide spectrum of platforms.
- **RQ2:** How can distributed algorithms be used to push computation closer to the sources of data, in order to utilize data locality and make efficient use of the limited resources found there?
- **RQ3:** How do system aspects of the different layers of fog computing influence the design of efficient algorithmic approaches?

RQ1 and *RQ3* become particularly relevant when considering applications that require high throughput processing of large volumes of data, such as pattern matching. The variety of features found in modern platforms (e.g. advanced vector instructions) offers new possibilities for faster processing, but requires novel, hardware-aware algorithmic designs that make efficient use of those features.

RQ2 and *RQ3* have an important role in the context of fog computing, where distributing computation is an important way to ensure scalability, in terms of bandwidth and latency. However, turning a centralized algorithm into a distributed one is challenging, especially when it is applied in a resource constrained setting (e.g. in wireless sensor networks).

We relate back to these research questions and how we address them in this thesis, in the context of the following research contributions.

1.5.2 (Paper I) Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization

In this paper, we introduce V-PATCH, a data-parallel algorithm for pattern matching, that uses vector instructions to process multiple bytes of input, in

parallel. This work builds on the observation that recent, state of the art approaches for pattern matching that rely on quick filtering of the input, have brought the problem close to the processor and achieve good cache locality. As a result, long memory latencies are no longer the dominant bottleneck and the computational part of pattern matching becomes significant. With that in mind, we target that computational part and show how to improve it further, through vectorization (see Section 1.2.1). In this way, we contribute towards *RQ1* and show how to make efficient use of the available hardware features.

We follow a two-step approach. First, we propose a refined and extended filtering strategy that: (i) performs filtering based on cache-resident data structures and is effective for the types of patterns found in real traffic, and (ii) is simple enough to allow efficient vectorization. As an example, we deal separately with small, but frequently found patterns and perform more targeted filtering for longer patterns. Second, we design a vectorized version that uses specialized instructions to parallelize the computation performed on the filters, together with optimizations (e.g. filter merging) that allow us to make the most of the filtering design.

We evaluate the effectiveness of V-PATCH using real malicious patterns from Snort [38], against both real and synthetic traffic mixes. The results on two platforms, an Intel Haswell processor and an Intel Xeon Phi co-processor, show up to 1.8x and 3.6x times faster processing throughput respectively, against the state of the art. Furthermore, we find that the vectorized approach retains a stable speedup of 1.4x over the scalar one, as the number of malicious patterns increases.

1.5.3 (Paper II) Geometric Monitoring in Action: a Systems Perspective for the Internet of Things

In this paper, we study geometric monitoring from a full-system perspective, when applied on real IoT networks. We propose a system design for geometric monitoring on top of a wireless sensor network stack. Then, we thoroughly evaluate the performance benefits achieved in practice, the run-time behavior of

the algorithm and the effects of packet losses. Through this work, we contribute towards *RQ2* and *RQ3*, by showing that distributing computation to resource-constrained devices can have significant benefits, as well as by evaluating the effect of the system aspects in the performance of the algorithm.

We design the system on top of multi-hop mesh networks, without the need for maintaining a topology. When a node detects a violation, it will trigger a network-wide broadcast and inform every other node of its new value. This is done by network flooding, where a node that receives "new" information will broadcast it further to its neighbors. In the event that a node fails to receive an update by any of its neighbors, that node will be *out of sync* until a subsequent broadcast from the same originator arrives.

Important considerations that are taken into account are the following. First, upon every new measurement or update, nodes need to check their local violation criteria, which typically involves finding whether two curves intersect. This can be computationally challenging for resource-constrained devices and in some cases, it might be impractical to compute it accurately and in time. To this end, we propose a simple relaxation of the violation check that introduces a trade-off between computational efficiency and communication reduction. Second, we investigate the important parameters of the network stack that affect the effectiveness of the algorithm in practice. As we show in the paper, the rate at which nodes wake-up to receive traffic (Channel Check Rate, CCR) greatly affects the energy savings of the GM method.

We evaluate our design using both full-system simulations and real IoT testbeds. Overall, we find that GM brings significant benefits to monitoring tasks, in terms of communication reduction. Specifically, when monitoring the variance and the average of real temperature data, GM achieves 3x and 11x reduction in duty-cycle, respectively. However, those benefits are limited compared to the communication reduction of the algorithm in isolation (4.3x and 44x respectively), due to baseline energy overhead of the network stack. Closer looks into the run-time behavior of the algorithm show that (i) the communication pattern varies greatly, and (ii) packet losses greatly impact the amount of time a node is *out of sync* and reduce the ability of the algorithm to detect

violations in a timely manner.

1.6 Conclusions and Future Directions

Motivated by the emergence of IoT and fog computing, this thesis targets the challenges of processing involved across a range of platforms, by focusing on two representative problems, namely pattern matching and continuous monitoring of sensor readings.

On the problem of pattern matching for Network Intrusion Detection, we focus on how to make efficient use of newly introduced hardware features to improve the processing throughput, thus contributing towards *RQ1* (cf. Section 1.5.1). The work in this thesis shows that, using advanced vector instructions, it is possible to improve the pattern matching throughput, across a range of data sets and different platforms.

On the problem of continuously monitoring distributed sensor readings, the work in thesis contributes insights towards *RQ2*. We demonstrate how geometric monitoring can be used to share the processing logic across the resource-constrained devices of IoT networks and give insights from real deployments. Through this work, important aspects of *RQ3* are also made clear and we find that a) algorithmic engineering is required to adapt to the needs of battery and CPU-constrained devices and we suggest appropriate approximations, and b) the underlying communication stack greatly influences the performance of the algorithm, in terms of energy, latency and accuracy.

The results presented above target challenges and techniques on two separate sides of the fog computing spectrum. The next challenge is how to bring them closer together and extend them across the different layers of fog.

In the context of pattern matching, it is interesting to consider approaches that would better fit in the intermediate layer, for the following reasons. First, the gateways found at the intermediate layer are points of entry into the IoT network. For this reason, fast intrusion detection is required to secure both the gateways themselves as well as the network to which they provide access. Second, performing intrusion detection closer to the sources of data and not

on remote servers, provides opportunities for bandwidth reservation as well as better response times. Finally, the hardware found at the intermediate layer is becoming increasingly powerful and offers new capabilities. As an example, embedded RaspberryPi-like devices come with multiple cores and even programmable GPUs [14], so they can take the role of gateways with support for processing. Hence, it is interesting to see how to make efficient use of new platforms and the hardware resources they have to offer.

Based on insights from the work in distributed monitoring, described above, there are two interesting aspects that are worth further investigation. First, distributed monitoring algorithms can be extended to take into account the reality of IoT communication. As an example, would it be possible to rely less on a network-wide broadcast mechanism and, instead, resolve violations in a local neighborhood? This would bring overall benefits for both the energy savings and the accuracy of the algorithm. In this direction, we are currently looking at ways to extend the existing literature [31] with a distributed algorithm that is less demanding in terms of communication and is not oblivious to the underlying network stack. Second, it is interesting to see how to design a tailored protocol that serves the communication needs of such distributed applications. Recently, advances in wireless sensor protocols (e.g. synchronous transmissions [16] and channel hopping [39]) have made it possible to achieve fast and highly reliable flooding in mesh networks. One can thus investigate to what extent these protocols can serve the needs of distributed monitoring and how they can be extended further.

Bibliography

[1] D. Evans, "The Internet of Things: How the next evolution of the internet is changing everything," Cisco White Paper `https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, January 2011, Accessed: 2018-05-07.

[2] N. Jazdi, "Cyber physical systems in the context of Industry 4.0," in *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, May 2014,

pp. 1–4.

[3] M. Chiang and T. Zhang, "Fog and IoT: An Overview of Research Opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec 2016.

[4] M. Brodie, "Data: The World's Most Valuable Resource," Lectures in Computer Science on Big Data and Applications, July 2017.

[5] Cisco, "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are," White Paper `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf`, 2015, Accessed: 2018-05-07.

[6] IDC FutureScape, "Worldwide Internet of Things 2017 Predictions," `https://www.idc.com/getdoc.jsp?containerId=US40755816`, November 2016, Accessed: 2018-05-07.

[7] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, Nov 2016, pp. 20–26.

[8] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018.

[9] "Intel vectorization tools," `https://software.intel.com/en-us/articles/intel-vectorization-tools`, Accessed: 2016-12-10.

[10] O. Polychroniou and Kenneth A. R., "Vectorized Bloom filters for advanced SIMD processors," in *Proc. of the Tenth Int. Workshop on Data Management on New Hardware*. 2014, DaMoN '14, ACM.

[11] InsideHPC, "Gather Scatter operations," `http://insidehpc.com/2015/05/gather-scatter-operations/`, Accessed: 2016-12-10.

[12] Intel, "Intel Xeon Phi product family," `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`, Accessed: 2016-12-10.

[13] Intel, "The importance of vectorization for Intel Many Integrated Core Architecture (Intel MIC architecture)," `https://software.intel.com/en-us/articles/the-importance-of-vectorization-for-intel-many-integrated-core-architecture-intel-mic`, Accessed: 2016-12-10.

[14] Hardkernel, "Odroid XU4," `http://www.hardkernel.com/main/main.php`, Accessed: 2018-05-07.

[15] A. Dunkels, "The ContikiMac radio duty cycling protocol," Tech. Rep. 2011:13, ISSN 1100-3154, 2011, Swedish Institute of Computer Science.

[16] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, Chicago, IL, USA, April 2011, pp. 73–84.

[17] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: Versatile and Efficient All-to-all Data Sharing and In-network Processing at Scale," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2013, SenSys '13, pp. 1:1–1:14, ACM.

[18] T. Istomin, A. L. Murphy, G. P. Picco, and U. Raza, "Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, New York, NY, USA, 2016, SenSys '16, pp. 83–95, ACM.

[19] P. Lin, Y. Lin, Y. Lai, and T. Lee, "Using string matching for deep packet inspection," *Computer*, vol. 41, no. 4, 2008.

[20] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proc. of the 13th USENIX Conf. on System Administration*, Seattle, Washington, 1999, USENIX Association.

[21] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[22] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 551–565, USENIX Association.

[23] G. Navarro, "NR-grep: a fast and flexible pattern-matching tool," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1265–1312, 2001.

[24] G. Navarro and M. Raffinot, *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.

[25] S. Antonatos, K. Anagnostakis, and E. Markatos, "Generating Realistic Workloads for Network Intrusion Detection Systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, 2004.

[26] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra, "On the statistical distribution of processing times in network intrusion detection," in *2004 43rd IEEE Conf. on Decision and Control (CDC)*, 2004, vol. 1.

[27] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, June 1975.

[28] I. Moraru and D. Andersen, "Exact pattern matching with feed-forward bloom filters," *J. Exp. Algorithmics*, vol. 17, Sept. 2012.

[29] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[30] S. Burdakis and A. Deligiannakis, "Detecting Outliers in Sensor Networks Using the Geometric Approach," in *2012 IEEE 28th International Conference on Data Engineering*, Washington, DC, USA, April 2012, pp. 1108–1119.

[31] I. Sharfman, A. Schuster, and D. Keren, "Aggregate threshold queries in sensor networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, March 2007, pp. 1–10.

[32] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to DDoS attack detection and response," in *Proceedings DARPA Information Survivability Conference and Exposition*, Washington, DC, USA, April 2003, vol. 1, pp. 303–314 vol.1.

[33] A. Barberis, L. Barboni, and M. Valle, "Evaluating energy consumption in wireless sensor networks applications," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 455–462.

[34] I. Sharfman, A. Schuster, and D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2006, SIGMOD '06, pp. 301–312, ACM.

[35] M. Garofalakis, D. Keren, and V. Samoladas, "Sketch-based Geometric Monitoring of Distributed Stream Queries," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 937–948, Aug. 2013.

[36] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster, "Prediction-based Geometric Monitoring over Distributed Data Streams," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2012, SIGMOD '12, pp. 265–276, ACM.

[37] M. Garofalakis, "Approximate Geometric Query Tracking over Distributed Streams," *IEEE Data Eng. Bull.*, vol. 38, no. 3, pp. 103–112, 2015.

[38] Cisco, "Snort Rules and IDS Software Download," `https://www.snort.org/downloads/#rule-downloads`, 2018, Accessed: 2018-05-07.

[39] M. Palattella T. Watteyne and L. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement," RFC 7554, RFC Editor, May 2015.

Part II

PAPERS

PAPER I

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization

Appeared in the 46th International Conference on Parallel Processing (ICPP)
Bristol, United Kingdom
August 16, 2017, pp. 472 - 482

2

PAPER I

Abstract

Pattern matching is a key building block of Intrusion Detection Systems and firewalls, which are deployed nowadays on commodity systems from laptops to massive web servers in the cloud. In fact, pattern matching is one of their most computationally intensive parts and a bottleneck to their performance. In Network Intrusion Detection, for example, pattern matching algorithms handle thousands of patterns and contribute to more than 70% of the total running time of the system.

In this paper, we introduce efficient algorithmic designs for multiple pattern matching which (a) ensure cache locality and (b) utilize modern SIMD instructions. We first identify properties of pattern matching that make it fit for

vectorization and show how to use them in the algorithmic design. Second, we build on an earlier, cache-aware algorithmic design and we show how cache-locality combined with SIMD gather instructions, introduced in 2013 to Intel's family of processors, can be applied to pattern matching. We evaluate our algorithmic design with open data sets of real-world network traffic: Our results on two different platforms, Haswell and Xeon-Phi, show a speedup of 1.8x and 3.6x, respectively, over Direct Filter Classification (DFC), a recently proposed algorithm by Choi et al. for pattern matching exploiting cache locality, and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems.

2.1 Introduction

Security mechanisms, such as Network Intrusion Detection Systems and firewalls, are part of every networked system and are analyzing network traffic to protect from attacks. An essential building block of many such systems is *pattern matching*, i.e., to discover if any of many predefined patterns exist in an input stream (*multiple* pattern matching), for whitelisting or blacklisting. In the context of Network Intrusion Detection, the data stream is the reassembled protocol stream of the *packets* on the monitored network and the set of patterns (usually in the order of thousands) represents *signatures* of malicious attacks that the system aims to detect.

Motivation and Challenges. Pattern matching represents a major performance bottleneck in many security mechanisms, especially when there is a need to employ analysis on the full packet's payload (Deep Packet Inspection). In intrusion detection, for example, more than 70% of the total running time in spent on pattern matching [1, 2]. Moreover, with the increasing interest in Network Function Virtualization (NFV) [3, 4], applications like firewalls and Network Intrusion Detection are now moved into the cloud, where they need to rely on commodity hardware features for performance, like multi-core parallelism and vector processing pipelines.

In this paper, we introduce a vectorizable design of an exact pattern match-

ing algorithm which nearly doubles the performance when compared to the state of the art on modern, SIMD capable commodity hardware, such as Intel's Haswell processors or Xeon Phi [5]. *Vectorization* as a technique to increase throughput is gradually taking a more central role [6]. For example, architectures with SIMD instruction-sets now provide wider vector registers (256 bits with AVX) and introduce new instructions, such as gathers, that make vectorization applicable to a wider range of applications. Moreover, modern processor designs are shifting towards new architectures, like Intel's Xeon Phi [5], that, for example, supports 512 bit vector registers. On those platforms, vectorization is not just an option but a must, in order to achieve high performance [7]. In this work we introduce algorithmic designs to utilize these capabilities.

Approach and Contributions. The introduction of *gathers* and other advanced SIMD instructions (cf. section 2.3) allows even applications with irregular data patterns to gain performance from data parallelism. For example, SIMD can speed up regular expression matching [8–10]. Here, the input is matched against a single regular expression at a time, represented by a finite state machine that can fit in L1 or L2 cache. Working close to the CPU is crucial for these approaches, otherwise the long latency of memory accesses would hide any computation speedup through vectorization.

The domain of multiple pattern matching for Network Intrusion Detection has challenging constraints that limit the effectiveness of these approaches: applications need to simultaneously evaluate thousands of patterns and traditional state-machine-based algorithms, such as Aho-Corasick [11], use big data structures that by far exceed the size of the cache of today's CPUs. The size of the patterns varies greatly (from 1-byte to several hundred byte patterns) and can appear anywhere in the input. That is why SIMD techniques have not been previously considered for exact multiple pattern matching – with a few exceptions discussed in Section 2.6 – for Network Intrusion Detection.

Building upon recent work [12, 13] that take steps in addressing the cache-locality issues for this problem, our approach fills this gap: we propose algorithmic designs for multiple pattern matching that bring together cache locality and modern SIMD instructions, to achieve significant speedups when compared

to the state of the art. Combining cache locality and vectorization introduces new trade-offs on existing algorithms. Compared to traditional approaches that perform the minimum required number of instructions, but on data that is away from the processor, our approach, instead, performs more instructions, but these instructions find data close to the processor and can process them in parallel using vectorization.

In particular, our works build on a family of recent methods [12, 13] that propose filtering of the input streams using small, cache efficient data structures. We argue that, as a result, memory latencies are no longer the dominant bottleneck for this family of algorithms while their computational part becomes more significant. In this work, we follow a two-step approach. First, we propose a refined and extended method, which is able to benefit from vectorization while ensuring cache locality. Second, we design its vectorized version by utilizing SIMD hardware *gather* operations. To evaluate our approach, we apply our techniques to the DFC algorithm [12], as a representative example that outperforms existing techniques in Network Intrusion Detection applications, including [13], on which our proposed approach can be applied as well. In particular, we target the computational part of pattern matching for performance optimization and make the following contributions:

- We propose algorithmic designs for multiple pattern matching which (a) ensure cache locality and (b) utilize modern SIMD instructions.
- We devise a new pattern matching algorithm, based on these designs, that utilizes SIMD instructions to outperform the state of the art, while staying flexible with respect to pattern sizes.
- We (implement the algorithm and) thoroughly evaluate it under both real-world traces and synthetic data sets. We outperform the state of the art by up to 1.8x on commodity hardware and up to 3.6x on the Xeon-Phi platform.

The remainder of the paper is organized as follows: Section 2.2 gives an overview of important pattern matching algorithms and background on vectorization. Section 2.3 describes our system model. In Section 2.4, we present our

approach leading to a new, vectorized design. Section 2.5 presents our experimental evaluation. In Section 2.6, we give an overview of other related work and we conclude in Section 2.7.

2.2 Background

In this section we present traditional approaches to pattern matching, followed by a brief description of the DFC algorithm (Choi et al. [12]) to which we apply our approach. Next, we introduce the required background on vectorization techniques.

2.2.1 Traditional Approach to Multiple-Pattern Matching

The most commonly used pattern matching algorithm for network-based intrusion detection is by Aho-Corasick [11]. It creates a finite-state automaton from the set of patterns and reads the input byte by byte to traverse the automaton and match multiple patterns. Even though it performs a small number of operations for every input byte, it implies– in practice and on commodity hardware – a low instruction throughput due to frequent memory accesses with poor cache locality [12]: As the number of patterns increases, the size of the state automaton increases exponentially and does not fit in the cache. Nevertheless, the method is heavily used in practice; e.g., both Snort [14], one of the best known intrusion detection systems, as well as CloudFlare's web application firewall [15], use it for string matching.

2.2.2 Filtering Approaches and Cache Locality

Besides state-machine based approaches, there is a family of algorithms that rely on *filtering* to separate the innocuous input from the matches. Recent work focuses on alleviating the problem of long latency lookups on large data structures. Choi et al. [12] present a novel algorithmic design called DFC (Direct Filter Classification), that replaces the state machine approach of Aho-Corasick

with a series of small, succinct summaries called *filters*. Such a filter is a bit-array that summarizes only a specific part of each pattern, e.g. its first two bytes, having one bit for every possible combination of two characters that can be found in the patterns. The algorithm is structured in two phases, the *filtering* and *verification*:

- In the *filtering* phase, a sliding window of two bytes over the input goes through an initial filter, as described above, to quickly evaluate whether the current position is a possible starting point of a match. The two-byte windows that passed the initial filter are fed to other, similar filters, each specializing on a family of patterns depending on their length. Since the filters are small (8KB each), they usually fit in L1 cache. Thus, the main part of the algorithm differs from Aho-Corasick and uses only cache-resident data structures, resulting in up to 3.8 times less cache misses [12].

- If a window of two characters passed all filters, there is a strong indication that it is a starting point of a match. For this reason, in the next *verification* phase, the DFC algorithm performs lookups on specially designed hash tables, containing the actual patterns and performs exact matching on the input and the pattern, to verify the match.

Other algorithms in this family, like [13] as well as this work, operate on the same idea: the input is filtered using cache resident data structures, and only the "interesting" parts of the input is forwarded for further evaluation.

2.2.3 Vectorization

Single Instruction Multiple Data (SIMD) is an execution model for data parallel applications, which utilizes processing units that operate on a vector of elements simultaneously, instead of separate elements at a time. SIMD vectorization is a desirable goal in computationally intensive, number-crunching applications, where computation is performed on independent data, *sequentially* stored in memory.

Vector instruction sets have evolved over time, introducing bigger registers and support for more complex instructions. Recently, vector instruction sets have been enriched with the *gather* instruction [16] that enables accessing data from *non-contiguous memory locations* (described in detail in Section 2.3). Polychroniou et al. [17] study the effect of vectorization with the *gather* instruction on Bloom filters, hash tables joins and selection scans among others.We are building on these works with SIMD instructions and extend their design to pattern matching with the applications we focus on.

2.3 System Model

In this section we introduce the assumptions and requirements that our approach makes on the hardware. We focus on mainstream CPUs, with vector processing units (VPUs) that support *gather* instructions. The latter make it possible to fetch memory from non-contiguous locations using only SIMD instructions[1]

The semantics of *gather* are as follows: let W be the vector length, which is the maximum number of elements that each vector register can hold. The parameters to the instruction are a vector register (I) that holds W indexes and an array pointer (A). As output, *gather* returns a vector register (O) with the W values of the array at the respective indexes. It is important to note that *gather* does not parallelize the memory accesses; the memory system can only serve a few requests at a time. Instead, its usefulness lies in the fact that it can be used to obtain values from non-contiguous memory locations using only SIMD code. This increases the flexibility of the SIMD model and allows to efficiency employ it for workloads previously not considered, i.e., where the memory access patterns are irregular. The alternative is to load the values using scalar code, then transfer them one by one from the scalar registers into vector registers. Generally, switching between scalar and vector code is not efficient [17, 18].

[1] In Intel processors, the *gather* instruction was introduced with the AVX2 instruction set and is included in the latest family of mainstream processors (Haswell and Broadwell); *gather* also exists in other architectures, such as the Xeon Phi co-processor [5].

Apart from *gather*, the rest of the instructions we use can be found across almost all the vector instruction sets available. Worth mentioning is the *shuffle* instruction, that makes it possible to permute individual elements within the vector register in any desired order. For example, we employ it for handling the input and output of the algorithm (cf. Section 2.4.2).

The size of the cache, especially the L1 and L2, is very important for the algorithmic design, as we describe later in Section 2.4. Common sizes in modern architectures is 32 KB of L1 data cache with 256 KB of L2 cache and we will use this as a running example. Our design is applicable to other cache sizes as well.

2.4 Algorithmic Design

In this section, we begin by introducing S-PATCH, an efficient algorithmic design for multiple pattern matching. It is designed with both cache locality and vectorizability in mind. Next, we propose our vectorization approach V-PATCH, Vectorized PATTern matCHing.

2.4.1 S-PATCH: a vectorizable version of DFC

To enable efficient vectorization, we introduce significant modifications to the original DFC design. The key insight for the modifications, explained later in detail, is that small patterns will be found frequently in real traffic, so they should be identified quickly without adding too much overhead. On the other hand, long patterns are found less frequently, but detecting them takes longer and requires more characters from the input to pinpoint them accurately.

As the original DFC, our approach has two parts, organized as two separate rounds. In the **filtering** round, we examine the whole input and feed it through a series of filters that bear some similarities to DFC, but adapted to consider properties of realistic traffic, as motivated above. The **verification** round is as in DFC and performs exact matching on the full patterns that are stored in hash tables. Compared with DFC, S-PATCH focuses on efficient filtering in the first

round, because this is the computationally intensive part of the algorithm that, as we show, can be efficiently vectorized. Splitting the two parts in separate rounds improves cache locality, since the data structures used in each round do not evict each other and, as shown in Section 2.4.2, makes vectorization more practical.

Filtering

In this first phase the goals are to (i) quickly eliminate the parts of the input that cannot generate a match and (ii) store the input positions where there is indication for a match. In general, key properties of the filtering phase include:

- Good filtering rate. A big fraction of the input is filtered out at this stage.
- Low overhead. Every filter introduces additional computations and memory accesses, so there needs to be a balance between its overhead and the amount of input that is filtered out.
- Size-efficiency. All the filters need to fit in L1 or L2 cache, while also leaving room for the input and the array for the intermediate results in cache. This is very important, because it ensures that the lookups on the filters will be fast and, as explained later, vectorization using the gather instruction will be feasible.

Our proposed filter design (cf. Figure 2.1) consists of three filters, each with a specific purpose. The first one stores information about the short patterns (less than 4 characters). It has one bit for every possible combination of two characters, and if a particular combination is the beginning of a pattern, the corresponding bit is set. Similarly, the second filter uses the same indexing and accounts for the longer patterns together with the third filter. In more detail, (cf. also Algorithm 1):

First filter. In the first part of the filtering, we examine two bytes of the input at a time and use them to calculate an index for filters 1 and 2. If the corresponding bit in the first filter is set, we directly store the current input position in an array for further processing (lines 5-7).

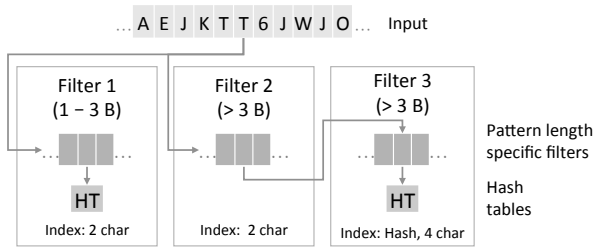


Figure 2.1: *Filter Design of S-PATCH. HT stands for the* Hash Tables *that contain the full patterns.*

Second filter. We also perform a lookup on the second filter using the same index, at line 8. A hit may indicate that we have a match with a longer pattern, but it may also be a false positive (e.g. compare the strings "**at**tribute" and "**at**tack"). Thus, before storing the current input position after a match with the second filter, the algorithm uses more bytes (in our case four) from the input stream with a third filter to gain stronger indications whether there is actually a match. Only when the match in the second filter is corroborated with a match from the third filter is the current position in the input stream stored for further processing (line 11).

Third filter. For the third filter, the index is calculated differently; we cannot have a filter with all combinations of four bytes, due to cache-size limitations. Instead, we use a multiplicative hash function for the four bytes of input to compute the index in the filter, at line 9. There is a trade-off between having a large enough filter to avoid collisions (thus providing a good filtering rate) and having it small enough to fit in cache. The reason why we choose four bytes as input will become clear in the next section (4 bytes fit in each one of the 32-bit vector register values).

Note that the performance of the filtering phase is intrinsically tied to the filter designs and the type of input. The reason why our proposed design is more effective is twofold. Short patterns, although few,[2] are likely to generate many matches. As an example, if strings like `GET` and `HTTP` are part of the pattern set,

[2] 21% of Snort's v2.9.7 patterns are 1-4 bytes long [12].

Algorithm 1: Pseudocode for S-PATCH.

```
   Data: D: data to inspect
 1 # A_short : temporary array for short patterns
 2 # A_long : temporary array for long patterns
 3 for i=0, i <D.length, i++ do
 4 |   index = Read two bytes from pos i in D
 5 |   if (Filter1[index] is set) then
 6 |   |   Store i in A_short
 7 |   end
 8 |   if (Filter2[index] is set) then
 9 |   |   new_index = hash 4 bytes from input
10 |   |   if Filter3[new_index] is set) then
11 |   |   |   Store i in A_long
12 |   |   end
13 |   end
14 end
15 for i=0, i <A_short.length, i++ do
16 |   Verification for small patterns
17 end
18 for i=0, i <A_long.length, i++ do
19 |   Verification for big patterns
20 end
```

they will frequently be found in real network traffic. Treating them separately in a dedicated filter allows us to focus on the longer patterns in other filters. Long patterns, found more rarely, require more information to be distinguished from innocuous traffic.

Verification

After the filtering, all the possible match positions in the input have been stored in a temporary array. At this point, we need to compare the input at these positions with the actual patterns, before we can safely report a match. As mentioned before, the verification phase is as described by Choi et al. [12], except that it is now done in a separate round, after the current chunk of input has been processed by the filtering phase. For ease of reference we paraphrase

here.

Among several optimizations, Choi et al. [12] use specially designed *compact hash tables* that are different for different pattern lengths. Translated to our improved filtering design, if the input at some position i passed the filtering, in the verification phase the algorithm will perform a match on the compact hash table that stores references to all the patterns of appropriate size. For example, if i passed the third filter that stores information on patterns that are four bytes or longer, in the verification phase, the algorithm performs a match on the compact hash table that stores patterns of four bytes or longer (lines 18-20). Each hash table is indexed with as many bytes as the shortest pattern that the hash table contains (in this case, four bytes of the input will be used as an index to the hash table). Each bucket in the hash table contains references to the full patterns and the algorithm has to compare each one of them individually with the input, before reporting a match. Eventually, the algorithm identifies all the occurrences of all the patterns, producing the same output as Aho-Corasick.

In general, the compact hash tables as we use them in this phase, do not fit L1 or L2 cache (but they might fit L3 cache) and accessing them incurs high latency misses. However, the success of the approach lies in the fact that the filtering phase will reject most of the input, so the algorithm resorts to verification only when it is needed (when there is a high probability for a match). That is why our efforts focus on the filtering part, where the data structures are close to the processor and can benefit from vectorization.

2.4.2 V-PATCH: Vectorized algorithmic design

A basic issue when vectorizing S-PATCH is its non-contiguous memory accesses. The sequential version accesses the filters at nonadjacent locations for every window of two characters, whereas in a vectorized design W indexes are stored in a vector register (of length W), each pointing to a separate part of the data structure. For this reason, we use the SIMD *gather* instruction that allows us to fetch values from W separate places in memory and pack them in a vector register.

Algorithm 2: Pseudocode for the V-PATCH filtering phase.

```
   Data: D: input data to inspect
1  # W : the vector register length
2  # A_short : temporary array for short patterns
3  # A_long : temporary array for long patterns
4  # M1→ : constant mask used to convert the input to 2 byte sliding window format
5  # M2→ : constant mask used to convert the input to 4 byte sliding window format
6  for i=0, i <D.length, i += W do
7      R→ = Fill register with raw input from D
8      Indexes→ = shuffle(R→,M1→)
9      V1→ = gather(filter1_address, Indexes→)
10     if at least one element in V1→ is set then
11         Store positions of matches in A_short
12     end
13     V2→ = gather(filter2_address, Indexes→)
14     if at least one element in V2→ is set then
15         NewIndexes→ = shuffle(R→,M2→)
16         Keys→ = hash(NewIndexes→)
17         V3→ = gather(filter3_address, Keys→)
18         if at least one element in V3→ is set then
19             Store positions of matches in A_long
20         end
21     end
22 end
```

Algorithm 2 gives a high level summary of the filtering phase of V-PATCH. The first step towards vectorizing the algorithm is loading the consecutive input characters from memory and storing them in the appropriate vector registers. Figure 2.2 shows the initial layout of the input and the desired transformation to W elements, each holding a sliding window of two characters. The transformation is efficiently achieved with the use of the *shuffle* instruction, allowing to manually reposition bytes in the vector registers (Algorithm 2, line 8).

Once the vector registers are filled, the next step is to calculate the set of indexes for the filters. Note that every 2-byte input value maps to a specific *bit* in the filter, but the memory locations in the filter are addressable in *bytes*. A

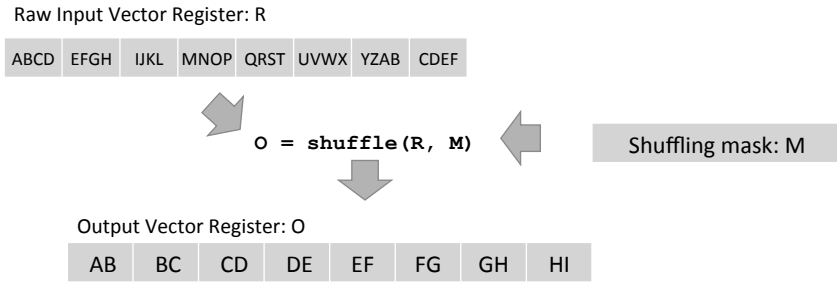


Figure 2.2: *Input Transformation from consecutive characters to sliding windows of two characters.*

standard technique used in the literature [12, 19] is to perform a bit-wise right shift of the input value to the corresponding index in the filter. The remainder of the shift indicates which bit to choose from the ones returned. Having computed the indexes, we use them as arguments to the *gather* instruction that fetches the filter values at those locations (Algorithm 2, lines 9 and 13).

Regarding the number of *gather* instructions used, to optimize in latency, note that the first two filters (lines 9 and 13) are specifically designed to use the same indexes for a given input value in *gather* but different base addresses for the filters. Thus, with the **filter merging** optimization where the filters are interleaved in memory (at the same base address), we can merge lines 9 and 13 into a single *gather*, to bring the information from both filters from memory simultaneously. This optimization is not shown in the pseudo-code but depicted in Figure 2.3, giving an example in which a single *gather* instruction fetches information from both filters. Using bit-wise operations we can choose one filter or the other, once the data is in the vector register.

If at least one of the W values has passed the second filter, they need to be further processed through the third filter. Remember that the third filter uses a window of four input characters as an index. Thus, we load a sliding window of four input characters in each vector element in the register (line 15) and create the hash values that we use as indexes in the third filter (lines 16-17).

Not all of the values in the vector register are useful; only the ones that

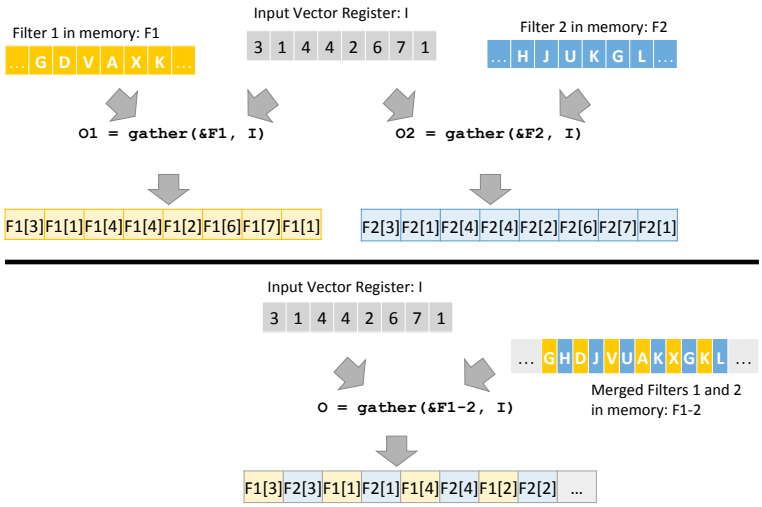


Figure 2.3: *Figure describing the **filter merging** optimization. In the upper half, lookups on two filters require two gather invocations. Once the filters are merged in memory in the lower half, one gather brings information from both filters to the registers.*

passed the second filter need to be processed further by the third filter. This is a common challenge when vectorizing algorithms with conditional statements, since for different input we need to run different instructions. There are approaches [19] that manipulate the elements in the vector registers, so that they only operate on useful elements. For this particular algorithm, experiments with preliminary implementations showed that the cost of moving the elements in the registers out-weighted the benefits. Thus, we choose to speculatively perform the filtering on all the values and then mask out the ones that do not pass the second filter. In our evaluation (Section 2.5), we observe that operating speculatively on all the elements is actually not a wasteful approach, especially with a large number of patterns to match.

As with the scalar algorithm, after a hit in the first or third filter we need to store the position of the input where a potential match occurred. We store the positions of the input that passed the filter from the set of W values in the

register (lines 11 and 19). Here, we postpone the actual verification to avoid a potential costly mix of vectorized and scalar code, where the values from the vector registers need to be written to the stack and from there read into the scalar registers. Such a conversion can be costly and can negate any benefits we gain from vectorization [18].

Furthermore, to fully exploit the available instruction-level parallelism, we manually unroll the main loop of the algorithm by operating on two vectors (R_j) of W values instead of one, a technique that has proven to be efficient especially for SIMD code [19]. This has the benefit that, while the results of a *gather* on one set of W values are fetched from memory (line 9), the pipeline can execute computations on the other set of values in parallel.

2.5 Evaluation

In this section, we evaluate the benefits that our vectorization techniques bring to pattern matching algorithms. Our evaluation criteria are the processing throughput and the performance under varying number of patterns. We show the improvements of V-PATCH with both realistic and synthetic datasets, as well as with changing number of patterns. For a comprehensive evaluation, we compare the results from five different algorithms: the original Aho-Corasick ([11]; implementation directly taken from the Snort source code [14]), DFC (Choi et al. [12], summarized in Section 2.2.2), Vector-DFC (a direct vectorization of DFC done by us), S-PATCH (the scalar version of our algorithm, described in Section 2.4.1, that facilitates vectorization and addresses properties of realistic traffic that were not addressed before), and V-PATCH (the final vectorized algorithm described in Section 2.4.2).

2.5.1 Experimental setup

Systems. For the evaluation we use both Intel Haswell and Xeon-Phi. More specfically, the first system is an Intel Xeon E5-2695 (Haswell) CPU with 32KB of L1 data cache, 256KB of L2 cache and 35MB of L3 cache. We use the

ICC compiler (version 16.0.3) with -O3 optimization under the operating system CentOS. Unless otherwise noted, the experiments in this section are run on this platform. The second system is the Intel Xeon-Phi 3120 co-processor platform. Xeon-Phi has 57 simple, in-order cores at 1.1 GHz each, with 512-bit vector processing units. The memory subsystem includes a L1 data cache and a L2 cache (32KB and 512KB respectively) private to each core, as well as a 6GB GDDR5 memory, but no L3 cache. We compile with ICC -O3 (version 16.0.3) under embedded Linux 2.6. We are only using Xeon-Phi in native mode as a co-processor. The next versions of Xeon-Phi are standalone processors, so the problem of processor-to-co-processor communication is alleviated. Since different hardware threads can operate independently on different parts of the stream, in our experiments with both platforms, we focus on the speedup achieved by a single hardware thread, through vectorization.

Patterns. We use two sets of patterns: a smaller one, named *S1*, consisting of approximately $2,500$ patterns that comes with the standard distribution of Snort[3] [20] – the de-facto standard for network intrusion detection systems – and a larger one, named *S2*, with approximately $20,000$ patterns, that is distributed by `emergingthreats.net` The patterns affect the performance of the algorithm and this is analyzed in detail in Section 2.5.3.

Data sets. In our evaluation, we use both real-world traces and synthetic data-sets. The real-world traces are the ICSX dataset [21, 22] (created to evaluate intrusion detection systems) and the DARPA intrusion detection dataset [23]. From ICSX, we randomly take 1GB of data from each of days 2 and 6 (thereafter named ICSX day 2 and ICSX day 6, respectively) and we also use 300MB of data from the DARPA 2000 capture. We are aware of the artifacts in the latter set, and the discussions in the community about its suitability for measuring the *detection capability* of intrusion detection systems [24]. In our experiments, we use it only for the purpose of comparing throughput between algorithms, allowing for future comparisons on a known dataset. The synthetic data set consists of 1GB of randomly generated characters.

An important point, considering the evaluation validity, is that, typically, not

[3] We used version 2.9.7 for our experiments.

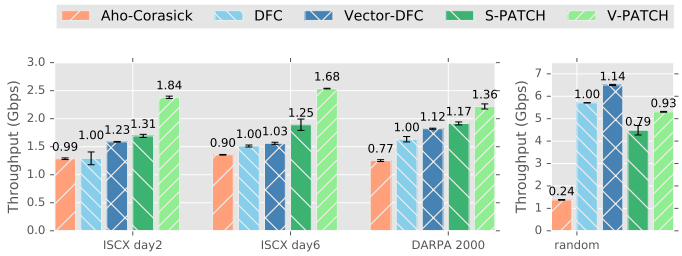
all the patterns are evaluated at the same time. In a Network Intrusion Detection System such as Snort, patterns are organized in groups, depending on the type of traffic they refer to. When traffic arrives in the system, the reassembled payload is matched only against patterns that are relevant (e.g. if the stream has HTTP traffic, it is checked against HTTP related patterns, as well as more general patterns that do not refer to a specific protocol or service). To evaluate our algorithm in a realistic setting, we also pair traffic with relevant patterns. Since, in our datasets, most of the traffic is HTTP [21], we focus on HTTP traffic and match it against the patterns that are applicable based on the rule definitions. A similar approach can be used for other protocols (e.g. DNS, FTP), but we focus on HTTP traffic as it typically dominates the traffic mix and many attacks use HTTP as a vector of infection.

2.5.2 Overall Throughput

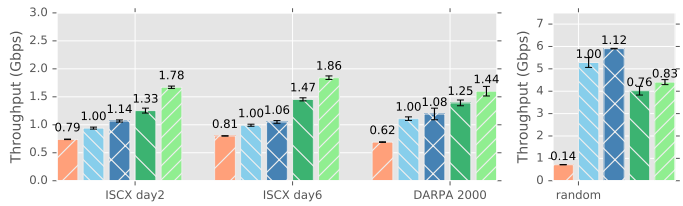
In this section we compare the overall performance between the different algorithms. Using the HTTP-related patterns of each set gives us 2K patterns from pattern set *S1* and 9K patterns from pattern set *S2*. All algorithms count the number of matches. We use 10 independent runs of each experiment. We report the average throughput values, as well as standard deviation as error bars.

Figure 2.4a shows the throughput of all algorithms under realistic traffic traces and synthetic traces, when matched against the small pattern set (*S1*). In Figure 2.4b we use the bigger pattern set (*S2*). The numbers above the bars indicate the relative speedup compared to the original DFC algorithm.

We first discuss the results by only considering each pattern set and each traffic set separately. For realistic traffic traces, our vectorized implementation consistently outperforms the DFC algorithm by up to 1.86x (left parts of Figure 2.4), due to the parallelization we introduce in the filtering phase. The direct vectorization of the original DFC algorithm (Vector-DFC) has limited performance gain, because much of the running time of DFC is spent on verification and not filtering. This is the main motivation for introducing a modified version of DFC, in Section 2.4.1, focused on improving the filtering phase. By treating



(a) Snort web traffic patterns (2K).



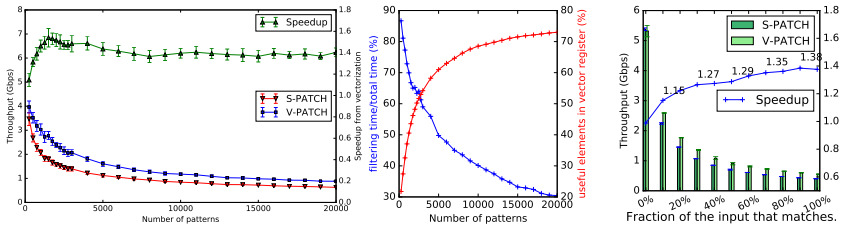
(b) ET open 2.9.0 web traffic patterns (9K).

Figure 2.4: *Performance comparison between the different algorithms for public and random data sets on the Xeon platform.*

small, frequently occurring patterns separately and by examining more information in the case of long patterns, S-PATCH outperforms the original by up to 1.47x. More importantly, it allows for much greater vectorization potential, since the biggest portion of the algorithm's running time is shifted to efficient filtering of the input, and verification is done much more seldom.

Next, we evaluate the impact of the size of the ruleset on the overall throughput (comparing Figure 2.4a with Figure 2.4b). The overall throughput of the algorithms decreases, since the input is more likely to match and identifying every match consumes extra cycles. The performance of Aho-Corasick, in particular, decreases by more than 40%, because the extra patterns greatly increase the size of the state machine. The rest of the algorithms experience a 23-34% drop in performance.

It is important to note that the performance gain of the algorithms (DFC versus Aho-Corasick, V-PATCH versus DFC) is influenced by the input as fol-



(a) Throughput as the number of patterns increases.

(b) Filtering to verification ratio and vectorization efficiency.

(c) Speedup from vectorization, as the numbers of matches in the input increases.

Figure 2.5: *a) comparison between the scalar and vectorized versions of our approach, as the number of patterns increases. b) filtering-to-verification ratio (left axis), as well as the average number of useful elements in the vector registers after filter 2 (right axis), as the number of patterns increases. c) comparison between the scalar and vectorized approach, as the fraction of matches in the input increases.*

lows: when feeding the algorithms a data set that contains random strings, DFC significantly outperforms AC (right part of Figure 2.4). In this case, we do not expect to find many matches in the input and the filtering phase will quickly filter out up to 95% of the input. This is also the reason why the modified versions of the algorithm (S-PATCH and V-PATCH) perform less efficiently compared to what they do in the different input scenarios; the design of the two separate filters as described in Section 2.4 shows its benefits in more realistic traffic mixes. In turn, this poses interesting questions for the future in how to best design the filters based on the expected traffic mix. Still, the vectorized versions provides speedups over the scalar ones.

2.5.3 The effects of the number of patterns

As shown in Section 2.5.2, it is important to account for the actual traffic mix the algorithms are expected to run upon when designing the filtering stage, as it has a large impact on the performance. As new threats emerge, more malicious

patterns are introduced and the performance of the algorithm must adapt to that change.

We measure the effects of the number of patterns on the two best performing algorithms and summarize the results in Figure 2.5a, also including the overall speedup of V-PATCH compared to S-PATCH. In this experiment, we randomly select the number of patterns from the complete set *S2* ($20,000$ patterns) in order to test our algorithms with as many patterns as possible. V-PATCH consistently performs better compared to S-PATCH, regardless of the number of patterns considered. Observe that:

(i) As the number of patterns increases, so does the input fraction that passes the filters. This causes the verification part, which is not vectorized, to take up more of the running time, essentially reducing the parallel portion and, by Amdahl's law [25], the benefit of vectorization. The portion of the running time spent in filtering, over the total running time is shown in Figure 2.5b (blue line).

(ii) As the number of patterns increases, the vectorization of the filtering becomes more efficient. Remember that V-PATCH will proceed with the third filter if at least one of the values in the vector register block passes the second filter. With a small number of patterns, we will seldom pass the second filter. When we do, it is likely we only have a single match, meaning that the rest of the values in the register are disabled and any computation performed for those values is wasteful work. Increasing the number of patterns results in more potential matches in the second filter and, as a consequence, less disabled values for the third filter and thus more useful work. In Figure 2.5b (red line) we measure this effect and show the average number of useful items inside the vector register every time we reach the third filter. Clearly, with an increasing number of patterns, the vectorization is performed mainly on useful data and therefore becomes more efficient.

(iii) The two trends essentially cancel each other out, keeping the overall performance benefit of V-PATCH compared to S-PATCH constant after a point (Figure 2.5a), even though the optimized filtering gradually becomes a smaller part of the total running time.

(iv) A similar effect is observed when we keep the number of patterns con-

stant, but increase the amount of matches in the dataset (Figure 2.5c). For this experiment, we created a synthetic input that contains increasingly more patterns, randomly selected from a ruleset of $2,000$ patterns. As more matching strings are inserted into the input, our vectorized portion of the algorithm becomes more efficient and the relative speedup compared to the scalar version slowly increases.

2.5.4 Filtering Parallelism

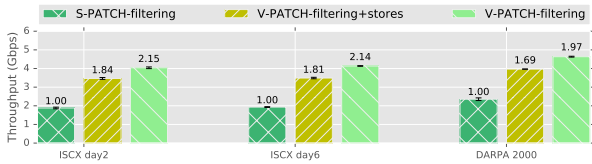
In this section, in order to gain better insights about the benefits of vectorization, we measure the speedup gained in the filtering part in isolation.

Figure 2.6 compares the filtering throughput of the scalar S-PATCH and V-PATCH, for pattern sets S1, S2, as well as the full pattern set (20K patterns). In the same figure, we also report the performance of the vectorized filtering, where we exclude the cost of storing the matches in the filtering phase in the temporary arrays. As we can see from the graph, the throughput of the filtering part is increased by up to a factor of 1.84x, on the small pattern set. Storing the matches of the filtering part in arrays comes with a cost; when it is removed, performance increases up to 2.15x for small pattern sets and up to 2.80x for the full pattern set. Even though there is a small decrease at the pattern set with 9K patterns (Figure 2.6b), the relative speedups of vectorized filtering increase with the number of patterns (Figure 2.6c).

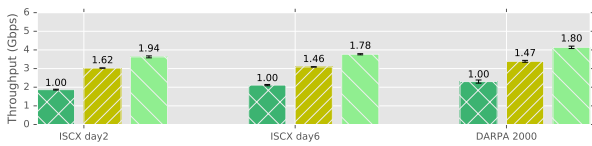
2.5.5 Changing the vector length: Results from Xeon-Phi

We have also evaluated the effectiveness of our approach on an architecture with a wider vector processing pipeline. The Xeon-Phi [5] co-processor from Intel supports vector instructions that operate on 512-bit registers, thus able to perform two times more operations in parallel, in the filtering phase.

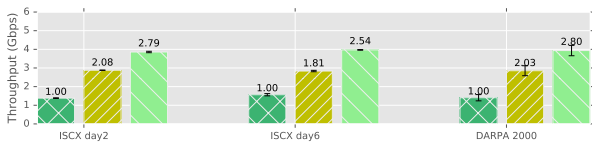
Figure 2.7 summarizes the results from Xeon Phi, where the experiments are identical with those described in Section 2.5.2. Note that we report the throughput of a single Xeon-Phi thread. V-PATCH takes advantage of the wider vector registers and outperforms the original scalar DFC algorithm, up to a



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).



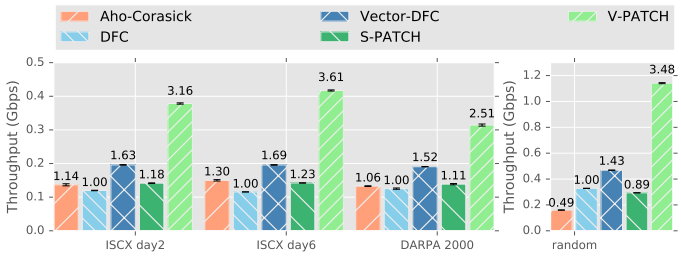
(c) Full pattern-set (20K).

Figure 2.6: *Measuring the performance of the filtering part only. "V-PATCH-filtering+stores" includes the cost of storing the results of the filtering phase to temporary arrays .*

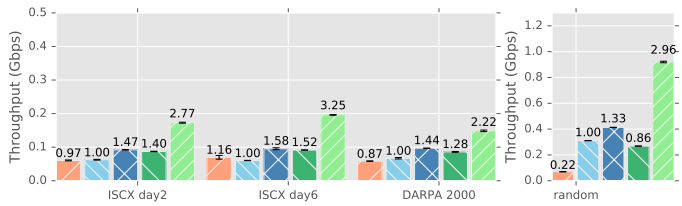
factor of 3.6x on real data and 3.5x on synthetic random data.

As Xeon-Phi threads have much slower clock (1.1 GHz) and the pipeline is less sophisticated (e.g. there is no out-of-order execution), it is not surprising that the absolute throughput sustained by a single Phi thread is smaller than that of the single thread performance of the Xeon platform used in the previous experiments. When dealing with multiple streams in parallel, due to the higher degree of parallelism, the aggregated gain will naturally be higher.

An interesting observation is that the DFC algorithm is slightly slower than AC on real data, where the number of matches in the input is significantly higher. In the original DFC algorithm, the filters are small and can easily fit L1 or L2 cache, and the hash tables containing the patterns are bigger, but still



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).

Figure 2.7: *Performance comparison between the different algorithms for public and random data sets on the* Xeon-Phi platform.

expected to fit L3 cache. In Xeon-Phi there is no L3 cache, so accesses to the hash tables in the verification phase are typically served by the device memory, negating the benefits of cache locality that is part of the main idea of the algorithm. Nonetheless, our *improved filtering* design reduces the number of times we resort to verification and access the device memory, thus resulting in 1.1x-1.5x increased throughput on realistic traffic, compared to the original DFC design.

2.6 Related Work

2.6.1 Pattern matching algorithms

Pattern matching has been an active field of research for many years and there are numerous proposed approaches. Aho-Corasick, explained before in Sec-

tion 2.2.1 is one of the fundamental algorithms in the fields. There are variants of Aho-Corasick that decrease the size of the state transition table (for example [26]) by changing the way it is mapped in memory, but they come at an increased search cost, compared to the standard version of Aho-Corasick used in our evaluation. Other approaches apply heuristics that enable the algorithm to skip some of the input bytes without examining them at all, such as Wu-Manber [27] where a table is used to store information of how many bytes one can skip in the input. The main issue with these approaches is that they perform poorly with short patterns. For the problem domain investigated here, the patterns can be of any length and the algorithm must handle all of them gracefully. Moreover, in both Aho-Corasick and Wu-Manber algorithms, there is no data parallelism because there are dependencies between different iterations of the main loop over the input.

Recent algorithms [12, 13] follow a different idea: Using small data structures that hold information from the patterns (directly addressable bitmaps in the case of [12], Bloom filters in the case of [13]), they quickly filter out the biggest parts of the input that will not match any patterns and fallback to expensive verification when there is an indication for a match. Our work is inspired by this family of algorithms, showing how they can be modified to perform better under realistic traffic and gain significant benefit from vectorization.

2.6.2 SIMD approaches to pattern matching

Even though pattern matching algorithms are characterized by random access patterns, SIMD approaches have been used before for pattern matching, especially in the field of regular expression matching. Mytkowicz et al. [8] enumerate all the possible state transitions for a given byte of input to break data dependencies when traversing the DFA. Then they use the *shuffle* instruction to implement gathers and to compute the next set of states in the DFA. The algorithm is applied on the case where the input is matched against a single regular expression with a few hundreds of states and does not scale for the case of multiple pattern matching where we need to access thousands of states for every

byte of input. Sitaridi et al. [9] use the same hardware gathers as we do, but apply them on database applications where the multiple, independent strings need to be matched against a single regular expression. There have been approaches that use other SIMD instructions for multiple exact pattern matching, but have constraints that make them impractical for the case of Network Intrusion Detection. Faro et al. [28] create fingerprints from patterns and hash them, but they require that the patterns are long, which is not always true for the typical set of patterns found e.g. in Snort.

2.6.3 Other architectures

Outside the range of approaches that target commodity hardware, there is rich literature on network intrusion detections systems that are customised for specific hardware. For example, SIMD approaches that target DFA-based algorithms have been applied on the Cell processor [29], as well as GPUs and FPGAs [30–32]. Vasiliadis et al. [30] build a GPU-based intrusion detection system that uses Aho-Corasick as the core pattern matching engine. Kouzinopoulos and Margaritis also experiment with pattern matching algorithms on GPUs and apply them on genome sequence analysis [31]. GPU parallelization has many similarities with vectorization; in fact GPUs offer more parallelism that can hide memory latencies. At the same time, it introduces additional challenges e.g. long latencies when transferring data between the host and the GPU. In this work we utilize vector pipelines that are already part of modern commodity architectures. Moreover, vectorization with CPUs requires careful algorithmic design that makes use of caches and advanced SIMD instructions. A main part of our work is showing how this problem can be tackled for the case of intrusion detection.

2.7 Conclusion

In this paper, we introduce an efficient algorithmic design for multiple pattern matching which ensures cache locality and utilizes modern SIMD instructions.

Specifically, we introduce V-PATCH: it employs carefully designed and engineered vectorization and cache locality for accelerated pattern matching and nearly doubles the performance when compared to the state of the art.

We thoroughly evaluate V-PATCH and its algorithmic design with both open data sets of real-world network traffic and synthetic ones in the context of network intrusion detection. Our results on Haswell and Xeon-Phi show a speedup of 1.8x and 3.6x, respectively, over single thread performance of Direct Filter Classification (DFC), a recently proposed algorithm by Choi et al. for pattern matching exploiting cache locality, and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems. Moreover, we show that the performance improvements of V-PATCH over the state of the art hold across open, realistic data sets, regardless of the number of patterns in the chosen ruleset. The experimental study also provides insights about the net effect of vectorization as well as trade-offs it implies in this family of algorithmic designs.

Acknowledgements

The research leading to these results has been partially supported by the Swedish Energy Agency under the program Energy, IT and Design, the Swedish Civil Contingencies Agency (MSB) through the project âĂIJRICSâĂİ, and by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC and the project LoWi, and by the Swedish Research Council (VR) through the project ChaosNet.

Bibliography

[1] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos, "Generating realistic workloads for network intrusion detection systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, 2004.

[2] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra, "On the statistical distribution of processing times in network intrusion detection," in *2004 43rd IEEE Conf. on Decision and Control (CDC)*, 2004, vol. 1.

[3] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, 2015.

[4] Yong Li and Min Chen, "Software-defined network function virtualization: a survey," *IEEE Access*, vol. 3, 2015.

[5] "Intel Xeon Phi product family," `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`, Accessed: 2016-12-10.

[6] "Intel vectorization tools," `https://software.intel.com/en-us/articles/intel-vectorization-tools`, Accessed: 2016-12-10.

[7] "The importance of vectorization for Intel Many Integrated Core Architecture (Intel MIC architecture)," `https://software.intel.com/en-us/articles/the-importance-of-vectorization-for-intel-many-integrated-core-architecture-intel-mic`, Accessed: 2016-12-10.

[8] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte, "Data-parallel finite-state machines," in *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2014, ASPLOS '14, ACM.

[9] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A. Ross, "SIMD-accelerated regular expression matching," in *Proc. of the 12th Int. Workshop on Data Management on New Hardware*. 2016, DaMoN '16, ACM.

[10] Peng Jiang and Gagan Agrawal, "Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2017, PPoPP '17, ACM.

[11] Alfred V. Aho and Margaret J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, June 1975.

[12] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han, "DFC: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, USENIX Association.

[13] Iulian Moraru and David G. Andersen, "Exact pattern matching with feed-forward bloom filters," *J. Exp. Algorithmics*, vol. 17, Sept. 2012.

[14] "Snort rules and IDS software download," `https://www.snort.org/downloads`, Accessed: 2016-12-10.

[15] "Scaling CloudFlare's massive WAF," `https://www.scalescale.com/scaling-cloudflares-massive-waf/`, Accessed: 2016-12-10.

[16] "Gather Scatter operations," `http://insidehpc.com/2015/05/gather-scatter-operations/`, Accessed: 2016-12-10.

[17] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*. 2015, SIGMOD '15, ACM.

[18] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein, "Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips," in *Proc. of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. 2014, WPMVP '14, ACM.

[19] Orestis Polychroniou and Kenneth A. Ross, "Vectorized Bloom filters for advanced SIMD processors," in *Proc. of the Tenth Int. Workshop on Data Management on New Hardware*. 2014, DaMoN '14, ACM.

[20] Martin Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conf. on System Administration*. 1999, USENIX Association.

[21] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers & Security*, vol. 31, no. 3, 2012.

[22] "UNB ISCX intrusion detection evaluation dataset," `http://www.unb.ca/research/iscx/dataset/iscx-IDS-dataset.html`, Accessed: 2016-12-10.

[23] "DARPA intrusion detection data sets," `https://www.ll.mit.edu/ideval/data/`, Accessed: 2016-12-10.

[24] Matthew V Mahoney and Philip K Chan, "An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection," in *Int. Workshop on Recent Advances in Intrusion Detection*. Springer, 2003.

[25] Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the April 18-20, 1967, Spring Joint Computer Conference*. 1967, AFIPS '67 (Spring), ACM.

[26] Marc Norton, "Optimizing pattern matching for intrusion detection," *Sourcefire, Inc., Columbia, MD*, 2004.

[27] Sun Wu and Udi Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, University of Arizona. Department of Computer Science, 1994.

[28] Simone Faro and M. Oğuzhan Külekci, *Fast Multiple String Matching Using Streaming SIMD Extensions Technology*, Springer, 2012.

[29] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the Cell processor," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[30] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis, *Gnort: High Performance Network Intrusion Detection Using Graphics Processors*, Springer, 2008.

[31] Charalampos S Kouzinopoulos and Konstantinos G Margaritis, "String matching on a multicore GPU using CUDA," in *Informatics, PCI'09. 13th Panhellenic Con. on*. IEEE, 2009.

[32] Ioannis Sourdis and Dionisios Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed nids pattern matching," in *Field-Programmable Custom Computing Machines, FCCM 2004. 12th Annual IEEE Symposium on*, 2004.

PAPER II

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Geometric Monitoring in Action: a Systems Perspective for the Internet of Things

under submission, 2018

3
PAPER II

Abstract

Many applications in the Internet of Things (IoT) continuously monitor sensor values and react when their network-wide aggregate exceeds a threshold. Geometric monitoring (GM) promises a several-fold reduction in terms of communication and coordination between the sensors for such applications. Previous work on GM has been limited to analytic or high-level simulation results and does not consider critical system aspects such as the radio duty-cycle and packet losses.

In this paper, we devise, realize and evaluate a system design for GM, enabling deployment possibilities on resource-constrained IoT devices and network stacks. In particular we provide (i) an algorithmic implementation for

commodity IoT hardware (ii) a study and insights regarding duty cycle reduction and energy savings on actual IoT nodes, in connection with existing analytic and high-level simulation results about GM, as well as (iii) a study and insights on the influence of packet losses on the effectiveness of the method, in terms of accuracy and responsiveness. Our results, both from full-system simulations and a publicly available testbed, show that GM indeed provides several-fold energy savings in communication; e.g., we see up to 3x and 11x reduction in duty-cycle when monitoring the variance and average temperature of a real-world data set, respectively but the results fall short of the analytic predictions (4.3x and 44x, respectively). Hence, we investigate the energy overhead imposed by the network stack as well as the effects of packet losses on the accuracy and responsiveness of the algorithm. Through the above insights, we offer guidelines for the adaption of GM and similar algorithms in IoT settings.

3.1 Introduction

Sensing and monitoring the state of a system or the conditions of the environment is one of the most fundamental uses for Wireless Sensor Networks (WSNs). In fact, it is the target application that drives a rich body of research on communication protocols that address the problem of efficient dissemination and collection of sensor readings.

Given a set of sensor nodes $n_1, n_2, ..., n_N$ with readings $\vec{v_1}, \vec{v_2}, ..., \vec{v_N}$ that vary over time, we want to continuously track whether a function f, defined over the network-wide, weighted average of those readings, is larger than a predefined threshold T. Keeping track of such a function is of particular interest and serves as a basis for many applications and control loops, e.g. for detecting outliers [2], hot-spots [25] or denial-of-service attacks [7, 8, 12]. The challenge associated with the problem is to let all nodes accurately determine whether the function is above or below the threshold locally, without having to share every reading with other nodes. For simple, linear functions (e.g. when monitoring just the average value), it is often easy to derive local constraints that minimize communication. However, for non-linear functions (that are, in fact, the most

interesting to monitor), deriving such constraints is challenging.

The problem of distributed monitoring has been studied extensively over the years, in the context of distributed stream monitoring, with many interesting solutions. Sharfman et al. [24] proposed a general method, called geometric monitoring (GM), that can monitor *any* function, linear or not, computed over network-wide aggregates and keep track of its value with respect to a threshold. The method suppresses unnecessary communication by deriving local constraints that individual nodes can check without communication. The effectiveness of the method has been thoroughly studied on distributed data streams, with different functions and data-sets, showing impressive communication reduction results. Since then, GM has become a very active research topic, with many interesting extensions that build upon the original approach. Variations of GM have been enhanced with sketches [10] and prediction models [11] and have been applied on outlier detection [2] and data stream queries [9]. The above mentioned extensions are orthogonal to the original GM algorithm. In this paper we focus on the basic principles of GM and tackle the challenges described next.

Research Challenges: Even though GM and similar threshold monitoring algorithms are designed with sensor networks in mind, there is lack of insights from full-system design, implementation and real deployments. Existing work on GM has focused on the algorithmic part, backed up with numerical, high level simulations where communication is assumed instant, reliable and without any overheads. However, the reality of WSNs for Internet of Things (IoT) environments is different: packet losses are frequent, nodes have severe constraints on processing power and lifetime, and message propagation is costly, both in terms of energy as well as latency. Moreover, the impact of systems properties that were not previously considered relevant for GM, such as radio duty cycling and idle listening needs to be understood. Recent work on data aggregation [23] has shown that such properties greatly influence the lifetime savings that can be achieved in practice, and allow room for cross-layer optimizations.

Thus, the feasibility of GM for WSNs and the impact of the system's properties imply open questions. Specifically:

1. What are the actual battery lifetime savings that we can achieve on real nodes?

2. What is the effect of packet losses on the accuracy and responsiveness of the algorithm?

3. How can the processing of such methods be efficiently approximated on CPU-constrained and energy-constrained devices suitable for the IoT?

To address these, we take a step beyond the existing analysis and consider the whole system stack, through (i) extensive, cycle-accurate, full-system simulations and (ii) validation from a real deployment. By doing so, we are able to evaluate if, and up to what degree, emerging results on distributed continuous monitoring algorithms can bring practical benefits on real WSN deployments.

Contributions:

1. We bridge the gap between numerical simulation results on efficient threshold monitoring and real IoT network environments.

2. We study the algorithmic implementation and the actual performance on a real deployment, using real data sets and offer valuable insights. Specifically:

    - As the computational complexity of the GM method is a serious challenge for CPU-constrained devices, we suggest an efficient approximation.
    - We show that the practical energy lifetime improvements may vary significantly and are often far from the savings estimated analytically. Specifically, we find that the overhead of idle listening is a dominant factor that can sometimes limit the effectiveness of GM.
    - The communication behaviour under GM varies greatly, where the underlying protocol must take into account periods of no activity as well as bursts of concurrent updates.

- As message losses are common in practice, we study their effect and identify that they cause the nodes to de-synchronize. We also show that this has cascading effects that decrease the accuracy-responsiveness of the algorithm.

The remainder of this paper is organized as follows. Section 3.2 summarizes GM and introduces its challenges in wireless sensor networks. Section 3.3 outlines system design challenges and our solutions as well as tunable parameters of the network stack that will influence properties of the system implementation. We evaluate our full-system implementation of GM in Section 3.5, through the experimental methodology presented in Section 3.4. We discuss related work in Section 3.6 and conclude in Section 3.7.

3.2 Background

In this section we start by summarizing the Geometric Monitoring as a general method for distributed threshold monitoring. We then outline background related to wireless sensor networks communication and the system model.

3.2.1 The Geometric Monitoring Method (GM)

In their seminal work [24], Sharfman et al. present a general method that is able to threshold-monitor arbitrary, possibly non-linear functions, defined over network-wide aggregates. Geometric Monitoring (GM) limits the number of broadcasts needed. Instead of having nodes broadcast at every *epoch* (sensor sampling period) of the execution, each individual node uses only information from the last available broadcast and its current (up-to-date) sensor reading to calculate a region of the input domain where the true value of the network-wide aggregate can be. As long as this region remains fully on one side of the threshold, communication is avoided. If a part crosses the threshold, broadcasts are needed to see whether the local change is offset by changes at other nodes (false alarm) or if the function has actually crossed the threshold. The key points

of the method are briefly explained here, but the interested reader is referred to [24] for the full explanation with proofs.

In GM the set of sensor readings $\vec{v_1}, \vec{v_2}, ..., \vec{v_N}$ are called *local statistics vectors*. These vectors are only known locally, but sporadically a node n_i will broadcast its $\vec{v_i}$ to every other node. The last broadcasted value from node n_i is denoted as $\vec{v_i'}$. The weighted average[1] of the local vectors (eq 3.1) is called the *global statistics vector*.

$$\vec{v} = \sum_{i=1}^{N} w_i * \vec{v_i} \tag{3.1}$$

Similarly, the weighted average of the last broadcasted values is called the *estimate vector* ($\vec{e}$). The estimate vector is known to all nodes and is essentially an estimation of the global statistics vector.

When a node measures a new set of sensor readings, its local statistics vector will drift ($\vec{\Delta v_i} = \vec{v_i} - \vec{v_i'}$). The *drift vector* $\vec{u_i}$ is defined as the displacement of the estimate vector because of the new drift, i.e. $\vec{u_i} = \vec{e} + \vec{\Delta v_i}$ and can be computed locally without communication.

The convex hull of the drift vectors is defined as the set of all the convex combinations of $\vec{u_i}$ ($\sum \theta_i \vec{u_i}$).[2] As such, it is clear that the weighted average of the drift vectors (defined similar to eq 3.1) would be part of this set. With simple substitution, one can also see that the global estimate is equal to the weighted average of the drift vectors and thus it must also lie in the convex hull of the drift vectors.

Figure 3.1 shows an example with f being the bi-variant function $f(x, y) = y - x^2$ and $T = 2$. We want to know if $f(\vec{v}) < T$, or equivalently, if $\vec{v}$ lies within a region where f has values less than T. This region is marked white in the figure. The convex hull created by the drift vectors is the yellow area, and $\vec{v}$ is guaranteed to lie somewhere inside that area. Thus, as long as the yellow area is inside the white region, also $\vec{v}$ is guaranteed to be inside the white region. However, nodes cannot locally determine the convex hull, since

[1] Without loss of generality, we assume $0 \leq w_i \leq 1$ and that the weights sum to one.

[2] See for example `https://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/VictorKlee.pdf`

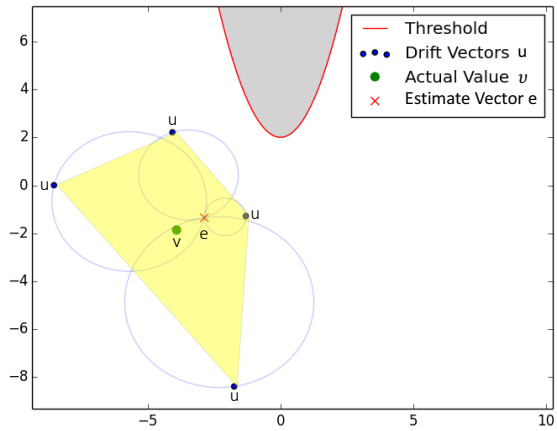


Figure 3.1: *An example illustrating the GM method.*

that would require knowing all of $\vec{u_1}, \vec{u_2}, ..., \vec{u_N}$.

This is where the final part of the method of Sharfman et al. [24] comes into play. Consider the set of spheres in Figure 3.1, each centered at $\frac{\vec{e}+\vec{u_i}}{2}$ with a radius of $\frac{\vec{e}-\vec{u_i}}{2}$. Each node can create such a sphere locally, since $\vec{u_i}$ is known to n_i and $\vec{e}$ is the same across all nodes at a given time. The authors of GM prove that the union of those spheres strictly covers the convex hull (yellow area). Therefore, using GM, a node only needs to track whether its locally computed sphere crosses the threshold: if not, it can remain quiet; else, it will send their local vector to everyone, subsequently updating the estimate vector.

3.2.2 In the context of wireless sensor networks

Deploying an algorithm over a wireless sensor network can be challenging. WSNs are multi-hop, as the radios have a limited range. The communication is typically done over noisy and unreliable links and packet losses are common, originating mostly from noise from the environment but also from interference between nodes when they try to transmit concurrently. As such, the perfor-

mance of an algorithm can be influenced by parameters and network properties, in ways that are hard to be included in a meaningful analytical study.

Design and implementation choices in WSNs emphasize the energy footprint, as the node battery capacity is limited. The radio is by far the most energy-hungry component, regardless of its mode of operation (send, receive, listen), often consuming up to 10 times more energy than the CPU. Consequently, the goal of most communication protocols is *radio duty cycling* (RDC), i.e. ensuring the radio is kept off as much as possible. In its simplest form, the RDC layer ensures that nodes turn their radio on a fixed number of times per second (called the *channel check rate* (CCR), aka wake-up time). On transmission, nodes keep transmitting the same packet for at least $1/CCR$ seconds, as in that time all neighbors have listened. High CCR suits frequent communication: broadcasts are shorter and nodes wake up more often to receive them; low CCR might be a better choice when communication happens rarely, so nodes do not have to check the medium often.

The channel check rate is a tunable parameter of the protocol that directly affects a node's duty cycle, communication latency as well as loss rate. We study the effects of this parameter on the GM in Section 3.5, when we evaluate the feasibility of GM on power- and CPU constrained devices.

3.3 Practical GM-based threshold-monitoring: design aspects and algorithmic implementation

For GM-based continuous threshold monitoring in IoT environments, we argue that focal points are: (i) Design challenges from the application's point of view and (ii) System properties and parameters affecting the design.

3.3.1 Addressing system challenges: processing, communication

Multi-hop, all-to-all communication: In traditional WSN communication scenarios, either all nodes send to a single, fixed node (data collection) or all traf-

fic is disseminated from a single, fixed node to all the others (data dissemination). With GM's communication requirements, every node can potentially be a source of information that needs to be disseminated to all other nodes (all-to-all communication) and even concurrently with other nodes (as shown in Section 3.5). In addition, sensor nodes typically form multi-hop networks, so an individual update generated by a single node needs to be propagated through the network in a reliable manner until every node receives that update.

We consider mesh, unstructured networks that follow a simple approach for multi-hop propagation: every node that receives a packet with new information, will broadcast it further on. Obviously, this leads to an increased amount of broadcasts for every update. This is a commonly considered baseline, motivated by its inherent property that the update will eventually propagate throughout the network with a high degree of reliability, without the need to maintain a routing topology. As the goal of GM is to reduce the number of updates that need to be propagated, we expect that network-wide flooding of updates will not happen often. We evaluate this further in Section 3.5.

To implement all-to-all communication and be able to distinguish new updates from already seen ones, every packet containing an update carries a sequence number, locally generated at the node that issued the update. All nodes keep track of the last known sequence number from every other node so that, upon arrival of a new packet, they know whether it contains new information and should be propagated further, or dropped.

Recovery from losses. The related literature on GM and similar methods does not typically consider packet losses but assumes reliable message delivery. In WSNs, losses are common and an important consideration for application design.

If an update from node A fails to reach B, node B will have stale information about A and the estimate vector will be *out of sync* (with respect to A), until the next update from A. An out of sync node has an inaccurate view of the network-wide aggregate being monitored and might miss a threshold violation or report a non-existing one.

We allow updates to get lost and rely on the application layer to eventually

converge to the correct estimate vector. We evaluate the effect of losses on the *responsiveness* of the algorithm and the time that a node might be out of sync in Section 3.5.

Threshold checking complexity. As explained in Section 3.2, GM relies on computing a sphere, based on the estimate vector and the local drift and checking whether that sphere crosses a threshold surface. This can be computationally challenging, even in low dimensions, considering that the threshold surface might have an arbitrary shape. Even if there is a closed-form solution that accurately solves this problem, it can be computationally expensive. This calculation needs to be performed for every new sampled local reading and every new received update. The problem is particularly hard for sensor nodes, where the computational resources are scarce: there is often not any native-floating point support and the nodes need to go to sleep as soon as possible, to save energy.

We address this by approximating the GM-spheres (Section 3.2) with a simpler shape that makes the computations significantly simpler, while ensuring that we do not introduce false negatives. As an example in 2D, the circles from Figure 3.1 can be replaced with squares containing the former, which results in simpler boundary conditions (as it is simpler to check whether the sides of a square, rather than points on a circle, cross a surface). Obviously, there will be cases where the square check will report a violation even though the circles do not actually cross the threshold, hence sacrificing accuracy for communication reduction. However, a similar trade-off is already inherent in GM as there will be cases when spheres cross the threshold even though the full convex hull (cf. Section 3.2) would not. This relaxation might not cope with high dimensions, but in many cases of monitoring statistics such as variance or correlations between nodes, it provides a simple and efficient solution. In Section 3.6 we discuss other alternative, shape-sensitive extensions to GM that reduce the computational cost of threshold checking by approximating the threshold surface. Contrary to those methods, we simplify the threshold checking with an approximation that is particularly useful for the important case of tracking two variables (e.g. when monitoring the variance, the computation time decreases from 20ms to just 2ms).

3.3.2 Tunable system-parameters

In this section, we go through system parameters that need to be tuned as they affect how the application of geometric monitoring interacts with the network stack.

Since a node might, at any time, propagate an update, we use a network stack based on asynchronous transmissions and RDC to save energy. In this setting, the main parameter of interest is the channel check rate (CCR).

As mentioned before, CCR affects: (i) how often nodes will wake up to check the medium for possible packets transmissions and (ii) for how long a broadcasting node should keep re-transmitting a packet to make sure that all nodes receive it.

As the main goal of the GM method is to reduce the amount of updates through the network, one would generally expect GM to benefit from lower CCR values, compared to a naive approach that shares every sensor reading. On the other hand, GM's communication behaviour is highly data-dependent and unpredictable. There are periods with high activity and low activity, depending on the value that is being monitored and how close it is to the threshold. In addition, when an update is received by a node, the resulting recalculated global estimate ($\vec{u}$) might also cause a violation, forcing the node to broadcast its readings immediately, creating periods of burst traffic.

Based on the above, it is clear that: (i) CCR is a system parameter that will affect the expected energy savings of the method, and (ii) it is hard to come up with a value for CCR that can match the communication of GM at all times. We evaluate it further in Section 3.5.

3.4 Experimental methodology

We implemented geometric monitoring in Contiki [5], a well-known operating system for IoT applications. We targeted the TelosB platform and used a mainstream stack that relies on ContikiMac [4] for radio duty cycling. In this section, we assess the performance of GM in practice over a real network stack.

Experiment Setup: We run our experiments in two settings: (i) A *full-system evaluation* on the Flocklab [18] testbed, a deployment of TelosB nodes on a university building with 26 nodes on a four hop topology. Flocklab has realistic interference due to the presence of people and Wi-Fi signals and allows us to evaluate the system in a realistic environment. (ii) A *full-system simulation* on Cooja [22], a cycle-accurate simulator where the *whole network stack* is simulated in software at every node. Cooja simulates a real deployment as accurately as possible and we use it to run long experiments (up to 20 hours per configuration) that would not be possible in the testbed due to usage restrictions. The simulation platform also allows for repeatable experiments and fine-grained control over network properties (notably packet loss) to stress-test the algorithm. The topology here is similar to the testbed (26 nodes, 4 hops). We use the accurate and extensive simulations in Cooja to reproducibly uncover trends and insights, which we then validate from the real deployment in Flocklab.

Data set: As our source of data values, we use the Intel Lab data set [1], a commonly used data set in the WSN literature. It contains temperature, light, humidity and voltage readings from 54 sensors, placed inside an indoor lab, over a period of 36 days. Nodes take a new reading every 31 seconds.

We select twenty-six nodes (the ones with IDs 22-47) that have good quality of readings and use their temperature values. The temperature values follow a periodic, daily cycle with the temperature rising during the day and falling moderately during the night, partly controlled by the heating and ventilation system. Out of the 36 days, we use the first day of readings for the experiments on Flocklab and in Cooja (20 hours). In our experiments, we replay the temperature values from the data set at each node, using the same sampling frequency.

Monitoring functions: We experiment with both linear and non-linear monitoring functions, namely the global *average* and *variance* of the sensor readings. The latter is of practical use in many different scenarios, e.g. to detect the presence of an area with irregularly high temperatures i.e. a hot-spot. The former is a simple case that can be solved even without the need for geometric monitoring but we include it for completeness. For the average we choose a

Experiment label	Duration (per exper.)	Platform	Parameters	Metrics of interest
A: Full system simulations	20 hours	Cooja	CCR	Loss rate, Latency, Comm. reduct., DC, Lifetime Impr.
B: Testbed validation experiments	2 x 3 hours	Flocklab	Data set section	Loss rate, Latency, Comm. reduct., DC, Lifetime Impr.
C: Runtime insights	20 hours	Cooja	Elapsed time	DC, Number of updates
D: Accuracy / Responsiveness	10 hours / 1 hour	Cooja	Artificial loss rate	Average time out of sync, Responsiveness

Table 3.1: *Summary of the experiments presented in this section.*

threshold of $T = 20°C$ and for the variance a threshold of $T = 2°C^2$, which is crossed twice during the daily cycle. Similar to Sharfman et al. [25], we track the variance of readings in the GM framework, by having nodes tracking a local statistics vector $\vec{v} = (t, t^2)$ where t is the current temperature reading. Unless stated otherwise in the description of the experiment, the variance will be used.

Experiments: Table 3.1 summarizes the different experiments presented in this section. In experiment series *A*, we use simulations to evaluate the impact of the network stack's parameters (CCR). In experiment *B* we deploy a selected configuration on the Flocklab testbed. Experiment series *C* takes a closer look at the execution and offers runtime insights. Finally experiment series *D* evaluates the impact of packet loss on the accuracy and responsiveness of the method. As a comparison, we adopt the *baseline* method (also used in [24]) where nodes broadcast at every *epoch*, i.e. every sensor reading, as soon as they get it. We choose this baseline since: (i) there is no other method but GM that solves the problem of threshold monitoring for the general case (apart from its variants, discussed in Section 3.6) and (ii) it allows us to directly compare the theoretical savings with the ones achieved in practice (c.f. Section 3.5.1).

Metrics: For each of the experiments mentioned above, we are interested in the following metrics:
(i) *Communication reduction* achieved by GM, measuring the number of updates propagated; it is a measure of the efficiency of GM, purely from the application point of view.

(ii) *Duty cycle (DC)* i.e. the fraction of total time that a node has its radio turned on, computed using Contiki's power profiler [6]. We also define the *lifetime improvement* achieved through GM, as the reduction in duty cycle achieved through the execution of GM, compared to the baseline.
(iii) *Loss rate* measured as the number of individual destinations of a packet that fail to receive it (e.g. if a node sends an update and only 24 out of the other 25 nodes receive it, the loss rate is 1/25).
(iv) *Communication Latency* is the average time that a packet originating from node A needs to reach another node B.
(v) *Accuracy* and *Responsiveness* where the later relates to the latency of detecting an actual threshold violation. See respective definitions later in Section 3.5.4.

3.5 Evaluation from a holistic system perspective

3.5.1 Full-system simulations

In the full system simulation (Table 3.1, A), 20 hours of data are used for each configuration. We collect results both for the geometric method and for the baseline. In every configuration, the GM method achieves **4.31 times** communication reduction when monitoring the variance (Table 3.2, col 9) i.e. only 23.2% of the sensor readings are actually propagated.

We start by discussing the impact of the *channel check rate (CCR)*. Table 3.2 summarizes the results for a channel check rate ranging from 8 to 64 Hz, averaged across all 26 nodes.
Loss rate: For a CCR of 8 Hz, both the baseline and the GM method have unacceptably high loss rate and latency values. This is due to the fact that, with this configuration, a single broadcast is 1/8 seconds long and there is not enough time to propagate an update around the network before another node issues a new update. The network is thus past its operational point. For the rest of the settings, GM has slightly smaller loss rate than the baseline. GM successfully reduces the total amount of data transmitted through the network, which also

Intel Lab Dataset (20 hours), Monitored Functions: **Variance** (T=2) and **Average** (T=20)														
	Baseline (variance/average) [3]			GM, Function: variance					GM, Function: average					
CCR in Hz	**Duty cycle**	Loss rate (%)	Latency (ms)	**Duty cycle**	Loss rate (%)	Latency (ms)	Lifet. Impr.	Comm. Red.	**Duty cycle**	Loss rate (%)	Latency (ms)	Lifet. Impr.	Comm. Red.
8	8,44	9,98	18348	2,68	1,19	3862	3,15x		0,76	0,46	3716	11,04x	
12	7,73	0,59	760	2,57	0,22	590	3,01x		1,03	0,41	2465	7,50x	
16	7,28	0,18	314	2,70	0,11	336	2,69x		1,37	0,02	2078	5,31x	
24	6,88	0,07	189	3,11	0,07	212	2,21x	4,31x	1,96	0,20	1078	3,51x	44x
32	7,26	0,04	162	3,70	0,04	190	1,96x		2,68	0,97	906	2,71x	
48	7,81	0,12	139	4,88	0,12	151	1,60x		3,85	0,45	581	2,03x	
64	9,11	0,14	134	6,18	0,12	157	1,47x		5,31	1,01	703	1,71x	

Table 3.2: ***Full system simulations****: Duty cycle, loss rate and latency for the GM method vs the baseline, with varying CCR.*

reduces the probability of collisions, hence improving the loss rate.

Latency: Regarding latency, GM is better for CCR value of 12 Hz. In these configurations, the baseline version pushes the required bandwidth close to the network's saturation point (increased average latency), while the message reduction of GM is important and reduces the latency. On the other hand, on higher CCR settings, GM's latency is higher than the baseline by a small, almost constant value. We attribute this increase to the small amount of extra processing required upon reception (checking for threshold violations), before a packet can be propagated further. Reducing the processing cost (using our approximation) from 20ms to 2ms is important here for two reasons:(i) the overall latency decreases (ii) there is enough time for processing between packet receptions (for CCR value of 64 Hz, nodes wake up to receive every 16ms).

Duty cycle: We next turn our attention to duty cycle, the metric that is directly related to a node's effective lifetime. GM results in significant reduction in duty cycle. As an example (Table 3.2, col 5, CCR=12 Hz), using GM reduces the duty cycle from 7.73% to just 2.57%, a three-fold improvement. However, this improvement diminishes as the CCR increases. The lifetime improvement between the best configurations is 2.8x (compare duty cycles between 12 Hz for GM and 24 Hz for the baseline), which is far from the 4.31x communication

[3]The baseline method behaves the same way, regardless of the function.

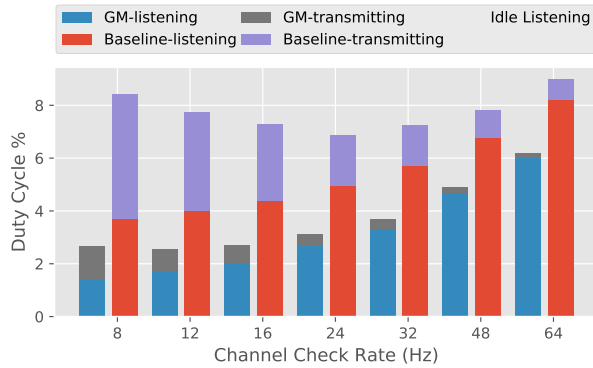


Figure 3.2: ***Full system simulations****: The duty cycle, broken down to sending and listening, as well as the cost of idle listening.*

reduction achieved by the method.

A brief look at the respective results from monitoring the *average* (Table 3.2, col 10), shows that the effects mentioned above for the variance are even more pronounced now. In this case, GM manages to reduce communication by 44 times, keeping nodes mostly quiet throughout the execution. In terms of duty cycle, GM reduces it by an impressive amount (up to 11 times for a CCR value of 8 Hz), but still, 4 times less than the achieved reduction in communication.

Duty cycle decomposition: A detailed look on the duty cycle *explains the aforementioned differences*. Figure 3.2 shows the duty cycle for GM and the baseline method (when tracking the variance), as well as its individual components. First, the percentage of the duty cycle that is spent on sending is greatly reduced using the GM method, directly matching the communication reduction ratio achieved by the algorithm. Subsequently, the GM version spends less time receiving data at each node. Also, notice that the time spent on transmitting decreases as the channel check rate grows. This is simply because broadcasts are shorter when the CCR is high. In this figure, we have also included the cost of idle listening, i.e. the cost of turning on the ratio periodically to check for traffic, even though there is nothing to receive. This cost is the same for

Flocklab Testbed, Monitored Function: **Variance**									
		Duty Cycle (%)		Comm.	Lifet.	Loss rate (%)		Latency(ms)	
CCR	Dataset section	GM	Baseline	Red.	Impr.	GM	Baseline	GM	Baseline
12	low comm. (0:00 - 03:00)	1,33	6,98	45,74x	5,26x	0,45	5,43	908	1704
	high comm. (8:00 - 12:00)	4,28		2x	1,63x	0,88		634	
24	low comm. (0:00 - 03:00)	2,05	6,42	45,74x	3,13x	0,91	0,84	223	268
	high comm. (8:00 - 12:00)	5,50		2x	1,17x	0,49		262	

Table 3.3: ***Testbed experiments****: Results from the Flocklab testbed, on two 3 hour periods from the dataset.*

both methods and is computed from the CCR value. It is evident that this cost dominates the duty cycle when the channel check rate increases. Even for small CCRs, *the idle cost represents a significant overhead*, that reduces the potential lifetime savings of the algorithm.

3.5.2 Validation through Testbed Experiments

Goal: We use the testbed experiments of this section as a way to validate the insights and trends gained from the full-system simulations that we presented above.

Experiment settings: In this section, we present the results from the execution on the Flocklab testbed (Table 3.1, B). Due to usage restrictions on the testbed, we do not replay full day measurements from the dataset. Instead we focus on sections of particular interest. We select two sections from the data set, 3 hours each (midnight and morning) where, as we detail further in the next experiment series, the communication pattern of the GM method is expected to be very different. For these experiments, we have picked a channel rate of 12 Hz, where the GM method had the lowest duty cycle on the simulations, as well as a rate of 24 Hz for comparison.

Results: Table 3.3 shows the overall results for the two sections of the dataset.[4] The topology of the testbed is slightly different than the one used in the simulation (more sparse), so the absolute values are different than Table 3.2, but we expect the general trends to hold. On the morning section (08:00 to 12:00), GM

[4]The baseline version has the same communication behaviour (every reading gets propagated) regardless of the data section, so we evaluate it only on the midnight section.

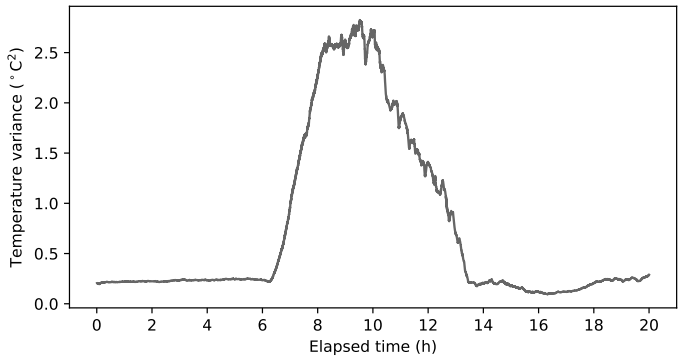
communicates 2 times less than the baseline. The lifetime improvement follows closely, and the duty cycle is reduced by 1.63 times. For the midnight section (00:00 to 03:00), GM achieves remarkable communication savings, reducing the number of readings that need to be propagated by 46 times, but the associated lifetime improvement is more modest (5.26 times). This indicates that, during this section (and unlike the previous one), even a check rate of 12 Hz is excessively high, and *most of the energy is spent on idle listening*. Similar results can be seen when CCR is set to 24 Hz. Here, the lifetime improvements decrease for both data set sections, especially for the period with low communication (3,13x lifetime improvement).

3.5.3 Runtime insights: a closer look

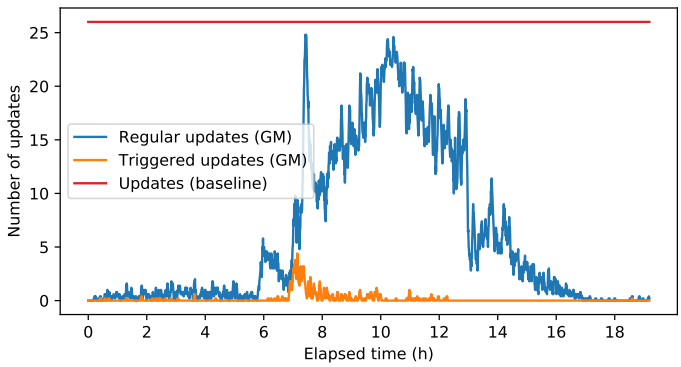
In Section A, we presented overall results after the completion of each 20 hour long experiment. We now take a closer look into a single experiment and provide insights for the communication behaviour of the algorithm (Table 3.1, C). We set the CCR to 12 Hz (that resulted in the best duty cycle for the GM case) and elaborate on detailed insights from the execution of the GM method, in order to distill deeper insights about the interplay between the algorithm and the communication stack.

Monitored value: Figure 3.3a shows the actual value that is being monitored: the variance of the temperature readings. Due to variation in the temperature between different rooms during working hours, the data set exhibits a period of approximately 7 hours where readings between nodes have increased variance, up to 2.8 $^{\circ}C^2$.

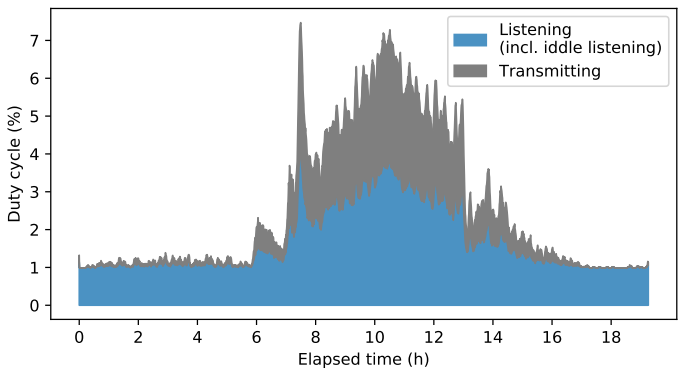
Update decomposition: Next, we count the number of updates that nodes propagate at every *epoch* of the execution. Recall that the epoch is defined as the sampling at which nodes take new measurements (for the Intel Lab data set this period is 31 seconds) and that the baseline method broadcasts all of them. For the GM method, we distinguish between two kinds of updates. *Regular updates* happen when a node gets a new sensor reading, detects a threshold violation and therefore decides to propagate this reading to all the other nodes. When



(a) Variance of the temperature between nodes.

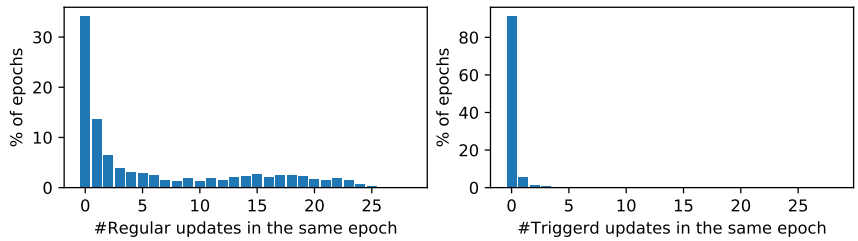


(b) Number of updates per epoch.



(c) The average duty cycle across nodes, during the execution.

Figure 3.3: ***Runtime insights*** *from the execution of GM over a period of 20 hours.*



(a) Regular updates. (b) Triggered updates only.

Figure 3.4: ***Runtime insights:*** *Percentage of epochs with concurrent updates (regular on the left part and triggered on the right part)*

other nodes received that update, they update their estimate vector and check for a threshold violation. If there is one, that node will in turn propagate the last stored reading to everybody. We call the latter a *triggered update*, since it is triggered by the reception of an update from another node and not by a local sensor reading. Triggered updates are interesting from the communication protocol point of view, because they cause traffic bursts where nodes want to concurrently share information across the network.

Temporal variation in comm. reduction: In Figure 3.3b we show the number of updates per epoch, for the GM method as well as the baseline, computed over a 1.5 minute sliding window and averaged across all nodes. Note that the x-axis has been translated to reflect the elapsed time in hours, in order to match Figure 3.3a. The baseline induces the same number of updates per epoch, equal to the number of nodes. On the contrary, the GM method significantly reduces the number of sensor readings that need to be updated per epoch. Especially during the periods when variance is small and away from the threshold, almost all communication is suppressed. As the variance comes closer to the specified threshold, nodes start detecting frequent violations and update more of their readings. We can also see a period close to the threshold where these updates trigger violations on other nodes, as well as a peak in the number of updates when the actual value of the variance is close to the threshold 2 $^{\circ}C^2$.

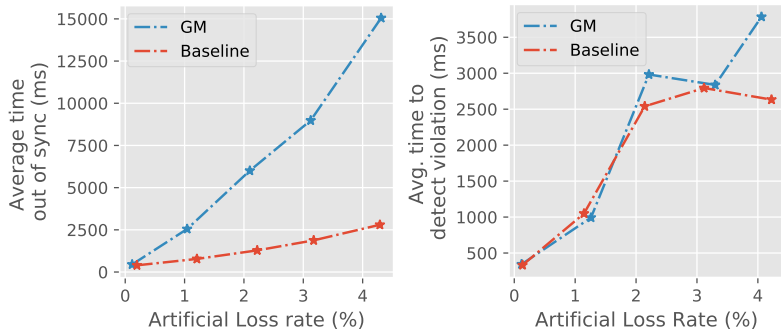
Figure 3.4 presents a different view of the same results. Here, we count the percentage of epochs with *concurrent* updates, both regular and triggered. Figure 3.4a shows that a significant amount of epochs (34%) do not contain any regular updates, but the majority of epochs contain concurrent updates. Also, *a noticeable percentage of epochs contain many concurrent updates* (15-26 concurrent updates). Triggered updates occur very rarely in this data set (Figure 3.4b): 91% of epochs do not have any triggered updates. However, there are epochs where several updates are triggered concurrently.

Temporal variation in duty cycle: In Figure 3.3c we take another look at the duty cycle and monitor how it changes during the execution. Here, the duty cycle is computed at every epoch and averaged across all nodes. The duty cycle follows the same trend as the number of updates in Figure 3.3b: at periods where communication is high, the radio needs to stay on longer in order to send or receive the extra traffic. From Figure 3.3c, it is also evident that *the idle listening cost is a dominant factor* that affects the duty cycle and limits the potential of the GM method: even during periods with no activity, nodes waste a constant amount of time to check the radio for transitions.

3.5.4 Accuracy/Responsiveness: The effect of packet losses

In this section we investigate the effect of losses on the *accuracy* and *responsiveness* of the algorithm.

Under the assumption of no packet loss, nodes should always have the same estimate vector (described in section 3.2) and the same "view" of the network-wide aggregate value to be monitored. In practice, losses are common and will cause the estimate vector on different nodes to drift. We want to measure and quantify this effect, as the packet loss rate increases. In the following, we will state that a node A is "out of sync" with respect to node B, when its global estimate is different from node B, either: (i) because an update from node B is still "in flight" or (ii) because an update from node B was lost. In the first case, node A will stay out of sync until the "in flight" update arrives. In the second case node A will stay out of sync until a later update from B successfully arrives.



(a) Average time (in ms) that nodes are out of sync

(b) Average time (in ms) to detect that the threshold was crossed.

Figure 3.5: ***Accuracy/Responsiveness*** *of the algorithm measured as we introduce packet losses.*

Define as *accuracy* the average time a node is out of sync with respect to any other node. Define also the *responsiveness* of the algorithm as the average time elapsed from the moment the value of the monitored function has crossed the predefined threshold, until the moment a node actually detects this. Note that accuracy is just a measure of the time nodes have stale information with respect to each other, which might not necessarily be an issue if the network aggregate is not close to the threshold. On the contrary, responsiveness captures the *critical time* during which the threshold has been actually crossed and the node has not yet detected it.

We next evaluate both metrics (cf. Table 3.1, D). We run simulations where we intentionally introduce packet losses, with a controlled rate, at each node and report the resulting loss rate. The CCR value is set to 16 Hz.

Figure 3.5a reports the average duration that a node is out of sync, for increasing loss rates. For small loss rates (0.1% approximately) nodes are out of sync mostly until "in flight" packets arrive (400 ms). Overall, both methods stay out of sync longer as we introduce packet losses, since more and more nodes will miss updates and will have to wait for at least one full epoch to get back in

sync. GM is affected to a much greater degree, simply because some nodes will suppress their transmissions, keeping others out of sync for longer.

Figure 3.5b shows how the *responsiveness* of the two methods changes as we introduce losses. On low loss rates (0.1%) nodes in both version quickly detect the threshold violations, within approximately 300 ms. The time to detect the violation increases rapidly for both versions as more losses are introduced. Even with a little over 1% loss rate, it takes 1 second on average to detect the violation, for both versions. On higher loss rate values, GM seems to take longer to detect the violation, since more and more nodes are out of sync and have stale information. Note that the gap between GM and the baseline is not as large for responsiveness as for accuracy. This is because, close to when the threshold violation happens, nodes in GM detect local violations frequently, so the behaviour becomes similar to the baseline, for a short time interval, until the threshold is exceeded.

Overall summary of results: The results show how GM is of practical use in IoT environments. The presented extensive simulations and testbed runs' outcomes suggest that GM can indeed bring several fold reduction to duty cycle. However, the full system perspective reveals that the benefits achieved in practice can be far from the expected ones, due to the overhead of idle listening. Also shown in the evaluation is the insight that to avoid that cost is hard in GM, due to changes in the communication pattern: during its execution, GM has periods of little to no activity that would benefit from a low CCR, as well as periods with high traffic that need a higher CCR value. This opens interesting questions as to whether approaches that dynamically choose CCR values [20, 21] would be efficient in managing the data-dependant communication pattern of GM. Finally, we show that packet loss has a direct impact on the accuracy and responsiveness of the algorithm, suggesting that the network stack should be tuned to ensure a low loss rate at all times, especially during the critical periods where the global estimate crosses the threshold. This also suggests that, for applications with no strict requirements on responsiveness, it might be beneficial to sacrifice some responsiveness in favor of a lower duty cycle (e.g. by relaxing the "eagerness" of the propagation protocol).

3.6 Related Work

In this section, we outline related work beyond the original concept of GM described in Section 3.2.

Geometric Monitoring (GM): In [10], the authors orthogonally augment GM with sketches, that further reduce the communication cost by keeping track of an approximation on the network-wide aggregate. Giatrakos et al. [11] combine the communication reduction of GM with prediction models that track the temporal evolution of sensor readings and only report when the model needs to be updated. A summary of the use of GM for query tracking in distributed streaming systems can also be found in [9]. This interest has been motivating also for the work in our paper.

In [15], the authors introduce shape sensitive geometric monitoring, that takes into account properties of the monitored function. Lazerson et al. [17] propose a variant of GM that addresses the computational complexity of checking for violations. They introduce a method of approximating the monitored function to convex/concave components, so that it can be checked for violations fast, without the need to construct the respective spheres. The approximation we propose in our work is orthogonal, in the sense that we leave the function intact and instead bind the local area that nodes have to keep track of for violations.

GM has been studied in the general context of WSN from a high level perspective [2, 25]. In [25] the authors present an adaption of GM that is designed for clustered topologies. In [2] GM is used for detecting outliers in the readings of wireless sensor nodes. In both of these lines of work, the network is only considered as a communication abstraction and practical systems aspects are not considered. Differently, we take a full system perspective and consider practical aspects such as the effect of the RDC protocol and packet loss.

Data prediction and aggregation for WSNs: For data aggregation in WSNs, usually the goal is to collect at a single, sink node, the sensor readings from every other node in the network. A significant amount of research effort has been put on reducing communication by aggregating data along the way from sources to a destination [13, 16, 19].

Data prediction extends this design space by building models that approximate (within some error guarantees) the sensed values at every node. An update is only sent to the sink when a node's value deviates from the ones predicted by the model [3]. For example, Raza et al. [23] propose a data prediction method and are the first to evaluate it on real sensor nodes. Similar to the spirit of our work and through a full system evaluation, they find that the sleeping interval of the MAC protocol and the cost of maintaining a topology have significant effects on the practical lifetime of the nodes. Istomin et al. [14] extend these findings and propose Crystal, a protocol based on synchronous transmissions that is tailored for the communication requirements of data prediction.

Differently from data prediction, in GM there is no need to model the physical quantities sensed at the nodes, which is often challenging and requires expert knowledge. Moreover, in data prediction, nodes model their stream of readings locally without dependencies on neighbouring nodes, whereas the threshold monitoring problem that GM addresses is inherently distributed, hence the work in [23] and [14] is not readily applicable to distributed threshold monitoring.

3.7 Conclusions and Future Work

Inspired by important results on the problem of continuous monitoring, we take a full-system approach on the applicability of Geometric Monitoring (GM) on real IoT environments. In particular, we focus on processing and communication , as well as a cross-layer perspectives. We provide a method that simplifies the threshold checking for resource-constrained IoT devices, with an approximation that is particularly useful for many common cases. We also study the GM-communication interplay: we confirm several-fold reduction in communication which in turn leads to battery lifetime improvements on the nodes. We observe however that the resulting improvement falls short of the theoretical savings, which, as our results underline, is due to the baseline energy overhead of the network stack. Moreover, we show that packet losses have a magnified effect on the accuracy and responsiveness of the algorithm. Both the above

motivate cross-layer approaches for practical purposes. We expect that these insights will enable the design of custom protocols and cross-layer optimization techniques that will unlock the full potential of GM threshold monitoring applications for IoT systems.

Bibliography

[1] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. Intel Lab Data. `http://db.csail.mit.edu/labdata/labdata.html`, 2004.

[2] S. Burdakis and A. Deligiannakis. Detecting Outliers in Sensor Networks Using the Geometric Approach. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1108–1119, April 2012.

[3] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, 2004.

[4] A. Dunkels. The ContikiMac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science, 2011.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov 2004.

[6] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, pages 28–32, New York, NY, USA, 2007. ACM.

[7] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to ddos attack detection and response. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 303–314 vol.1, April 2003.

[8] Z. Fu, M. Almgren, O. Landsiedel, and M. Papatriantafilou. Online temporal-spatial analysis for detection of critical events in cyber-physical systems. In *IEEE Int'l Conf. on Big Data*, pages 129–134. IEEE, 2014.

[9] M. Garofalakis. Approximate geometric query tracking over distributed streams. *IEEE Data Eng. Bull.*, 38(3):103–112, 2015.

[10] M. Garofalakis, D. Keren, and V. Samoladas. Sketch-based Geometric Monitoring of Distributed Stream Queries. *Proc. VLDB Endow.*, 6(10):937–948, Aug. 2013.

[11] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. Prediction-based geometric monitoring over distributed data streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 265–276, New York, NY, USA, 2012. ACM.

[12] V. Gulisano, M. Almgren, and M. Papatriantafilou. Metis: a two-tier intrusion detection system for ami. In *Int'l Conf. on Security and Privacy in Comm. Sys.*, pages 51–68. Springer, 2014.

[13] V. Gulisano, M. Almgren, and M. Papatriantafilou. When smart cities meet big data. *Smart Cities*, page 40, 2014.

[14] T. Istomin, A. L. Murphy, G. P. Picco, and U. Raza. Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 83–95, New York, NY, USA, 2016. ACM.

[15] D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape Sensitive Geometric Monitoring. *IEEE Transactions on Knowledge and Data Engineering*, 24(8):1520–1535, Aug 2012.

[16] L. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 575–578, 2002.

[17] A. Lazerson, D. Keren, and A. Schuster. Lightweight monitoring of distributed streams. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1685–1694, 2016.

[18] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 153–165, April 2013.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.

[20] C. J. Merlin and W. B. Heinzelman. Duty cycle control for low-power-listening mac protocols. In *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, pages 497–502, Sept 2008.

[21] X. Ning and C. G. Cassandras. Optimal dynamic sleep time control in wireless sensor networks. In *2008 47th IEEE Conference on Decision and Control*, pages 2332–2337, Dec 2008.

[22] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 641–648, Nov 2006.

[23] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical Data Prediction for Real-World Wireless Sensor Networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2231–2244, Aug 2015.

[24] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 301–312, New York, NY, USA, 2006. ACM.

[25] I. Sharfman, A. Schuster, and D. Keren. Aggregate threshold queries in sensor networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.