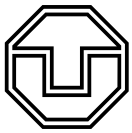


TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Professorship of Systems Engineering

COMMUNITY-BASED INTRUSION DETECTION

Stefan Weigert

DISSERTATION

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Professorship of Systems Engineering

COMMUNITY-BASED INTRUSION DETECTION

Stefan Weigert
Born on: 24th January 1983 in Dresden, Germany

DISSERTATION

to achieve the academic degree

DOKTOR INGENIEUR (DR.-ING.)

Supervisor
Prof. Christof Fetzer, Ph.D.
Technische Universität Dresden, Germany

Advisor
Prof. Dr. rer. nat. Hermann Härtig
Technische Universität Dresden, Germany

Submitted on: 4th March 2015

PREFACE

Today, virtually every company world-wide is connected to the Internet. This wide-spread connectivity has given rise to sophisticated, targeted, Internet-based attacks. For example, between 2012 and 2013 security researchers counted an average of about 74 targeted attacks per day. These attacks are motivated by economical, financial, or political interests and commonly referred to as "Advanced Persistent Threat (APT)" attacks. Unfortunately, many of these attacks are successful and the adversaries manage to steal important data or disrupt vital services. Victims are preferably companies from vital industries, such as banks, defense contractors, or power plants. Given that these industries are well-protected, often employing a team of security specialists, the question is: How can these attacks be so successful?

Researchers have identified several properties of APT attacks which make them so efficient. First, they are adaptable. This means that they can change the way they attack and the tools they use for this purpose at any given moment in time. Second, they conceal their actions and communication by using encryption, for example. This renders many defense systems useless as they assume complete access to the actual communication content. Third, their actions are stealthy — either by keeping communication to the bare minimum or by mimicking legitimate users. This makes them "fly below the radar" of defense systems which check for anomalous communication. And finally, with the goal to increase their impact or monetisation prospects, their attacks are targeted against several companies from the same industry. Since months can pass between the first attack, its detection, and comprehensive analysis, it is often too late to deploy appropriate counter-measures at businesses peers. Instead, it is much more likely that they have already been attacked successfully.

This thesis tries to answer the question whether the last property (industry-wide attacks) can be used to detect such attacks. It presents the design, implementation and evaluation of a community-based intrusion detection system, capable of protecting businesses at industry-scale. The contributions of this thesis are as follows. First, it presents a novel algorithm for community detection which can detect an industry (e.g., energy, financial, or defense industries) in Internet communication. Second, it demonstrates the design, implementation, and evaluation of a distributed graph mining engine that is able to scale with the throughput of the input data while maintaining an end-to-end latency for updates in the range of a few milliseconds. Third, it illustrates the usage of this engine to detect APT attacks against industries by analyzing IP flow information from an Internet service provider. Finally, it introduces a detection algorithm- and input-agnostic intrusion detection engine which supports not only intrusion detection on IP flow but any other intrusion detection algorithm and data-source as well.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Christof Fetzer for giving me his trust and providing me with the freedom to do what I was really enthusiastic about. He also gave me the opportunity to work on several exciting projects — one of which allowed me to visit AT&T Labs Research where I had the pleasure to work with Matti Hiltunen. To this day I remember and value the many hallway conversations with other researchers and the meetings with Matti and his colleagues. He has since been a continuous source of ideas and motivation and it is thanks to him I could use the infrastructure, provided by AT&T.

I am also very thankful to Etienne Rivière for his strong commitment to StreamHub. Apart from Etiennne's and Pascal Felber's supervision and engagement, StreamHub would not have been possible without the help of Emanuel Onica and Raphaël Barazzutti who contributed libfilter and Jean-François Pineau who contributed libcluster. Finally, Raphaël also helped with the comparison of StreamHub to Padres and generated the input subscriptions.

As much as I want to thank my direct co-authors I also feel the need to express my gratitude to my colleagues and friends Diogo and Martin. Our discussions have always been a source of inspiration for me and more than once turned desperation into courage. I will always remember the many funny moments and fruitful conversations Diogo, Martin, Sebastian, Jons and I had in our time together, sharing the same office. Guys, I wish you all the best for you and your families!

I also want to thank my parents who supported and encouraged me throughout my entire life, who were so dedicated to my education that they spent their evenings explaining things like trigonometry to their son who (again) hadn't prepared himself for the next day's exam.

Finally, I want to thank Annett for supporting me throughout all these years, for proof-reading this dissertation, and for making sure I would not miss out on "real life". Since 16 months our son Tonio is supporting her in that endeavor with great success. You both rock my world!

CONTENTS

1 INTRODUCTION

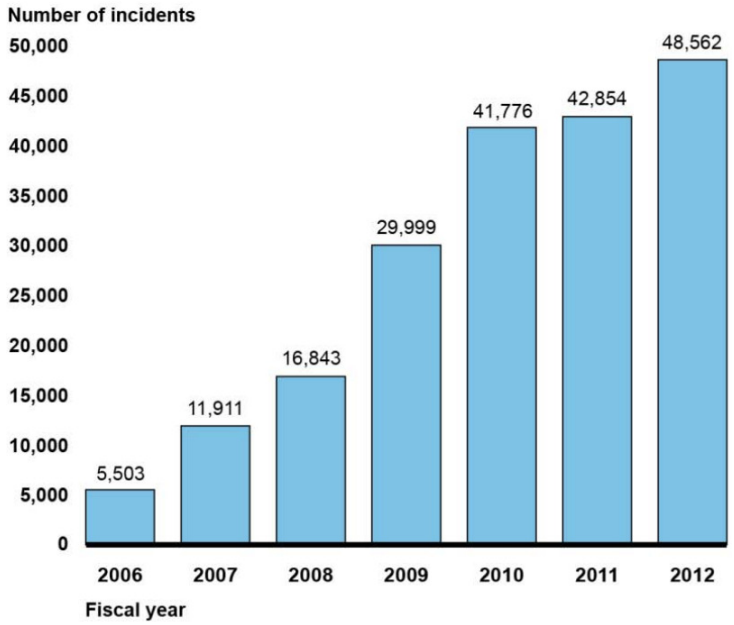


Figure 1.1: Number of incidents reported to US-CERT between 2006 and 2012 [Wil13]

The Internet has become a major driving force for economies around the world. Between 2006 and 2011 alone the Internet was responsible for 21% of the gross domestic product (GDP) growth[1], creating 2.6 jobs for every single job lost (due to the rise of the Internet), as reported by McKinsey [Rau+11]. The study also finds that 75% of Internet impact on economies arises from traditional industries and that businesses making heavy use of web technologies grow and export twice as much as others. Given these observations, it comes as no surprise that today virtually every company world-wide is connected to the Internet. Unfortunately, this connectivity also exposes the companies to a vast array of online security threats. Even more so because running their daily business on the web requires their business data to be accessible from the Internet. This creates a financial incentive to attack these companies — customer credit card information, for example, can be sold on the black market or used for blackmailing. It also creates an economical incentive — industrial espionage, while not new, is now expanding into cyberspace. Finally, it creates a political incentive — terrorists and even states use cyberattacks to enforce their political agenda.

A report published by the United States Government Accountability Office [Wil13], offers a glimpse of how severe this problem has become. The authors counted the number of incidents reported by US-based federal agencies. From 2006 to 2012 alone this number has increased by 782%, as shown in Figure 1.1. While 37% of the incidents were still under investigation at the time the report was published, all but 7% of the remaining incidents were successful attacks.

Now the question is: How does such an attack work? In order to gain access to a system, attackers must find a vulnerable part in the IT infrastructure that they can exploit. Such vulnerabilities may be found in the software-stack as programs contain bugs or have been misconfigured, for example. Vulnerable are, however, also the humans who operate the software-stack for their daily business: For instance, email attachments are opened without authenticating the sender or prior virus scans. In fact, targeting people's carelessness has a long and often successful tradition in human history and mythology. For example, after the ten year siege of Troy during the Trojan War, the Greeks built a large wooden horse with a hidden force of soldiers inside. They withdrew their remaining forces, seemingly terminating

[1]The study selected 13 countries which account for more than 70% of the global GDP.

the siege. But they left the large wooden horse — the perfect war trophy in the eyes of the triumphant Trojans. The Trojans pulled the horse inside the city walls of Troy and with it their inevitable defeat. The very same tactic is used by attackers in cyberspace today: Since it is hard to break into a computer system from the outside, they try to make insiders (e.g., employees of a company) their involuntary allies — for example, by tricking them into executing malicious email attachments. Such a malicious attachment is accurately referred to as a "Trojan" or "Trojan Horse".

Since neither software nor humans can be guaranteed to be invulnerable, IT infrastructures are secured by a multitude of defense mechanisms, such as firewalls to restrict the number of open ports or credential management to restrict the accessibility of valuable assets to a small number of users. However, all defense mechanisms are ultimately part of the software stack and operated by humans and, therefore, suffer from the same fundamental problems. It is hence desirable to at least detect when an attacker has compromised a part of the infrastructure in a timely manner in order to deploy adequate countermeasures. The research area that deals with detecting that a part of a system has been compromised is called *intrusion detection*.

1.1 OVERVIEW

An *Intrusion Detection System (IDS)* "[...] discriminates intrusion attempts and intrusion preparation from normal system usage" [Hal95]. These systems are based on the hypothesis that security violations can be detected by monitoring certain aspects of the system [Den87]. Such aspects range from network traffic to OS-level events (e.g., system-calls) to the data, stored on disk, or a combination of those resulting in a manifold of different approaches. In order to organize these different approaches into a common framework, Reilly and Stillman [RS98] proposed the *Common Intrusion Detection Framework (CIDF)*. Among its suggestions, two have since influenced the classification of research contributions in the field of intrusion detection. First, they suggest to distinguish IDS by *where* events are monitored. Related work mostly concentrates on either network events (Network-based Intrusion Detection — or NIDS), changes to files on disk (Storage-based Intrusion Detection — or SIDS), or changes on the host operating system itself (Host-based Intrusion Detection — or HIDS). Second, they suggest to further distinguish by *how* these systems analyze such events. Here, the research community is divided between signature-based analysis, which searches for pre-defined patterns in events, and behavior-based analysis, which aims at modeling normal system behavior (e.g., by means of machine learning).

A rough overview of this classification is given in Figure 1.2. Unsurprisingly, all design choices come with benefits and drawbacks. Here, we give some examples. While signature-based IDS such as Snort [Roe99] detect known vulnerabilities reliably, they fail at detecting previously unknown attacks. Behavior-based IDS are not restricted to detecting known attacks, but suffer from an inherently high false-positive rate, preventing their adoption in the industry [SP10]. Host-based IDS can monitor a vast array of events - going as far as creating normal usage patterns on a per-user basis [YD03]. However, as we will show, once attackers gain control over a computer, they may manage to shut down a host-based IDS before it can detect the intrusion (e.g., Stuxnet). Network-based IDS are hard to attack since they are usually running on a different machine, but they can only act on network events. Finally, storage-based IDS [Pen+03] can directly operate on a storage device and are therefore hard to tamper with. However, they are restricted to storage-events only.

The remainder of this chapter is concerned with examining some recent high-impact attacks to understand whether there are important commonalities among them that can be of use for our analysis. These commonalities guide the selection of the security challenges which a community-based IDS needs to solve. This is followed by a presentation of an architecture

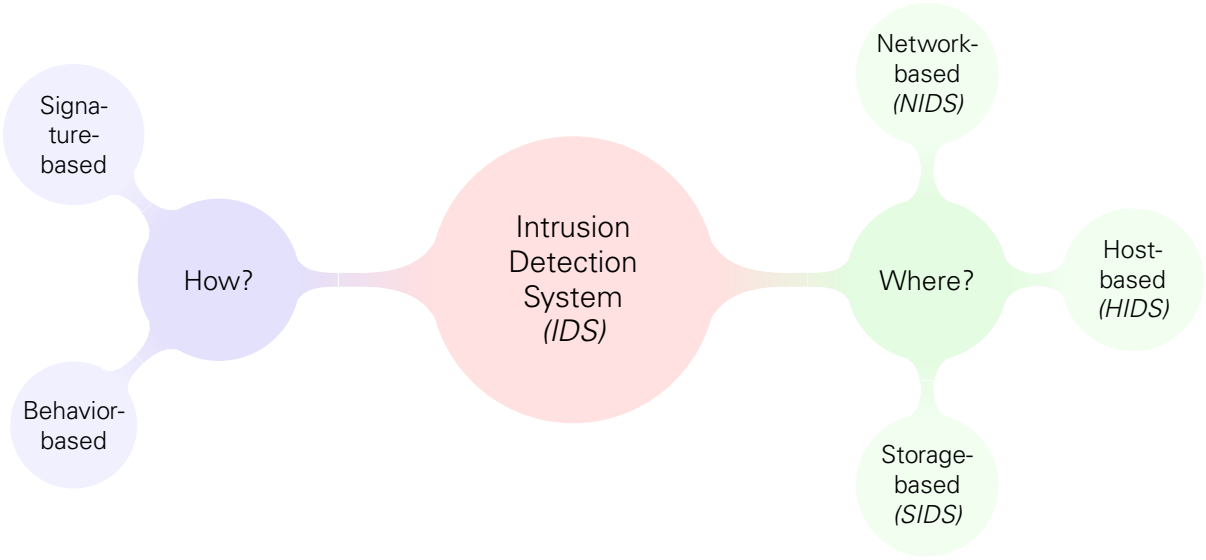


Figure 1.2: Intrusion Detection Overview

for a community-based IDS tackling those security challenges. The main contributions of this thesis conclude this chapter.

1.2 SECURITY THREATS

Wuesst and Candid [Wue14] monitored targeted attacks worldwide in a period from July 2012 to June 2013. The study, published by Symantec, observed an average of 74 of such attacks per day with the most common targets being the government/public sector, directly followed by the energy sector. From their analysis, the authors conclude that cyberespionage and sabotage attacks are becoming increasingly common. However, even more worrisome is their statement that these attacks also become increasingly more sophisticated and that individual campaigns vary significantly in exploited vulnerabilities and malware in use.

Such targeted and sophisticated attacks are classified as *Advanced Persistent Threat (APT)* attacks [Bej10]. To understand more precisely what sets them apart from ordinary attacks, we use the definition of an APT attack as proposed by Juels and Yen [JY12]. They divide an APT attack into four different stages:

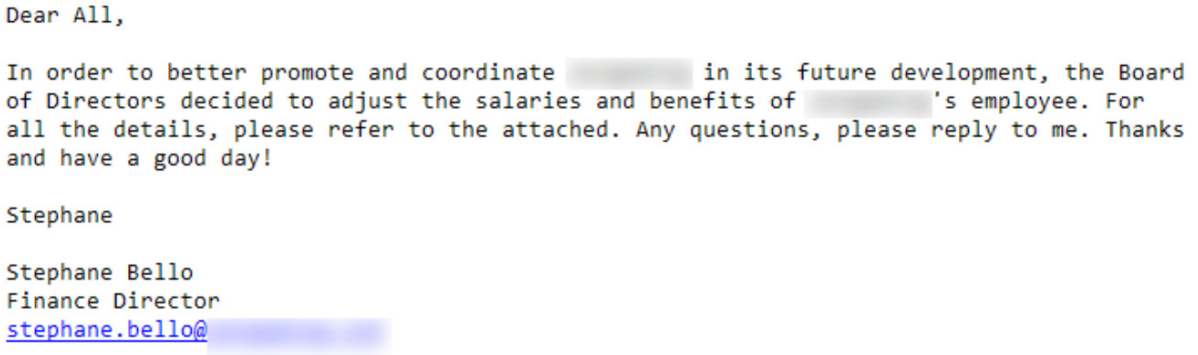
1. *Social Engineering:* Instead of directly targeting the IT infrastructure itself, attackers first target the humans behind those systems by researching their social background and crafting highly personalized phishing emails, so called *spear phishing* [Hon12]. Although these emails are often considered trustworthy by the receiver, they contain malware - either directly as attachment or by linking to websites containing malware. An example is given in Figure 1.3.

2. *Command-and-Control:* The malware usually opens a backdoor on compromised machines which opens those machines to remote control from a command-and-control server.

3. *Lateral movement:* The attacker uses the compromised infrastructure to spread the infection further. This may be facilitated by using stolen credentials, elevated privileges, or by exploiting software vulnerabilities.

4. *Data exfiltration:* Finally, the attacker downloads the desired data to the command-and-control server. To conceal the action the data is compressed and encrypted.

In the end, Juels and Yen summarize their definition in a single statement: "An APT isn't a playbook. It's a campaign" [JY12]. This refers to the fact that APT attacks are not automated and, therefore, need a command-and-control facility in order to complete their objective. As we will see, many of the recent high-impact attacks classify as APT attacks. After looking closer at some of them, we will extract commonalities among them which will guide the design of our system.


Dear All,

In order to better promote and coordinate [redacted] in its future development, the Board of Directors decided to adjust the salaries and benefits of [redacted]'s employee. For all the details, please refer to the attached. Any questions, please reply to me. Thanks and have a good day!

Stephane

Stephane Bello
Finance Director
stephane.bello@[redacted]

Figure 1.3: Example for a spear-phishing email which was involved with the "Winnti" attacks. The picture is taken from the Kaspersky Labs report [Lab13].

1.2.1 ATTACKS AGAINST THE CHEMICAL INDUSTRY [CO11]

The "nitro attacks" are a set of attacks against the chemical industry in the United States. These attacks started in late July and continued into mid-September 2011. Their goal was to download intellectual property. To this end, the attackers carefully crafted spear-phishing emails in order to trick employees into opening a malicious attachment. By opening the attachment, a malware was installed and this malware then opened a connection with a command-and-control server. The connection was encrypted to conceal the attack and used port 80 since most firewalls allow traffic through this port. The information exchanged contained details about the network the infected host was connected to in order to support the lateral movement. It was also possible to receive new instructions from the command-and-control server over this connection. The initial attack was launched from a VPS (virtual private server) in the US, registered with a static IP. This IP also hosted one of the command-and-control servers.

> "This attack campaign focused on the chemical sector with the goal of obtaining sensitive documents such as proprietary designs, formulas, and manufacturing processes." [CO11]

1.2.2 ATTACKS AGAINST THE ENERGY INDUSTRY [Wue14]

In recent years there have been several attacks against the energy industry. Even now, as of writing this thesis, Reuters quotes the FBI on warning government and energy agencies about an ongoing highly sophisticated, targeted attack [Fin+14].

Researchers from Symantec have looked at the history of such attacks [Wue14]. In particular they investigated the three most sophisticated and damaging attacks: Stuxnet, Night Dragon and Shamoon/Disttrack.

STUXNET

The first version of the command-and-control servers of Stuxnet were registered in November 2005 while the first appearance of the malware itself was in November 2007. Lateral movement was difficult as its main target, industrial control systems, are rarely connected to the Internet. Therefore, Stuxnet was not only able to spread through the network but also through storage media (e.g., USB). The ultimate goal was to sabotage gas pipelines and power plants. To this end, it employed a vast array of attack vectors, such as zero-day exploits, rootkits, detection evasion, code injection, network infection, peer-to-peer updates, and a command-and-control interface. In fact, Stuxnet's activity was monitored by wiretapping the encrypted traffic to its command-and-control servers [FMC11].

NIGHT DRAGON

Night Dragon was an attack against global oil companies, directed at finding project and financial details about oil and gas field exploration and bids. It started in late 2009 and was discovered in 2010. The attackers first infected public facing web servers of the companies and then used the infected servers to contaminate internal computers in order to advance to the lateral movement stage. The attack was carried out through a trojan on infected computers that communicated back to a command-and-control server in order to exchange information and commands.

SHAMOON/DISTTRACK

The goal of Shamoon/Disttrack was to take down as many computers as possible. These attacks took place in August 2012. It is assumed that the attackers were so called "hacktivists" who use the impact of their attacks for political reasons. The attack consisted of a dropper, a wiper and a reporter module. The dropper installed all necessary files and made sure it was automatically started on each reboot. For lateral movement, it tried to copy itself to as many network shares as possible. On a hard-coded date, the wiper would wipe the system's harddisk. Subsequently, the reporter used HTTP/GET to sent the domain-name, the IP address and the number of overwritten files back to the command-and-control server.

CARETO - THE MASKED APT

Careto [Lab14] was a sophisticated attack targeted at the government and the energy/oil sector. It used spear-fishing to make victims click on infected websites, which in turn resulted in an automatic download of malware. The malware collected files as well as VPN configurations, SSH keys and additional private information, necessary for further spread and espionage. It communicated with a command-and-control server through an encrypted channel. This connection was used for data exfiltration. Moreover, Careto executed any set of instructions coming from the command-and-control server.

All detected command-and-control servers had statically assigned IP addresses. While the height of the attack took place in 2012, it was still ongoing in 2013. The first verified attacks even date back as far as 2007. The malware exists for Windows as well as Mac OS X. There are also malware-traces of Android, iOS and Linux, but Kaspersky was unable to retrieve samples.

1.2.3 ATTACKS AGAINST THE DEFENSE INDUSTRY [Fis12]

While it is unknown whether this attack is still active, it was reported as ongoing in summer 2012 by security researchers. Similar to previous attacks, the adversaries sent spear-phishing

emails to employees containing a malicious PDF attachment which is actually an executable. This program reaches to a command-and-control server for instructions. In order to remain undetected, the attackers gained control of servers in universities that are deeply involved with DARPA research efforts. These servers were most probably used as proxies to the central command-and-control server as they used a tool that bounces traffic between hosts (*HTran*[2]). The attack was targeted at US government and defense contractors.

1.2.4 ATTACKS AGAINST THE ONLINE GAMING INDUSTRY [Lab13]

Kaspersky Labs uncovered a series of attacks (called "Winnti") against online video gaming servers. Their attention was first drawn to this attack in autumn 2011, although its first occurrences date back as far as 2010. As of spring 2013 the attack was still ongoing. Kaspersky suspects a Chinese group is behind these attacks with the goal of stealing source code, architectural design concepts, and digital certificates.

Interestingly, the authors note that while "[. . .] it is tempting to assume that Advanced Persistent Threats (APTs) primarily target high-level institutions: government agencies, ministries, military and political organizations, power stations, chemical plants, critical infrastructure networks, and so on [. . .] any company which hosts data that can be effectively monetized is at risk from APTs" [Lab13].

The Winnti attackers managed to install the trojan on online game servers either by using spear-phishing emails or by exploiting previously stolen digital signatures. The malware that was installed on the servers allowed for complete remote control access and used a special network driver to communicate to the command-and-control server. With that, its network connections remained invisible to tools like netstat or tcpview. In order to evade detection while communicating with the command-and-control server, messages were compressed, omitting the compression-header for obfuscation, and in later versions the messages were also encrypted. Message transmission was facilitated via TCP, using ports 53 (DNS), 80 (HTTP), and 443 (HTTPS). The actual examination of infected computers was carried out in a completely manual fashion by the attackers using the command-and-control facility.

1.2.5 CARBANAK: ATTACKS AGAINST THE FINANCIAL INDUSTRY [Lab15]

Carbanak is an ongoing attack against the financial sector, targeting banks world-wide. It uses spear-phishing emails to make victims open malicious attachments (usually MS Word documents). Once a computer is compromised, the attackers install Ammyy, a remote control software, or an ssh-server. Ammyy is preferred as it is also often used by administrators and is, therefore, often white-listed in firewalls. Most networks were compromised for 3 to 4 months until actual action was taken. During these months the attack is in the lateral movement phase where the attackers try to compromise services or devices that are directly involved in financial transactions, such as ATMs. However, the attackers do not only use this time to get access to the right critical systems and customers. During these months, they actually learn how the employees execute their daily business. They were also able to hack video surveillance and literally look over the employees' shoulders to steal passwords, for example. With that knowledge the attackers are able to mimic the employee's actions, rendering the attack almost invisible.

Researchers from Kaspersky assume that the attackers have extensive knowledge of the software and networks used in banks. One of their tricks was to use regular maintenance software for ATMs to withdraw money and have an accomplice pick the money on-site. They also used SWIFT directly to transfer money to their account. The command-and-control servers that could be examined so far contained secret bank documents, such as emails, manuals,

[2] HTran: `http://www.secureworks.com/cyber-threat-intelligence/threats/htran/`

crypto keys, passwords and more. Sergey Golovanov, Principal Security Researcher at Kaspersky Lab's Global Research and Analysis Team stated, that "these bank heists were surprising because it made no difference to the criminals what software the banks were using. So, even if its software is unique, a bank cannot get complacent. The attackers didn't even need to hack into the banks' services: once they got into the network, they learned how to hide their malicious plot behind legitimate actions. It was a very slick and professional cyber-robbery".

1.2.6 SECURITY CHALLENGES

In retrospect it seems that the applied intrusion detection techniques have failed at protecting the aforementioned systems. Virvilis and Gritzalis [VG13] came to a similar conclusion. They analyzed several recent APT attacks that were targeted against vital industries. All analyzed attacks evaded firewalls by using commonly open ports, such as ports 22, 80, and 443. All analyzed attacks evaded intrusion detection (usually based on packet inspection) by obfuscating or encrypting their traffic. As a consequence, all analyzed attacks were active for several years and managed to exfiltrate high-profile data. Given what we learned so far, we derive four essential security challenges that will guide our IDS design:

Security Challenge 1: Polymorphism APT attacks vary significantly in how they are executed. Although there is a somewhat consistent story-line in every APT attack campaign as detailed in this section, the actual implementation varies in the malware used, the exploited vulnerabilities, and traffic patterns. This means that an intrusion detection system must be able to dynamically add new rules in order to look for additional patterns.

Security Challenge 2: Encrypted, obfuscated, or compressed traffic with command-and-control servers The in- and out-bound traffic used to control the malware and download data employs encryption, obfuscation, compression, or any combination of these techniques to conceal the attack. Thus, the intrusion detection system must not rely on packet inspection.

Security Challenge 3: Industry-oriented attacks APT attacks, by their very meaning, are targeted. However, as our analysis and the analyses of others [VG13; Wue14] have shown, the actual target may be as broad as a complete industry. This means that an effective defense against such threats needs to operate at the industry-level as well.

Security Challenge 4: Stealthy attacks Since APT attacks act with a specific goal in mind they remain stealthy throughout the complete attack until they have completed their agenda. Such "below the radar" attacks are extremely hard to discriminate from the background-noise which is inherent to large-scale communication.

This thesis therefore tries to envision and implement a system which is capable of monitoring an entire industry for suspicious activity.

1.3 AN ARCHITECTURE FOR A COMMUNITY-BASED IDS

This section is concerned with a system design capable of handling the aforementioned security challenges. We will discuss design decisions step-by-step, starting with the choice of input datasets and then proceeding to how these data-sources are processed. This section will be closed by discussing the contributions, presented within this thesis.

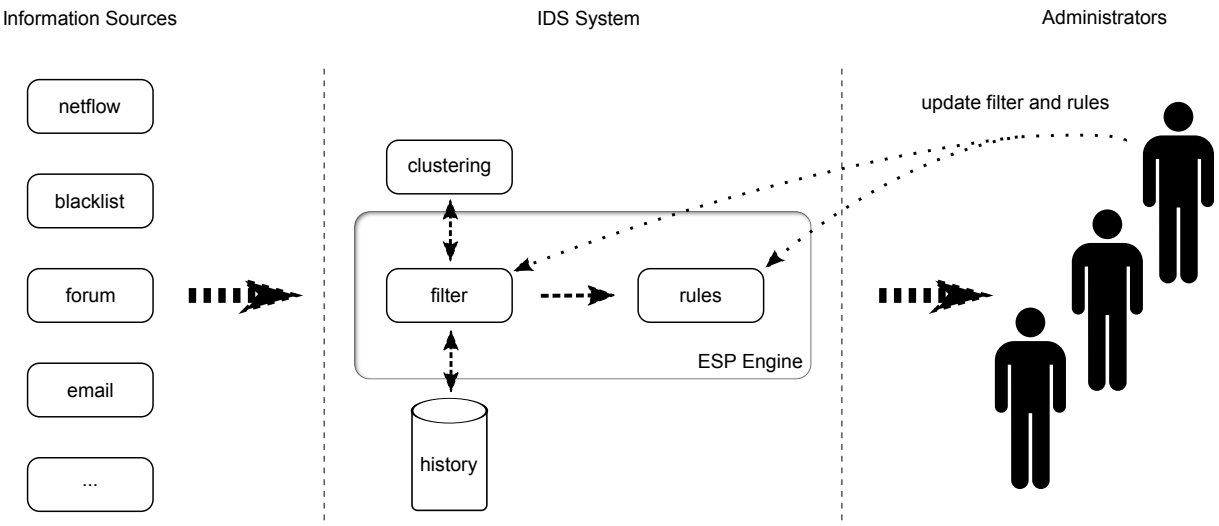


Figure 1.4: Architecture of a community-based intrusion detection system.

1.3.1 INPUT DATA

Figure 1.4 is an abstract depiction of an IDS, as envisioned in this thesis. Our primary input-data is IP flow information provided by an *Internet Service Provider (ISP)*. This choice is guided by Security Challenge 2: Encrypted, obfuscated, or compressed traffic with command-and-control servers, which makes packet inspection impossible, and Security Challenge 3: Industry-oriented attacks, which calls for a cross-domain dataset. IP flow information is usually gathered at the edge routers of ISPs in a format called *netflow*. Roughly speaking, it contains information of every route established from a source IP to a destination IP, including duration, amount of transferred bytes, used protocol, and many more properties. Another important input-dataset are online blacklists, such as *DShield*[3]. At least in some of the aforementioned attacks, previously hacked systems have either been used as command-and-control servers, for the lateral movement, or for data exfiltration. Blacklists do not only offer means to detect communication from such IP addresses early, but they have proven to be reliable predictors of future attacks [ZPU08]. Finally, they can also be used to warn possible future victims of detected threats. A similar argument can be made for scanning forums (both security and hacker forums) for activity. Especially campaigns driven by hacktivists have reportedly been deliberately initiated from forums [Den01]. Another interesting source of information could be email traffic. For example, Wuesst and Candid [Wue14] have successfully monitored targeted attacks worlwide by scanning email traffic for spear-phishing campaigns. Our conclusion from these considerations is that an effective intrusion detection system must provide the ability to monitor and cross-correlate any number of sources. This observation raises the question: How does a processing architecture that can handle such a diverse set of input look like? Or, more specifically, what are the challenges, imposed by the input data-sets?

Data Challenge 1: Variable throughput We expect substantial differences in the throughput of the different information sources. For example, individual IP flows (like netflow) are small in size, but an ISP collects millions of entries per second. In contrast, information from distributed blacklists is updated less frequently, but individual updates are larger than individual flows. Still, a large attack may result in a sudden burst of updates as reports are coming in. Finally, information from forums or mail servers are similarly subject to bursts. We also assume that updates from sources, such as mail servers or forums, are significantly larger in

[3] `https://www.dshield.org/`

size than updates from other sources, such as netflow.

Data Challenge 2: Diversity of datatypes The input data may be structured such as IP flow and blacklists, or semi-structured such as emails and forum posts. The structure of the input data has an impact on how such information is represented and processed — for example, IP flows can be naturally represented as graphs which offer several well-defined algorithms for analysis. While forum and email databases can also be represented as graphs (e.g., with emails as vertices and edges as actions, such as reply-to, forward, and so on), the actual content of the email is unstructured and needs to be processed differently, by using text-mining algorithms, for example.

Data Challenge 3: Hidden community structure As APT attacks are targeted against complete industries (such as the complete energy industry), intrusion detection should be a community concern as well. However, a prerequisite for cross-domain event correlation to work is the ability to assign data-items to their respective communities. Unfortunately, this assignment is not explicit in any of the datasets we wish to analyze. For example, we do not know all IP addresses which belong to the energy industry. Even a manual inspection of DNS entries would not yield a correct result: Infrastructure-as-a-service has eliminated the need to host every service within the company's network (e.g., web-servers are often hosted in the cloud). But even more importantly, such an industry does not have a sharp boundary. Rather, specialized law-firms or supply companies should be counted as industry members as they are often less well protected and present the perfect entrance card into an otherwise hard to intrude business community. We expect, however, that these characteristics are reflected in how the industry members interact with one another. Hence, we assume we are able to detect an industry in our input data.

1.3.2 DATA PROCESSING

The processing architecture can be split into five main components where the data-stream flows from the left to the right and may or may not generate alarms as depicted in Figure 1.4.

The *filter* component discards any information that is not of any use for the downstream components. For example, it drops any IP flows where neither source- nor destination IP address belong to a protected industry. In order to cope with Security Challenge 1: Polymorphism, the filter needs to be adaptable, such that filtering rules can be altered, added, and removed on the fly. To accommodate for Data Challenge 1: Variable throughput, the filter needs to be scalable, such that any amount of incoming data, including spikes, can be handled. Finally, the filter must support a rich filtering language that is not tied to a specific kind of input data in order to tackle Data Challenge 2: Diversity of datatypes.

The *history* component stores all incoming traffic for a given amount of time. This may be necessary to understand exactly what actions an intruder has performed and supports the administrators in deploying effective counter-measures.

The input is also forwarded to the *clustering* component, which is responsible for maintaining an up-to-date list of IP addresses belonging to the protected industry. Protecting the industry at large allows for statistical correlation between individual companies, which makes it much harder for a targeted attack to pass below the radar [ZLK10] and, therefore, tackles Security Challenge 3: Industry-oriented attacks and Security Challenge 4: Stealthy attacks. The clustering component must, hence, be able to infer community structure from that data as noted in Data Challenge 3: Hidden community structure.

The *rules* component applies user-defined rules to the data-streams. It supports rules that operate on meta-data, such as graphs representing IP or email communication. With that, this component tackles Security Challenge 2: Encrypted, obfuscated, or compressed traffic with

command-and-control servers. However, rules that operate on content are not at all obsolete. Forum posts or blacklist entries are not encrypted or obfuscated and mining such sources is of great help in understanding and predicting attacks. Therefore, this component requires a flexible (see Data Challenge 2: Diversity of datatypes) and scalable (see Data Challenge 1: Variable throughput) event stream processing architecture. The possibility for users to change, add, and remove rules accommodates Security Challenge 1: Polymorphism and is necessary to facilitate a feedback-loop, which connects the system and the administrators. With that, an administrator is able to mark alarms as false positive or true positive, helping to improve the performance of the employed algorithms.

1.4 CONTRIBUTIONS

This section presents a short overview of the major contributions of this thesis. They are organized into sub-aspects of the architecture for a community-based IDS, described above.

1.4.1 FILTERING

Chapter 4 presents a highly scalable implementation of the filter component. However, our implementation does not support dynamic addition and removal of filters during runtime and is restricted to the filtering of IP flows. Chapter 5 presents a more general implementation, based on the popular Publish/Subscribe paradigm. This implementation makes it possible to filter any kind of information. Should new insights on the maliciousness of an IP address require to re-evaluate input from the past, the *history* component can be queried for such information. Its implementation is out of the scope of this thesis, but related work on providing a history within Publish/Subscribe Systems exists [Cil+03; Li+07] and could be added to our system.

1.4.2 CLUSTERING

Chapter 2 contains a novel algorithm for finding industries in IP flows. IP flows are a natural candidate for detecting industries, since there is already a large body of related work that is concerned with clustering entities, given how they communicate. Moreover, we decided to base our industry detection on IP addresses because IP addresses are not only present in IP flow data (albeit most prominently), but also in the other datasets. Distributed blacklists refer to suspected or convicted IP addresses. Spear-phishing emails are sent to company mail servers, hence, making it possible to identify the target company. And finally, forums also often contain references to IP addresses, either those of possible targets or attackers.

1.4.3 RULES

Chapter 3 presents a generic, highly scalable graph-based rule-engine. This graph-based rule engine is the basis for a prototypical community-based IDS, presented in Chapter 4, that uses IP flow information to protect vital industries (i.e., energy, financial, and defense industries). A general processing engine, which makes it possible apply any kind of rule-engine to its inputs is presented in Chapter 5. Both engines are implemented on StreamMine, an event stream processing (ESP) framework, and can be integrated seamlessly. StreamMine itself is presented in detail in separate publications [Mar+11; Bri+11].

2 FINDING THE NEEDLE IN THE HAYSTACK: IDENTIFYING BUSINESS COMMUNITIES IN INTERNET TRAFFIC*

*The original version of this chapter appeared at WIC '14 [WHF14].

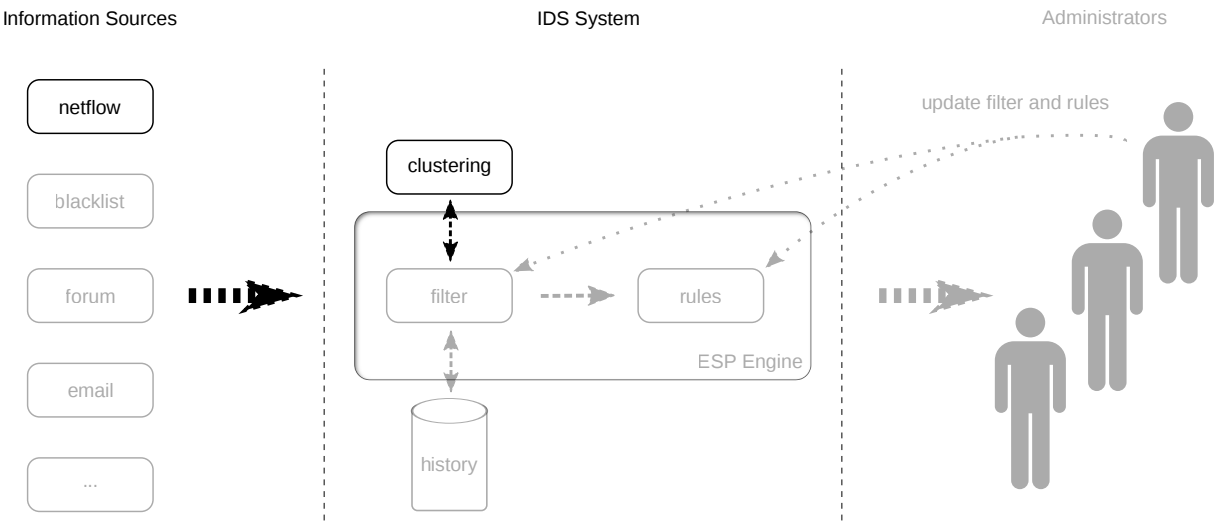


Figure 2.1: This chapter is concerned with the clustering component of a community-based IDS

This thesis advocates that intrusion detection should be a community concern, given Security Challenge 3: Industry-oriented attacks and Security Challenge 4: Stealthy attacks. However, this community structure is not directly available in our input data (see Data Challenge 3: Hidden community structure). Research in graph analysis has shown that at least for some graphs, such as firewall logs [Din+12], authorship graphs [Gup+12], link graphs from web pages [AL06; BCZ13], membership graphs [RAK07], online social networks [Leu+09; WW13], or news networks [MS09] the inherent community structure can be made explicit by applying community detection algorithms. This chapter tries to answer the question whether similar mechanisms can be used to detect business communities in IP flow data.

Identifying real-world business communities, e.g., energy, finance, defense, in Internet traffic is a challenging problem. Seed-based community detection identifies a community in a graph by iteratively adding the 'closest' vertices to an initial set of seed-vertices which are known to belong to the community. Previous research focused on unambiguous networks, where edges describe a specific intention in a fixed domain (e.g., a 'friend' in a social network) and tightly-knit communities whose members are better connected to each other ('close') than to the rest of the network. However, looking at a complete day of raw Internet traffic, we found that (1) the intend of a communication is ambiguous (e.g., ad-downloads are indistinguishable from web-page downloads) and (2) real-world industries manifest themselves as loosely-coupled communities, i.e., with more edges to non-community members than to community members. The quality of a community detection algorithm is measured in terms of *recall* and *precision*. Recall is the percentage of all known community members that the algorithm finds automatically. Precision is the percentage of actual community members which are part of the detected community. The higher both values, the better. This chapter presents a new seed-based community detection algorithm that provides higher precision and recall in our setting than the related work. This enables the detection of loosely-knit communities, such as the energy, defense, and financial industries. For instance, our solution detected 111 individual energy companies with only 6 false positives, starting from eight ISOs (Independent System Operators) and RTO (Regional Transmission Operators) in the US.

2.1 INTRODUCTION

Internet-based attacks often target a specific industry rather than random targets or a single institution. Recent examples include attacks against the energy industry [Lab14], major U.S. defense contractors [Fis12], and financial institutions [SE12]. Meanwhile, security research has shown that communities of related entities, such as companies within an industry, can be used effectively for network intrusion detection and resilience by applying anomaly detection [OKA10; McD+06], outlier detection in evolving communities [Gup+12; CJ12], classification of malicious network traffic [Din+12; WHF11a], or behavioral analysis [Ver+06]. However, the precondition for these methods is knowing the members of the community to be protected. Unfortunately, identifying the members of a community in the IP address space is difficult and labor intensive. It is, for example, not enough to perform a DNS lookup of a company's website since websites are often hosted in the cloud and the lookup will return the respective cloud provider. Furthermore, many companies own multiple subnets or host their publicly accessible resources at several different sites.

It is therefore desirable to identify the member companies of such industries in Internet traffic logs (e.g., netflow) automatically. However, Internet communication is a major challenge for community detection since the communication intend is extremely ambiguous. As a result, previous work detects only coarse-grained communities in Internet traffic logs (i.e., countries) [Eri+03]. Instead of aiming at detecting all communities, seed-based community detection [AL06; ACL06; AP09; MS09; GS12; YL15; Bal+13] detects a community around the seed vertices. However, we found two major obstacles in applying these results to our network.

First, research in this domain focused on unambiguous networks with precise definitions of what an edge between two entities represents (e.g., a 'friend' in a social network, a 'link' in a web-graph, . . .). However, an Internet traffic log contains web traffic to different destinations, peer-to-peer traffic, DNS lookups, and a range of other kinds of traffic with unknown user intentions and varying domains. For example, a bank employee may browse a social web site at work, while also using web-based tools to monitor the stock market. Both actions appear as traffic on port 80 (i.e., web traffic). Both will be represented by an edge from the bank employee's computer to the corresponding social network and stock market IP addresses. Hence, both have an equal chance of appearing as a part of the employee's community, but only one is related to the industry of interest for our analysis (financial sector). Thus, considering the social web site as part of this community would be a false positive for our purposes. It is impossible to determine a priori which Internet traffic is important for identifying a specific industry.

Second, the target communities in the related work exhibit a very low conductance (i.e., they are tightly-knit). This means that community-vertices are significantly better connected with each other than they are with non-community vertices. However, we found that members of industries within the Internet are better connected to the rest of the network than to each other, that is, these communities have a large conductance. Hence, the traditional methods will tend to select non-community members for inclusion. We argue that this is by no means special to our communities in particular but a general property of our network as it is simply a consequence of the ambiguity in Internet communication.

The question is whether such communities can be detected with the available information in the network. Or more specifically, can we grow a relatively small set of loosely connected industry members (such as the eight energy organizations in the US) into a larger community that is still dominated by the given industry? Our contributions are as follows:

- We analyze the structural properties of three representative industry-communities in an ISP's Internet traffic log. Our results show that, contrary to the findings in [Les+08], the network-community profile (NCP) suggests the existence of large low-conductance

cuts in Internet traffic. However, the conductance of the three industry-communities is several orders of magnitude larger than the best cuts of similar size.

- We show that personalized page-rank (PPR) and conductance optimization algorithms perform poorly, given our seed-sets. We attribute this to the ambiguity of Internet communication that makes it significantly harder to select the right members for inclusion.
- We find that using binomial probabilities [MS09] yield better results than conductance or PPR. However, this algorithm is unnecessarily complex and impractical to implement.
- We propose a new algorithm: normalized inverse conductance. We show that it performs equally to binomial probabilities in terms of recall and precision without inheriting its practical limitations.

The remainder of this chapter is structured as follows. We analyze the structural properties of our graph and its communities in Section 2.2. Equipped with that knowledge we present our algorithm in Section 2.3. We evaluate our own algorithm and several other algorithms from the related work in Section 2.4. Section 2.5 presents additional related work before Section 2.6 concludes this chapter.

2.2 PROBLEM CHARACTERISTICS

In this section, we take a closer look at our dataset and the communities we thrive to find. This information will guide the design of our algorithm. It will also be useful for the interpretation of the results obtained by applying the different algorithms.

2.2.1 DATA

Netflow is a standardized format used by network routers to log each connection, including source and destination IP address, protocol, ports, intermittent routers, etc. Our dataset is a heavily sampled netflow log from a large Internet service provider (ISP), recorded over a period of a complete day. Most of the fields present in netflow can be discarded since they are of no use to solve our problem. However, two fields, namely the used port and the transferred bytes, deserve a closer evaluation. While it would seem prudent to consider the ports, this information is actually less useful than one might think. For example, even if we assert for argument's sake that port 80 is exclusively used for http connections, we are still unable to distinguish between private browsing (e.g., social networks) and community-related browsing (e.g., stock-market monitoring for the finance community). Furthermore, our experiments with the amount of transferred bytes showed that this information does not turn out to be of significance to our problem either. Therefore, we only consider source and destination IP addresses in our analysis. Since netflow contains one flow record for each individual connection between two IP addresses, we aggregated all flows between two IP addresses into one flow, resulting in 227, 292, 837 unique edges. We remove any flows where one of the endpoints is located within a reserved range, e.g., private use or cable-tv[1].

[1] `http://www.rfc-editor.org/rfc/rfc3330.txt`

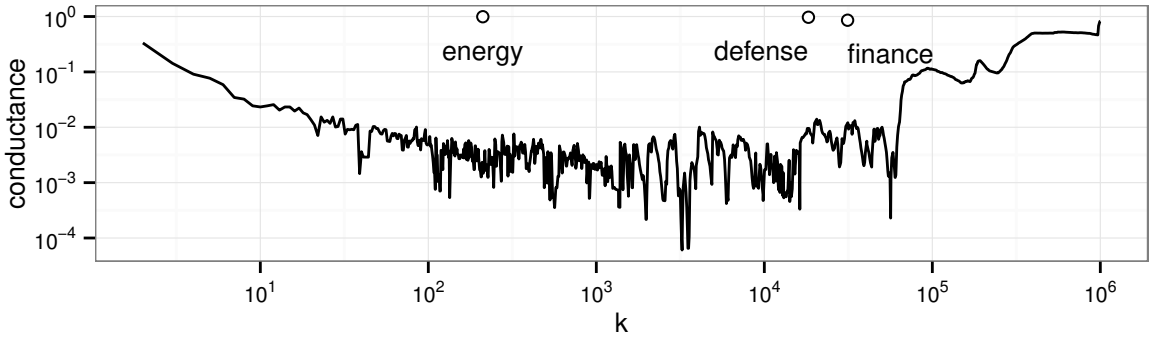


Figure 2.2: Network community profile (NCP) plot for one day of netflow

2.2.2 STRUCTURAL PROPERTIES

In the following, we evaluate several structural properties of the network and the communities.

Conductance A small conductance has consistently proven to be one of the most robust structural properties of real communities in social networks [YL15; GS12; Alv+13]. The conductance of a community S in a graph is defined as the fraction of edges between the community and the rest of the graph (cut) against all edges that have at least one endpoint in the community (vol):

$$\varphi(S) = \frac{cut(S)}{vol(S)}$$

The network community profile [Les+08] allows to analyze at which size-scale low conductance cuts can be found in a given graph. It selects for each size within a given range, a community of that size with minimal conductance from the given network. If we look at the ncp plot of the Internet traffic log (Figure 2.2), we can see that our graph supports low conductance cuts of significant size. Note that while [Les+08] observed minimal conductance only for very small clusters (e.g., 10^1 to 10^2 vertices) in social networks our network contains small conductance clusters with significantly more vertices (e.g., 10^2 to 10^5).

Unfortunately, these low conductance communities are not the communities we hope to detect. We plotted the conductance of our seed sets alongside the ncp plot (denoted as points in Figure 2.2). Obviously, all three communities exhibit large conductance values—several orders of magnitude above the optimal conductance for equally sized communities. After all, it also seems counter intuitive to expect low conductance for industries in the Internet: Why would energy utilities or banks be communicating with the rest of the Internet less frequently than they do with industry peers? Even if we assume for the moment that social websites would be inaccessible from work, search tools, ad-servers, news sites, or teleconferencing services (just to name some examples) still make up for a large part of the traffic. All we might expect is that there is a reasonable amount of communication between them but not that this communication surpasses the one with the remaining Internet. This is also supported by the conductance of the verified communities, listed in Table 2.1. The verified communities are all true positive IP addresses that were identified by our algorithm. While the conductance between the seed set and the verified community decreases (albeit remains at the same order of magnitude) for the energy and the defense communities, the conductance of the finance community increases slightly.

Table 2.1: Structural properties of seed sets, the verified communities and the complete graph

		Vertices	Cut	In-community Edges	Conductance	Edge Density	Vertex degree	
							mn	sd
Energy	Seeds	211	7000	56	0.99	1.26×10^{-3}	35.6	103.5
	Verified	39000	249000	80000	0.76	5.11×10^{-5}	8.3	109.9
Defense	Seeds	18000	260000	8000	0.97	2.36×10^{-5}	14.6	174.9
	Verified	134000	698000	476000	0.60	2.65×10^{-5}	8.8	139.6
Finance	Seeds	31000	447000	74000	0.86	7.57×10^{-5}	16.6	348.9
	Verified	78000	1600000	218000	0.88	3.61×10^{-5}	23.4	356.8
Network	–	39000000	–	217000000	–	1.40×10^{-7}	–	–

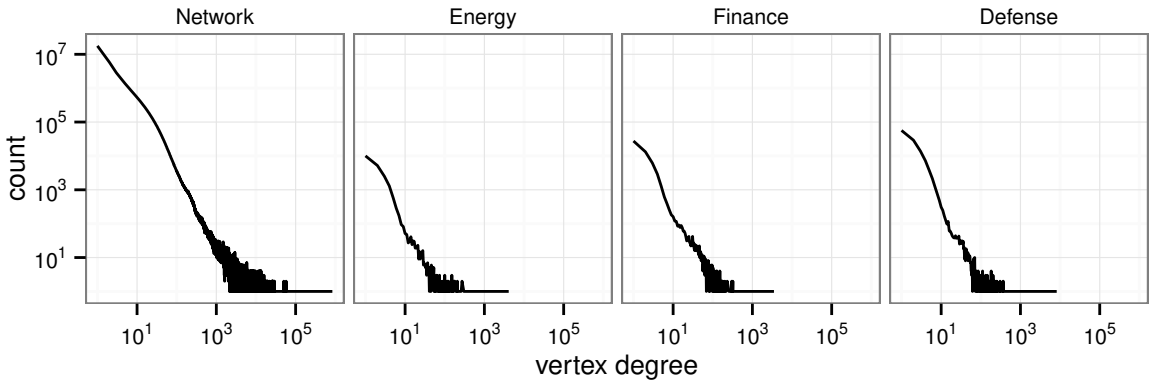


Figure 2.3: Frequency plot of vertex degrees in netflow

Vertex Degrees Figure 2.3 shows the frequency of vertex-degrees per verified community and for the complete network. As expected, most vertices in the network are low-degree vertices, while some vertices reach degrees of almost a million. This is consistent with other social networks [Mis+07b]. The degree-distribution of our target communities exhibits a similar shape as the one for the whole network. This comes as no surprise, since the communities are social networks themselves. It is noteworthy, however, that the degree distributions of the communities do not reach as high vertex degrees as the degree distribution of the complete network. We attribute this to the absence of widely used services in our communities, such as the main DNS servers, ad-servers, social web-sites, etc. Although most vertices have a degree of 1, Table 2.1 shows that the mean degree of each community is considerably higher. Together with the high conductance of the seed set, this means that a breadth first search, contrary to the assumption in [AL06], can indeed find enough candidates in the seed set's neighborhood to grow the community.

Edge Density In [MS09], a natural community was defined as a component of the graph that exhibits significantly higher edge density than the network as a whole. The edge density of a directed graph $G = (V, E)$ is defined as the relation between the actual edges and the number of possible edges for the given vertices:

$$D = \frac{|E|}{|V| * (|V| - 1)}$$

The density of a community inside a graph is calculated simply by replacing $|E|$ and $|V|$ with the numbers of the edges and vertices inside the community. As shown in Table 2.1, our verified communities exhibit an up to two orders of magnitude higher edge density than the complete graph.

2.2.3 DISCUSSION

A lesson we learned here is that Internet traffic, although a social network by definition, is substantially different from what has been studied so far. While it does contain large low-conductance cuts, the conductance of the business sector communities we aim to detect is orders of magnitude higher. We argued that this is not an artifact of our communities in particular but a consequence of the diverse usage of the Internet. At the same time, the edge density proves that these communities do exist as a structure, substantially different from the rest of the network. Hence, given the seed set one should be able to distinguish the remaining community from the rest of the network. The degree-distribution of the vertices is not unexpected for a social graph but also shows an interesting aspect that we will use to

construct our solution: While low-degree vertices dominate the degree-distribution, the verified communities exhibit a significant amount of high-degree vertices which is also reflected in the mean degrees of the communities and their seed-sets. The significance of a structural property, defined over the edges of a vertex, is directly related to the degree of the vertex. A high-degree vertex with 70% of its edges connected to the community is more likely a member of that community than a low-degree vertex with 100% of its edges connected to the community. Hence, we will explore the possibility of using the mean degree of the seed-set and the degree of the candidate as an additional factor to drive our search.

2.3 NORMALIZED INVERSE CONDUCTANCE

Given the structural properties of our network and the communities we hope to find, we will now construct an algorithm which can grow our seed sets into larger communities that are still dominated by the given industry. We followed [YL15] and divided our algorithm into two phases: candidate selection and candidate ranking. It takes the graph G, the set of seeds S, and a size for the target community K as input, and returns a community of size K as output.

2.3.1 CANDIDATE SELECTION

The candidate selection phase in our algorithm is implemented as a breadth first search. We excluded the option of selecting all vertices of the network as candidates [MS09] since our network is too large. We also decided against random walks ([AL06; YL15; GS12]) for two reasons. First, the cut of our seed sets is high enough such that we will find a sufficient number of candidates in the direct neighborhood of the seeds. Second, random walks might also choose vertices that have no direct connection to the seed set but our ranking function only considers direct connections. Note that we can still find members that are not directly connected to the seed set by running the algorithm for several iterations where in each iteration, we use the obtained community of the previous iteration as the seed set.

2.3.2 CANDIDATE RANKING

Conductance is one of the most robust structural properties of natural communities. However, given such high-conductance communities as ours, simply aiming at low conductance might select the wrong members. For example, any vertex with a single edge (a large fraction of vertices have a degree of 1) into the community, has a conductance of 0, that is, the optimum. On the other hand, it is unlikely that vertices with larger degrees exclusively communicate with their community. Thus, these vertices are less likely to be chosen. The fundamental issue with conductance is that it does not provide any measure of significance: A low-degree vertex with minimal conductance might just be a coincidence (our data is sampled, provider might switch IP address of customer, short connection, ...) but a large degree vertex is probably not. To reduce the impact of low-degree vertices, we use a dampening factor based on the mean degree of the seed set (M) and the vertex degree (n). We apply this dampening factor to the inverse conductance (k/n) which is the fraction of edges that connect to known community members (k).

$$nic = \frac{n}{M+n} * \frac{k}{n}$$

The first term of the formula is a dampening factor with values within the range of $(0, 1)$. It's influence is highest for low-degree vertices, cutting the inverse conductance by more than a half for below-the-mean vertices. For high-degree vertices, its influence vanishes. This provides us with a measure to evaluate the significance of a vertex: A low-degree vertex with

a high inverse conductance will rank lower than a high-degree vertex with a (possibly) smaller inverse conductance.

2.4 EVALUATION

In this section, we will evaluate our algorithm using three sample communities, namely energy, finance, and defense companies. For each industry, we used publicly available knowledge (Wikipedia) to identify the major members and selected those as seeds for which we could identify one subnet - e.g. by performing a DNS lookup of their websites. For the energy industry, we selected eight ISOs (Independent System Operators) and RTO (Regional Transmission Operators) in the US[2], resulting in 211 seed-vertices, as shown in Table 2.1. For the defense industry, we selected eight defense contractors[3], resulting in 18K seed-vertices. Finally, for the finance industry, we selected 10 banks[4]/life insurers[5], resulting in 31K seed-vertices. For comparison, we evaluate PPR [AL06], minimal conductance [YL15; GS12], and binomial probabilities [MS09] using the energy industry. Note that none of the mentioned algorithms from related work was tested with or developed for high conductance communities in ambiguous networks. Thus, it comes as no surprise that PPR and minimal conductance fail to identify the members of the energy industry. However, we were able to achieve good results using binomial probabilities. Unfortunately, it has severe practical limitations, which we will detail below. First, we will discuss how we evaluate the output of the community detection algorithms. All algorithms are implemented, using the *SNAP*[6] library, version 2.

2.4.1 OUTPUT

All evaluated algorithms return a vector of graph vertices that are supposed to be new community members. In our case, each vertex in the graph corresponds to one IP address. We can assign the members in the resulting community into four classes.

- *True positives (TP)* are the IP addresses of companies belonging to the industry in question.
- *Weak members (WM)* are the IP addresses of companies closely related to the industry. This could be, for example, enterprise security software companies, law firms, or teleconferencing providers.
- *Unknowns (NA)* are the IP addresses that do not belong to any company. Some of them do not even answer a ping (may be disconnected), some are used by ISPs for dynamic IP allocation to private customers, and some are simply not running any services or provide any useful hostname information.
- *False positives (FP)* are the servers of content distribution networks (CDNs), radio streaming services, social websites, or other non industry-related services.

After the communities have been identified, we evaluate the community by counting the true and false positives, weak members, and unknowns. Since each vertex corresponds to an IP address, we can use reverse DNS lookup (i.e., *whois*) to determine to which subnet the vertex belongs. The DNS lookup may directly give the name of the company to which the IP address belongs. However, in many cases the IP address resolves to some ISP or cloud

[2] http://en.wikipedia.org/wiki/Regional_transmission_organization
[3] http://en.wikipedia.org/wiki/Defense_contractor
[4] http://en.wikipedia.org/wiki/Big_Four_(banking)
[5] http://www.ffiec.gov/nicpubweb/nicweb/Top50Form.aspx
[6] https://snap.stanford.edu/snap/index.html

provider. In this case, it is necessary to check further (e.g., by using *nmap*) to determine the owner of the IP address. In cases where the company cannot be determined, the IP address is classified as unknown.

Since the reverse DNS lookup provides information to which subnet and company an IP address belongs, we can enhance the interpretation of the counts per class. In the default case, called "IP", the community consists exclusively of the detected IP addresses. But instead of counting each IP individually, we can also count the distinct subnets they belong to (the unknown IPs are still counted individually since it would be wrong to mark the entire subnet of a cloud provider as unknown). We call this count type "subnet". It is a useful measure since in an extreme case, all discovered IP addresses might be contained in a single subnet. However, for the true positives, we would rather discover IP addresses from different subnets since some of the community members own several subnets which are difficult to identify otherwise. Conversely, for the false positives and weak members we would rather detect few distinct subnets since this greatly simplifies the task of excluding them from the community. Finally, we count distinct companies (we use the name of the ISP or hosting service as the company name for the unknown class). We call this count type "company". This measure is useful for a similar reason: In an extreme case all discovered subnets might belong to a single company. However, a perfect algorithm would be good at both count types: detecting many subnets from many distinct companies.

To determine the quality of the output of an algorithm, we use precision, recall, and f-measure. Informally, the precision is the fraction of community members (tp) to all selected members (fp + unknown + wm), the recall is the fraction of selected community members (tp) to all known community members (i.e., the verified community), and the f-measure is the geometric mean between recall and precision. Since it is not realistic to expect all the IP addresses of a company to appear in the netflow, it is also not realistic to expect a community detection algorithm to find all IP addresses. To this end, we only compute recall, precision, and f-measure using the company counts. Note that our definitions of precision and recall are very pessimistic: One could also argue that the weak members are to be counted as true positives, because they may open less protected back-doors into an industry and, therefore, must be part of the community. An upper bound to the precision, recall, and f-measure would be to count all classes except the false positives as true positives. We believe, however, that the lower bound gives a better measure of quality than the upper bound would.

Table 2.2: Evaluation results — counts do not include the seeds

		Energy				Defense				Finance			
		FP	TP	NA	WM	FP	TP	NA	WM	FP	TP	NA	WM
Normalized Inverse Conductance	IP	7	55	31	7	10	42	10	38	4	91	1	4
	Subnet	7	46	31	6	3	22	10	9	2	11	1	1
	Company	5	42	18	6	2	13	9	8	2	6	1	1
Normalized Inverse Conductance (after four iterations)	IP	22	191	141	46	54	195	61	90	28	348	10	14
	Subnet	17	132	141	25	13	69	61	25	10	37	10	10
	Company	6	111	49	21	5	38	38	22	9	21	5	10
Minimizing Conductance	IP	40	264	507	71	-	-	-	-	-	-	-	-
	Subnet	31	108	501	61	-	-	-	-	-	-	-	-
	Company	23	98	152	61	-	-	-	-	-	-	-	-
Personalized PageRank (α = 0.9)	IP	7	33	42	12	-	-	-	-	-	-	-	-
	Subnet	7	27	42	12	-	-	-	-	-	-	-	-
	Company	5	25	25	12	-	-	-	-	-	-	-	-
Minimizing Binomial probabilities	IP	5	56	34	5	18	41	7	34	2	92	2	4
	Subnet	5	43	34	5	5	26	7	9	2	16	2	2
	Company	3	39	17	5	3	17	6	8	2	7	2	2

2.4.2 COMPETING APPROACHES

In the following we will evaluate several approaches which have been applied with high success in related work.

MINIMIZING CONDUCTANCE

While [YL15] used random walks in order to find vertices around the seed set that minimize conductance, [GS12] found that low conductance cuts can often be found in the direct neighborhood of vertices. Hence, in order to evaluate conductance, we selected all direct neighbors of the seed set with a minimal conductance (i.e., $\varphi = 0$). This set contains more than 800 vertices. After classifying all vertices in this set, we obtained the numbers shown in Table 2.2. The dominant class is without a doubt the unknown class. While the total number of distinct IP addresses in the true positive class is quite high, it only transforms to about 100 energy companies. We have already argued that conductance will select those vertices with a zero cut, as long as they exist. In the case of the energy community, 68% of the vertices with $\varphi = 0$ have a degree of only 1 (and 20% a degree of 2). Of all one-degree vertices, 73% belong to the unknown class while only 28% belong to the true positives.

PERSONALIZED PAGERANK

Intuitively, the PPR algorithm works as follows [AL06; ACL06]: First, it calculates for each vertex in the neighborhood of the seed vertex the probability that a random walker visits the given vertex. This step returns a list of vertices V, sorted (in descending order) by how likely they are to be visited by a random walker. Second, the algorithm performs a sweep-cut: It selects a $k : 1 \leq k \leq |V|$ which minimizes the conductance of the vertices in $top_k(V)$. These steps are repeated for all seed vertices. Finally, the union of all sweep cuts for all seed vertices is the community of the seed set.

The results for personalized pagerank for the energy-community are similar to those of minimizing the conductance in that most detected IP addresses belong to the unknown class. Given that the sweep cut also minimizes conductance, this is no surprise. Moreover, [GS12] have found that a minimal conductance cut of the direct neighborhood of a seed vertex is sometimes as good as a random walk. A crucial parameter of PPR is the teleport probability (α): Intuitively, it governs how far away the random walker can travel from the seed vertex. A high probability means that it will be "teleported" back to the seed vertex more frequently. We evaluated the results for varying α: 0.1, 0.5 and 0.9. As expected, for lower α the algorithm selects also vertices that are not directly connected to the seed set. Although, the recall of the resulting community is larger in that case, it also exhibits an even lower precision. Conversely, if we set α as high as 0.9, the returned community has a higher precision but the recall is very small. This remains true, if we increase the size of the target community.

MINIMIZING BINOMIAL PROBABILITY

Using binomial probabilities for community detection was proposed by [MS09]. The algorithm works as follows: Suppose, each vertex has n edges in total, k of which connect to already known community members. The probability p of a vertex being a community member is the fraction of seed vertices divided by the number of all vertices in the graph. Then we can compute the binomial probability that a given vertex has at least k community edges as

$$Pr[n, k] = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}$$

Intuitively, the candidates are ranked by how probable the combination of k and n is, given the graph and the community to be found: The smaller this probability for a given candidate, the more likely it is an actual member. This probability is computed for each vertex in the graph and the vertex with the smallest probability is selected to become a new member. In the next round, the probabilities of vertices in the neighborhood of the new member are updated and the algorithm selects again the vertex with smallest probability for inclusion. Since computing the probabilities for all vertices in a graph as big as ours is infeasible, we only compute the probabilities for the direct neighborhood of the seed set. This is an equivalent solution, since $k = 0$ for all other vertices. For our experiments we ran the algorithm until the seed set was grown by 100 new vertices.

Results Looking at the results, we can see that the algorithm consistently returns more IPs from the true positive class than from any other class. The IP - count of the finance community scores best, with a true positive count one order of magnitude above the other classes. This relation remains constant, even when counting how many different subnets and companies are identified by these IP addresses. We note, however, that most IP addresses belong to the same company: A large bank with many distinct subnets. Finding all these subnets manually would be very cumbersome. While the total IP count for the energy community is lower, these IP addresses belong to a larger set of distinct companies. Although the difference between the true positives and the other classes is not as large as for the finance community, the true positives are still dominating the selected candidates. Finally, the algorithm selects a notably higher amount of weak members and false positives for the defense community as compared to the other communities. Still, the number of false positives is less than half of the true positive count. We attribute the good results of this algorithm to the fact that it incorporates the degree of vertices into the ranking.

In order to get a better understanding of how well the binomial probability separates the nearest neighbors of the seed set into the four different classes, we computed the binomial probability on all direct neighbors of the seed set of each community. Once classified, we can plot the observation count (the width of the violins) of the classes as a function of the property's value: The wider the violin for a given class at a given point on the y-axis, the more candidates have been observed with the corresponding binomial probability in the corresponding class. Figure 2.4 depicts the classification performance of the binomial probability.

Mehler [MS09] selected those candidates which had the lowest binomial probability. Thus, when selecting candidates, the algorithm starts from the lowest end of the violins. In order to magnify this end of the scale, we only plot the candidates with probabilities below the 10th percentile. If we plot all candidates, the violin plots are dominated by the huge count of observations at the opposite side of the scale, making it impossible to reason about the separation performance. We can observe that there are vertices from all classes with a very small binomial probability: Except for the energy community all classes have binomial probabilities in the same order of magnitude. It is, however, the observation count which is highest for the true positives and low binomial probabilities. As expected this is most visible for the finance community: It was also the community with the highest count of true positive candidates. It is also apparent why the defense community had such a high count of weak members: Many weak members are observed at a low binomial probability. Similarly, in the energy community, unknown IP addresses are observed quite frequently at a low binomial probability, while false positives and weak members have higher probabilities.

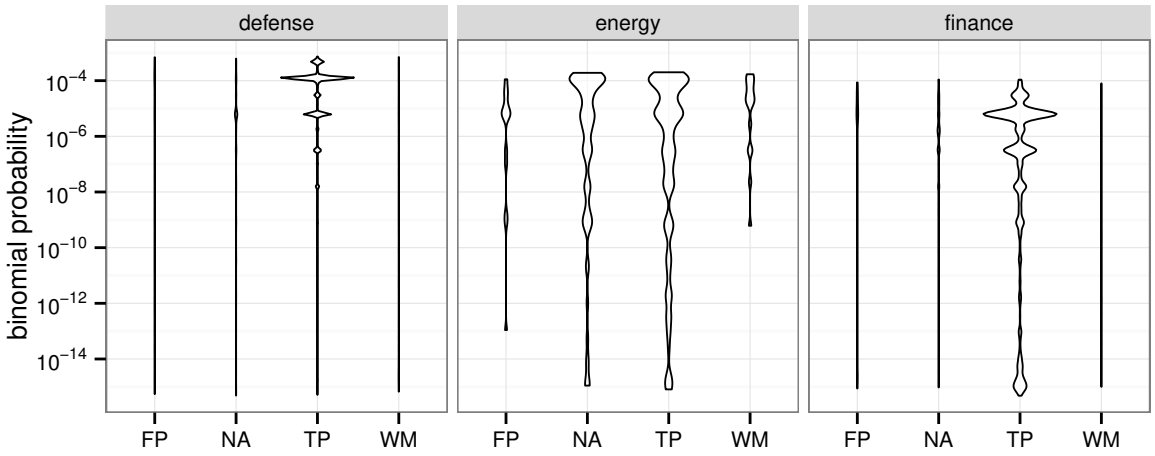


Figure 2.4: Class separation performance of binomial probabilities

Practicability Computing the binomial probability that a vertex has at least k neighbors that are part of the community has a linear runtime complexity of $O(n-k)$. Since our communities have a large conductance, $n >> k$ is true for most vertices. Therefore, we can alleviate this issue to some extend by computing the binomial probability that a vertex has at most k edges (complexity of $O(k)$):

$$Pr[n, k] = 1 - Pr[0, k-1] = 1 - \sum_{i=0}^{i<k} \binom{n}{i} p^i (1-p)^{n-i}$$

Moreover, the binomial coefficient has to be computed using its multiplicative representation since, otherwise, the factorials quickly overflow the available data-types.

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n-(k-i)}{i}$$

Unfortunately, once $k \approx n/2$, this alternative provides no benefit anymore.

The linear runtime complexity is, however, not our only concern. In the original work, p is computed as the fraction of the number of community vertices and the total number of vertices in the graph. We had to fix p to 0.05 since otherwise either p^i or $(1-p)^{n-i}$ resulted in an overflow[7] for vertices with a degree larger than 500 which led to incorrect results and, therefore, a wrong ranking of the candidates. Note, that our communities contain vertices with degrees of more than 10^3 vertices. If the neighborhood of the seed-set contains such vertices, fixing p at 0.05 may not be an appropriate solution either. We believe these issues put a limit to the practicality of the binomial probability for large social graphs.

2.4.3 NORMALIZED INVERSE CONDUCTANCE

We evaluated the normalized inverse conductance (nic) similar to the binomial probability: Among all candidates, we selected the 100 vertices with the highest rating. First, the distribution of classes is similar to the results we achieved using the binomial probability. However, when looking closer some points are worth mentioning. For the defense community, the unknown class is slightly larger, while there are fewer false positives. Also, the selected vertices belong to fewer distinct companies and subnets. For the finance community, we can observe

[7] On our test-machine (64 Bit) and for a value of $p \approx 5e^{-5}$ (energy community), an exponent larger than 255 resulted in an overflow of the double data-type.

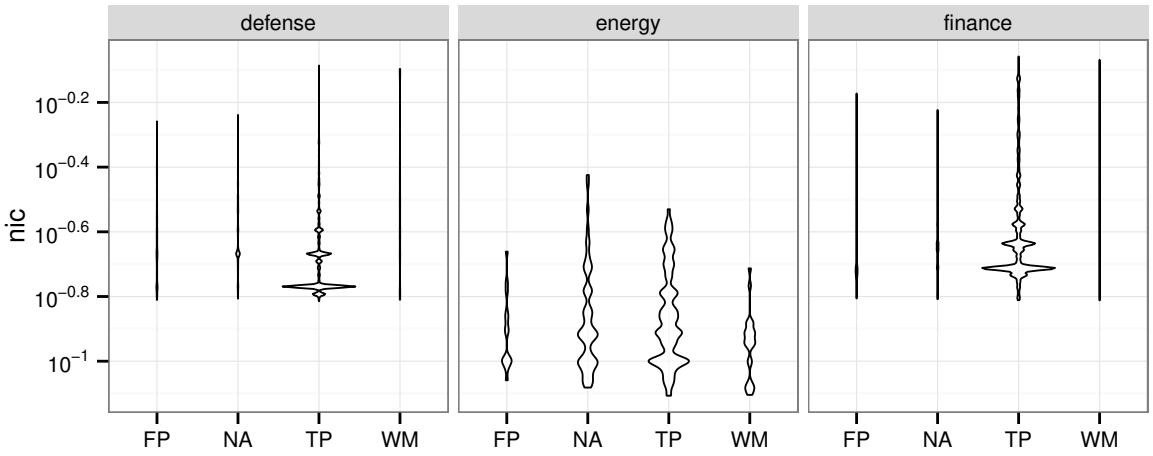


Figure 2.5: Class separation performance of normalized inverse conductance (nic)

that the binomial probability selected candidates from more distinct subnets than nic did. Finally, with nic, we detected more distinct companies from the energy community than we did with the binomial probability while the counts for the other classes are almost the same. As reflected in the obtained counts, the classification performance of nic is comparable to the performance of the binomial probability.

In order to get a better understanding of how well nic separates the nearest neighbors of the seed set into the four different classes, we computed it on all direct neighbors of the seed set of each community. The normalized inverse conductance is defined in such a way that most true positives are located when it reaches its maximum for a given community: Vertices with a high degree (n) and a high count of in-community edges (k), also have a high nic value. In order to magnify this end of the scale, we only plot the candidates with nic-values above the 90th percentile in Figure 2.5. First, the violins do not start at the same order of magnitude for each class and community. This is because the seed set for the energy community is several orders of magnitude smaller as compared to the other communities (see Table 2.1). Therefore, k (and nic with it) is expected to be much smaller than for the other communities. For the finance and defense communities, the candidates with the highest nic are also true positives. Moreover, the violins of the true positive class are significantly wider for both communities which means that high nic values are dominated by the true positive class. Finally, although the candidates with the maximal nic within the energy community belong to the unknown class, most observations for high values of nic still belong to the true positives.

2.4.4 GROWING FURTHER

To evaluate whether we can improve the our results further, we grew the communities throughout several iterations, using our algorithm. We let it run for four iterations, growing the seed set by selecting the best 100 candidates each time. If we first look at the energy community in Table 2.2, we see that the amount of unknown IP addresses grew faster than the amount of true positive IP addresses albeit the true positive IP addresses still dominate the results. The subnet count is dominated by the unknown IP addresses since each selected candidate from this class also counts as one subnet as explained above. The company count gives an impressive result of as many as 111 detected distinct energy companies with only 6 false positive companies. This is 2.64 as many distinct energy companies as for the first iteration. The IP count of the finance community follows a similar shape as the IP count after the first iteration. Thus, while keeping the precision high, we were able to increase the amount of detected distinct finance companies by a factor of 3.5. Finally, the defense community also

Table 2.3: Community quality after first and fourth iteration

	Iteration	Precision	Recall	f-measure
Defense	1	0.41	0.11	0.17
	4	0.37	0.32	0.35
Finance	1	0.60	0.02	0.03
	4	0.47	0.06	0.10
Energy	1	0.60	0.15	0.24
	4	0.59	0.41	0.48

shows a similar shape as for the first iteration. The amount of detected distinct companies was increased by a factor of 2.92. Table 2.3 shows precision, recall, and f-measure for our algorithm after the first and the fourth iteration. As expected from the counts, the recall was more than doubled for all communities. Moreover, the precision remained almost constant for all communities, except for the finance community where the precision decreased slightly. Still, we were able to increase the f-measure for all communities.

2.4.5 DISCUSSION

If we come back to the initial motivation for detecting these communities, including IP addresses from other classes is not as bad as it seems at first: The fact that these IP addresses rank high for any of the ranking functions proves that there is indeed communication between these entities. An attacker might very well use such less protected back doors into an industry. In order to detect such attacks, it is useful to include these IP addresses as well. However, we must be careful not to end up with a community where true positives are a minority. In that case, determining whether an attack against a large number of community members is in fact an attack against the industry or just an attack against one of the three other classes becomes impossible. Figure 2.6 shows f-measure, recall, and precision for all algorithms and communities, computed on the company count. If we first consider a single run of the algorithms only, conductance optimization has the highest f-measure: This is because it's recall is highest (almost 100 energy companies), however it's precision is worst since it also detected a high number of unknowns, weak members, and false positives. We discard conductance as unsuitable since it does not return a community that is dominated by true positives. The same is true for personalized page rank, although its precision is higher. Binomial probabilities and our algorithm achieve the highest precision while maintaining a high recall as well. Both returned communities are dominated by true positives.

This opens the possibility to re-apply our algorithm on the community, obtained in the previous iteration. Note, that this approach is only possible if the precision is not too low as it is the case with the conductance optimization and personalized page rank. In that case, re-running these algorithms on the obtained output community would quickly diverge to an unrelated community. Running our algorithm four times, each time taking the output community of the previous run as the input community of the next run, we are able to improve the f-measure of our algorithm significantly and beat all other algorithms.

2.5 RELATED WORK

Minimal conductance [GS12; YL15] has proven to be the most robust structural property of communities in several social networks. In many cases, the conductance of the seed set is so small that random walks are necessary to expand the seed-set [AL06; ACL06; YL15]

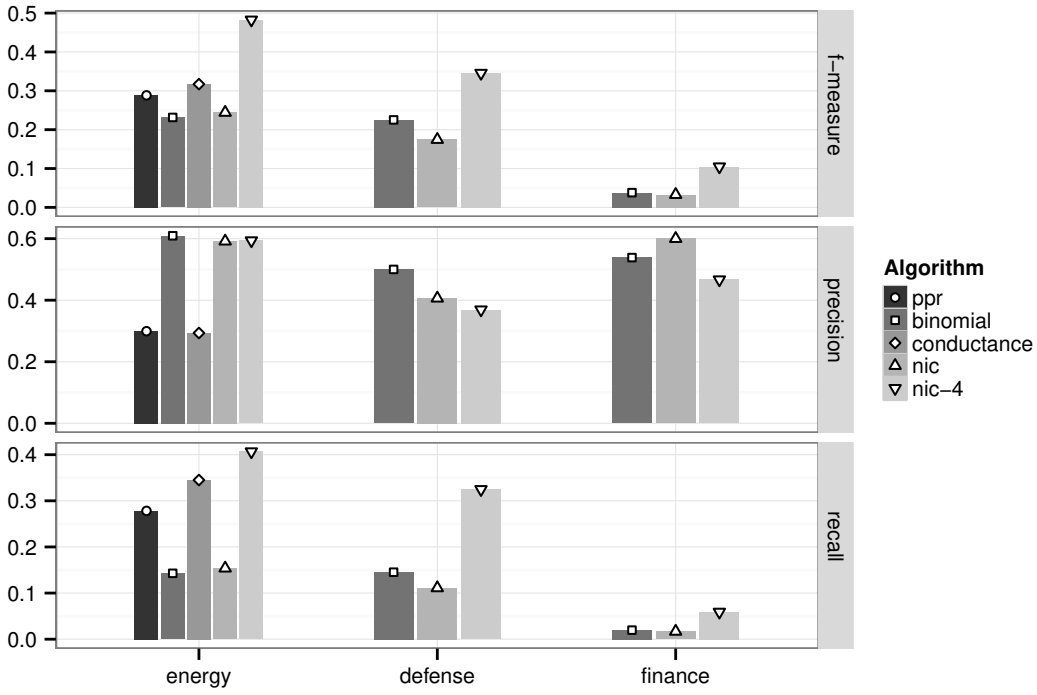


Figure 2.6: Comparison of all three ranking functions

because expansion is impossible otherwise. However, random walks are closely related to conductance: If the conductance of the target community is high, a random walker is less likely to remain inside this community. Finally, [Tsi+13] uses conductance to estimate congestion of ISP network topologies. Its ultimate goal is to find low-conductance communities: The links into those communities are most prone to congestion. Another approach that is closely related to conductance is called (α, β) clustering [Mis+07a]: The goal is to find clusters where each vertex inside the cluster is adjacent to at least a β-fraction of the cluster and any vertex outside the cluster is adjacent to at most an α-fraction of the cluster with $0 \leq \alpha \leq \beta \leq 1$. Hence, the conductance of communities needs to be smaller than 0.5 (but not minimal as opposed to conductance optimization or random walks). An extension of (α, β) clustering is proposed in [Bal+13], where overlapping communities are detected using the relative affinities among the vertices of a graph. However, they also assume that communities need to be internally dense and externally sparse i.e., the conductance of communities needs to be smaller than 0.5. In [PPD13], attribute homogeneity of vertices is used to detect communities in authorship and software-package content graphs. Since defining homogeneity among IP addresses is difficult (our communities include IP ranges of any size), we did not investigate such algorithms. Finally, [MS09] proposed to use binomial probabilities for community detection. This was motivated by the fact that although their goal was to detect low-conductance communities (e.g., baseball players in the US) in news-site networks, the ncp plot showed that large low-conductance cuts did not exist in their graph. Hence, using methods like personalized page rank or random walks was out of question.

2.6 SUMMARY

Given an Internet traffic log, our goal was to grow a relatively small set of loosely connected industry members into a larger community that is still dominated by the given industry. We have shown that such industries manifest themselves as dense, high conductance clusters in Internet traffic in contrast to the communities that have been studied so far in the field of seed based community detection. Therefore, two of the most prominent algorithms from related work (that is, PPR and conductance optimization) did perform poorly on our dataset and our communities. While we were able to detect our communities using binomial probabilities, they exhibit severe limitations in practice, such as numerical overflows and high computational complexity. We showed that our algorithm achieves comparable results without inheriting the practical limitations of binomial probabilities. Moreover, we were able to grow the seed set beyond its immediate neighbors by recursively applying our algorithm on the expanded seed set.

3 MINING LARGE DISTRIBUTED LOG DATA IN NEAR REAL TIME*

*The original version of this chapter appeared at SLAML '11 (co-located with SOSP) [WHF11b].

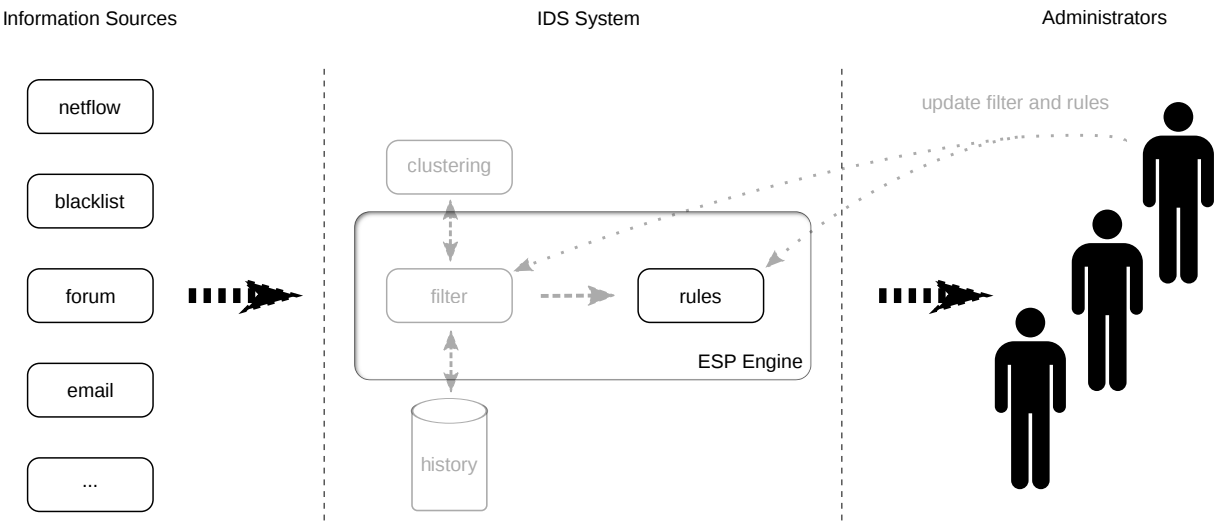


Figure 3.1: This chapter is concerned with deriving an efficient graph database as part of the rule component for a community-based IDS

Now that we have an understanding of how to detect a community in communication logs (such as IP flow), we need to gdevelop a graph processing and mining engine that is able to monitor all data concerning such a community in real time.

All input datasets that are depicted in Figure 3.1 can be represented as a graph as they exhibit explicit inter-dependencies. For example, emails can be represented as edges, which connect sender and receivers. Likewise, we can connect members of discussions in a forum. Both are just examples of how such representations could be made. A graph-based representation for blacklists has already been successfully applied to intrusion prediction in [ZPU08]. Finally, netflow data is naturally represented in graphs [ATK14]. For example, the work by Ding et al. [Din+12] creates a projection from IP flows where two vertices are connected by an edge if they communicated with the same target. With that, they show that intrusion can be defined as "anti-social behavior" in that an intruder tends to select communication peers more randomly than a normal user does. Or McDaniel et al. [McD+06] propose to capture common communication peers of computers in an enterprise network with communities of interest (COI) [CPV01]. They show that infection-rates of worms can be slowed down effectively, or even contained in many cases.

While this list is by no means complete, it shows that graph-based intrusion detection has received sustained attention from the research community. In particular, by only relying on the meta-data of communication (e.g., no deep packet inspection, no text-mining on emails) we can alleviate Security Challenge 2: Encrypted, obfuscated, or compressed traffic with command-and-control servers to some extend: Whether the traffic is obfuscated, encrypted, or compressed makes no difference as long as we can access the meta-data. Furthermore, relying on a graph-based representation of communication patterns helps with conquering Security Challenge 1: Polymorphism since the polymorphism is mostly found in the employed malware, whereas the communication-patterns remain unchanged.

In order to apply graph-based intrusion detection, we need a graph processing and mining engine. In this chapter, we design, implement, and evaluate a distributed graph mining engine, capable of scaling with throughput of the input data, tackling Data Challenge 1: Variable throughput. Moreover, this helps with solving Security Challenge 3: Industry-oriented attacks as it permits to scale the IDS with the size of the industry to be protected. Its design allows to perform any kind of input-data to graph conversion — e.g. telephony call record, netflow, emails, blacklists or forums.

Aside from managing the high throughput in graph updates the graph can be queried con-

currently to other queries and to the updates. Based on that, this engine will allow us to employ a broad range of well studied graph algorithms for intrusion detection on realtime IP flow data. We therefore use it as our rule-engine throughout this thesis. Our graph mining engine is able to process around 39 million log entries per second on a 50 node cluster while providing processing latencies below 10 ms. We validate our approach by presenting two example rules, namely telephony fraud detection and internet attack detection. A thorough evaluation proves the scalability and near real-time properties of our system.

3.1 INTRODUCTION

The volume of log data in complex distributed systems can become very large due to the amount of data generated by individual nodes (e.g., a network router generating netflow data) as well as the number of nodes in a complex system (e.g., 100K+ compute nodes) each generating log data. Processing and analyzing this data to identify events of interest or to store the data for user initiated queries becomes a challenging task. This task is particularly challenging if the data needs to be analyzed in real time, for example, applications such as fraud detection require new entries in the log files to be analyzed in less than a second. Given the volume of data, it is typically not possible to store all the required data in the memory of one processing node.

Graphs are used in various applications to process log-data. Examples include clique analysis [FBO09], query-log analysis for search-engines [Don10], graph databases [WHW07], pattern matching [WY85], and fraud detection [CPV01; Ver+06]. Graphs can generally be used to store data representing interactions between entities such as phone calls between calling and called phone numbers, Internet communication between two end points (each defined by IP address and port number), or interactions between software components (e.g., call graph).

In this chapter we will demonstrate that large graphs can be updated and queried in near real time by distributing the graph onto multiple nodes (i.e., physical machines). We show that the space complexity for the distributed graph is linear.

Our contributions can be summarized as follows:

1. We construct dynamic, distributed graphs with linear space complexity.
2. We make these graphs easily accessible for queries.
3. The system scales linearly with the number of processing nodes.
4. We avoid unnecessary data copies by using the log-generating nodes for processing.

The rest of this chapter is structured as follows. We describe the approach and architecture in Section 3.2, followed by two example applications (telephony fraud detection and internet attack detection) in Section 3.3. We evaluate the performance of the system with these example applications in Section 3.4. Section 3.5 provides a survey of related work, followed by conclusions in Section 3.6.

3.2 APPROACH AND ARCHITECTURE

In this section, we will present our graph abstraction. We show how it enables us to distribute the graph across multiple machines with linear space overhead. Thereafter, we focus on the processing architecture, needed to fulfill our real-time and concurrent query requirements despite the massive amount of input.

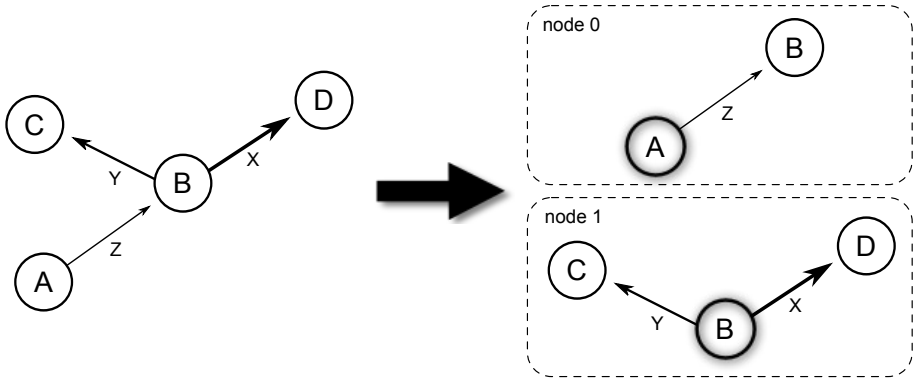


Figure 3.2: Distributed graph

3.2.1 DISTRIBUTED GRAPH

We propose a distributed, dynamic graph structure as depicted in Figure 3.2 as means for log processing. The graphs are directed, potentially cyclic, and do not need to be fully connected (i.e., there may be a vertex v_1, not reachable from a distinct vertex v_2). With this structure, correlations and relationships can be expressed easily. For example, the vertices *A, B, C,* and *D* could be physical machines, sending error messages *X, Y,* and *Z* to each other. This information could be used to determine how an error propagated through a complex system.

The graphs consist of a set of vertices V and a set of edges E: $G = (V, E)$ with $E \subseteq \{(v_1, v_2) \mid v_1, v_2 \in V\}$. Each edge $e \in E$ has a weight $w(e) \in \mathbb{R}$ and can store any kind of additional information (e.g., the different ports used, for every observed communication between v_1 and v_2).

SUB-GRAPH CONSTRUCTION

To distribute such a graph onto multiple nodes (i.e., physical machines or separate address spaces), we construct for each vertex $v \in C$, a sub-graph $G_v = (V_v, E_v)$ with:

$$V_v = \{v_a \in V \mid \exists e \in E \wedge e = (v, v_a)\} \cup \{v\}$$

where the set of vertices C contains only those vertices $v \in V$ which have at least one outgoing edge. The new set of vertices V_v now contains all vertices v_a that are connected with an edge from v to v_a, as well as v itself. To construct the set of edges E_v we add all outgoing edges $e \in E$ of v:

$$E_v = \{e \in E \mid \exists v_1 \in V : e = (v, v_1)\}$$

The vertex v is now called the center vertex of the sub-graph G_v. To determine on which node to store sub-graph G_v, we apply a hash-function h to the center vertex v and compute the modulo of $h(v)$ with the number of available nodes. Since each node has a unique id, which starts at 0, the result of the modulo operation equals the unique id of the node on which to store G_v. For example, in Figure 3.2, $h(A) \bmod 2 = 0$ matches the unique id of *node 0* and $h(B) \bmod 2 = 1$ matches the unique id of *node 1*.

SPACE COMPLEXITY

Because the sum of all $|E_v|$ is equal to $|E|$, the space needed to store all edge information is not changed by the sub-graph construction. The transformation does, however, increase the space needed to store the vertices, that is, the sum of all $|V_v|$ is greater or equal to $|V|$.

This is because of the vertices that are, at the same time, center-vertices and part of other sub-graphs, such as vertex B in Figure 3.2.

The multiset of vertices in the distributed graph $V^D = \biguplus_{v \in C} V_v$ increases linearly with the number of edges in each sub-graph:

$$|V^D| = \sum_{v \in C} |V_v| = \sum_{v \in C} (|E_v| + 1)$$

If the number of outgoing edges for each vertex $v \in C$ is limited by a constant K, then $|E_v| + 1$ can be substituted with $K + 1$ and the overhead is constant:

$$\frac{|V^D|}{|V|} \leq \frac{\sum_{v \in C}(K+1)}{|V|} = \frac{|C| \cdot (K+1)}{|V|}$$

In the worst case, all vertices in V have outgoing edges. Hence, C is equal to V and thus

$$\frac{|V^D|}{|V|} \leq \frac{|V| \cdot (K+1)}{|V|} = K + 1$$

Note, that the space complexity of the distributed graph is $O(|V|)$, independent of how many physical nodes are used.

Our algorithm maintains two graphs for every center vertex. One contains the information of all the data received within a given window (e.g., the last 100 updates for each center-vertex) and the other one stores the summary information of the complete data that has been received so far. The latter uses a top-k algorithm and the former its window size to limit the number of outgoing connections per vertex to K. We show an example of how to use and merge both graphs in Section 3.3. Both graphs are bounded in size. Therefore, the amount of data that is stored per node is bounded by the size of the two graphs times the number of center vertices, stored on that particular node. An algorithm for reconstructing the original graph G is given in Section 3.2.4.

3.2.2 PROCESSING ARCHITECTURE

Our design choices aim at maximizing pipeline, task, and data parallelism in order to support high-throughput and scalability. The resulting architecture shares these design principles with big-data processing systems such as MapReduce [DG04], and in particular with the online models of computation on data *streams* that were inspired from it [Neu+10; Fou; Bri+11; Bac+12].

STREAMMINE STREAM PROCESSING ENGINE

StreamMine uses the base construction principles illustrated by Figure 3.3. It is composed of a set of *operators*, sharing the same code and supporting pipeline parallelism. Operators are organized as a directed acyclic graph (DAG). Communication takes place in the form of *events* flowing through the DAG of operators, where an event is a <key,value> pair. Each operator can scale horizontally by using an arbitrary number of operator *slices*, each running on a different server and managing an independent state. Slices scale vertically by partitioning the received event load between all available cores on each server. There is no communication or shared state between the slices of an operator. Event forwarding between operators can use one three different primitives. The *unicast* primitive determines the *id* of the slice in the next operator by using a key and a hash-function (a simple modulo by default). A specific hash-function can be specified by the event handler function, which may implement more sophisticated stream partitioning mechanisms. The *anycast* primitive sends the event to a random slice of the next operator. The *broadcast* primitive sends the event to all slices

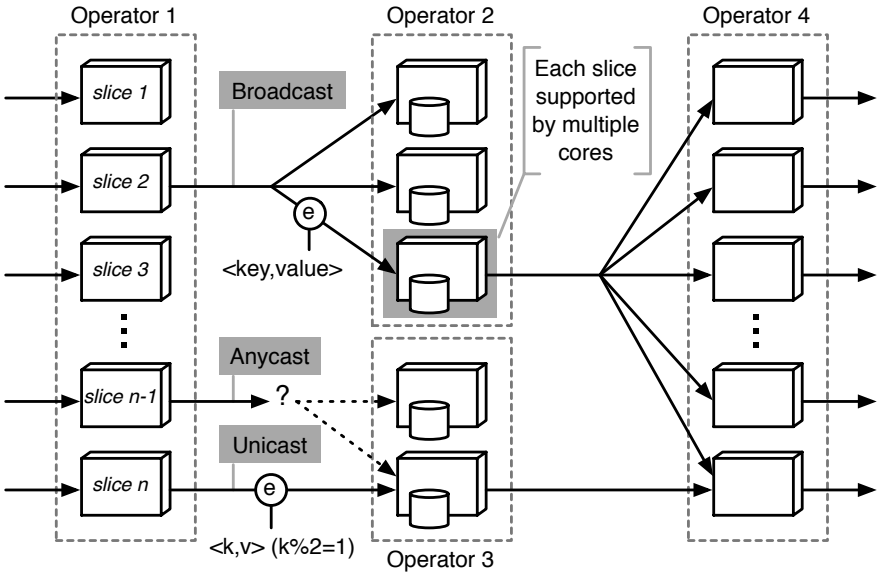


Figure 3.3: Architectural principles.

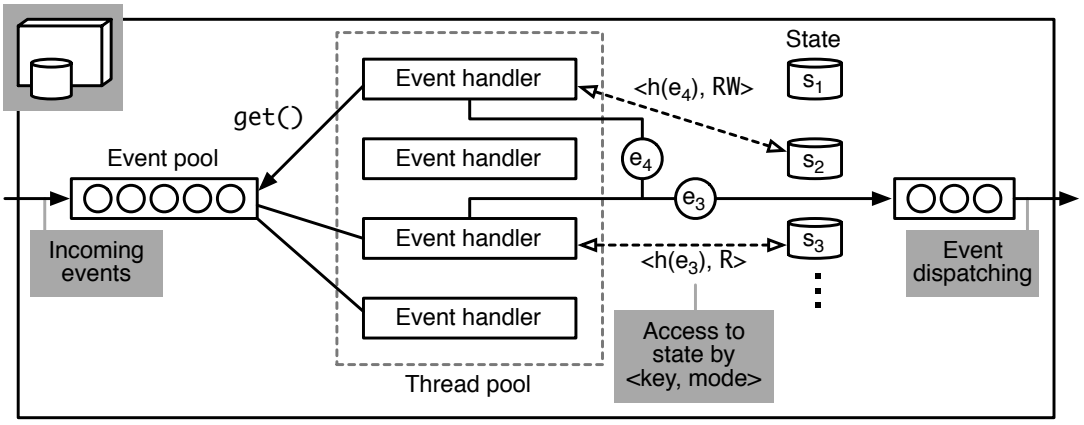


Figure 3.4: Details of an operator slice from Figure 3.3 supported by 4 threads.

of the next operator. All communications take place on pre-established and persistent TCP connections: all slices of one operator are persistently connected to all slices of the next operator(s) in the DAG.

All threads of all slices of a given operator support the execution of the same event handling function. Figure 3.4 presents a detailed view of the state of the greyed slice from Figure 3.3. Threads from a pool process incoming events from an input queue using a part of the state determined according to the event identifiers. The state of a slice (such as a window of events or summary information from previously processed events) is managed by StreamMine, and is partitioned using the same keys that are used for unicast routing between operators. Hence, there is no communication or shared state between the slices of an operator. Each thread accesses the state corresponding to the event to process using the appropriate read-only (`R`) or read-write (`RW`) mode. An event with key k_1 can be processed in parallel with another event with key k_2 as long as $k_1 \neq k_2$ or, if $k_1 = k_2$, only when both accesses are read-only.

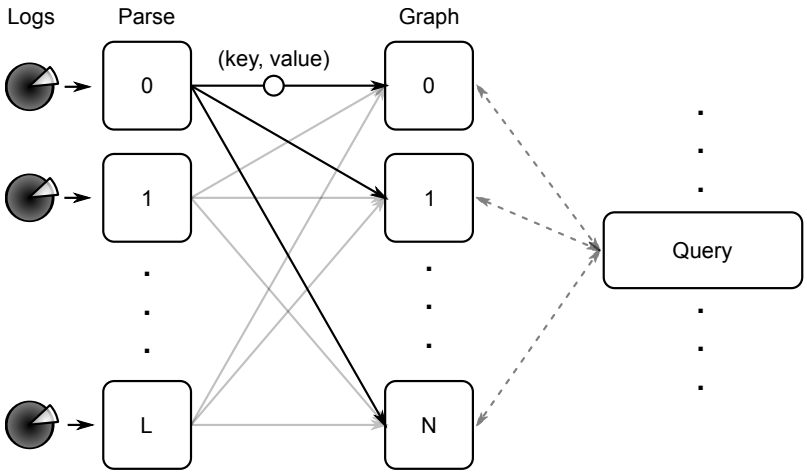


Figure 3.5: DAG of operators for the distributed graph architecture

GRAPH PROCESSING ON STREAMMINE

Figure 3.5 depicts the architecture of the distributed graph processing. The log data can be read from any socket. This includes disks, as well as TCP-connections and Unix-pipes. The *parse operator* is used to extract and convert the necessary information from the log data. For example, the parse operator can be used to construct a numerical representation of a human-readable IP address, such as "141.30.2.2".

Subsequently, the parse operator generates a new event and sends it downstream to the *graph operator* using the *unicast* primitive. As mentioned, all communication in StreamMine is event based and events are key-value pairs. Considering the example from Figure 3.2, if we want to send an event to update the sub graph with the center vertex *B*, our event has to have the key *B*. We will show how we chose the keys for our two example applications in Section 3.3.

Since the system uses back pressure, the processing speed of each component is not only limited by itself but also by all downstream components. For example, the parse operator slices cannot send more events than the graph operator slices can process.

3.2.3 SIMPLE QUERIES

The graph maintained by the graph operator slices can be accessed and queried. There is no limit on the number of queries running simultaneously. Moreover, each query itself may be distributed using the same concepts as the distributed graph for communication and parallelization. Since the routing scheme is static, it is straight forward to compute where the graph of a specific item can be found.

The query interface provided by the graph operator is based on events. So called *control messages* can be sent to request the graph of a given item and will be answered with a *response message*. Algorithm 3.1 shows how a query is performed.

3.2.4 MULTI-LEVEL QUERIES (TRANSITIVE CLOSURES)

To obtain the transitive closure of a sub-graph or even the complete graph, the previous algorithm has to be repeated recursively. Algorithm 3.2 shows this in pseudo-code. The algorithm takes the initial vertex and the desired depth as inputs. Initially, the set *visited* and the graph are initialized to empty sets. First, we check if the sub-graph of a vertex has already been processed. This ensures each vertex is included only once in the output. Thereafter, we check for each adjacent vertex in the sub-graph if it has already been processed, and if

Algorithm 3.1: Query a single sub-graph

```
   input  : vertex
   output: sub-graph
1  begin function query
2      control_message m
3      m.key = vertex
       // emits the event and waits for the answer
4      sub_graph = unicast(m)
5      return sub_graph
```

the remaining search depth is still greater than 0, we request the sub-graph of the current adjacent vertex. The result is the union of all the collected subgraphs.

Algorithm 3.2: Multi-level Query (Transitive closures)

```
1  global visited = ∅
2  global graph = ∅
   input  : vertex, depth
   output: graph
3  begin function closure
4      if vertex ∈ visited then
5          return ∅
6      visited.insert(vertex)
       // the query function is shown in algorithm 3.1
7      sub_graph = query(vertex)
8      graph = graph ∪ sub_graph
9      if depth > 0 then
10         foreach adjacent_vertex ∈ sub_graph do
11             if adjacent_vertex ∈ visited then
12                 continue
13             sub_graph = closure(adjacent_vertex, depth −1)
14             graph = graph ∪ sub_graph
15     return graph
```

3.2.5 QUERY EXAMPLES

This section will present some implementations of popular graph algorithms on top of our infrastructure. Since it is impossible to present all of them we focused on the non-trivial ones which are actually applied in practice.

LOCAL CLUSTERING COEFFICIENT

Details The local clustering coefficient of a vertex v is the fraction of pairs of neighbor vertices which are connected over all pairs of neighbor vertices of v. Intuitively speaking, the local clustering coefficient is high, if the neighborhood of a vertex is well connected.

Applications Suppose, the vertices in a graph are actual users in a social network. The edges connecting them represent a relation of some sort – e.g. friendship, professional relationships, and so on. Now, we want to find malicious users who connect to many random users. One reason to follow this goal is to increase their influence within the network. However, if a user connects to a random subset of other users, those random users are unlikely to be well connected. Hence, the local clustering coefficient of the malicious user will be small [Alv+13]. Ding et al. have shown that the local clustering coefficient can even be used to detect intrusions [Din+12].

Algorithm 3.3: Local Clustering Coefficient (lcc)

input : vertex
output: lcc

```
1 begin function local_clustering_coefficient
      // get neighbors of vertex
2     N = closure(vertex, 1)
      // N contains also the neighbors of the neighbors of "vertex" which we don't need
3     N = N /{v_j : ∄e_{vertex,j} ∈ E(N)}
      // The local clustering coefficient is the fraction of actual links between the
         neighbors of "vertex" over all possible links between the neighbors of "vertex"
4     lcc = |{e_jk : v_j, v_k ∈ V(N), e_jk ∈ E(N)}| / V(N)(V(N) − 1)
5     return lcc
```

Algorithm Algorithm 3.3 first acquires the closure of *vertex* with depth 1 and stores the returned subgraph in *N*. However, this will also include the vertices which are neighbors of the neighbors of *vertex*. Since the local clustering coefficient only considers the direct neighbors of *"vertex*, we remove all vertices from *N* which are not direct neighbors of *vertex* in the next step. Now, calculating the local clustering coefficient is easy. The enumerator of the fraction in line 5 is the actual number of edges which connect the direct neighbors of *vertex*. The denominator is simply the number of possible edges in the direct neighborhood of *vertex*.

SEED-BASED COMMUNITY DETECTION

Details Community detection algorithms thrive to find all communities in a graph. However, Abrahao et al. [Abr+12] argued that research should focus on giving those algorithms some guidance as to which community should be found exactly. One example is seed-based community detection which detects communities around a given set of seed vertices. Examples are personalized page-rank [ACL06], low-conductance cuts [GS12], or binomial probabilities [MS09].

Applications Applications of community detection are broad. Google's famous pagerank algorithm is in fact a community detection algorithm [ACL06]. Researchers have also shown that community detection can predict future attack targets [ZPU08]. However, such algorithms are most prominently used in social network analysis [Les+08].

Algorithm In Algorithm 3.4, we present a general framework with which all community detection approaches can be implemented. It is based on the work by Yang et al. [YL15]. A programmer only needs to supply the *Select* and *Rank* functions. The former selects a set

Algorithm 3.4: Local Community Detection, based on the framework by Yang et al. [YL15]

```
input : S, depth = 1
output: C
1 begin function local_community_detection
      // get neighbors of vertex (can be executed in parallel)
2     foreach seed ∈ S do
3         N = N ∪ closure(seed, depth)
      // In a first phase, the algorithm implements a function which selects possible
         community members, i.e. canidate selection
4     N = Select(S, N)
      // In a second phase, those selected candidates are ranked by how "close" they
         are to the seeds
5     C = Rank(S, N)
6     return C
```

of good candidates from the graph. All seed-based community detection algorithms select vertices in the neighborhood of the seeds. When using algorithms like minimal cut [GS12] or normalized inverse conductance [WHF14] only direct neighbors of the seeds are selected. Algorithms which use personalized pagerank [ACL06] for candidate selection will search deeper into the neighborhood of the seeds, however, how deep is bounded by the jump-back probability e (i.e. the maximum depth is e^{-1}). For such cases, an optional argument *depth* can be supplied to the function. Note, that the *closure* function 3.2 can be executed in parallel as well - hence, the most time and communication consuming part of the algorithm can be completely parallelized. The latter then performs ranking on the previously selected neighbors, creating a mapping C, which maps each selected candidate to a rank or score which determines how well the given candidate fits into a community.

3.3 USE CASES

In the following, we will provide two use-cases that we have investigated using our approach. The first is a fraud detection application for the telephony domain and the second is an attack detection application for the internet domain. Both examples use community of interest graphs (COI) [CPV01]. Algorithm 3.5 shows how the COI graph is constructed. We first add the received entry to the window, which is the first graph we store for every center vertex. If more than a pre-defined number of connections have already been added to the window, the window is merged with the (potentially not yet) existing COI, which is the second graph we store for every center vertex (topk). To this end, the application needs to define an attribute by which to measure the weight of the connections. With that, the weight (sum of all the attribute values) in the window, multiplied with a damping factor $1 - \theta$, are added to the weight (multiplied with θ) in the COI. Since $\theta = 0.85$ in both examples, the influence of the new connections in the window is damped. Thereafter, the weights of all contacts in the COI that have not been observed during the current window are decayed by multiplying them with θ. To keep the COI at a maximum size of K, we remove the weakest link until the size of the COI is smaller or equal to K. Finally, the window and the counter are reset.

Algorithm 3.5: Top-k graph construction

```
   input  : (window, topk, counter, new_contact, attribute)
   output: None
1  begin
       // Save new contact in window
2      window[new_contact].weight += attribute
3      counter++
       // Merge window into top-k after N events
4      if counter > N then
5          foreach contact ∈ window do
               // θ has a value of 0.85 in our analysis.
6              ww = window[contact].weight
7              tw = topk[contact].weight
8              topk[contact].weight = 1 - θ * ww + θ * tw
           // Decay weight of old connections
9          foreach contact ∈ topk ∧ contact ∉ window do
10             topk[contact].weight = θ * topk[contact].weight
           // Copy additional information I which can be saved with the edges
           // Remove the weakest links
11         while |topk| > K do
12             remove_weakest_link_from(topk)
13         window = ∅
14         counter = 0
```

3.3.1 TELEPHONY APPLICATION

In this application, we parse *call-detail-records* (CDRs) that contain various information for each telephone call. The CDRs are stored in a text file with each line representing one CDR. The parse operator extracts the caller, callee, and call-duration information from each CDR. Thereafter, two events are sent to the graph operator: one with the caller's hash as the key and one with the callee's hash as the key. Each event is about 24 bytes large. In the graph operator, we construct a sub-graph for each event's key, i.e., there is a sub-graph for each phone number observed in the CDRs—be it the caller or the callee. This approach is similar to [CPV01], with the exception that we dynamically update each sub-graph individually, instead of updating the complete graph after a fixed time interval.

GRAPH

Figure 3.6 shows the top-k sub-graph of a phone number. The adjacent vertices are other phone numbers that have either been callers or callees with respect to communication with the center vertex. Edges are weighted by the sum of all call durations of each communication between two vertices. The graph was obtained using the simple query function (see Algorithm 3.1). All links were stored into a file and then rendered using the *neato* tool, contained in the *graphviz* [Ell+02] package.

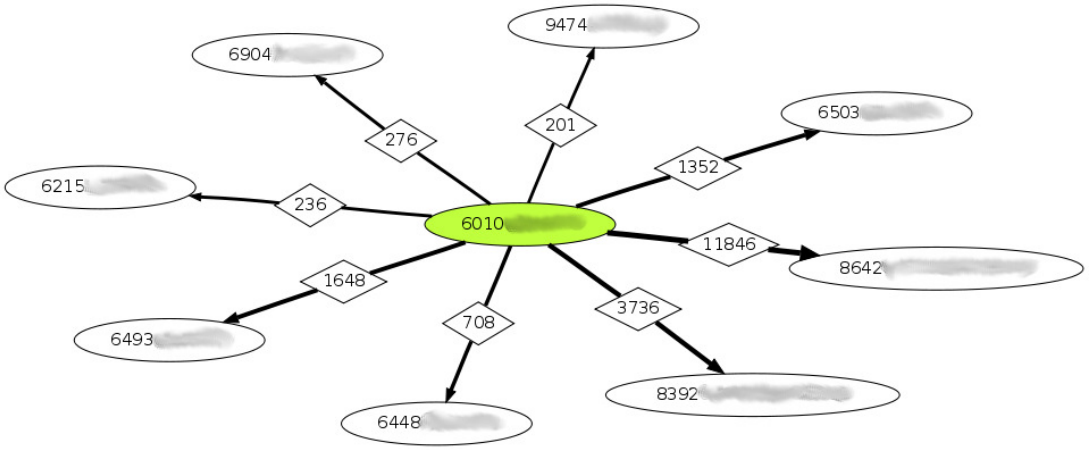


Figure 3.6: Community of interest example for an anonymized phone number

SOCIAL FINGERPRINTING

One application of communities of interest is social fingerprinting. The idea is that an individual can be identified, with a certain probability, using only its community of interest. The probability depends highly on the size of the COI. Research has shown that a COI of size 9 is sufficient [CPV01]. This, in turn, can be used to check whether the same individual lies behind two or more different numbers. For example, if we have a set of known fraudsters, we can check if the COI of a suspicious customer is sufficiently similar (e.g., 90%) to one of the stored fraudster's COIs. This check needs to be done in real time, since action must be taken (e.g., block a call) before the actual call is completed.

Implementing such a fingerprint query on top of our graph-mining system is straight forward and shown in Algorithm 3.6. First, we query the COI of the number in question. Then we construct the set intersection for the received COI with each of the stored fraudster's COIs. If at least one of the intersections is as large as 90% of the COIs size, a possible fraudster has been found and a fraud analyst should further investigate the case.

Algorithm 3.6: Fingerprinting

```
input : suspect_number
output: alarm
1 begin
2     suspect_coi = query(suspect_number)
3     foreach coi ∈ fraudster_cois do
4         intersection = coi ∩ suspect_coi
5         if |intersection| >= 0.9 * |coi| then
6             return true
7         else
8             return false
```

3.3.2 NETFLOW APPLICATION

In this application, we parse netflow logs, recorded by routers, which contain various information about internet based communication connections. We describe this application in detail in Chapter 4. However, we give a short overview here in order to motivate the usefulness of our engine for intrusion detection. Each netflow entry is a line in a text file. We use the parse

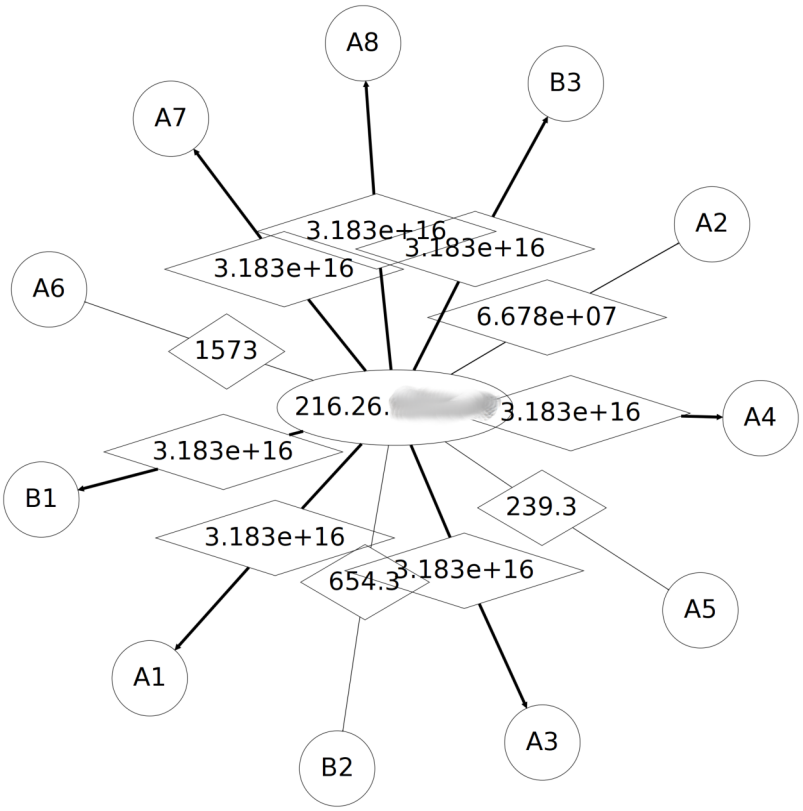


Figure 3.7: Community of interest example for an anonymized IP address

operator to extract the source-ip, target-ip, source-port, target-port and number of transferred bytes from each netflow entry. Thereafter, an event, representing one netflow entry, is sent to the filter operator. Each event is about 32 bytes.

The filter operator is a new operator, introduced for the netflow application, to discard unimportant traffic before it reaches the graph operator. The decision to discard an event depends on various factors, such as the ports used and source and destination IPs. A detailed description of the filter operator is out of the scope of this chapter, but fundamentally we use this operator to filter traffic that is not related to the community or uses trusted protocols. Finally, if the original event was not filtered, the filter sends an event to the graph operator slices using the source-ip's hash as the key.

With that, we construct a sub-graph for each unfiltered IP address. We use the number of bytes transferred to determine the weight of each edge and also store the used ports as additional information for each edge.

GRAPH

Figure 3.7 shows a top-k sub-graph as it is used in the internet attack detection application. The center vertex depicts an IP address that connects to various members of the community "A" and three members of the community "B". We omit the real community names for privacy reasons. The edges depict communication between the two vertices and the diamond boxes depict the weights of the connections. These weights are determined by the sum of the transferred bytes by each communication between two vertices. While not shown in the figure, the edges also contain every source- and destination-port combination ever used by the two vertices. The graph in the figure was obtained using the simple query function (see Algorithm 3.1).

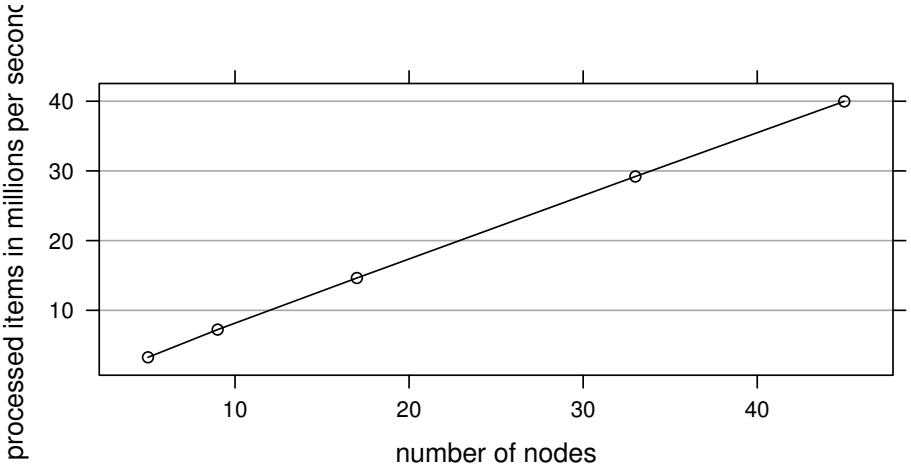


Figure 3.8: Scalability of dynamic graphs

SECURITY INCIDENT DETECTION

The netflow application is used to detect APT attacks. The basic idea of this application is to identify IP addresses outside the community that communicate with a number of different organizations in the community.

3.4 PERFORMANCE EVALUATION

We executed all benchmarks of the telephony application on a 50 node cluster, each equipped with two Intel Xeon quad-core processors and 8GB of RAM. The relation of nodes for this experiment is $(n : n + 1)$: n parse nodes and $n + 1$ graph nodes.

The experiments, using the netflow application were executed on a 9-node cluster, each equipped with 4 Intel Xeon quad-core processors and 24GB of RAM. The relation of nodes for this experiment is $(n : n : 1)$: n parse nodes, n filter nodes, and 1 graph node. Each node is a physical machine.

The measurements were conducted by processing accumulated (i.e., historic) data, because the volume of the real time data was not high enough (for example, we can process one day of netflow data in 40 minutes on a single machine) to show how the system behaves (i.e., with regard to latency and throughput) at its limit. Of course, we plan to integrate our system with real-time analysis at a later point in time.

3.4.1 TELEPHONY APPLICATION

Figure 3.8 shows the scalability of the dynamic graphs. It can be seen that the dynamic graphs scale linearly with the number of nodes used and, more importantly, with the input traffic. The linear characteristic is due to the fact that we use the same set of CDRs for every setup. Thus, the number of sub-graphs per node decreases as more nodes are added. In the largest configuration, we can process up to 39 million log-entries per second.

This is still being done in near real time: Figure 3.9 shows the mean latency from reading a new log entry until the processing is completely finished for different configurations. It can be observed that by adding more nodes, the overall latency declines. The reason for this is the same as for the linear scalability in the throughput measurement. Note, however, that the throughput also increases with larger setups (as shown in Figure 3.8). Therefore, by increasing the number of nodes, latency can be maintained at a constant level even if the

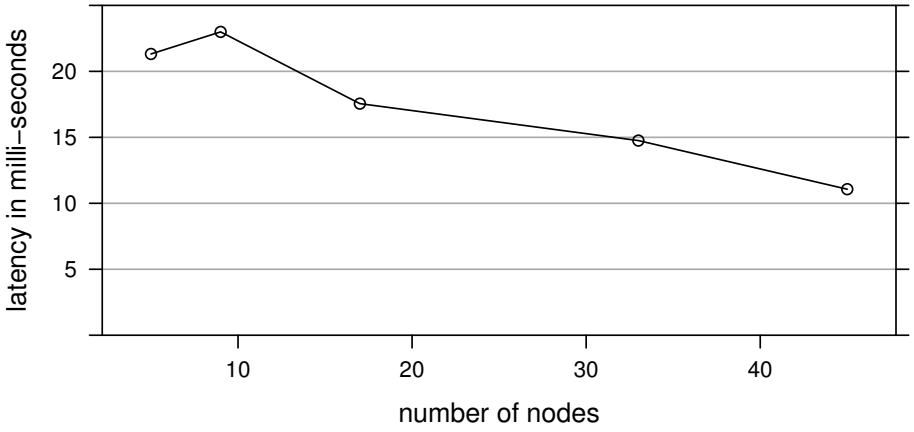


Figure 3.9: Latency of processing individual log entries

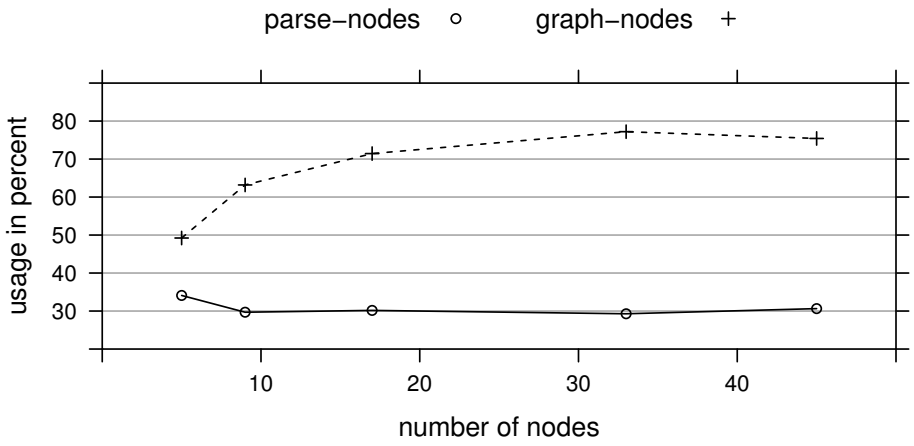


Figure 3.10: CPU utilization, separated for source and graph nodes

workload increases.

One of the motivations for mining log data with a distributed system is that the computational overhead can be distributed. For example, the nodes that parse the entries from the log files will not have to dedicate all their resources to the mining and may be used for other purposes. This argument is supported by Figure 3.10. It shows that the CPU of the parse operator slices is much less utilized than that of the graph operator slices. However, the network connections of the parse operator slices are fully utilized (i.e., 115MB/s) because the parsed entries need to be sent to the graph operator slices.

It can also be seen that larger configurations yield a higher CPU utilization on the nodes running the graph operator slices. This is caused by the fact that we always use 1 graph operator slice more than parse operator slices ($n : n + 1$), which is the setup in which the telephony application performs optimally. Since this is constant, its influence is higher for smaller configurations and the graph nodes in smaller configurations will be less utilized.

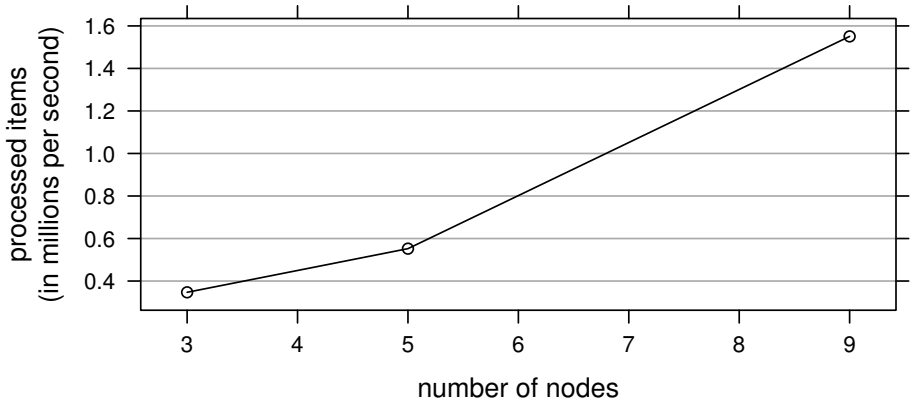


Figure 3.11: Scalability of dynamic graphs

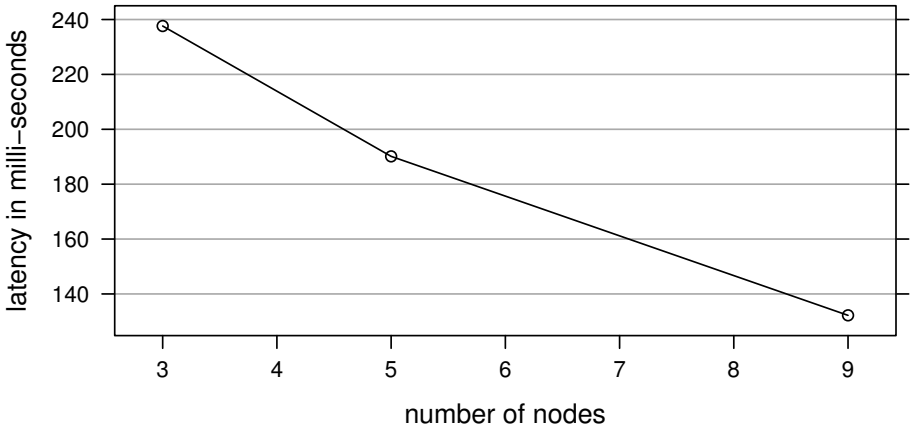


Figure 3.12: Latency of individual log-entries

3.4.2 NETFLOW APPLICATION

We executed the same set of measurements for the netflow application. However, the results cannot be compared directly since the hardware is different[1] and the netflow application uses one additional operator (the filter).

Figure 3.11 shows that the netflow application scales almost linearly with the number of nodes. Figure 3.12 depicts the mean latency from the time a netflow entry is read until the graph operator finished processing it completely. As expected, the latency is significantly higher than for the telephony application. This is due to (1) a non-optimal implementation of the filter and (2) the network delay, introduced by the additional filter operator. The reason that the latency decreases for larger configurations is that the filter operator slices, which dominate the overall latency, are less loaded.

The CPU utilization (see Figure 3.13) shows that the filter operator slices have the highest CPU utilization. The parse operator slices do not require as much CPU since they are limited by I/O operations (reading the log entries from disk and sending events to the filter operator slices). Again, the reason for the decrease of CPU utilization in larger configurations is mostly due to lower load on the filters.

[1] Due to legal reasons we were not able to execute all experiments on the same infrastructure.

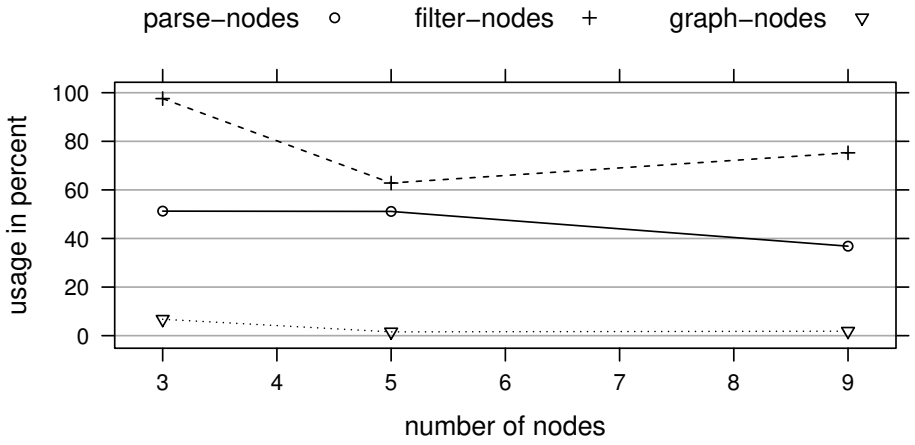


Figure 3.13: CPU utilization, separated for sources, filter and graph slices

Interestingly, however, the graph operator slice is much less utilized than for the telephony application. This is true for the CPU utilization (see Figure 3.13) as well as the network traffic. As depicted in Figure 3.14, the network throughput from the parse operator slices to the filter operator slices (the graph on the left side) is by orders of magnitude higher than the network throughput from the filter operator slices to the graph operator slices (the graph on the right side). Therefore, also the CPU utilization of the graph operator slice is by orders of magnitude lower than for the telephony application, where the graph operator slices have to process all the traffic sent from the parse operator.

Consequently, CPU utilization can be traded for network traffic: Either the CPU usage is lower but the network traffic is higher (as shown in Figure 3.10) or the CPU usage is higher but the network utilization is lower (as shown in Figures 3.13 and 3.14).

3.5 RELATED WORK

The related work can be split into three distinct categories: data mining, graph mining, applications for graphs mining. The latter includes distributed graphs as well as fraud detection on graphs.

3.5.1 DATA MINING

A system close to our mining platform is IBM's SPC [Ami+06]. It provides a rich API for defining operators. However, it only reaches high throughput with large event sizes (more than $128KB$) and the processing elements can only operate on fixed windows over the input stream, while our system allows a state to be stored independently from a window (such as the top-k graphs).

Dryad [Isa+07] is a streaming system by Microsoft. It relies on the availability of a distributed file system and processing elements are executed in a single-threaded manner. No latency measurements or absolute numbers on the achieved throughput are provided in [Isa+07]. SCOPE [Cha+08] is a system by Microsoft, specifically designed to parse log files. It relies, like Dryad, on the availability of a distributed file system. Furthermore, it provides only limited customizability of operators, i.e., operators need to be defined by means of the three constructs: PROCESS, REDUCE, and COMBINE. No absolute latency or throughput results are provided in [Cha+08] and the scalability is sub-linear.

Hadoop [Fou10] is the main open-source implementation of the MapReduce paradigm

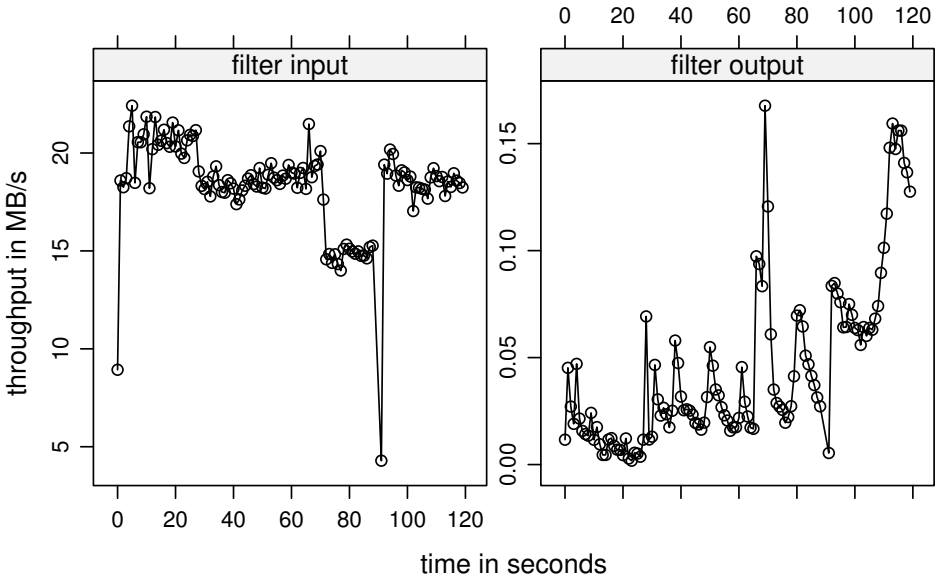


Figure 3.14: Influence of filtering on throughput

[DG08]. It is intended for batch processing and therefore provides latencies far beyond the seconds range. Several variations of the model have also been proposed. Hadoop Online [Con+10] improves Hadoop's efficiency and latency by enabling a direct communication between mappers and reducers. In the original publication, the authors consider the processing of continuous data streams, but provide only a minimal example and no performance evaluation. State is incorporated into MapReduce in [Log+10]. However, the goal is to provide better support for incremental batch jobs and, thus, it will not support applications requiring low latency. Better incremental support for batch systems is also the focus of the new Google system, named Percolator [PD10]. Unfortunately, the observed latencies in Percolator can range up to minutes.

Borealis [TÇZ07] addresses scalability by optimizing the placement of operators, balancing load, and applying sophisticated techniques for load shedding. However, it only supports SQL-like operators. Moreover, it does not address the problem of parallelizing a single operation among a large number of nodes. GSDM [IR05] addresses operations that are partitionable, like is the case with MapReduce, but it does not consider operators that maintain state—operator process whole windows at a time, similar to our jumping windows.

Finally, StreamMine [BFF09; Bri+08] addresses low latency, fault tolerance, and parallelization of stateful operators, but the speculative approach does not scale horizontally (i.e., among nodes within a cluster).

3.5.2 GRAPH MINING

The *Gather - Apply - Scatter (GAS)* model has become the standard programming abstraction in distributed graph processing engines. Algorithms in this model are formulated as vertex programs. The vertex program is then executed on each vertex individually. The algorithm halts if no more vertex has scheduled itself or was scheduled by one of its neighbors (who is in charge of scheduling vertices depends on the actual engine). In the *gather*-phase, the vertex program collects data from its neighboring vertices. In the *apply*-phase, the vertex program may perform any type of computation with these values. Finally, in the *scatter*-phase the vertex program may communicate new values to its neighbors. Or, depending on the engine

in use, may even schedule its neighbors for execution. Distributed graph engines differ in the way they schedule execution of vertex programs, how vertices (or edges) are distributed over the set of computing nodes, and additional optimization techniques like caching intermediate results.

Pregel [Mal+10] is a distributed graph processing engine, based on Apache Hadoop. It was later released as part of the Apache project under the name "Apache Giraph". The execution of vertex-programs is divided into so-called "supersteps". In each superstep, the vertex programs run in parallel and may receive messages from their neighboring vertex which have been sent in the previous superstep. They may also send new messages to their neighbors which they will receive in the next superstep. Each vertex decides itself if it wants to be executed at the next superstep. The execution of supersteps is strictly sequential, i.e. the execution of one superstep has to be finished completely before the next superstep may be run. The exchange of messages between the vertices is facilitated by message passing.

GraphLab [Low+12] takes a different approach in that vertex-programs can access the state of their neighbors. In contrast to Pregel, they may also schedule the execution of neighboring vertices. This can help with some algorithms, such as pagerank, where vertices only need to update their rank if the rank of their neighbors changed significantly. Instead of scheduling vertex programs in supersteps, GraphLab achieves serializability by scheduling only those vertices in parallel which cannot share any state, i.e. have a disjoint set of neighbors. This scheduling scheme has proven to especially efficient with algorithms that converge to a final solution (e.g. PageRank).

GraphLab and Pregel scale well as long as inter-machine communication is limited. In both systems, vertices are distributed over computing nodes through the application of a hash-function on the vertex-id. This has the drawback, that vertices which are connected by an edge may be assigned to different computing nodes. A problem which is especially prominent with graphs whose degree distribution follows a power-law: They are hard to partition efficiently as some of the vertices have several orders of magnitude more neighbors than most other vertices. Since the communication complexity of vertex programs is $O(n)$ with n being the number of neighbors, this results in some vertex programs having to exchange many messages with remote computing nodes in the *gather* and *scatter* phases. Therefore, PowerGraph [Gon+12], distributes the graph using so-called vertex cuts, as it is the case with our solution, presented in Chapter 3. As we have seen in Chapter 3, the result is that an edge never spans different computing nodes. In addition to our solution, they minimize the number of additional vertices which need to be stored.

Finally, instead of creating another highly specialized distributed graph processing engine, GraphX [Gon+14] provides a *GAS* API on top of Apache Spark – a distributed dataflow system. The authors show that a graph parallel computing abstraction as used by the *GAS* model can be mapped onto traditional *JOIN* and *GROUPBY* SQL operations on property graphs. The authors then propose several optimizations to how the *JOIN* and *GROUPBY* are executed, such as caching intermittent results. With that, GraphX can reach a higher performance than any of the aforementioned systems.

3.5.3 GRAPHS

The use of graph structures for different kinds of applications has been addressed in number of research efforts [WY85; FBO09; Don10; Ver+06]. However, there is very little related work on how to efficiently distribute huge graphs. The authors in [III+05] propose a distributed graph algorithm but focus only on finding the strongly connected components. It is not clear if the graphs can be updated and with what latencies. Taentzer [Tae99] provides a theoretical analysis of distributed graphs, but no implementation is provided.

In the field of community of interest-based fraud detection, the closest related work is

[CPV01]. However, the authors do not consider constant updates but rather batch processing. Moreover, the computation and the graph are not distributed.

The indexing of graph structures in databases has been investigated in [WHW07]. Nevertheless, the analysis considers only one computer (not distributed), and latencies for queries and updates are in the range of in the seconds with larger databases.

3.6 SUMMARY

We have demonstrated how to use graphs for many interesting log processing problems. The evaluation showed that we can process huge amounts of log data in near real time (i.e., 10 ms). This is especially due to the very good scalability of our system. Since the index is deterministic and easy to compute, queries are very simple to implement. Addition of new operators, such as our filter operator, is easy and can result in significant reduction of data transferred and processed.

4 COMMUNITY-BASED ANALYSIS OF NETFLOW FOR EARLY DETECTION OF SECURITY INCIDENTS[1]

[1]The original version of this chapter appeared at LISA '11 [WHF11a].

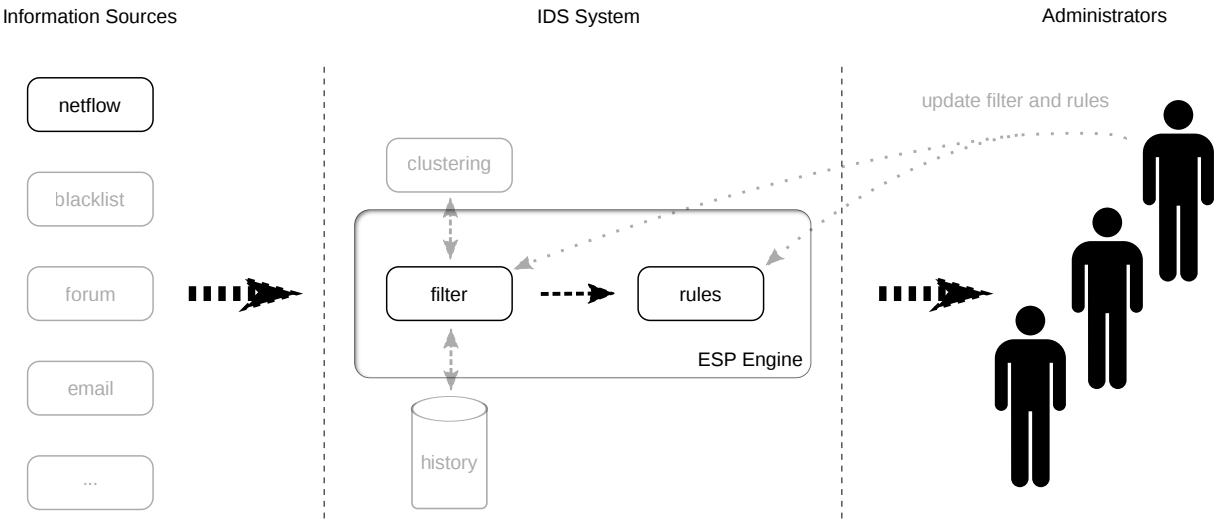


Figure 4.1: This chapter is concerned with an efficient implementation of the filter component and with the evaluation of a graph-based intrusion detection algorithm

In the last two chapters we have not only learned how to detect industries in IP flow data but also how to update and process huge distributed amounts of input data. Since we now know which IP addresses belong to a given industry and have a highly scalable and adaptable graph-based rule-engine, we will design and implement a community based IDS in this chapter. With that, we are finally in the position to tackle Security Challenge 4: Stealthy attacks. Given a scalable, efficient graph-based rule engine, an efficient filter and the knowledge about the boundary of a given industry in our input data, we can correlate events from different industry members to lift "below the radar" attacks above the radar. To this end, we first implement an efficient filter which drops irrelevant events. Furthermore, we implement a complex graph-based rule, which is able to detect intrusions automatically in netflow traffic. We present several case-studies and conclude with an outlook on how to enhance this engine to support additional information sources.

4.1 INTRODUCTION

Detection and remediation of security incidents (e.g., attacks, compromised machines, policy violations) is an increasingly important task of system administrators. While numerous tools and techniques are available, novel attacks and low-grade security events may still be hard to detect in a timely manner. Specifically, system administrators typically have to base their actions on observing the local traffic to and from their own networks as well as global security incident alerts from organizations such as SEI CERT[1], Arbor Atlas[2], or software and hardware vendors. However, stealthy targeted attacks may slip below detection thresholds both in the local data alone or on the global scale.

Furthermore, the nature of internet-based attacks is changing from random hacking to financially or politically motivated attacks. For example, botnets are increasingly leased out to highest bidders and DDoS attacks are often used as a means for blackmail. Moreover, attacks targeting industries with financial information (e-commerce, banking, gaming, insurance) are increasing and the threat of attacks against SCADA (supervisory control and data acquisition) systems in electrical power generation, transmission, and distribution (among other industrial

[1] http://www.cert.org/
[2] http://atlas.arbor.net/

process control systems) is even considered a potential target for terrorism [ILW06].

Targeted attacks might not leave a large traffic footprint in the targeted organization since one machine with access to the desired information or control system may be sufficient for the attacker to achieve their goals. It is often difficult to detect such low-footprint attacks based on local monitoring alone because it is often necessary to set local alerting thresholds high enough not to generate too many false positives and overwhelm the system administrators. But as a result, a stealthy attack or compromise may lay undetected. Therefore, it is possible for an attacker to target many such organizations without being detected. For example, the attacker may want to maximize profit by attacking multiple financial organizations concurrently before the vulnerability used is detected and corrected. Similarly, terrorists may require the control of many companies to achieve their goal of large scale damage.

In this chapter, we present a novel approach for detecting stealthy, low-grade security incidents by utilizing information across a community of organizations (e.g., banking industry, energy generation and distribution industry). We will show by using an example that we can find possible attacks (or attempts) that only transfer very little data (e.g., a few bytes) and thus would remain undetected by conventional approaches.

The remainder of this chapter is structured as follows. In Section 4.2, we present the technical approach based on netflow data and construction of communities of interest. Section 4.3 describes the implementation of the system, including the algorithms used for the analysis. We evaluate the performance of our system in Section 4.4 and present selected case studies of suspicious activity we have identified in Section 4.5 . Section 4.7 outlines related work in the area and Section 4.8 concludes the chapter.

4.2 APPROACH

4.2.1 SERVICE VISION

Our technique is based on the concept of *community*, in our case defined as a collection of (at least two) organizations. A community can be specified based on any criteria relevant for attack detection. For example, it could consists of businesses in a particular industry (e.g., banking, health care, insurance, etc), organizations within a country (e.g., businesses and government agencies in one country), or organizations with particular type of valuable information (e.g., industrial espionage or customer credit card information). We detect stealthy security attacks by observing the communication to/from the member organizations of a community. The intuition being that within each organization only very few machines may be attacked or compromised and as a result an attack can be very hard to detect within each organization. However, by observing the communication behavior across multiple organizations in the community, such stealthy behavior may become visible.

Given that we analyze communication in the Internet, each organization is defined by the list or range of IP addresses belonging to the organization. We consider Internet communication connections (reported by netflow, for example) within the communities and between communities and external IP addresses who do not belong to any community. For our analysis, all the IP addresses within an organization can be collapsed into one identifier representing the organization. Any communication between two IP addresses where neither belongs to one of our communities and neither has communicated with a community in the past can be ignored. Furthermore, communication with IP addresses belonging to commonly used Internet services (e.g., search, news, social media) can be white listed and removed from consideration.

We construct a communication graph for each IP address that communicates with at least one organization in a community as illustrated in Figure 4.2. This figure shows the communication graph for an external IP address (i.e., some IP address outside any of the communities

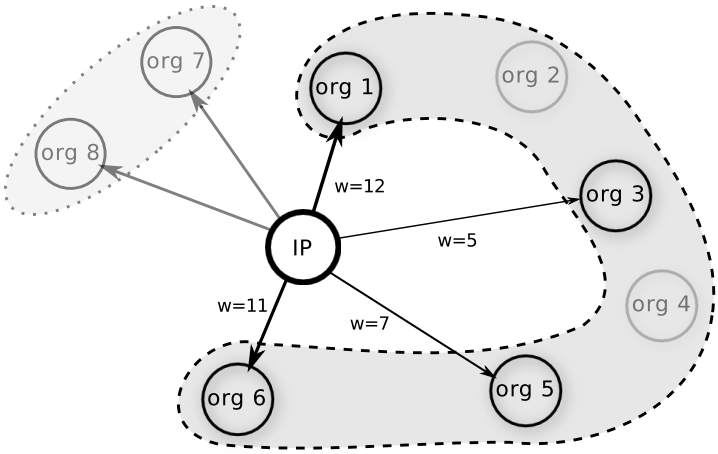


Figure 4.2: Communication graph for an IP address

of interest). This node has communicated with two communities, one consisting of organizations 7 and 8, and the other consisting of organizations 1 through 6. A directed edge from some node A to some other node B in the graph indicates that A has sent messages to B. Although not depicted in the figure, each edge may contain additional information, such as the combinations of source and destination ports used.

The weight of the edge is used to quantify the importance of the communication. The importance can be based simply on the number of messages or bytes sent, or the number of contacted individual members in the targeted organization. However, some communication may be more important than others from security point of view. For example, some port numbers are more often involved with malicious activity (e.g., based on CERT reports) and communication using such ports can be weighted more heavily.

The weight is also used to limit the size of each graph. The size of the graph is determined by the number of nodes it contains. If the size exceeds a given threshold, we remove the weakest links until the threshold is reached. This is necessary because storing all communications would require too much space even for a single day. For example, in our data set consisting of heavily sampled netflow, a given weekday contains about 860 million entries. These 860 million recorded netflows originate in 28 million distinct IP addresses. Therefore, if we would not filter unimportant IP addresses, we would need to store 28 million graphs. Moreover, each of these 28 million IP addresses often connects with 1 to 2 million other IP addresses. Thus, if we did not limit the size of each graph, we would have some graphs that are too large to fit into memory. The situation would be even more challenging if we analyzed the data for one month or a week instead of the current one day at a time.

As already stated, we also consider communication within a community and across communities. With that, we are able to detect already compromised computers inside an organization when they try to attack further organizations as shown in Figure 4.3. To reduce the number of false positives (many organizations have frequent contact with other organizations of the same or other communities), a computer inside an organization that belongs to a community (or is contained in the whitelist) has to show more suspicious behavior than an external IP address before an alarm is generated. For example, we do not consider communication via port 443 with or across communities.

Given such communication graphs, a potential security incident is suspected when an IP address communicates with a specified number of community members. Typical examples of security threats that can be detected using this approach include botnet controllers managing a number of bots in the community, compromised machines downloading stolen information on a dedicated server, an attacker targeting machines in multiple organizations, as well as many

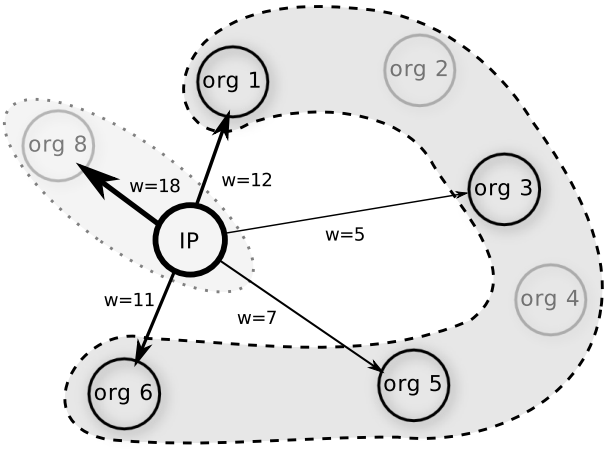


Figure 4.3: Communication graph for a community member

security policy violations (e.g., illegal software download sites, etc). The number of alarms can be controlled using thresholds and the system can memorize IP addresses that have already been reported recently. When there are false positives, the system administrators can extend the whitelist.

An IP address may contact a large number of community members either because the community is actually targetted or if the attacker is targetting all or most of the Internet (e.g., broad port scan). The system administrators may want to react differently to these alternative scenarios. Therefore, for each IP address that has contacted a community member, our system keeps track of how many times it has communicated with IP addresses outside our communities of interest.

4.2.2 INPUT DATA

Our community-based alerting service uses netflow as its input data source (although other types of information could be utilized as well). Netflow is a standard data format collected and exported by most networking equipment, in particular, network routers. It provides summary information about each network communication passing through the network equipment. Specifically, a network flow is defined as an unidirectional sequence of packets that share source and destination IP addresses, source and destination port numbers, and protocol (e.g., TCP or UDP). Each netflow record carries information about a network flow including the timestamp of the first packet received, duration, total number of packets and bytes, input and output interfaces, IP address of the next hop, source and destination IP masks, and cumulative TCP flags in the case of TCP flows. Note, however, that the netflow record does not contain any information about the contents of the communication between the source and destination IP addresses.

The community-based alerting service requires access to netflow to/from each of the organizations in the community. Such data can be collected by each of the organizations in the community at their edge routers and then collected at a central location for processing. Alternatively, it can be provided by an ISP that serves a number of the organizations in the community. Note that the netflow data may be sampled (to reduce the volume of the data) and the actual IP addresses of the computers within each organization can be obfuscated prior to the analysis (e.g., all IP addresses belonging to an organization can be collapsed into one address) if desired.

Given the collected netflows and the IP address ranges belonging to each member organization in the community, our alerting service analyses the data (either real time or in daily or hourly batches) and generates alerts to the system administrators. The analysis algorithm is

described in Section 4.3. A whitelist can be used to eliminate any legitimate communication destinations from consideration (e.g., search engines, CDNs, banking, on-line retailers, etc).

4.3 IMPLEMENTATION

4.3.1 ARCHITECTURE

The architecture of our system is based on StreamMine and presented in Figure 4.4. We use three different types of operators: the parse, the filter, and the graph operators. Each operator can be executed by any number of operator slices (e.g., *L*, *M*, *N*1, and *N*2 in the figure). We use the *unicast* primitive to route events between the operator slices. Additionally we enhanced the distributed graph processing engine of Chapter 3 to support multiple different kinds of graph operators in one system configuration as illustrated by *Graph 1* and *Graph 2* in the figure. Different graph operators can be used to realize different alerting conditions as we will describe below.

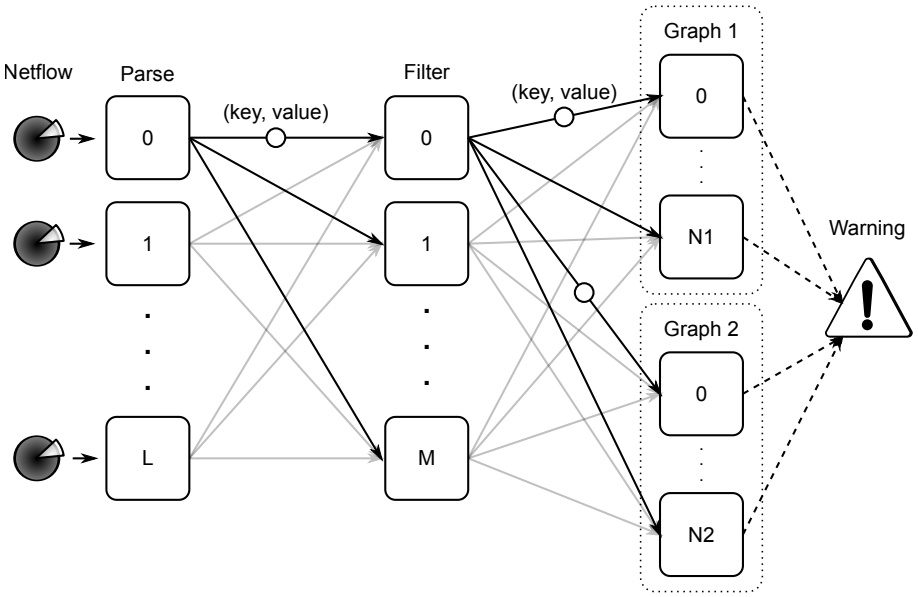


Figure 4.4: Data processing architecture

Each network flow is processed as follows. First, the netflow data is read from a local storage device (it could also be received in real time from a router). The parse operator transforms the IP addresses from their original string representation (i.e., "AAA.BBB.CCC.DDD") into an integer representing the IP address[3] and constructs an event with 5 fields: sourceIP, source-port, destinationIP, destination-port, and transferred-bytes. The parse operator sends this message to the filter operator. It uses the sourceIP field as the event's key. The filter operator either forwards (using the same key) or discards the received event. This decision is based on various factors, like used ports and source and destination IPs. If the event is forwarded, it is forwarded to one slice of every graph operator (e.g., *Graph 1* and *Graph 2*). Finally, the graph operator constructs a *community graph* for each source IP. The filtering and community graph construction are described in detail below.

[3]We will continue calling this identifier an IP address to enforce the one to one connection between these numerical IDs and the IP addresses.

4.3.2 FILTERING

The filter is an essential part of our analysis and its role is to remove irrelevant flow records and to reduce the amount of data that needs to be processed by the graph operator. For example, commonly used search, news, social media, and entertainment web sites are used so frequently that they would appear with almost every community. Furthermore, any traffic that does not involve at least one community member is not relevant for the analysis and is filtered out. Other filtering actions can be chosen based on data volume and perceived threat vectors. For example, HTTP-traffic may be filtered to reduce data volume, but at the risk of missing attacks that use HTTP (port 80).

Algorithm 4.1: Example Filter algorithm

```
input : (src-IP, src-port, dst-IP, dst-port, transferred-bytes)
output: The same as the input, if not filtered

// collapse IP addresses
src-IP, dst-IP = collapse(src-IP), collapse(dst-IP)
// filter IPs of commonly used web sites
if src-IP ∈ whitelist then
    return ∅
// filter web-accesses to community-members
if dst-IP ∈ community then
    if src-IP ∉ community then
        if src-port = 80 then
            return ∅
// only forward if one of the IPs is in the community
if dst-IP ∈ community OR src-IP ∈ community then
    return (src-IP, src-port, dst-IP, dst-port, transferred bytes)
```

Algorithm 4.1 shows an example filter operator that filters connections based on their ports, and source and destination IP addresses. First, the algorithm collapses IP addresses for an organization into one address. If, for example, an organization has the IP range from `141.1.0.0` to `141.85.255.255` and either the src-IP or dst-IP are within this range, it is set to `141.1.0.0`. We then discard every connection from IP addresses that are contained in the whitelist. Second, accesses to a community member's web-server are filtered. Finally, we only forward the event message if at least one of the connection end-points is contained in the community.

4.3.3 COMMUNITY GRAPH

We build a fixed size (K) Community of Interest (COI) graph for each IP address that is received by the graph operator. Essentially, we use a windowed top-K algorithm, as described in [CPV01]. However, there are two significant differences in our implementation compared to [CPV01]. First, our window is not based on a fixed time interval, but rather on the observed connections. This has the benefit that the COIs of IP addresses with many connections will be updated more often than of those with very few. Second, we introduce several COI views ($\{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$) that use different methods to determine the weight of a connection. We can, for example, favor connections that transfer many bytes over those that only transfer a few by using the transferred bytes as the edge weight. Obviously, in this case we would not be able to detect attacks that transfer only a small set of data if these connections are dominated by large file transfers. Therefore, we define another view that uses the port numbers involved in security incidents to weight the edges (i.e., the more reported security incidents for a port,

the larger the weight). Our system supports any number of such views running in parallel, as depicted in Figure 4.4 (with *Graph 1* implementing a different view than *Graph 2*).

Algorithm 4.2 shows how the COI is constructed in more detail. The algorithm uses two main data structures: a window that is used to collect recent data and a COI graph that stores the COI graph as seen from the beginning of the analysis run. We first add the received connection to the window. If more than 1000 connections have already been added, the window is merged with the COI graph. To this end, for each *IP* in the window, the weight of each edge is calculated, multiplied with a damping factor $1 - \theta$ and added to the weight in the COI, which is first multiplied with θ. Since $\theta = 0.85$, the influence of the new connections in the window is dampened. We also merge the port-mapping per destination-IP. It maps the source-port to the destination-port and a counter, counting how often this port-combination was used. Thereafter, the weights of all contacts in the COI that have not been observed during the current window are decayed by multiplying them with θ. To keep the COI at a maximum size of K, we remove the weakest links until the size of the COI is equal to K. Finally, the window and the counter are reset.

Algorithm 4.2: Example Community graph construction

```
   input : (src-IP, src-port, dst-IP, dst-port, transferred-bytes), s = State[src-IP], F
   output: None

   // Save connection in window
1  s.window[dst-IP].transferred_bytes += transferred-bytes
2  s.window[dst-IP].port_map[src-port][dst-port]++
3  s.counter++
   // Merge window into topK after 1000 events
4  if s.counter > 1000 then
5      foreach IP ∈ s.window do
           // θ has a value of 0.85 in our analysis.
6          s.topk[IP].weight = 1 - θ * V(s.window[IP])
7                              + θ * s.topk[IP].weight
           // Merge the window's port map with the top-k's
8          foreach {source-port, dest-port} ∈ s.window[IP].port_map do
9              s.topk[IP].port_map[source-port][dest-port] +=
                 s.window[IP].port_map[source-port][dest-port]
       // Decay weight of old connections
10     foreach IP ∉ s.window do
11         s.topk[IP].weight = θ * s.topk[IP].weight
       // Remove the weakest links
12     while size(s.topk) > K do
13         remove_weakest_link_from(s.topk)
14     s.window = ∅
15     s.counter = 0
```

4.3.4 GENERATING ALARMS

We showed above how the COI graph is constructed. Here, we provide two complementary algorithms to detect suspicious IP addresses.

The first, shown in Algorithm 4.3, is used to pre-filter all IP addresses that belong to a community. However, if a computer inside the community is compromised, we still want it to be checked further. To this end, we iterate over all connections in the IP's top-K and check each pair of ports. The pairs of ports, considered suspicious, are specified using a configuration file.

Algorithm 4.3: Suspicious IP detection (1)

```
input : IP, community, s = State[IP]
output: IP, if suspicious; ∅, if not

// blacklisted IPs are always suspicious
if IP ∈ blacklist then
    return IP

// check if IP is in the community
if IP ∈ community then
    // iterate over all of IP's connections
    foreach conn ∈ s.topk do
        // iterate over all ports of one connection
        foreach p ∈ s.topk[conn].port_map do
            // check if src_port and dst_port are suspicious
            if is_suspicious(src_port, dst_port) then
                return IP

    // no strange ports → skip
    return ∅

// not in community → check
return IP
```

We call Algorithm 4.4 for all IP addresses returned by Algorithm 4.3. It assures that (1) only those IP addresses that connected to at least `min_cnt` members of the community will be reported and (2) that the connections to the community make at least `min_part` percent of all the connections of the current IP address.

Algorithm 4.4: Suspicious IP detection (2)

```
input : IP, community, min_cnt, min_part, s = State[IP]
output: Alarm

// check if top-K connections of this IP are in the community often enough
cnt = count_community(community, s.topk)
part = cnt / size(s.topk)

if IP ∉ blacklist then
    if cnt ≤ min_cnt OR part ≤ min_part then
        return false

return true
```

The detection algorithm can be run either for all IP addresses at once or individually for each IP address. Therefore, it is possible to provide different detection latencies. For example, to detect a suspicious IP address the earliest possible, the algorithm must be executed as soon as a message is received for its source-IP's top-K. If this is not necessary, the algorithm can be run for all top-Ks in one graph process at any desired interval.

The generated alarms can be emailed to the system administrators in the affected organizations or posted on a security dashboard. The reports contain the complete top-K for each suspicious IP address, including the port mappings.

4.4 EVALUATION

4.4.1 INPUT DATA AND GENERAL SETUP

We currently run the experiment on a per-day basis. This means we fetch the netflow entries of the last 24 hours and run our analysis. We do not carry any state from one daily run to the next. In principle, we could leave the system running continuously or checkpoint the graph operator and re-initiate its state on the next day. However, we found it useful to start with a clean system every day since this makes it easier to reason about the impact of changes in the community and white lists.

Moreover, we introduced the concept of different views in June 2011. Since then, we use three different views: one that weighs the bytes transferred, another that weighs the number of connections made, and the last one that weighs the security risk for the ports used (as described in Section 4.3). For any measurements that were conducted before this date, we only used the view based on the bytes transferred.

Our input data-set is heavily sampled netflow from an ISP. In the first step, we remove all unimportant fields, leaving only the source-IP, destination-IP, source-port, destination-port, and the number of transferred bytes. This sums up to roughly 50GB of processed netflow per day.

The community lists define a community with the IP address ranges of all its members and each community is stored in a separate file (the white list is simply a "special" community). For example, if we wanted to add "TU-Dresden" to a "universities community" we would add the following line into the corresponding file:

```
141.1.0.0 - 141.85.255.255 TU.DRESDEN.DE
```

If a company or institution has more than one IP address range assigned, we can simply add each range as a separate entry. Moreover, an entry in one community is allowed to be a member in other communities as well.

4.4.2 PERFORMANCE

We implemented the parse, filter, and graph operators on top of StreamMine [Mar+11], a highly scalable stream processing system. While StreamMine supports scaling to hundreds of physical machines, a scalability and performance evaluation involving multiple machines has already been presented in the previous chapter. Therefore, we only used a single machine with 24GB of RAM and 16 processing cores for the analysis. For the top-K algorithm we used a value of 100 for K.

Figure 3.11 shows the read-throughput of the parse operator of one such run in which we processed one day of netflow data (using only one view). The measurement was taken every second throughout the whole run. The parse operator can read around 400,000 netflow entries per second with this single machine. Each entry is converted into an event and sent to the filter operator. The filter operator discards a large fraction of these messages and only

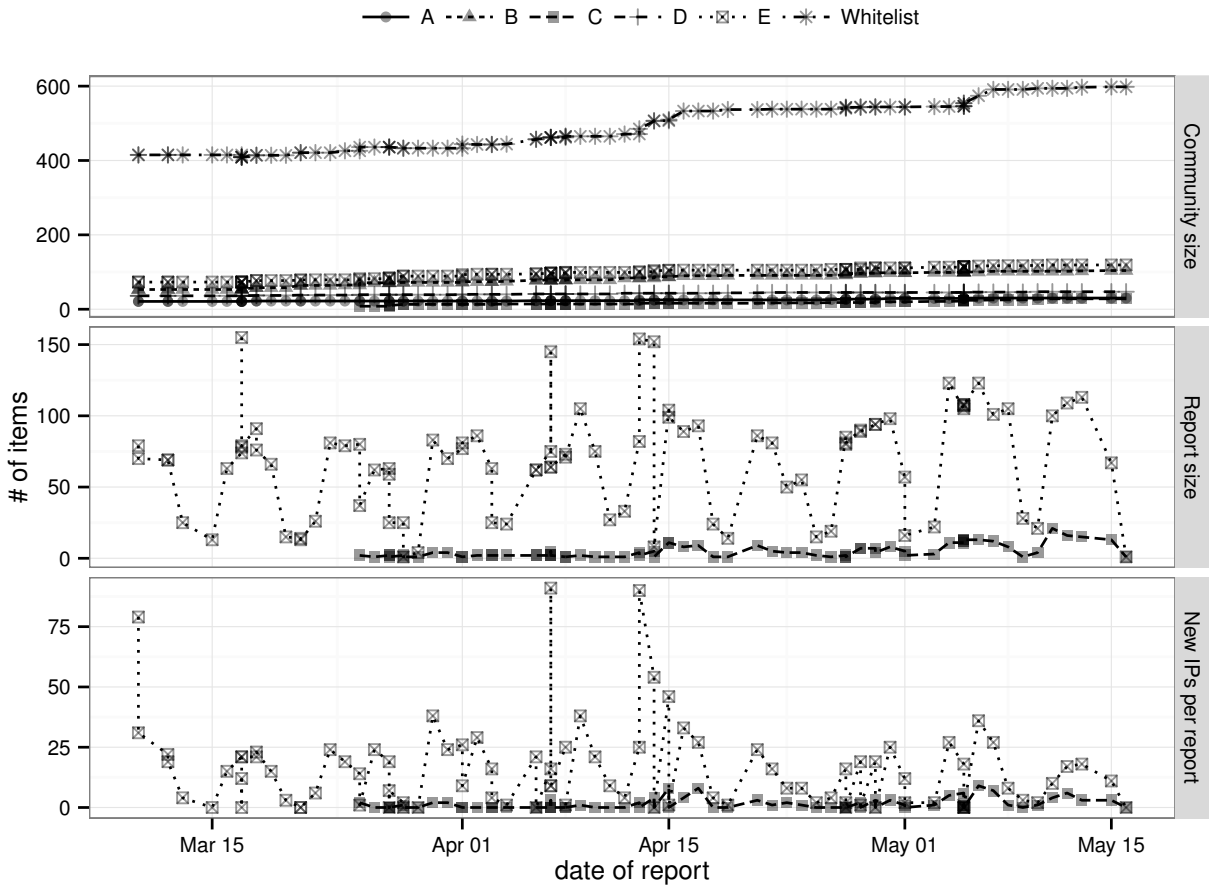


Figure 4.5: Community and alarm sizes over time

sends around one in a hundred of the incoming events to the graph operator. Naturally, the read throughput varies over time, since the amount of processing that needs to be done in the system depends heavily on the content of the input data. However, it is important to note that the mean throughput stays constant, i.e., the system performance does not decline with time as more graphs are added.

In the experiments reported in this chapter, the filter operator uses 13 of the available cores, since it has to filter the 400,000 netflow entries arriving every second. The graph operator uses only one core since the amount of data it has to process is only a fraction of the data the filter receives. Note that even if one would assign more processing resources (i.e., cores) to the graph operator, it would still be impossible to process unfiltered traffic (i.e., system without the filter operator)— the system would simply run out of memory. The parse operator uses the remaining two cores for reading the input files and parsing their contents.

To avoid queuing, StreamMine uses the TCP back-pressure mechanism on the network-connections. Hence, if an event cannot be processed by the filter operator because all its threads are already busy processing other messages, the parse operator will eventually stop sending new events (the TCP send blocks if events are not read fast enough on the other side). This will eventually lead to the parse operator not reading any new netflow entries, because all its threads are blocked trying to send messages.

Figure 4.5 shows the size of the daily alarm reports (= number of suspicious IPs communicating with the community) and community sizes (approximately the number of member organizations) over time for several months. We split the daily report sizes into two: The "Report size" shows how many alarms have been generated in total for a particular community.

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	6000	E	1433	2
X.Y.Z.W	6000	E	1433	2
X.Y.Z.W	6000	E	1433	1
X.Y.Z.W	6000	E,C	1433	1
X.Y.Z.W	6000	B	1433	1
X.Y.Z.W	6000	E,C	1433	1

Table 4.1: Anonymized report-snippet (port-mapping) from May 13th, 2011

The "New IPs per report" shows how many of those alarms are in fact previously unseen IP addresses. We expect that administrators deal with alarms immediately and thus, once an IP address is classified as an attacker it would be blocked and therefore, not re-appear the next day. The size of the alarm report is subject to a weekly pattern with larger sizes for weekdays (for alarm reports produced from Tuesday to Saturday) and smaller for weekend traffic. The community lists and the white list were updated manually on a daily basis. Given a fixed community, the community list would typically stay relatively fixed but in our case we occasionally identified additional community members. For the alarm reports, we only plot the report sizes for communities E and C. We did not generate reports for the other communities because (1) we found E and C to be the most interesting ones and (2) because of time-constraints as we need to scan the reports manually for attacks and new members of the community or white lists. It is natural that the reports, especially initially, contain a number of false positives. Some of them will be new community members that have to be added to the community list, while others are companies and organizations that can be added to the white list. The white list is used to filter out trusted traffic, i.e., from well known search engines, entertainment web sites, social media, popular CDNs, banking, government services, etc.

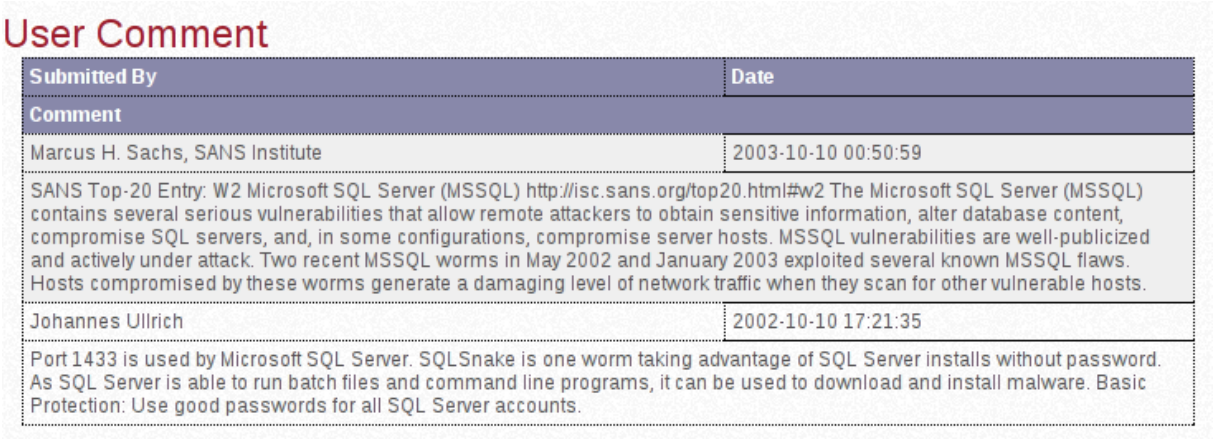
In an actual usage of the system, the system administrators analyzing the alarm reports would also add other known "good" IP addresses to the white list to prevent them from being reported daily. Lacking such domain knowledge, our experiments used the white list conservatively. The bottom plot approximates the size of the daily alarm report under real usage scenario where suspected IP addresses are processed daily and either added to the white list or the suspect communication is stopped (e.g., clean up infected machine, add firewall rules). This alarm size is approximated simply by only listing the IP addresses that have not been reported before.

4.5 CASE STUDIES

While we do not typically know the ground truth, we have observed a number of suspicious cases in our analysis. In this section, we outline some of these examples.

4.5.1 CASE 1

Table 4.1 shows an anonymized part of the report, generated for the netflow on May 13th, 2011. The report was obtained using the view based on the number of bytes transferred. It depicts the anonymized source-IP address (X.Y.Z.W) and the communities it was connected to, which ports were used (to help identify the application or service used), and a measure of the frequency of communication—the "Occurence" field indicates how often this connection was observed in the COI. In the actual report, the IP address and the exact community member are visible, of course.



User Comment

Submitted By	Date
Comment	
Marcus H. Sachs, SANS Institute	2003-10-10 00:50:59
SANS Top-20 Entry: W2 Microsoft SQL Server (MSSQL) http://isc.sans.org/top20.html#w2 The Microsoft SQL Server (MSSQL) contains several serious vulnerabilities that allow remote attackers to obtain sensitive information, alter database content, compromise SQL servers, and, in some configurations, compromise server hosts. MSSQL vulnerabilities are well-publicized and actively under attack. Two recent MSSQL worms in May 2002 and January 2003 exploited several known MSSQL flaws. Hosts compromised by these worms generate a damaging level of network traffic when they scan for other vulnerable hosts.	
Johannes Ullrich	2002-10-10 17:21:35
Port 1433 is used by Microsoft SQL Server. SQLSnake is one worm taking advantage of SQL Server installs without password. As SQL Server is able to run batch files and command line programs, it can be used to download and install malware. Basic Protection: Use good passwords for all SQL Server accounts.	

Figure 4.6: Screenshot of "`http://isc.sans.org/port.html?port=1433`" from September 8th, 2011

In the next step, we usually use the *whois* service, to determine to whom the IP address belongs. This way, we may also find new members of the community by looking up the company names, displayed in the *whois* information. For this particular example, the only information we could get, was that it belongs to an Asian ISP. Since the IP address likely does not belong to a company that the community members would typically collaborate with, we have a closer look at the ports being used. We assume that the lower port number (1433) belongs to the server and the higher port-number to the client (6000). Figures 4.6 and 4.7 show the output of the "SANS Internet Storm Center" web-site[4] related to port 1433. The web-site shows the services that usually run on these ports—in this example, "Microsoft-SQL-Server". The SANS reports indicate many potential vulnerabilities, which may be used, for example, to steal data.

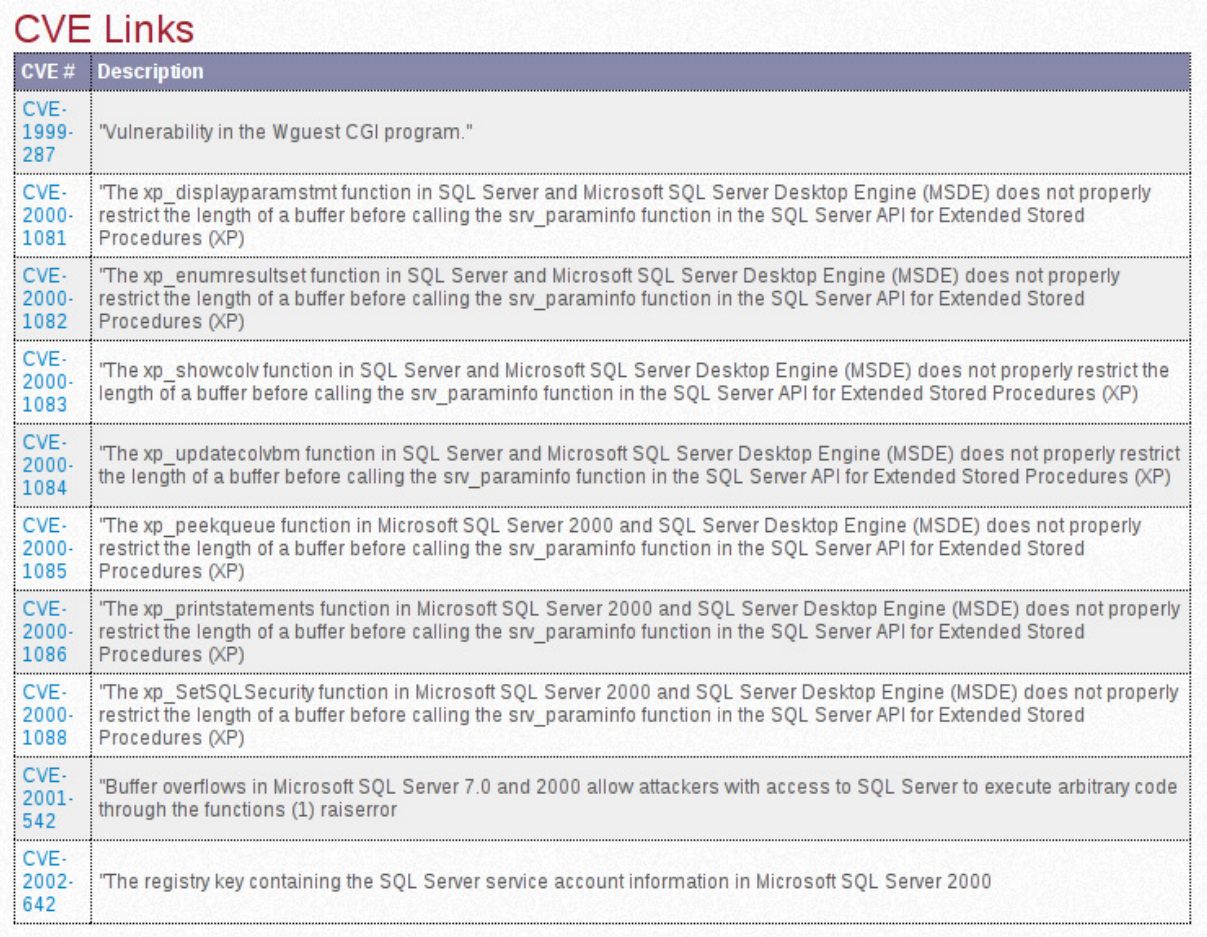
Unfortunately, this is usually everything we are able to derive from the netflow alone. While we consider this to be a potential attack, final certainty could only be provided by the system administrators of the individual companies, given they have deeper knowledge about legitimate communication connections of each organization and access to lower-level logs on the targeted machines.

4.5.2 CASE 2

Table 4.2 shows a summary of the COI of another anonymized IP address for August 8th, 2011. It shows the IP address, each community and two numbers. The report was generated using the view based on the security risk of used ports. The first number is simply a count of how many members of the current community had an entry in the COI of this IP address. The second number shows how often the IP address connected to other IP addresses that are in none of the communities. We stated in Section 4.2 that this number is a good indicator of the severity and specificity of an attack. Here, it is relatively low, which leads to the assumption that the connections were not driven by a brute-force or port-scan-like technique.

To verify this intuition, Table 4.3 shows the used ports for each community member individually. In contrast to the previous example, the source port is not constant anymore but seems to be chosen randomly. The destination port, however, is constant 445. Port 445 is usually used by "Win2k+ Server Message Block". Note that every connection only appeared once in the netflow. This either means there was in fact just one connection being used or the attempt to connect failed.

[4] `http://isc.sans.org`



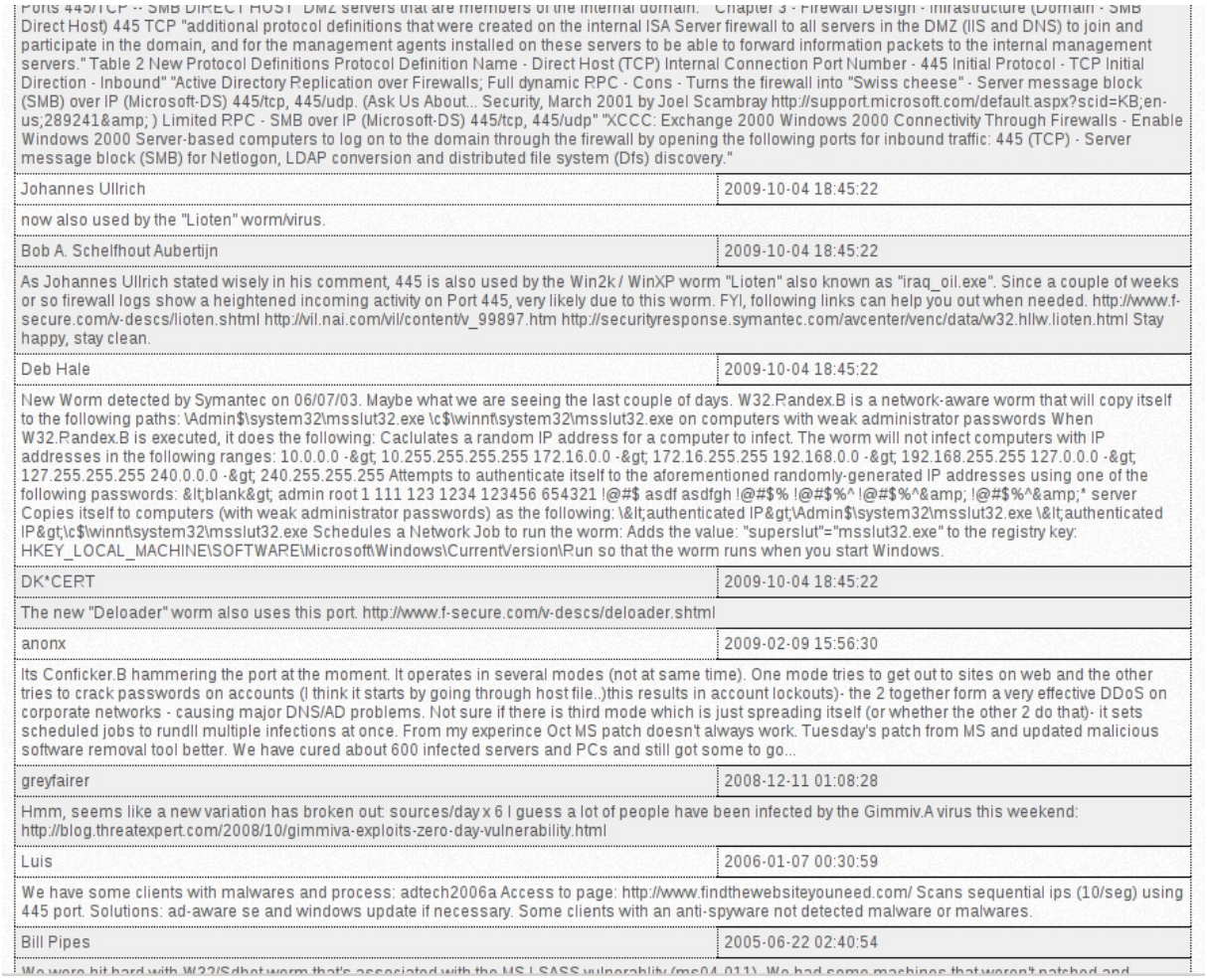
CVE Links

CVE #	Description
CVE-1999-287	"Vulnerability in the Wguest CGI program."
CVE-2000-1081	"The xp_displayparamstmt function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1082	"The xp_enumresultset function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1083	"The xp_showcolv function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1084	"The xp_updatecolvbm function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1085	"The xp_peekqueue function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1086	"The xp_printstatements function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1088	"The xp_SetSQLSecurity function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2001-542	"Buffer overflows in Microsoft SQL Server 7.0 and 2000 allow attackers with access to SQL Server to execute arbitrary code through the functions (1) raiserror
CVE-2002-642	"The registry key containing the SQL Server service account information in Microsoft SQL Server 2000

Figure 4.7: Screenshot of "`http://isc.sans.org/port.html?port=1433`" from September 8th, 2011

IP Address	Community	# in Top-K	# outside Community
X.Y.Z.W	A	0	42
X.Y.Z.W	B	0	42
X.Y.Z.W	C	1	42
X.Y.Z.W	D	0	42
X.Y.Z.W	E	1	42
X.Y.Z.W	F	6	42

Table 4.2: Anonymized report-overview-snippet from August 8th, 2011. The last two columns contain the following numbers: (1) Number of members of the current community which had an entry in the COI of the current IP address and (2) number of connections to non-community members after the first connection to a community-member.


Ports 445/TCP -- SMB DIRECT HOST "DMZ servers that are members of the internal domain." "Chapter 3 - Firewall Design - Infrastructure (Domain - SMB Direct Host) 445 TCP "additional protocol definitions that were created on the internal ISA Server firewall to all servers in the DMZ (IIS and DNS) to join and participate in the domain, and for the management agents installed on these servers to be able to forward information packets to the internal management servers." Table 2 New Protocol Definitions Protocol Definition Name - Direct Host (TCP) Internal Connection Port Number - 445 Initial Protocol - TCP Initial Direction - Inbound" "Active Directory Replication over Firewalls; Full dynamic RPC - Cons - Turns the firewall into "Swiss cheese" - Server message block (SMB) over IP (Microsoft-DS) 445/tcp, 445/udp. (Ask Us About... Security, March 2001 by Joel Scambray http://support.microsoft.com/default.aspx?scid=KB;en-us;289241&) Limited RPC - SMB over IP (Microsoft-DS) 445/tcp, 445/udp" "XCCC: Exchange 2000 Windows 2000 Connectivity Through Firewalls - Enable Windows 2000 Server-based computers to log on to the domain through the firewall by opening the following ports for inbound traffic: 445 (TCP) - Server message block (SMB) for Netlogon, LDAP conversion and distributed file system (Dfs) discovery."

Johannes Ullrich — 2009-10-04 18:45:22

now also used by the "Lioten" worm/virus.

Bob A. Schelfhout Aubertijn — 2009-10-04 18:45:22

As Johannes Ullrich stated wisely in his comment, 445 is also used by the Win2k / WinXP worm "Lioten" also known as "iraq_oil.exe". Since a couple of weeks or so firewall logs show a heightened incoming activity on Port 445, very likely due to this worm. FYI, following links can help you out when needed. http://www.f-secure.com/v-descs/lioten.shtml http://vil.nai.com/vil/content/v_99897.htm http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.lioten.html Stay happy, stay clean.

Deb Hale — 2009-10-04 18:45:22

New Worm detected by Symantec on 06/07/03. Maybe what we are seeing the last couple of days. W32.Randex.B is a network-aware worm that will copy itself to the following paths: \Admin$\system32\msslut32.exe \c$\winnt\system32\msslut32.exe on computers with weak administrator passwords When W32.Randex.B is executed, it does the following: Caclulates a random IP address for a computer to infect. The worm will not infect computers with IP addresses in the following ranges: 10.0.0.0 -> 10.255.255.255 172.16.0.0 -> 172.16.255.255 192.168.0.0 -> 192.168.255.255 127.0.0.0 -> 127.255.255.255 240.0.0.0 -> 240.255.255.255 Attempts to authenticate itself to the aforementioned randomly-generated IP addresses using one of the following passwords: <blank> admin root 1 111 123 1234 123456 654321 !@#$ asdf asdfgh !@#$% !@#$%^ !@#$%^& !@#$%^&* server Copies itself to computers (with weak administrator passwords) as the following: \<authenticated IP>\Admin$\system32\msslut32.exe \<authenticated IP>\c$\winnt\system32\msslut32.exe Schedules a Network Job to run the worm: Adds the value: "superslut"="msslut32.exe" to the registry key: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run so that the worm runs when you start Windows.

DK*CERT — 2009-10-04 18:45:22

The new "Deloader" worm also uses this port. http://www.f-secure.com/v-descs/deloader.shtml

anonx — 2009-02-09 15:56:30

Its Conficker.B hammering the port at the moment. It operates in several modes (not at same time). One mode tries to get out to sites on web and the other tries to crack passwords on accounts (I think it starts by going through host file..)this results in account lockouts)- the 2 together form a very effective DDoS on corporate networks - causing major DNS/AD problems. Not sure if there is third mode which is just spreading itself (or whether the other 2 do that)- it sets scheduled jobs to rundll multiple infections at once. From my experince Oct MS patch doesn't always work. Tuesday's patch from MS and updated malicious software removal tool better. We have cured about 600 infected servers and PCs and still got some to go...

greyfairer — 2008-12-11 01:08:28

Hmm, seems like a new variation has broken out: sources/day x 6 I guess a lot of people have been infected by the Gimmiv.A virus this weekend: http://blog.threatexpert.com/2008/10/gimmiva-exploits-zero-day-vulnerability.html

Luis — 2006-01-07 00:30:59

We have some clients with malwares and process: adtech2006a Access to page: http://www.findthewebsiteyouneed.com/ Scans sequential ips (10/seg) using 445 port. Solutions: ad-aware se and windows update if necessary. Some clients with an anti-spyware not detected malware or malwares.

Bill Pipes — 2005-06-22 02:40:54

We were hit hard with W32/Sdbot worm that's associated with the MS LSASS vulnerability (ms04-011). We had some machines that weren't patched and

Figure 4.8: Screenshot of "`http://isc.sans.org/port.html?port=445`" from September 8th, 2011

In the next step, we use again the *whois* service, to determine that the IP address belongs to an European ISP. However, it is not clear if this address belongs to a community member. An attempt to *ping* the address did not succeed. A query to "SANS Internet Storm Center" (Figure 4.8) shows a long list of reports about worms using this port with the famous "Conficker" being one of them.

As with the previous example, we cannot determine if this case is a true attack. To this end, we would need the help of the system administrators of the various community members who have access to the log-files of the corresponding machines. However, there are two interesting points concerning this IP address. First, there are only a total of 69 entries in the netflow, where this address is the source of communication. Second, all connections transfer only a very small amount of data—around 60 bytes each. Even in total, this only sums up to several kilo bytes. Therefore, this address only appears in the ports view and not in the other views that consider either the number of bytes or connections. Hence, an administrator would need to set the detection threshold very low to see an alarm concerning this address.

4.5.3 CASE 3

In contrast to the previous two cases, this case is not an attack. It occurred in all views and if one only looks at the report (an excerpt is shown in Table 4.4), it is not immediately clear what service is being used since the address seems to be using random ports on both ends of the

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	4798	F	445	1
X.Y.Z.W	1238	F	445	1
X.Y.Z.W	1256	F	445	1
X.Y.Z.W	1682	F	445	1
X.Y.Z.W	3143	C,E,F	445	1
X.Y.Z.W	4243	F	445	1

Table 4.3: Anonymized report-snippet from August 8th, 2011

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	13397	B	38426	1
X.Y.Z.W	41748	F	41387	1
X.Y.Z.W	49534	C	23068	1
X.Y.Z.W	16249	C	22654	1
X.Y.Z.W	29167	C	43183	2
X.Y.Z.W	20	F	7205	4
...	...	...	...	...

Table 4.4: Anonymized report-snippet from August 8th, 2011

communication. The query to *whois* does also not reveal any useful information, except that the address belongs to a US ISP.

However, looking at the connections with IP addresses outside of the communities provides a hint that this is not targeted against any of our specified communities as shown in Table 4.5. Moreover, the use of port 20 (the last line in Table 4.4) gives a hint that at least some part of the communication involved anonymous ftp, which uses port 20 to initiate the connection but uses random ports thereafter. Finally, using an ftp-client (i.e., a web-browser) revealed indeed that this is simply an ftp server hosting software updates. As a result of this analysis, we added the address to the white list.

IP Address	Community	# in Top-K	# outside Community
X.Y.Z.W	A	0	14250
X.Y.Z.W	B	2	14250
X.Y.Z.W	C	1	14250
X.Y.Z.W	D	0	14250
X.Y.Z.W	E	0	14250
X.Y.Z.W	F	6	14250

Table 4.5: Anonymized report-overview-snippet from August 8th, 2011. The last two columns contain the following numbers: (1) Number of members of the current community which had an entry in the COI of the current IP address and (2) number of connections to non-community members after the first connection to a community member.

netflow recorded on: 2012-04-06

src-ip	name	ipvoid.com	rcc	community-count	topk-size	other-count
98.	OrgName:	CLEAN	80	4	5	4
65.	CustName:	SUSPICIOUS	100	4	4	4
8.	OrgName:	CLEAN	100	7	7	5
208	OrgName:	CLEAN	100	4	4	5
216.	CustName:	CLEAN	80	4	5	7
206.	OrgName:	CLEAN	100	4	4	11
8.	OrgName:	CLEAN	100	4	4	16
64.	OrgName:	CLEAN	100	4	4	22

Figure 4.9: Ranking of alarms

4.5.4 BUILDING COMMUNITIES

In real use of the system, the community members might be known a priori and even stay relatively fixed. However, in our case we built the community lists incrementally by identified new community members based on the COIs generated. Specifically, we assumed that members of a community exchange information with one another and often the data exchange is encrypted. Therefore, we focused on new IP addresses that used the https-port (443) for communication. However, a certain minimal set of known members is needed before reports can be generated. This set should be as large as possible for two reasons. First, the likelihood that an unknown address that belongs to the community (and thus, should be added) connects to one or more entries of a large set of members is higher than if the set contains only very few entries. Second, if the set is large, one can set the reporting threshold higher and reduce the amount of noise.

Building a community this way is a task that lasts for weeks, depending on how much communication is observed between the individual members and how large the community is initially. We start by adding the new community to the list of communities. With every subsequent report, we scan for new members and add them to the corresponding lists. This way, the community grows every day, and with it the likelihood of finding any missing members. The community stabilizes eventually with fewer and fewer new members per day.

4.6 EXTENDING THE ANALYSIS

In this section we augment our analysis with further input data. We correlate the additional information with our alarms in order to rank them by severity. Figure 4.9 shows a sample report of our extended analysis. It is essentially an extension of the reports we used in the previous section (e.g. Table 4.2). The first column shows the IP address which triggered the alert. The second column shows the organization which registered this IP address. To this end, we query DNS databases for this information for every suspected IP address. We assume that the name of the registering organization is of some help for an administrator to assess the severity of an incident. The third column shows whether this IP address is considered suspicious or clean by IP Void. IP Void[5] is an online blacklist which facilitates the detection of IP addresses involved in malware incidents and spamming activities. To this end IP Void includes several external sources, such as honeypots, multiple DNS blacklists, IP reputation engines, and so on. IP

[5]Visit `ipvoid.com` for further information.

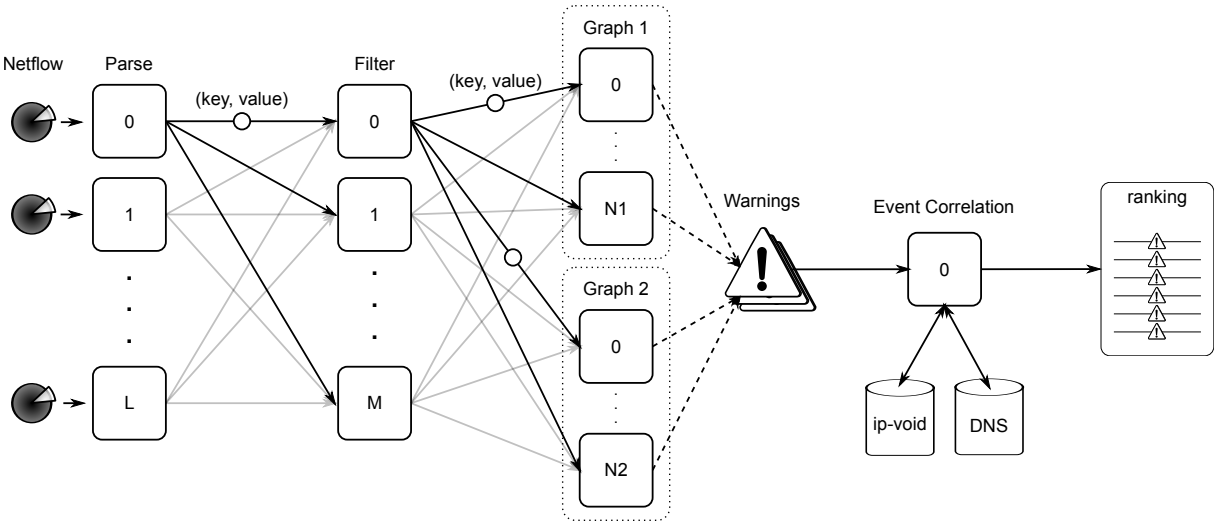


Figure 4.10: Extended Architecture for Alarm Ranking

addresses can be queried to obtain their status which can either be "clean" or "suspicious". Here, "clean" means that no report has been found in the databases. Contrary, "suspicious" means that this IP address was involved in malicious activity in the past. The fourth column shows the *Relative Community Count (rcc)*. It gives a direct indication of how targeted the suspect's communication was. The rcc is simply the percentage of members of the current community in the COI of the suspect. It is computed with # in Top-K ∗ 100/topk-size which corresponds to the fifth and sixth column, respectively. Finally, the incidents are sorted by the seventh and last column, *other-count* which corresponds to the *# outside Community* in the previous Section.

Figure 4.10 shows the processing architecture of the extended analysis. For each generated alarm, a separate module queries DNS databases, IP Void, and calculates other metric, like rcc. From that, it generates a list of alarms with the appropriate information. This list can then be distributed to the company administrators for further investigation.

4.7 RELATED WORK

This section presents related work in the field of intrusion detection.

4.7.1 NETFLOW

A number of tools and techniques have been developed to process and visualize netflow data(see [Sol09] for a survey). Netflow processing tools include OSU flow-tools [Rom00], SiLK [Gat+04], and Nfdump[6]. In addition to command line tools, numerous graphical user interfaces exist to visualize and query network activity, including NTOP[7], Nfsen [Haa05], NfSight [Ber+10], VisFlowConnect [Yur06], FlowScan [Plo00], NetPY [Cir+09], FloVis [Tay+09], VIAssist [DAm+07], and NFlowVis [Fis+08]. While visualization tools allow the users to view the netflow data from different perspectives to locate suspicious activity, our approach analyzes the data and produces small number of meaningful alarms each day. Also, our focus on communities allows us to detect attacks and suspicious behavior that is focused on a potentially small community, but would not show significantly on a global scale.

[6] http://nfdump.sourceforge.net
[7] http://www.ntop.org

Detection of similar communication behavior in multiple hosts has been used previously to raise suspicion that hosts with the correlated behavior may be members of the same botnet. For example, [ZHS10] uses netflow data to identify sets of suspicious hosts and then uses host level information (collected on each host by a local monitor) to confirm or reject the suspicions. However, detection of botnets is simplified by the fact that the bots typically act in unison (e.g., start spamming or DDoS attack against a target at the same time). Indeed, much of the work in this area (e.g., BotMiner [Gu+08]) specifically build detection mechanisms based on the assumptions of the communication behavior required for a botnet. Furthermore, to our knowledge, prior work is limited to detecting similar behavior within one organization.

4.7.2 SIGNATURE-BASED IDS

Snort [Roe99] is one of the most prominent network-based intrusion detection systems. It applies a set of pre-defined rules against the packet content. These rules are expressed as conjunctions of predicates which may maintain state throughout flows. Predicates can be simple string matches or even regular expressions. In order to cope with the high bandwidth of modern network devices, Wun et al. [WCR09] proposed to run Snort an multiple cores in parallel. To this end, each core runs a Snort instance. In order to avoid state-sharing between those instances, Wun et al. employ flow pinning which makes sure that all packets of a given flow are processed exclusively on one core. Dharmapurika et al. [Dha+03] improve the packet matching speed of Snort by pre-filtering packets using bloom filters which they implement on FPGAs.

The biggest shortcoming of Snort is that it uses exploit signatures, i.e. a signature of an actual attack. This means that a signature can only be created once an attack has been witnessed. This puts security experts in the bizarre situation that they often already know about vulnerabilities but cannot create exploit signatures because for that, they have to wait until the first attackers actually exploit the vulnerability. And more so, often a single vulnerability can be exploited in several different ways rendering Snort especially vulnerable to polymorphic attacks. Therefore, Shield [Wan+04a] and NetShield [Li+10] went into a different direction. The signatures in both systems do not describe exploits but rather the vulnerabilities themselves. In contrast to snort, both systems have to be run on the protocol layer, i.e. both systems need to have access to the actual application messages and not the mere TCP/UDP packets. Unfortunately, Shield is much slower than Snort because its filters are much more complex. NetShield circumvents that shortcoming by combining filters and performing common matching operations only once. Moreover, NetShield translates filters into C++ directly to reduce matching times. Vigilante [Cos+05] is a system which instruments applications with any sort of detection engine. The authors present two engines, one based on stack and heap protection, the other based on dynamic dataflow analysis. If the detection engine detects a violation, it generates an alert which can be verified by other hosts running the same software. Since the alert contains a description of the exploited vulnerability, future attacks can be prevented. For example, the described vulnerability may be used to generate a filter for Shield or NetShield.

4.7.3 COMMUNITY-BASED IDS

The concept of using a community to help detect security events has been used in the past. For example, the Ensemble [Qia+09] system detects applications that have been hijacked by using the idea of a trusted community of users contributing system-call level local profiles of an application to a common merging engine. The merging engine generates a global profile that can be used to detect or prevent anomalies in application behavior at each end-host in real time. A similar concept of collaborative learning for security [Per+09] is applied to automatically generate a patch to the problematic software without affecting application functionality.

PeerPressure [Wan+04b] automatically detects and troubleshoots misconfigurations by assuming that most users in the community have the correct configuration. Cooperative Bug Isolation [Lib07] leverages the community to do statistical debugging based on the feedback data automatically generated by community users. Vigilante [Cos+05] apply the community concept for containment of Internet worms by community members running detection engines on their machines, where the detection engines distribute attack signatures to other community members when a machine is infected.

Collaborative intrusion detection systems (CIDS) [ZLK10; Ani+11; BDQ13] try to detect attackers who may pass "below the radar" of local intrusion detection. This is often the case as attackers manage to either obfuscate their actions or imitate legitimate users. By aggregating statistics across domains of related entities, CIDS increase the detection probability. However, so far none of these systems considers the community structure of the targeted systems. Rather a "fake community" is built in an ad hoc fashion by whoever collaborates. Consequently, this "fake community" does not necessarily represent an actual industry, for example. Moreover, [Ani+11; BDQ13] do only consider port scanning. However, port-scanning is rarely necessary in APT attacks since the adversaries make use of common ports, such as ports 22, 80, or 443. Furthermore, they focus on attackers who coordinate their actions with other attackers. Again, APT attacks are executed by rather small groups and, hence, there is neither evidence nor the need for such coordination.

4.7.4 INTRUSION RESILIENCE

PeerReview [HKD07] logs incoming and outgoing messages of each node in a distributed system. It is assumed that a protocol which runs on one node can be re-executed on any other node. Such re-executions are preformed periodically. To this end, the previously logged input of the original node is replayed to another node. The latter then verifies that it would produce the same output messages as the original node if the same protocol is applied. Byzantine faults (which include malicious attackers) can thus be detected by a correct node and will eventually be linked to the faulty node. While PeerReview can only detect the intrusion after the fact, Nysiad [Ho+08] aims at preventing the spread of an intrusion (or any other byzantine fault) altogether. To this end, it assigns each node in a system a set of guard nodes. It assumes that each node in the system executes a deterministic state machine which is replicated onto the guard nodes in order to validate messages sent by the original node. The guards employ a gossip protocol to prevent the original node from sending different messages to different hosts. Levin et al. [Lev+09] show that using a trusted hardware component, the message overhead of both, PeerReview and Nysiad can be reduced considerably. The work by Haeberlen et al. [Hae+10] employs a similar idea to that of PeerReview for virtual machines. The authors propose to use the virtual machine monitor to log input and output messages of a virtual machine. By applying their approach to the virtual machine monitor directly, they are able to observe and log non-deterministic events. These logs can then be checked by a trusted virtual machine. If the trusted VM produces the same output, given the original inputs and non-deterministic events, the original execution is proven correct.

4.7.5 INTRUSION PREDICTION

Using global blacklists, such as DShield [8] is a common practice in intrusion prevention. Such blacklists are created by a community of organisations which contribute their firewall logs. With that, DShield is able to create "global worst offender lists" containing those IP addresses from which most attacks originated. Zhang et al. [ZPU08] went one step further. They noted that a "global worst offender" might merely select targets at random without any specific

[8] www.dshield.org

intention. However, a greater threat is posed by attackers whose target selection follows a specific intention, such as it is the case with advanced persistent threats. Their question was: Given past attacks, can the attackers be ranked by how relevant they will be for a given future target? To find an answer they created weighted link between DShield contributors. The weight determines the attacker overlap between both contributors. If it is high, a large fraction of the attacks against those contributors can be attributed to the same attackers. For contributors who were never attacked by the same attacker there is no link. Now given an attacker attacks contributor A, how relevant is this attack for the other contributors connected directly or indirectly to A? It turned out that the answer was already given 10 years ago: The personalized pagerank algorithm [Pag+99], determines the relevance of any web-page, given a random surfer browses through the web, always starting at a predefined start location (i.e. the homepage). Zhang et al. [ZPU08] redefined this notion: Given an attacker always starts at contributor A and moves along the links of the DShield contributors what is his next most likely target. They showed that this technique improves attack prediction by 20% to 30%.

4.8 SUMMARY

In this chapter, we have presented a community-based analysis and alerting technique for detecting small-footprint attacks targeting communities of interest for attackers such as financial institutions, e-commerce web site, or the electricity generation and distribution infrastructure. By comparing communication behavior across the member organizations in the community, it is possible to detect suspect behavior that may fall below detection thresholds at individual member organizations. A white list can be used to avoid repeating false positives. We have implemented the analysis algorithm in a scaleable distributed architecture that can process large volumes of netflow data efficiently.

*8††

5 A MASSIVELY PARALLEL ARCHITECTURE FOR HIGH-PERFORMANCE CONTENT-BASED PUBLISH / SUBSCRIBE[1]

[1]The original version of this chapter appeared at DEBS '13 [Bar+13].

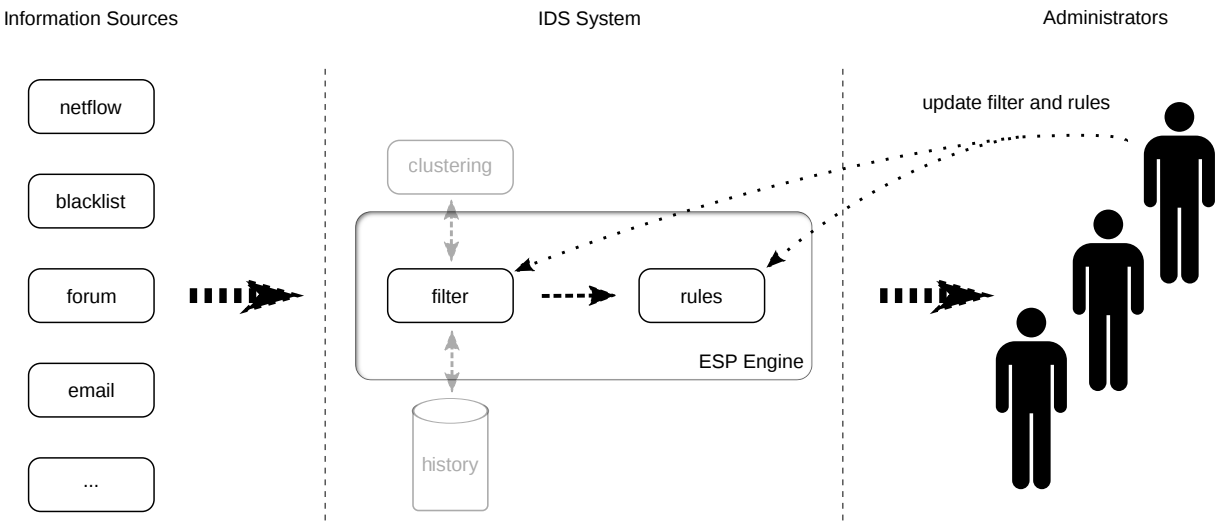


Figure 5.1: This chapter is concerned with a general and adaptable implementation of the filter and rule component for a community-based IDS

So far we have built a high throughput, low latency, graph- and community-based IDS. However, there are four issues that the current architecture does not solve.

1. The filtering cannot be adapted during runtime. This is a major burden as administrators who investigate the alarms and find them to be false positives may want to add some IP addresses to the whitelist. Contrary, if an investigation proves an actual attack, administrators may want to add the offending IP address(es) to the blacklist.

2. The rules are static. This means that we cannot add or change the definition for when an IP address is considered suspicious during runtime. Although we already argued that attack-polymorphism is mostly found in the applied malware, there is no reason to assume that attackers cannot change their communication patterns as well. Moreover, as soon as we depart from a strictly meta-data based rule-engine this issue automatically becomes more prominent. Therefore, Security Challenge 1: Polymorphism will eventually require the rules to be adapted. To give an example, Algorithm 4.1 discards IP addresses which access community members over port 80. This was necessary to keep the number of alarms manageable. However, web servers are often targeted by attackers.

3. We are limited to graph-based rules. However, especially for inputs such as emails or forums, it would be desirable to additionally be able to define rules on the content of that input. What we need is an architecture which allows us to execute any kind of rule-engine as dictated by Data Challenge 2: Diversity of datatypes. This will also help with Security Challenge 4: Stealthy attacks, as it is easier to pass "below the radar" of an IDS relying on a single input dataset than to pass "below the radar" of an IDS that allows to correlate many different inputs.

4. We cannot route alarms. If we consider the addition of many different rules, tackling diverse aspects of the various inputs, we must also think about how the alarms of these rules are routed: Some should be correlated and then forwarded, some might only be of interest to some administrators. For example, consider the attacks against web servers, mentioned above. It would be beneficial if these alarms could be gathered within a separate stream of alarms in order to apply some sort of post-processing. This would

allow to correlate them with other events in order to separate the scripted attacks against the actual APT attacks.

If we choose a high-enough abstraction, it turns out that this is a group communication problem: We want to send a single message, be it a netflow entry or an email, to n administrators (n can be 0). Let us look at some examples to illustrate that. For many of the netflow entries or emails the IDS receives, no alarm will be generated, hence $n = 0$. However, for others, n might be as high as the number of administrators from all the domains the IDS should protect. Even more interestingly, while some messages generate no alarms, they might do so after they occurred a given number of times. For instance, the intrusion detection algorithm in the previous chapter only generates an alarm after an IP address has communicated with community members sufficiently often.

Publish/Subscribe is a group-communication paradigm which decouples sender and receiver of messages in space and time by assuming the role of the message broker, routing the messages between sender and receiver. Content-based Publish/Subscribe allows to route messages based on their content. In this chapter we present a new Publish/Subscribe architecture which departs from the peer-to-peer broker overlay. Instead, we base it on StreamMine to take advantage of its scalability and adaptability. Here, publications are the events, coming from the various input sources (IP flow updates, emails, forum entries, and so on). Subscriptions are rules and filters at the same time since our Publish/Subscribe permits to run arbitrarily complex matching engines.

This has several advantages. First, it allows us to define complex, content-based rules for any of our inputs. For example, our graph-based rule engine can be easily embedded into the Publish/Subscribe engine as yet another matching engine. Second, it is possible to post-process (e.g. correlate) alarms from a specific set of rules: An alarm, generated by a rule is only sent to the subscribers of that rule — this could be an administrator but it could also be a correlation engine, such as our ranking component from Section 4.6 of the previous chapter.

5.1 INTRODUCTION

Content-based publish/subscribe (pub/sub) [Eug+03] is a strong contender for offering an efficient, yet *natural* communication paradigm to developers of large-scale applications. It supports decoupled interactions between the producers (*publishers*) and the consumers (*subscribers*) of information by the means of messages (*publications*). Decoupling occurs both in terms of space and time: publishers and subscribers do not need to know the existence or identity of one another, and no particular synchronization between them is necessary. They only communicate indirectly through a *pub/sub system*. It is the responsibility of this system to *route* publications from the publishers to interested subscribers. Routing is based on *subscriptions* registered by the subscribers to express their interest in specific content. The operation of *matching* the content of the publications against the subscriptions stored in the system is called *content filtering*.

A typical use of pub/sub systems is for composing a collection of independent applications running on different administrative domains or geographical locations. Communication between these applications takes place via a common pub/sub service running on a set of *dedicated servers*, typically set up in a *public cloud* or a *cluster equipped with a public address*, interconnected through a local area network and exposing access points to client applications.

The decoupled and data-centric nature of the pub/sub communication model allows for seamless integration and evolution of large-scale applications. A typical example is *QoS Monitoring as a Service* [Rom+11], where an application running on a private cloud is monitored and key performance indicators (KPIs) are generated as publications. These KPIs are propagated to a third-party monitoring service, based on subscriptions generated from a service

level agreement (SLA) in order to detect violations of this SLA. Communication takes place via a pub/sub service deployed on a public cloud accessible by both parties. Other applications include e-Health systems [IRC10b] that bridge several medical and healthcare institutions sharing information about patients cases, or the canonical example of stock trading [Gup+04]. We note that for all these applications, the use of a third-party infrastructure for communication may raise concerns about privacy and data security: publications and subscriptions represent sensitive data that should not be leaked to a third party. As a result, *encrypted content filtering schemes* have gained interest in the recent years [Bar+12; CGB10; IRC10a; IRC10b; RR06] as they support filtering of encrypted publications against encrypted subscriptions without needing decryption. Such approaches suffer, however, from a high computational cost and disallow some optimizations, in particular those based on containment relationships between subscriptions (i.e., the fact that a subscription will match a subset of the publications matching another subscription) or on the aggregation of a set of subscriptions into a single one.

Objectives. We argue that the key properties of a pub/sub system running on a public cloud or cluster and supporting large-scale application composition should be as follows.

(1) High throughput and low, predictable delays. The raw performance of the pub/sub service deployed on a public cloud or cluster must be sufficient to support demanding applications, such as high-frequency trading or network monitoring. This requires exploiting parallel processing of incoming subscriptions and publications as much as possible. Since the filtering operation itself is costly, the design must avoid filtering an incoming publication against a given subscription multiple times, which typically happens in overlay brokers systems. Furthermore, delays between the generation of a publication and its dispatching to interested subscribers must remain of the same order as the delay a coupled communication between the producer and consumer of information would take. As a corollary, there should not be significant deviation in the notification time for all subscribers interested in a given publication.

(2) Scalability. The ability to support increasing numbers of publishers/publications, subscribers/subscriptions, and notifications, as well as more computationally intensive filtering schemes, requires several levels of scalability. *Vertical scalability* is required to take advantage of additional resources available on a given node, notably multi- and many-core architectures that can process the pub/sub traffic in parallel. *Horizontal scalability* allows supporting a higher load by adding more nodes to the cluster. Ideally, a linear increase in the number of nodes should result in a linear increase in maximum supported throughput.

(3) Filtering scheme agnosticism. The design and architecture of distributed pub/sub systems should not be dependent on a particular filtering scheme and in particular on the semantics and representations of publications and subscriptions. Most existing distributed pub/sub systems [CRW01; CJ11; Jac+09; CF08; YMJ11] support filtering schemes based on conjunctive predicates ($<$, $\leq$, $=$, ...) over discrete attribute values (integers, strings, ...), and their designs are closely tied to the nature of this particular representation. This applies, for instance, to the construction and maintenance of routing tables between brokers that drive the flow of publications. To minimize inter-broker traffic, these systems typically rely on the ability to determine containment relationships between subscriptions and/or to construct aggregated subscriptions. Yet, such features are not available with all content-based filtering schemes, notably with encrypted approaches [Bar+12; CGB10; IRC10a; IRC10b; RR06]. As a matter of fact, there exist no fundamental reasons why content-based routing should be restricted to attribute- and predicate-based filtering: a pub/sub service should be able to integrate virtually any filtering scheme operating on the content of exchanged data using stored filters, as required by the application. Examples include not only encrypted filtering for privacy preservation, but also statistical methods such as Bayesian filtering [Sah+98], template matching for digital images (for instance, for face recognition) [Bru09], or even complex graph-based filters as presented in the previous two sections.

The architecture of the pub/sub system should be independent of the nature of the filtering scheme, while still allowing for specific optimizations at the level of a single node.

Contributions. In this chapter, we revisit the design of a distributed content-based pub/sub engine for supporting high throughput, low latency, and horizontal and vertical scalability. We propose a novel approach based on a tiered architecture and inspired by dataflow programming techniques, which exploits parallelism in ways similar to MapReduce [DG04] and stream processing engines inspired from it [Neu+10; Fou; Bri+11; Bac+12]. A set of independent operators, each spanning an arbitrary number of servers and taking advantage of multiple cores on individual servers, implement the three fundamental operations of content-based pub/sub: *subscription partitioning*, *publication filtering*, and *publication dispatching*. Interactions with the pub/sub system are managed by a set of independent *data converters and connection points* (DCCP) that maintain persistent connections with clients (publishers and subscribers).

We implement our approach in StreamHub, a pub/sub engine designed for operating on a public cluster or cloud. StreamHub leverages the runtime support of an existing stream processing engine such as S4 [Neu+10], Storm [Fou], or StreamMine [Bri+11]. We use the latter engine in our prototype implementation.

Our evaluation on a cluster with up to 384 cores on 48 physical machines indicates that StreamHub is able to sustain high-throughput workloads: up to 150 K subscriptions registered per second; and up to almost 2 K publications filtered per second with a population of 100 K stored subscriptions, resulting in an output flow of nearly 400 K notifications per second to interested subscribers.

We note that our contribution is not on the actual filtering scheme itself, which is supported by an independent library that can be chosen arbitrarily as long as it implements a simple and schema-oblivious API. We demonstrate the performance of StreamHub using the well-established counting algorithm of SIENA [CW03], and we leave the integration and comparison of other filtering libraries, such as those providing privacy-preserving encrypted matching [Bar+12; CGB10; IRC10a; IRC10b; RR06] and in particular security-related filters, for future work. Similarly, while StreamHub is designed with elastic scalability in mind (i.e., the ability to dynamically adapt the number of servers associated with each operator according to the experienced workload), we leave the implementation of elastic server provisioning for future work and concentrate on the performance and scalability of the architecture with various static configurations.

Outline. The remainder of this chapter is organized as follows. We present and motivate our proposed architecture in Section 5.2. We describe the implementation of StreamHub in Section 5.3, as well as the libraries used in our evaluation for filtering publications and clustering subscriptions. We evaluate our approach and compare it to a broker-based pub/sub system in Section 5.4 and survey previous work on distributed pub/sub systems in Section 5.5, before concluding in Section 5.6.

5.2 ARCHITECTURE

Our design choices aim at maximizing pipeline, task, and data parallelism in order to support high-throughput and scalable pub/sub. It is based on StreamMine. We use a set of three operators. Each implements a different aspect of the pub/sub service: subscription *partitioning*, publication *filtering*, and publication *dispatching*. Thanks to the scalability properties of operators, one can easily adapt the number of physical machines and cores to the load experienced by each of these three operations. This load varies with the nature of the workload, such as the number, complexity, or selectivity of subscriptions. The load also varies with the nature of the filtering schemes. For instance, encrypted filtering requires more processing power than non-encrypted filtering. To sustain the same publication throughput, one should allocate

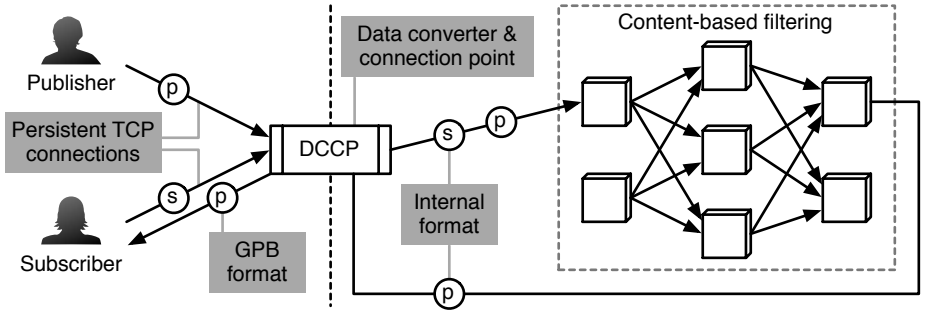


Figure 5.2: User view of StreamHub.

more slices (servers) to the publication *encrypted filtering* operator.

We present in this section our operators and support mechanisms. We start by describing the endpoints used by external clients to access StreamHub. Afterwards, we present the operators that support content-based filtering, as well as the partition of the load onto different slices at each operator. The filtering operation itself is delegated to one or more *filtering libraries*. StreamHub also provides optional support for *clustering libraries*, which can partition the state of subscriptions in elaborate ways and speed up the filtering operation. As these libraries are pluggable components whose algorithms do not represent novel contributions of this chapter, we describe them in Section 5.3.

5.2.1 CONNECTION TO AND FROM CLIENTS

The pub/sub operators are typically deployed on a cluster or a cloud, i.e., a set of machines with limited hardware heterogeneity. In our implementation, StreamHub, operators are implemented using the same language (C++). As a result, the internal communication and serialization formats between the elements forming the architecture can be selected based on performance criteria. Our implementation uses the efficient binary format provided by Boost libraries[1] for internal propagation of events. In contrast, clients may execute on different platforms and use a variety of languages. The choice of the external format is thus driven by its hardware and language independence. Our implementation uses Google Protocol Buffers (GPB),[2] which provide efficient serialization primitives for subscriptions, unsubscriptions, and publications while hiding language and platform heterogeneity.

Publishers and subscribers need a persistent and public connection point to the cluster or cloud supporting the pub/sub service. Connecting to any of the nodes supporting the pub/sub service is impractical in clouds (due to VM migrations) and often impossible in clusters (as most nodes do not have a public IP address). Our design features components external to the operators implementing the pub/sub service, that act as such persistent connection points. These are also in charge of translating between the external and internal representation format, and henceforth named *Data-Converter & Connection-Points* or DCCPs. Figure 5.2 presents a user-centric view of the system. Clients connect to a DCCP via a *persistent* TCP connection to enable low end-to-end delay for communication with the pub/sub service and, more importantly, to support asynchronous notifications of matching publications, as clients may not be directly reachable (for instance, they may be located behind a NAT or firewall). We note that this practical impossibility to reach clients directly limits the applicability of rewiring schemes for solutions based on brokers overlays [CJ10a; Li+12; CJ10b; CJ11].

[1] http://www.boost.org/
[2] http://code.google.com/apis/protocolbuffers/

Operator	Role	Description
AP Access Point	Subscription *partitioning*	– Receives subscription events and dispatches each to a single slice of an M operator. Optionally applies subscription clustering using a libcluster library. – Receives publications, forwards them to all slices of an M operator.
M Matching	Publication *filtering*	– Receives subscriptions and forwards them to the libfilter library. The libfilter library stores the subscription and corresponding subscriber identifier in the operator slice state. – Receives publication events and forwards them to the libfilter library, which returns a set of matching subscriber identifiers. The M operator slice forwards each publication and list of matching subscriber identifiers to the EP operator by unicast, using the publication identifier as key.
EP Exit Point	Publication *dispatching*	– Receives a publication and list of matching subscriber identifiers. When all lists are received, prepares the notifications, splits the list of matching identifiers, and dispatches them to corresponding DCCPs.

Table 5.1: Operators supporting scalable CBR.

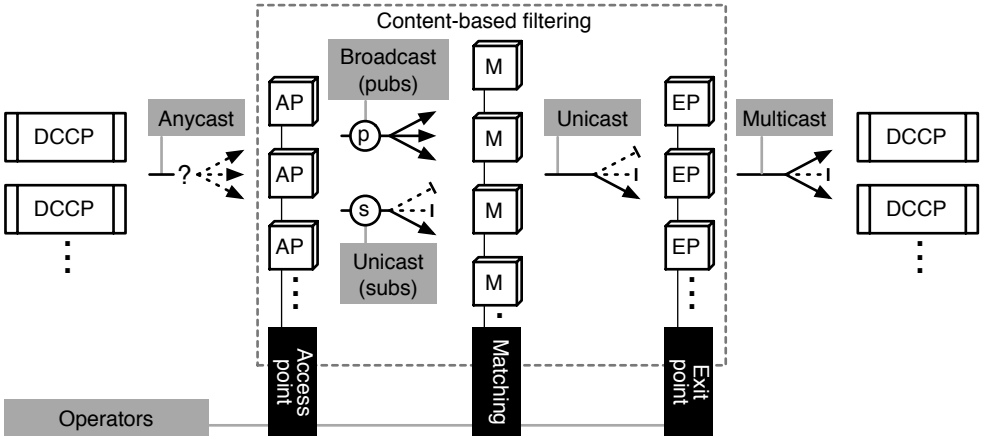


Figure 5.3: StreamHub processing operators (libraries and states not shown for clarity).

Several DCCPs can be used for the same StreamHub deployment, e.g., when the number of opened connections or the necessary bandwidth becomes too high for a single machine, when the cost of conversion creates a bottleneck, or on the same host when several network adapters are available.

5.2.2 CONTENT-BASED ROUTING OPERATORS

In this section, we present the three operators that form the *core engine* of our scalable pub/sub architecture. These three operators are organized as a pipeline. They are listed in Table 5.1 and illustrated by Figure 5.3, together with the communication primitives used for propagating events between them. A detailed example of the path taken by subscriptions and publications within the StreamHub engine is shown by Figure 5.4.

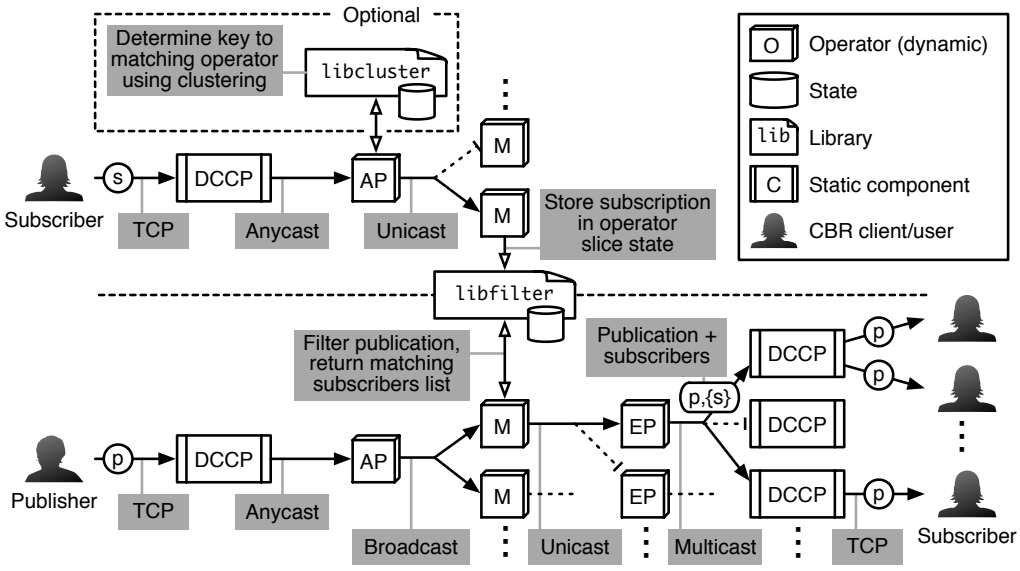


Figure 5.4: Path taken by subscriptions (top) and publications (bottom) in the StreamHub architecture.

ACCESS POINT OPERATOR

The *Access Point* (AP) operator plays the role of the input operator. It receives events from any of the DCCPs. The selection of an AP operator slice by a DCCP is done at random to guarantee good load balancing properties. The role of the AP operator is to *partition* incoming subscriptions among all slices of the *Matching* (M) operator as follows.

Each incoming event has a *key*, which is a data structure that indicates the type of the client request (i.e., a new subscription, an unsubscription, or a publication). Subscriptions are not stored by the AP operator slices but are instead forwarded to the M operator that implements the filtering operation as we describe next. Our architecture can simultaneously support different filtering schemes (such as flat vs. structured data, encrypted publications and/or subscriptions, declarative vs. executable filters). Each filtering scheme is supported by a separate M operator. The choice of the destination operator depends on the *filtering scheme identifier* embedded in subscriptions. The same applies to publications.

Only one of the slices of the M operator holds any given subscription.[3] AP slices hence use unicast communication to select the appropriate M operator slice that will be responsible for an incoming subscription. The default mechanism relies on unicast and routes the subscription based on the hashing of the *subscription identifier* specified in event keys. We note that this selection mechanism is *stateless* and *reproducible*: an unsubscription will be routed from the AP to the M operator using unicast and will arrive at the M operator slice that actually holds the subscription.

Alternatively to this default mechanism, the user can decide to defer selection to a library, denoted by libcluster in Figure 5.4, which supports more complex forms of subscription clustering (see Section 5.3.1). A clustering algorithm can maintain a slice-supported state, which allows for deciding on subscription placement based on the content of that subscription. This incurs additional costs at the AP operator level, in return for a better performance at the M operator level. In case the selection mechanism is not deterministic and reproducible, unsubscriptions need to be broadcast to all slices of the corresponding M operator.

Publications need to be matched against all subscriptions. They are thus broadcast from the

[3]Resilience of subscriptions in the presence of nodes faults can be handled at the level of the underlying stream processing engine, for instance using active replication or checkpoint/replay techniques.

AP operator to all slices in the corresponding M operator. Note that our architecture targets deployments in clouds or clusters, which are typically supported by dedicated, high-performance network infrastructures. The broadcast operation of publications in our architecture is designed to take advantage of the availability of IP multicast in such settings for dispatching publications, although our current evaluation does not exploit this feature.

MATCHING OPERATOR

The *Matching* (M) operator supports publication *filtering*. An M operator is associated with a library, denoted by libfilter in Figure 5.4, operating on the independently-maintained state at each of its slices. This library matches incoming publications against registered subscriptions. Different filtering implementations can be used as libfilter for different M operators, but they must comply with a simple API supporting two main operations: (1) storing/removing subscriptions based on their identifiers; and (2) processing a publication and returning a list of matching subscriber identifiers. At this stage, the content of the subscriptions and publications themselves is only accessed by the filtering library, making our architecture oblivious to the nature of the matching operation. The default filtering library provided with StreamHub is based on the SIENA counting algorithm [CW03] and is described in Section 5.3.1. Privacy-preserving filtering can be easily implemented using asymmetric scalar-product preserving encryption [CGB10] or other mechanisms [IRC10a; IRC10b; RR06]. Recent proposals to reduce the cost of privacy-preserving encrypted filtering through the use of a pre-filtering stage [Bar+12] can also trivially be integrated to libfilter libraries.

Subscriptions are stored in the state maintained for each slice. This state can be accessed concurrently using read and read-write locks (see Section 5.3 for implementation details). As filtering only requires reading the subscription set, and since most pub/sub workloads are dominated by publications, this allows for vertical scaling of the filtering operation for each slice of the M operator on multiple cores.

An M operator slice calls its libfilter for each incoming publication and generates an output event composed by the publication p and a list of matching subscriber identifiers, $s_1, s_2, \ldots, s_n$. When this list is empty, an output event indicating the lack of matching subscription is generated. The event is then sent to the next operator, the *Exit Point* (EP), using unicast. The routing key for selecting the slice of the EP operator is the identifier of the publication p. As a result, each slice of the M operator processing p will send its list of matching identifiers to the same slice of the EP operator.[4]

EXIT POINT OPERATOR

The *Exit Point* (EP) operator acts as the output operator of the engine. It shares similarities with the *reduce* phase in the MapReduce terminology [DG04]. Each incoming publication will be filtered at all slices at the M operator level, but will be processed by a single slice at the EP operator level. An EP operator slice receives the publication and lists of matching subscriber identifiers from all slices of the M operator (or notifications of empty matching lists). Once lists have been received from all M operator slices (or after a timeout to avoid slow M operator slices to delay notifications), the EP operator proceeds with *publication dispatching*: it contacts each DCCP maintaining a connection to at least one interested subscriber and sends it a notification message together with the identifiers of matching subscribers connected to that DCCP. The latter is then in charge of propagating the notification to the actual subscribers.

[4] If publications are of significant size, it is possible to have a single slice of the M operator send p and the others sending only their lists.

5.3 IMPLEMENTATION

We implement our architecture on top of a stream processing engine, StreamMine [Bri+11]. Other frameworks also present the features and abstractions our implementation requires, such as S4 [Neu+10], Storm [Fou], or Continuous-MapReduce [Bac+12]. We present an overview of the libfilter and libcluster libraries supported by our prototype StreamHub.

5.3.1 FILTERING AND CLUSTERING LIBRARIES

In the following, we present the filtering and clustering libraries (libfilter and libcluster) that StreamHub currently supports. While these libraries are based on known algorithms and do not represent novel contributions *per se*, they contribute to the overall performance of StreamHub as studied in Section 5.4.

FILTERING LIBRARIES

StreamHub can support any filtering scheme if implemented through an appropriate libfilter library. We listed variants of filtering schemes in Section 5.5.2. We note that our architecture also supports filtering schemes that need to maintain a state for each of the subscription they store, across the processing of several publications. For instance, a subscriber might wish to receive only the n^{th} publication that matches a given subscription, or be able to send subscriptions on the statistical evolution of publications attributes (e.g., over a window of publications). The corresponding state can be maintained by the libfilter in the slice-supported state.

StreamHub currently features an attribute-based filtering scheme library (libfilter) that uses a counting algorithm similar to that of SIENA [CW03]. It organizes predicates for received subscriptions in a forwarding table. An incoming publication will traverse and match in this forwarding table the predicates organized in conjunction sets. Each subscription is associated with a counter that specifies how many of its predicates have been matched so far. When a predicate is satisfied, the counters of all associated subscriptions are increased, and the whole subscription is marked as matched when all predicates of a subscription have been satisfied. Numerical predicates are indexed according to their type (=, <, >) and sorted by values in order to speed up traversals. Therefore, typically only a small part of the graph is traversed by publications. The algorithm generally scales sublinearly in the number of evaluated conjunction sets.

CLUSTERING LIBRARIES

When using multiple slices in the matching operator, each of these slices holds a subset of all subscriptions and filters publications concurrently with other matchers. By default, we partition subscriptions in a simple and deterministic way using a hash. This splits the load among all M operator slices. However, for many filtering schemes, filtering performance can be improved when subscriptions are partitioned in a content-aware manner. These types of subscription clustering are more costly than hash-based partitioning but they result in gains for the publication filtering performance. This is the case of the attribute-based filtering scheme described previously. As similar subscriptions are stored in the same M operator slice, the filtering algorithm may be able to better factorize common predicates and achieve higher filtering performance (this typically depends on the ability of the filtering operator to support containment determination between subscriptions). For pub/sub systems that process more publications than subscriptions, the relative gain can be significant as we show in our evaluation.

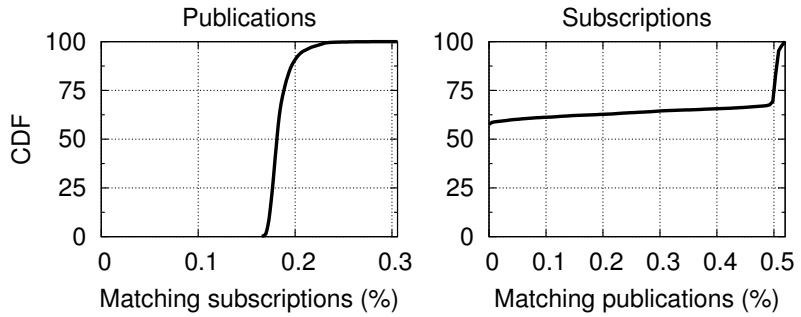


Figure 5.5: Workload characteristics: cumulative distribution of matching set sizes for publications (left) and matching ratios for subscriptions (right).

StreamHub supports various clustering algorithms by the means of libcluster libraries, that can optionally maintain state about previous subscriptions. When multiple filtering schemes are supported by multiple M operators, each slice of the AP operator supports a different libcluster (or default hash-based unicasting) for each such M operator. Deterministic clustering allows unicasting unsubscriptions while non-deterministic clustering require broadcasting unsubscriptions. StreamHub features the two clustering libraries described below.

K-Means [WKM00]: This clustering algorithm performs a partitioning of the subscriptions into K groups and a repetitive re-assignment based on the distance between subscriptions and groups until convergence. The algorithm is stateful and non-deterministic. We implement it in an online manner (*sequential* K-Means) for the dynamic clustering of subscriptions.

Event Space Partitioning (ESP) [Wan+02]: The space of subscriptions is represented as a d_s dimensional space, where d_s is the number of attributes. Each M operator slice is responsible for subscriptions that fall within its portion of the space. Subscriptions that intersect multiple domains are managed by the M operator slice that hosts the first attribute in lexicographic order. As the value d_s cannot be known in advance with content-based routing, it will increase when encountering subscriptions with unknown attributes. This clustering mechanism is stateful but deterministic.

5.4 EVALUATION

In this section, we present the experimental validation of StreamHub on a cluster of 48 nodes, each with two quad-core Intel Xeon (E5405) 2 GHz processors and 8 GB of RAM (384 cores total), interconnected with full-duplex 1 Gbps Ethernet. Our implementation uses the C++ language. We configure StreamMine to use batching between operators. Up to 16 KB of events can be stored in output buffers for each operator, and sent in batches or after a time limit. Batching allows increasing maximal supported throughput but has an impact on delays, as we demonstrate at the end of this section.

We first present the characteristics of the pub/sub workload used for the evaluation, followed by the baseline performance of the counting algorithm (libfilter). We then proceed to a operator-by-operator evaluation of the StreamHub architecture, highlighting performance and scalability of each of the operators. We describe the impact of subscription clustering (libcluster) and evaluate how the system scales when using an increasing number of nodes. Finally, we present a comparison of our approach with a broker overlay solution running on the same cluster.

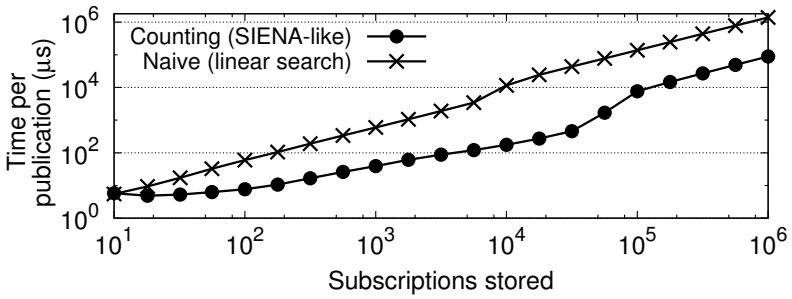


Figure 5.6: Performance of the counting libfilter for filtering incoming publications with respect to the size of the stored subscriptions set.

5.4.1 EXPERIMENTAL WORKLOAD

We constructed an experimental workload similar to the one used for the evaluation of Meghdoot [Gup+04]. It targets an attribute-based filtering scheme. We gathered five years of quotes for 200 randomly selected stocks from Yahoo! Finance [Yah]. This corresponds to over 250,000 publications. We built synthetic subscriptions based on the same categories as in [Gup+04]. These subscriptions contain a variety of ranges and equality predicates on the attributes of stock quotes, namely the symbol, date, exchanged volume, and daily statistics on their price (open, close, high, low). The characteristics of the workload are detailed in Figure 5.5. They represent a moderately selective type of pub/sub workload: a publication needs to be dispatched to a median of 0.18% of all subscriptions (with 100,000 subscriptions, each publication generates a median of 180 notifications), while a large part of subscriptions do not find publications of interest in the workload but yet need to be processed by the M operator.

5.4.2 BASELINE FILTERING PERFORMANCE

We first evaluate the raw performance of the libfilter filtering library based on the SIENA-like [CW03] counting algorithm. We compare it to a naive linear-search filtering mechanism acting as a baseline. Figure 5.6 indicates that the filtering operation cost evolves sublinearly and is at least an order of magnitude better than the naive algorithm above 500 stored subscriptions. Nonetheless, the cost of filtering can grow quite high with large sets of subscriptions, as can be observed on the right side of the graph. This highlights the importance of scaling the processing of incoming publications horizontally and vertically to sustain a high filtering throughput, and to process an incoming publication against subsets of the overall set of subscriptions to reduce dispatching delays.

5.4.3 PERFORMANCE OF OPERATORS

We now proceed to an operator-by-operator evaluation of StreamHub. Our methodology is to add one operator at a time, replacing the operators downstream the DAG by *sink* operators that receive the events but do not process them further. We denote such sink operators as S(AP), S(M), and S(EP). We focus our evaluation on the scalability aspects of each operator. All experiments are based on 180-seconds runs of StreamHub, during which the system is fed with subscriptions and/or publications as fast as possible to observe the maximal achievable throughput. When observing the performance of the publication filtering operation, subscriptions are registered before starting the measurement.

In our evaluation, the DCCPs are replaced by *generators* that inject the workload into the AP operator. We first verified that these generators can provide the system with a sufficient

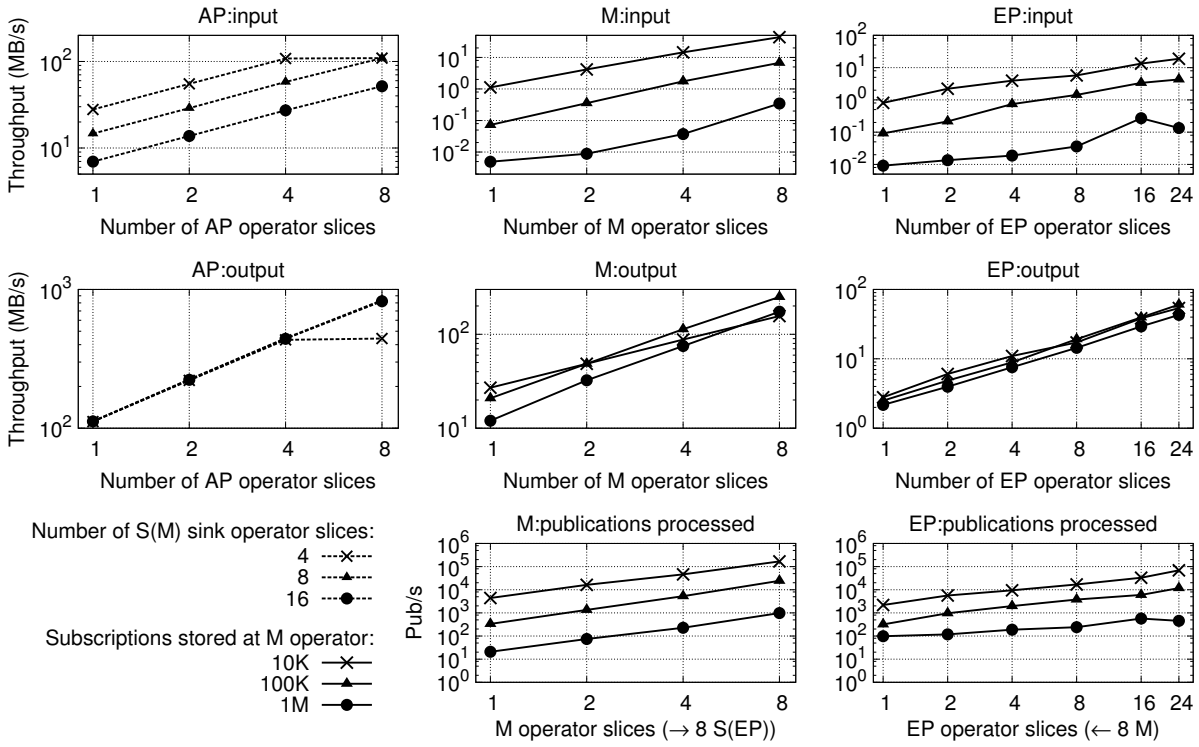


Figure 5.7: Scaling of StreamHub operators: Input and output throughput for all operators when varying the number of slices and subscriptions. Each operator is evaluated with the downstream operator replaced by a sink. We use one physical machine per slice.

throughput of publications and subscriptions and do not represent a bottleneck. Our results (not shown) indicate that the generators are able to nearly saturate the input bandwidth capacity of the nodes that host the AP operator slices. This indicates they will not impair the remainder of the evaluation.

We present the complete operator-by-operator evaluation results in Figure 5.7. We look primarily at the throughput in terms of bandwidth and events processed.

AP OPERATOR SCALABILITY

The first column of two plots in Figure 5.7 presents the AP operator scalability. It depicts the maximal input and output throughput of the operator with a publications-only workload. This corresponds to the worst case scenario since publications, unlike subscriptions, need to be broadcast by each AP operator slice to all sink S(M) operator slices. As expected, the input throughput of the AP operator is inversely proportional to the number of sink S(M) operator slices: a copy of each publication needs to be made for every S(M) operator slice and the bottleneck becomes the output bandwidth of AP operators. The planned support for IP multicast between the AP and M operators would boost performance for this operation. We observe nonetheless that this output throughput nearly saturates the cross-bandwidth of the connections between the AP and S(M) operator slices, and is able to saturate the input bandwidth of the S(M) operator slices (serving 104 MB/s of publications to each of them). This indicates that the AP operator will not hinder scalability when the S(M) sink operator slices are replaced by real M operator slices that need to perform the computationally-intensive filtering operation, as confirmed by our next experiment.

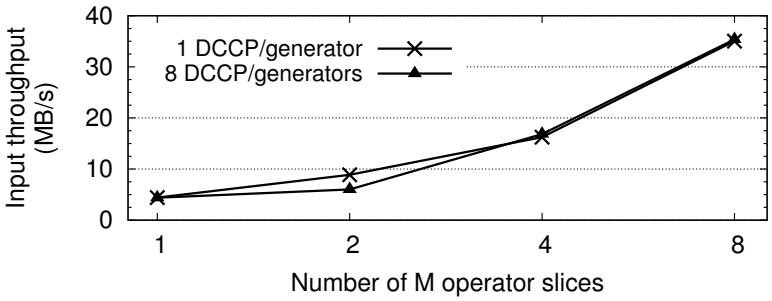


Figure 5.8: Scaling of the M operator receiving and storing a subscriptions-only workload. The throughput corresponds to the traffic between the DCCP/generators and the AP operator, with 8 AP operator slices.

M OPERATOR: SUBSCRIPTION STORAGE SCALABILITY

We start the evaluation of the M operator by assessing the scalability of the subscription storage with a subscriptions-only workload. Figure 5.8 presents the average bandwidth that the generators are able to push through the AP operator for storage at the M operator level. We use a set of 8 AP operator slices so that the AP operator does not form a bottleneck. We use 1 to 8 M operator slices. We observe that the scalability of the subscription storage is almost linear and StreamHub is able to register a flow of 35.4 MB/s with 8 M operator slices, which corresponds to a constant flow of 150,000 subscriptions stored per second. We observe a slight degradation of the throughput when using 8 generators and only 2 M operator slices. The reason was tracked down to overflows in input buffers of the M operator slices, leading to retransmissions of messages from the AP operator and some loss of bandwidth.

M OPERATOR: PUBLICATION MATCHING SCALABILITY

We now investigate the scalability of the filtering operation at the M operator level, i.e., matching each publication against the set of stored subscriptions. We use a set of 8 sink S(EP) operator slices as the downstream operator and 8 AP operator slices for the upstream operator. The second column of three plots in Figure 5.7 presents the achieved input/output throughput and the number of publications that the M operator filters per second, including transmission to the downstream S(EP) operator. We clearly observe that the architecture scales: the addition of new operator slices to the M operator results in linear increase of its processing capacity. Note that, as expected from the workload characteristics (median matching set of 0.18% of stored subscriptions), the bandwidth requirements are higher for output than for input because the publications are augmented with a potentially large list of matching identifiers.

EP OPERATOR SCALABILITY

We complete the operator-by-operator scalability evaluation by replacing the S(EP) sink operator slices with their real counterparts that perform publication dispatching. We use 8 generators, 8 AP, and 8 M operators slices. We observe in the third column of three plots of Figure 5.7 the input/output throughput of the EP operator and the number of publications that are effectively dispatched. With a configuration of 8 EP operator slices, StreamHub is already able to filter and dispatch from 3.8 K to 17 K publications per second, for respectively 100 K and 10 K stored subscriptions (corresponding to 684 K and 306 K notifications sent out to subscribers per second, respectively).

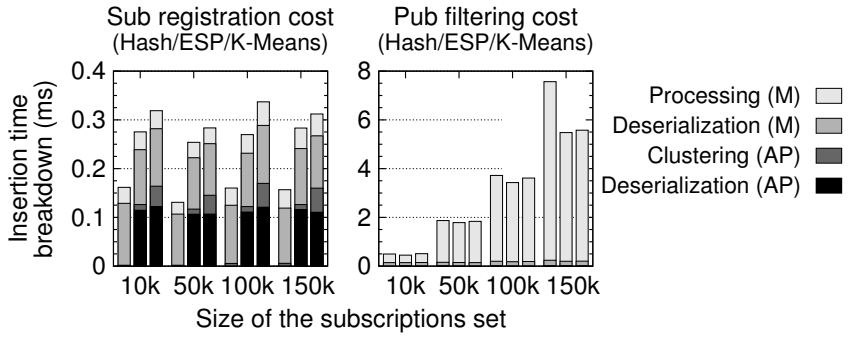


Figure 5.9: Overhead of clustering for storing subscriptions (left) and impact on filtering efficiency (right). Each group of stacked bars shows the breakdowns of average costs for one subscription or one publication matched against the corresponding subscription set. Each group has three bars for hash-based (no clustering), ESP, and K-Means.

DISCUSSION

The results of the operator-by-operator evaluation clearly show that the subscription registration and publication filtering operations scale by adding more slices (and thus physical machines) to the operator that supports them. Adequately provisioning the architecture allows handling an arbitrary number of publications and subscriptions. One should point out that the specific workload considered in our evaluation is costlier for the operators that deal with publication dispatching (EP) and matching (M) than for handling and forwarding incoming publications (AP).

5.4.4 IMPACT OF CLUSTERING

We now investigate the impact of using a libcluster library at the AP operator level for clustering subscriptions. Our objective is to evaluate if the additional cost for registering a subscription in the system using content-aware clustering is compensated by the subsequent performance gain when filtering publications against stored subscriptions. We present in Figure 5.9 the time required to store a subscription at the M operator level (left) and process an incoming publication against the set of stored subscriptions (right). Times are obtained by averaging over 10,000 events. The breakdown distinguishes between the different operations at the AP and M operators: deserializing the event at the AP operator level (for subscriptions when using clustering) and processing it (clustering, storing, or matching) at both the AP and M operators levels.

We observe that the cost of subscription insertion increases due to the additional deserialization and treatment at the AP operator level. On the other hand, the use of clustering yields significant performance gains when matching publications against a large set of subscriptions: for 150 K subscriptions matching is 25 to 27% faster when using K-Means or ESP. This supports our claim that using a libcluster library, when applicable to the filtering scheme being used, may significantly increase the filtering performance or reduce the number of M operator slices required to sustain a given publication filtering throughput requirement. At the same time, the use of a libcluster library does not break the separation of concerns and filtering schema agnosticism that underpins the complete architecture.

Configuration	Batching	Max. Publications/s	Avg. delay	Std. dev.
4 AP / 4 M / 8 EP = 16	No	200	0.36s	0.17s
	Yes	500	1.06s	0.28s
8 AP / 8 M / 16 EP = 32	No	500	0.22s	0.15s
	Yes	1000	0.98s	0.30s

Table 5.2: End-to-end delays (settings as Figure 5.11).

5.4.5 OVERALL PERFORMANCE

Our last experiment with StreamHub relates to the overall provisioning and scaling of the complete architecture. We consider the case where a cluster or a cloud virtual environment needs to be scaled up to increase the throughput of the pub/sub process, with the objective of offering a linearly increasing performance as more physical nodes are added to support the service, and to sustain low and predictable end-to-end delays. As previously demonstrated, the optimal assignment of slices (and thus, physical machines) to operators depends on the nature of the workload. For the purpose of this evaluation, we determined the best configuration for a given budget of machines based on the operator-by-operator experiments. Our architecture is designed to easily support dynamic scaling by migrating slices between physical machines. We leave the integration of such mechanisms and the appropriate decision-making systems to future work.

Figure 5.11 presents the evolution of the publication throughput with clusters of 8 to 32 machines (our other machines are used for 8 generators and 16 sink DCCPs). We observe that the scalability objectives of StreamHub are met: there is a linear gain in performance between 8 to 32 machines. The maximal supported throughput between the smaller and the larger configuration when using the ESP partitioning is actually 4.26x higher, which is mostly due to the reduced contention on the AP operator slices. We note that the impact of clustering is consistent with what we observed in Figure 5.9: throughput is from 10% to 28% better with clustering (see Figure 5.7).

Table 5.2 presents the average end-to-end delays (between the reception of a publication and the reception of the corresponding notifications by the subscribers) observed by clients, and their variations. In this case, we selected the throughput to be around half of the maximal supported throughput in the 16 and 32 nodes configurations. We consider two settings, with batching and without. Batching increases throughput but also introduces extra delays. Disabling it divides the maximal supported throughput by approximately a factor of 2. In both cases though, the delays are low (around a second with batching, or a fraction of a second without it) and predictable as they show only slight variations between publications.

Figure 5.10 shows the relationship between target throughput (i.e., maximum number of publications/s the generators are sending), the actual throughput (i.e., number of publications/s the generators are able to emit, given TCP-backpressure from downstream operator slices), and the end-to-end latency. The actual system throughput is depicted on the left side. The lines and points show the mean throughput of the given experiment over time and the errorbars the 10th (lower end) and 90th (upper end) percentile. As a rule of thumb: The larger the errorbars the more the throughput fluctuates and, hence, the more likely the system is overloaded. For that reason we selected a target throughput of 500 publications/s for the small 16 machines configuration with batching: It is the experiment with the highest target throughput where the actual system throughput stays constant over time. In contrast to that, the throughput of the same configuration but without batching already varies a lot, as shown by the errorbar. This is even more prominent on the right side of the figure which depicts the mean latency with the errorbars again showing the 10th and 90th percentile. As one can

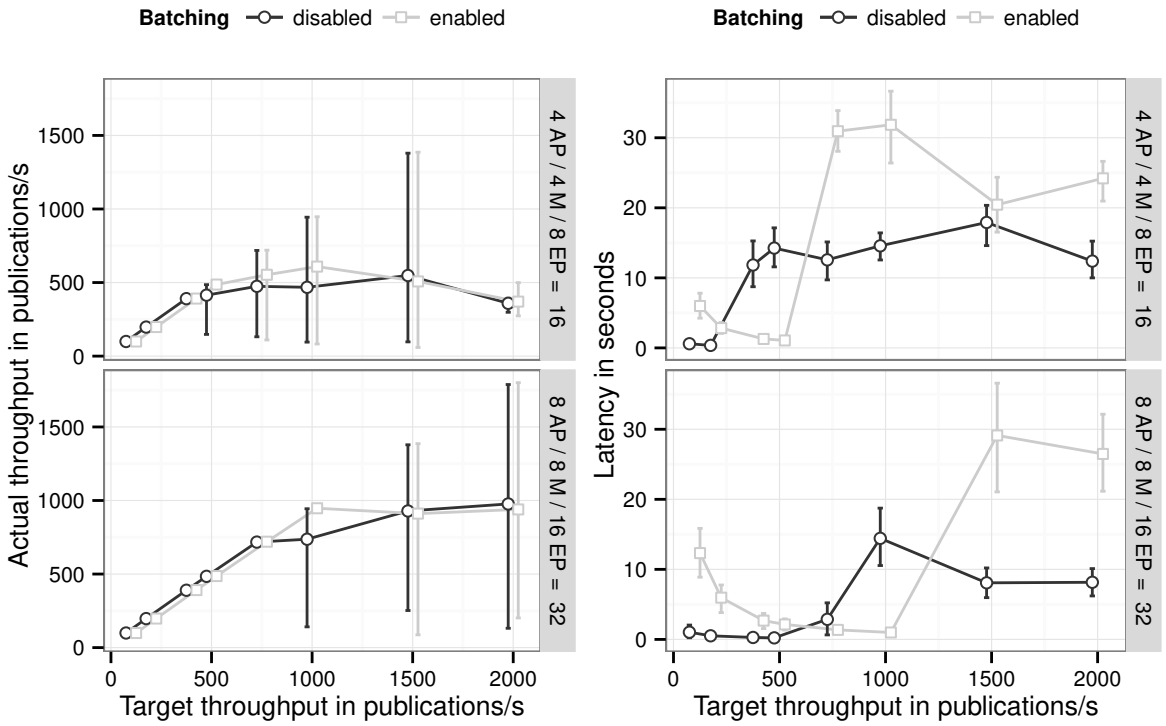


Figure 5.10: System throughput and latency.

see, with 16 machines, a target throughput of 500 publications/s, and enabled batching the latency is small and predictable. Contrary, the latency of the same configuration but without batching is not only high (events queue up in the TCP queue) but it also becomes unpredictable as it varies significantly. While the larger configuration shows the same properties, albeit supporting higher publication throughput, there is a another property of batching which becomes evident. If the throughput is small, batching can actually increase the latency. This is because the batches have a fixed size and need to be full before they can be sent. Hence, the latency actually becomes a jigsaw-like curve over time which is the reason for the larger errorbars when batching is enabled and the target throughput is small.

5.4.6 COMPARISON WITH PADRES

For completeness, we have also performed experiments with the same set of subscriptions and publications using the most recent version of PADRES [Jac+09] (v2.0). As detailed in our introduction, PADRES is based on different design choices than StreamHub: it establishes and maintains a network of brokers that collectively implement pub/sub functionality and is specific to a particular filtering scheme, while our design dedicates different machines to each operation and is independent from the actual filtering scheme that is used.

Our PADRES setup consists of one publisher, one subscriber, and a varying number of brokers. We verified that neither the publisher nor the subscriber represent a bottleneck in the experiments. The publisher and the subscriber are connected to every broker. Subscriptions and publications are randomly partitioned among brokers. Every machine executes 4 broker instances, which corresponds to half of its available cores. Using all cores on each machines yielded lower performance figures, probably due to contention on resources. Results are averaged over 100 publications, measured after an initial warm-up phase of 2,000 messages to enable JIT optimizations.

Figure 5.12 shows the throughput achieved with the same 100,000 subscriptions as for

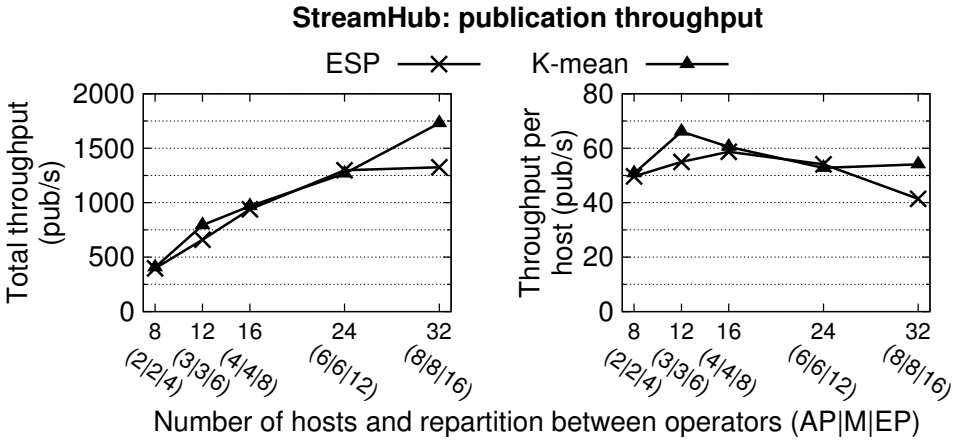


Figure 5.11: Throughput of StreamHub with 100,000 subscriptions, using libcluster and the workload-optimal configurations for each number of available machines. The number of slices at each operator is indicated within parentheses.

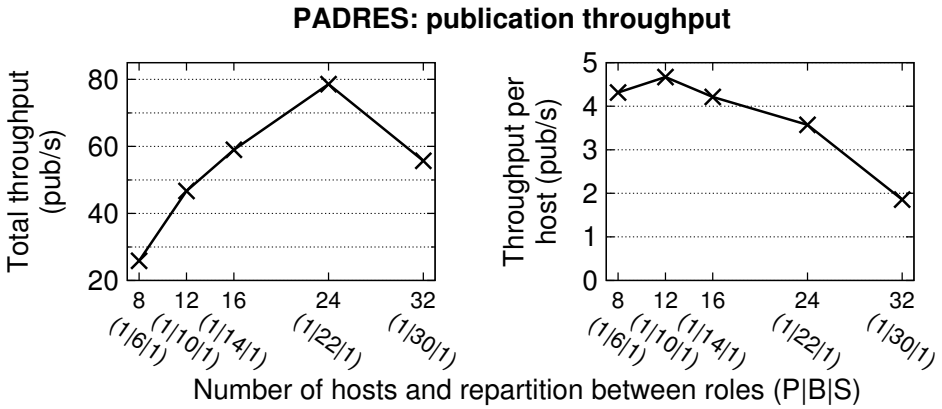


Figure 5.12: Throughput of PADRES with 100,000 subscriptions. The number of hosts is indicated within parentheses: a single publisher (P) and a single subscriber (S) were sufficient to fully load the brokers (B).

Figure 5.11 and various numbers of brokers. We observe that PADRES scales well for up to 88 brokers (i.e., 22 machines, each running 4 broker processes) but seems to suffer when adding more brokers. Actually, each publication has to be filtered by multiple brokers for propagation to other brokers due to the use of routing tables that are constructed according to the filtering schema that is used. This adds to the overall load and reduces the contribution of each broker to overall throughput (Figure 5.12, right). We finally observe that the maximal raw throughput achieved in our cluster is two orders of magnitude higher with our architecture than with PADRES. While a part of this difference can be accounted to language and implementation differences, the higher parallelism and independence of operations in our architecture clearly helps improving the filtering throughputs.

5.5 RELATED WORK

We start by reviewing related work on distributed content-based pub/sub systems. We focus on high-efficiency middleware operating on dedicated machines and do not specifically elaborate on peer-to-peer approaches. Similarly, we do not discuss work targeting the simpler topic-based filtering model.

5.5.1 PUBLISH/SUBSCRIBE ENGINES

Most earlier work on scalable pub/sub has relied on networks of *brokers*, which are dedicated machines, each performing the whole range of operations that compose the content routing task: (1) management of subscriptions from users and other brokers, (2) filtering of incoming publications against stored subscriptions and dispatching to local interested subscribers, and (3) filtering of incoming publications against routing tables for dispatching to other brokers. Brokers are typically organized in a *broker overlay*, with subscriptions and publications flowing between brokers according to its logical structure, typically a tree or a mesh.

Well-known examples of broker-based pub/sub middleware are SIENA [CRW01], Gryphon [Agu+99], and PADRES [Jac+09]. In these systems, a client (publisher or subscriber) connects to one of the brokers, which then acts as its single point of contact. Brokers forward subscriptions registered by their clients towards neighboring brokers. These systems are based on a filtering scheme where subscriptions are defined as conjunctions of predicates ($<$, $\leq$, $=$, ...) on a set of discrete attributes values (integers, strings, ...). They rely on the ability to (1) determine containment relationships between subscriptions, and (2) construct aggregated subscriptions representing the interests of sets of subscriptions. Subscriptions are aggregated along the way from consumers to producers of information, taking advantage of containment relationships between subscriptions: a single aggregated subscription may represent the interests of many downstream subscriptions, thus reducing the number of subscriptions managed by the broker and improving filtering performance (see for instance [JE11]). This approach works well with few publishers and with subscriptions that have certain locality properties (e.g., subscribers with similar interests connect to the same broker). However, it requires complex algorithms for maintaining the consistency of forwarding tables and provides only limited benefits when information flows from many sources or when the subscription representation does not allow for containment or aggregation [Bar+12; CGB10; IRC10a; IRC10b; RR06; Sah+98; Bru09].

In broker overlays, under some workloads, publications may have to traverse a number of brokers that have no local interested subscriber but still have to filter the publication against stored subscriptions. These are called *forwarder-only* brokers. While techniques for rewiring the broker overlay have been proposed to tackle this problem [KJ12], the presence of such forwarder-only brokers is intrinsic to a design where communication flows depend on the filtering scheme and on the current workload of stored subscriptions. Since all brokers play all roles in the pub/sub operation, the allocation of publishers and subscribers to brokers has a strong impact on the balance of load and on the overall filtering efficiency. A bad placement may result in a high number of messages being propagated between brokers. Some optimizations were proposed to address this problem by connecting subscribers with similar subscriptions to the same brokers [CJ10a], or by linking publishers and their expected subscribers to the same brokers [Li+12; CJ10b]. Cheung *et al.* [CJ11] proposed to use similar techniques to rewire the PADRES overlay in order to reduce the environmental footprint of the pub/sub system. Again, these mechanisms are dependent on the filtering scheme and require the ability to determine proximity relations between subscribers and publishers. This would not be possible, for instance, with encrypted filtering approaches.

In contrast to systems based on overlay of brokers, the logical connections between the elements in our proposed architecture are independent of the nature of the subscription and

publication workloads and of the nature of the filtering scheme. We support scaling each of the pub/sub operations independently by simply adding more processors to the set of nodes that support this operation. We do not require any specific optimization support from the filtering scheme, though we can leverage their existence for improving single-node performance inside filtering libraries. Note that our approach is readily applicable to architectures like Google's GooPS [Reu09], where pub/sub is implemented by regional data centers consisting of clusters of brokers and interconnected by dedicated network links.

5.5.2 FILTERING MECHANISMS

Our architecture supports *pluggable* filtering mechanisms. As the design of new such mechanisms is not the focus of this paper, we only briefly discuss below a few well-known algorithms that can be readily used in our StreamHub implementation (see Section 5.3.1). Note that this list is far from being exhaustive.

SIENA uses a counting algorithm [CW03] for efficiently matching publications against subscriptions. Individual predicates are stored in a forwarding table and a subscription is detected as matching when all its predicates have been encountered. We use an implementation of this counting algorithm as the default filter in StreamHub for non-encrypted matching. Additional details on its operation are given in Section 5.3.1. Encrypted filtering can be supported for instance by *asymmetric scalar-product preserving encryption* [CGB10], combined with pre-filtering [Bar+12] for efficiency. Gryphon [Agu+99] inserts the set of subscriptions into a matching tree: leaves contain subscriptions, non-leaf nodes contain tests, and outgoing edges represent the results of the tests. A publication traverses down the tree by following all matching paths and reports a matching subscription for each leaf node reached. PADRES uses a scalable filtering engine [Far+09] that can leverage multiple cores on a shared memory architecture. By splitting the state of subscriptions and using multiple threads synchronized using either locks or transactional memory, the filtering throughput is significantly improved. We also exploit all the cores available on a machine and provide synchronization mechanisms for concurrent accesses to a shared state.

Other examples of filtering mechanisms that can be leveraged in the context of StreamHub include, but are not limited to, the following. RAPIDMatch [Kal+05] is a tree-based filtering mechanism that takes into account the sparseness of criteria definitions over the whole attribute set in some pub/sub workloads for greater efficiency. TAMA [ZW11] trades accuracy and space complexity for efficiency by clustering range-based subscriptions in predefined sets based on discrete cuts of the definition range of each attribute. The use of discrete cuts leads to the presence of false positives, while the presence of subscriptions in multiple buckets leads to higher memory consumption, in return for faster filtering. Fabret *et al.* [Fab+01] use schema-clustering to minimize the number of filtering operations performed, along with techniques to improve cache performance of the algorithm. Filtering mechanisms also exist for boolean expressions [BH07; Fon+10], XML documents and XPath expressions [AF00; Cha+02], and compact data representation using Bloom filters [JF08].

5.5.3 CLUSTERING SUBSCRIPTIONS

Similarly to filtering mechanisms, we support subscription clustering by the means of pluggable libraries. Subscription clustering, as described in Section 5.3.1, splits the whole set of subscriptions maintained by the pub/sub system into clusters according to similarity (if a proximity metric is available on the subscription). This typically increases the level of containment and the potential for aggregation, which in turn improves the performance of the filtering operation. Classical clustering algorithms include K-means [WKM00], Event Space Partitioning (ESP) [Wan+02], or R-trees [Bec+90].

5.6 SUMMARY

We presented a novel design for high-throughput pub/sub services. We focused on the support of large-scale applications communicating through a managed environment providing the pub/sub service, such as a publicly available cluster or a public cloud deployment. Our architecture is highly parallel and scalable, and can readily support arbitrary complex filtering schemes, including encrypted or state-based filtering. We do so by departing from previous approaches based on broker overlays and by decoupling the architecture and communication flows of the pub/sub system from the filtering scheme(s) and the subscriptions workload. Our implementation, StreamHub, splits the pub/sub service into fundamental operations, allocated to horizontally and vertically scalable operators supported by a scalable stream processing engine. The evaluation of StreamHub on a cluster with up to 384 cores indicates that it can sustain high throughputs of subscription registrations and publication filtering: we filter thousands of publications against hundreds of thousands of registered subscriptions, resulting in hundreds of thousands notifications sent to clients every second.

6 CONCLUSION AND FUTURE WORK

Securing large computing infrastructures is a major task for decades to come. As intrusion detection evolves, attacks become more sophisticated as well and either side is unlikely to ultimately win or loose. In fact, we witness an arms race on both sides since Internet connectivity became wide-spread among businesses.

The most prominent example are APT attacks. Such attacks do not follow a strict playbook and, hence, their strategy can be easily adapted if necessary. Usually, these attack campaigns have more than one vector of attack and several pieces of malware to their availability. Some even change their modus operandi, for instance, switching from plain-text to encrypted communication with the command-and-control server. APT attacks are also specifically designed for a carefully selected group of related targets such as banks, energy companies, or defense contractors. Often, attackers have insider knowledge of how these target systems are operated (e.g., carbanak or stuxnet). Moreover, attackers manage to evade detection by using obfuscation, encryption, and mimicry — effectively "flying below the radar" of commonly deployed IDS. In fact, all presented attacks and those investigated by other researchers were running for months, sometimes years before the victims became aware of them — often by coincidence.

In all cases they had to consult highly specialized security companies to investigate their suspicion. These specialists then looked at a combination of sources, such as emails, forum posts, honeypots, or IP flow information from several victims at once. Only the combination and correlation of a multitude of such datasets made it possible to confirm or contradict the initial suspicions.

Hence, intrusion detection must not be tied to a specific dataset or detection algorithm. Rather, IDS must permit the addition and removal of data sources and the adaptation of detection algorithms. At the same time their design must support scalability. First, because different detection algorithms have different computational overhead. And second, because additional data sources may require more computing power. Moreover, intrusion detection must become a community concern where related entities cooperate to detect "below the radar" attacks.

This chapter summarizes the main achievements of this thesis in the field of scalable distributed processing architectures and intrusion detection techniques. It will also provide a brief outlook of possible future research directions.

6.1 CONCLUSION

This thesis presents a novel community detection algorithm to detect industries in IP flow data in Chapter 2. This is necessary to understand which entities of our input data belong to one of the business communities we want to protect. In contrast to the networks which are commonly studied for community detection, Internet flow data exhibits some distinguishing characteristics. First, its edges are ambiguous, for example, a communication between two IP addresses might mean that a user accesses another machine intentionally but it may also mean that an advertisement has been downloaded or that a click-analysis site was accessed. It is hard to differentiate between such unintended connections and actual user-intended connections. Second, the business communities do not manifest themselves as tightly knit communities, as assumed by conventional community detection. Rather, they are loosely-coupled with more connections to the rest of the Internet than to other members of the industry. Consequently, we found that traditional community detection algorithms were unable to detect our business communities reliably. Therefore, we propose a new algorithm, which instead of finding a closely knit community around a seed-set, tries to expand a given set of seeds by including those neighbors with a very strong connection to the community. And, most important, without preferring those neighbors, which are otherwise barely connected to the rest of the network. Our algorithm is able to detect three sample industries — energy,

financial, and defense — reliably in IP flow data while maintaining a low number of false positives.

In Chapter 3, we continue with the design and implementation of a distributed scalable graph mining engine. This enables us to store and process the vast amounts of input data, such as netflow data from distributed ISP routers and properly represent its inter-dependencies. It supports in-memory storage and updates to the graph with high throughput and below 10ms latency. In fact, those updates can be arbitrarily complex. For example, we show how to construct a COI graph on the fly by performing window-based aggregation of graph-updates. Furthermore, it is possible to query the graph — concurrent to other queries and concurrent to the graph updates. Using these queries, we implement several important graph algorithms like the local clustering coefficient or community detection algorithms. Finally, we use our engine to implement fraud detection in telephony networks and intrusion detection in IP flow data. Our engine shows excellent scalability, throughput, and latency for both applications.

A deeper evaluation of the intrusion detection application is given in Chapter 4. In order to tune the intrusion detection for different aspects of the input data, we added different views, all running concurrently. In the case of netflow, the chapter presents three views: A COI graph whose weight is based on the amount of transferred bytes, a COI graph whose weight is based on the vulnerability of the ports used, and a COI graph whose weight is based on the number of attempted connections. We ran our engine for several months and found that we were able to detect suspicious IP addresses which maintained an otherwise very low traffic profile per company. Throughout this period, the daily alarm rate remained below 20 for most of the days. We consider this to be a manageable amount of reports for a community of security administrators. The chapter presents several example cases of possible intrusions. Post-processing alarms is exemplified by using distributed blacklists and DNS lookup services. This post-processing has been successfully applied for ranking the issued alarms.

The next step was to design a system on top of which we cannot only implement our graph-based intrusion detection technique but which would also allow for adding, removing, and adapting any arbitrarily complex detection algorithm in Chapter 5. Furthermore, the goal was to put administrators in control of which alerts they want to receive and which not. In theory, content-based Publish/Subscribe permits exactly that: Subscribers (the administrators) subscribe to a given kind of event-content using a subscription language (e.g., a conjunction of predicates). Subscribers can add, remove, or change their subscriptions at any time. Publishers (our data sources) may send publications at different throughput and new publishers can be added during runtime. While, so far, it seemed to be the perfect solution, the state-of-the-art content-based Publish/Subscribe systems support only a fixed kind of filter (e.g., stateless conjunction of predicates) and were not suited for high-throughput and low-latency processing. Hence, we designed and implemented a new architecture for Publish/Subscribe, called StreamHub, allowing to run any number of arbitrarily complex filters within the same system. Our key contribution is that we implement the different stages, involved in filtering publications, as individual operators which we can scale separately to the current workload. Moreover, departing from the traditional peer-to-peer overlay used for Publish/Subscribe, allows us to support any kind of filter - even stateful ones, such as our graph-based intrusion detection. Our evaluation shows that StreamHub maintains low predictable latency at high throughput and out-performs the state-of-the-art by an order of magnitude.

6.2 FUTURE WORK

This thesis leaves several interesting open issues which are worth to be investigated in the future.

One direction of future work is to integrate our graph-based intrusion detection as a filter in StreamHub. While this is trivial by itself, it is interesting to think about possible combinations of different filters: For example, combining our graph-based intrusion detection with text-mining approaches that may be able to detect spear-phishing emails. This could be enhanced by the blacklist-based attack prediction. With such a system, an administrator would be able to follow an attack from the very beginning on. First, the attackers send spear-phishing emails to the corporations. These actions can be detected by a text-mining filter in StreamHub. Once the attackers have access to some victim computers they will start the lateral movement phase or directly exfiltrate data. In either case these actions require communication with the command-and-control servers and this can be detected by our graph-based intrusion detection filter in StreamHub. Finally, next targets could be predicted with a dedicated filter in StreamHub. Such a filter would gather information from distributed blacklists, for example, and build knowledge about past attack spreads. It would use this knowledge to predict the spread of ongoing attacks. Especially the attack prediction might enable victims to not only implement counter-measures but also to deploy additional sensors. This may be of great help — first, for knowing which exact parts of the system had been attacked (e.g., what data has been stolen) and second, to determine the identity of the attackers.

Another direction would be to continue the research in community detection. We have learned that IP flows are fundamentally different from the networks that community detection has been applied to traditionally. More effort is needed to further evaluate how loosely-knit communities, such as our industries, may be detected automatically. Moreover, detecting and updating such communities in real-time with the graph updates is a challenging and interesting research direction. Although related work in that direction exists, they assume a slow rate on graph-updates which is contradictory to the high update rates of IP flows, for example. Furthermore, community detection should also be applied to other input sources. While research on community detection has evaluated emails, forums, and blacklists to a large extend, the detected communities were tightly knit. However, it is unclear whether our industries are actually tightly coupled in such datasets. For example, it has yet to be determined if the exchange of emails between companies is significant enough to be considered a tightly knit community.

Finally, it would be interesting to evaluate our system using actual APT attack traces. Unfortunately, the only publicly available IP flow dataset which contains attacks is an artificial dataset, created more than 15 years ago. Given its age, this dataset is in fact considered to be irrelevant for modern intrusion detection. Another option would be to use traces from honeypots or honeynets. Those are, however, almost exclusively attacked by coincidence. For example, unskilled attackers often run scripted attacks and select their targets rather randomly. Consequently, honeypots are used for extracting malware samples but do not help in learning about new APT attack campaigns. Although the matter is often highly classified, a close collaboration with security companies might open possibilities to verify the approach.

BIBLIOGRAPHY

[Abr+12] Bruno Abrahao, Sucheta Soundarajan, John Hopcroft, and Robert Kleinberg. "On the Separability of Structural Classes of Communities". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 624–632. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339631. URL: http://doi.acm.org/10.1145/2339530.2339631.

[ACL06] Reid Andersen, Fan Chung, and Kevin Lang. "Local Graph Partitioning Using PageRank Vectors". In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 475–486. ISBN: 0-7695-2720-5. DOI: 10.1109/FOCS.2006.44. URL: http://dx.doi.org/10.1109/FOCS.2006.44.

[AF00] Mehmet Altinel and Michael J. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 53–64. ISBN: 1-55860-715-3. URL: http://dl.acm.org/citation.cfm?id=645926.671841.

[Agu+99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. "Matching Events in a Content-based Subscription System". In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 53–61. ISBN: 1-58113-099-6. DOI: 10.1145/301308.301326. URL: http://doi.acm.org/10.1145/301308.301326.

[AL06] Reid Andersen and Kevin J. Lang. "Communities from Seed Sets". In: *Proceedings of the 15th International Conference on World Wide Web*. WWW '06. Edinburgh, Scotland: ACM, 2006, pp. 223–232. ISBN: 1-59593-323-9. DOI: 10.1145/1135777.1135814. URL: http://doi.acm.org/10.1145/1135777.1135814.

[Alv+13] Lorenzo Alvisi, Allen Clement, Alessandro Epasto, Silvio Lattanzi, and Alessandro Panconesi. "SoK: The Evolution of Sybil Defense via Social Networks". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 382–396. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.33. URL: http://dx.doi.org/10.1109/SP.2013.33.

[Ami+06] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. "SPC: a distributed, scalable platform for data mining". In: *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. DMSSP '06. Philadelphia, Pennsylvania: ACM, 2006, pp. 27–37. ISBN: 1-59593-443-X. DOI: http://doi.acm.org/10.1145/1289612.1289615. URL: http://doi.acm.org/10.1145/1289612.1289615.

[Ani+11] Leonardo Aniello, GiuseppeAntonio Di Luna, Giorgia Lodi, and Roberto Baldoni. "A Collaborative Event Processing System for Protection of Critical Infrastructures from Cyber Attacks". English. In: *Computer Safety, Reliability, and Security*. Ed. by Francesco Flammini, Sandro Bologna, and Valeria Vittorini. Vol. 6894. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 310–323. ISBN: 978-3-642-24269-4. DOI: 10.1007/978-3-642-24270-0_23. URL: http://dx.doi.org/10.1007/978-3-642-24270-0_23.

[AP09] Reid Andersen and Yuval Peres. "Finding Sparse Cuts Locally Using Evolving Sets". In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 235–244. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536449. URL: http://doi.acm.org/10.1145/1536414.1536449.

[ATK14] Leman Akoglu, Hanghang Tong, and Danai Koutra. "Graph based anomaly detection and description: a survey". English. In: *Data Mining and Knowledge Discovery* (2014), pp. 1–63. ISSN: 1384-5810. DOI: 10.1007/s10618-014-0365-y. URL: http://dx.doi.org/10.1007/s10618-014-0365-y.

[Bac+12] Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, and Ugur Cetintemel. "C-MR: Continuously Executing MapReduce Workflows on Multi-core Processors". In: *Proceedings of Third International Workshop on MapReduce and Its Applications Date*. MapReduce '12. Delft, The Netherlands: ACM, 2012, pp. 1–8. ISBN: 978-1-4503-1343-8. DOI: 10.1145/2287016.2287018. URL: http://doi.acm.org/10.1145/2287016.2287018.

[Bal+13] Maria-Florina Balcan, Christian Borgs, Mark Braverman, Jennifer Chayes, and Shang-Hua Teng. "Finding Endogenously Formed Communities". In: SODA '13 (2013), pp. 767–783. URL: http://dl.acm.org/citation.cfm?id=2627817.2627872.

[Bar+12] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, and Etienne Rivière. "Thrifty Privacy: Efficient Support for Privacy-preserving Publish/Subscribe". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. Berlin, Germany: ACM, 2012, pp. 225–236. ISBN: 978-1-4503-1315-5. DOI: 10.1145/2335484.2335509. URL: http://doi.acm.org/10.1145/2335484.2335509.

[Bar+13] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. "StreamHub: A Massively Parallel Architecture for High-performance Content-based Publish/Subscribe". In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 63–74. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488260. URL: http://doi.acm.org/10.1145/2488222.2488260.

[BCZ13] Qing Bao, William K. Cheung, and Yu Zhang. "Incorporating Structural Diversity of Neighbors in a Diffusion Model for Social Networks". In: *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 01*. WI-IAT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 431–438. ISBN: 978-0-7695-5145-6. DOI: 10.1109/WI-IAT.2013.61. URL: http://dx.doi.org/10.1109/WI-IAT.2013.61.

[BDQ13] Roberto Baldoni, GiuseppeAntonio Di Luna, and Leonardo Querzoni. "Collaborative Detection of Coordinated Port Scans". English. In: *Distributed Computing and Networking*. Ed. by Davide Frey, Michel Raynal, Saswati Sarkar, RudrapatnaK. Shyamasundar, and Prasun Sinha. Vol. 7730. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 102–117. ISBN: 978-3-642-35667-4. DOI: 10.1007/978-3-642-35668-1_8. URL: http://dx.doi.org/10.1007/978-3-642-35668-1_8.

[Bec+90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles". In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. Atlantic City, New Jersey, USA: ACM, 1990, pp. 322–331. ISBN: 0-89791-365-5. DOI: 10.1145/93597.98741. URL: http://doi.acm.org/10.1145/93597.98741.

[Bej10] R. Bejtlich. "CIRT-level response to advanced persistent threat". SANS Forensic Incident Response Summit. 2010.

[Ber+10] Robin Berthier, Michel Cukier, Matti A. Hiltunen, Dave Kormann, Gregg Vesonder, and Dan Sheleheda. "Nfsight: Netflow-based Network Awareness Tool". In: *Proceedings of the 24th International Conference on Large Installation System Administration*. LISA'10. San Jose, CA: USENIX Association, 2010, pp. 1–8. URL: http://dl.acm.org/citation.cfm?id=1924976.1924988.

[BFF09] Andrey Brito, Christof Fetzer, and Pascal Felber. "Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems". In: *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 173–182. ISBN: 978-0-7695-3659-0. DOI: http://dx.doi.org/10.1109/ICDCS.2009.35.

[BH07] Sven Bittner and Annika Hinze. "The Arbitrary Boolean Publish/Subscribe Model: Making the Case". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 226–237. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266938. URL: http://doi.acm.org/10.1145/1266894.1266938.

[Bri+08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. "Speculative out-of-order event processing with software transaction memory". In: *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. Rome, Italy: ACM, 2008, pp. 265–275. ISBN: 978-1-60558-090-6. DOI: http://doi.acm.org/10.1145/1385989.1386023.

[Bri+11] Andrey Brito, André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. "Scalable and Low-Latency Data Processing with StreamMapReduce". In: *3rd IEEE International Conference on Cloud Computing Technology and Science*. Athens, Greece: IEEE Computer Society, Nov. 2011, pp. 48–58. DOI: 10.1109/CloudCom.2011.17.

[Bru09] Roberto Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley Publishing, 2009. ISBN: 0470517069, 9780470517062.

[CF08] Raphael Chand and Pascal Felber. "Scalable Distribution of XML Content with XNet". In: *IEEE Trans. Parallel Distrib. Syst.* 19.4 (Apr. 2008), pp. 447–461. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.70816. URL: http://dx.doi.org/10.1109/TPDS.2007.70816.

[CGB10] Sunoh Choi, Gabriel Ghinita, and Elisa Bertino. "A Privacy-enhancing Content-based Publish/Subscribe System Using Scalar Product Preserving Transformations". In: *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I*. DEXA'10. Bilbao, Spain: Springer-Verlag, 2010, pp. 368–384. ISBN: 3-642-15363-1, 978-3-642-15363-1. URL: http://dl.acm.org/citation.cfm?id=1881867.1881908.

[Cha+02] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. "Efficient Filtering of XML Documents with XPath Expressions". In: vol. 11. 4. Secaucus, NJ, USA: Springer-Verlag New York, Inc., Dec. 2002, pp. 354–379. DOI: 10.1007/s00778-002-0077-6. URL: http://dx.doi.org/10.1007/s00778-002-0077-6.

[Cha+08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. "SCOPE: easy and efficient parallel processing of massive data sets". In: *Proc. VLDB Endow.* 1 (2 Aug. 2008), pp. 1265–1276. ISSN: 2150-8097. DOI: http://dx.doi.org/10.1145/1454159.1454166. URL: http://dx.doi.org/10.1145/1454159.1454166.

[Cil+03] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. "Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware". In: *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*. DEBS '03. San Diego, California: ACM, 2003, pp. 1–8. ISBN: 1-58113-843-1. DOI: 10.1145/966618.966631. URL: http://doi.acm.org/10.1145/966618.966631.

[Cir+09] Andreea Cirneci, Stefan Boboc, Catalin Leordeanu, Valentin Cristea, and Cristian Estan. "Netpy: Advanced Network Traffic Monitoring". In: *Proceedings of the 2009 International Conference on Intelligent Networking and Collaborative Systems*. INCOS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 253–254. ISBN: 978-0-7695-3858-7. DOI: 10.1109/INCOS.2009.13. URL: http://dx.doi.org/10.1109/INCOS.2009.13.

[CJ10a] Alex King Yeung Cheung and Hans-Arno Jacobsen. "Load Balancing Content-Based Publish/Subscribe Systems". In: *ACM Trans. Comput. Syst.* 28.4 (Dec. 2010), 9:1–9:55. ISSN: 0734-2071. DOI: 10.1145/1880018.1880020. URL: http://doi.acm.org/10.1145/1880018.1880020.

[CJ10b] Alex King Yeung Cheung and Hans-Arno Jacobsen. "Publisher Placement Algorithms in Content-Based Publish/Subscribe". In: *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*. ICDCS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 653–664. ISBN: 978-0-7695-4059-7. DOI: 10.1109/ICDCS.2010.86. URL: http://dx.doi.org/10.1109/ICDCS.2010.86.

[CJ11] Alex King Yeung Cheung and Hans-Arno Jacobsen. "Green Resource Allocation Algorithms for Publish/Subscribe Systems". In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 812–823. ISBN: 978-0-7695-4364-2. DOI: 10.1109/ICDCS.2011.82. URL: http://dx.doi.org/10.1109/ICDCS.2011.82.

[CJ12] Zhuhua Cai and Christopher Jermaine. "The Latent Community Model for Detecting Sybils in Social Networks". In: *Proceedings of the 19th Annual Network & Distributed System Security Symposium*. San Diego, CA, USA, Feb. 2012.

[CO11] Eric Chien and Gavin O'Gorman. *The Nitro Attacks: Stealing Secrets from the Chemical Industry*. www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_nitro_attacks.pdf. July 2011.

[Con+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. "MapReduce online". In: *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*. San Jose, California: USENIX Association, 2010, pp. 21–21.

[Cos+05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. "Vigilante: End-to-end Containment of Internet Worms". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 133–147. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095824. URL: http://doi.acm.org/10.1145/1095810.1095824.

[CPV01] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. "Communities of Interest". In: *Advances in Intelligent Data Analysis*. Ed. by Frank Hoffmann, David Hand, Niall Adams, Douglas Fisher, and Gabriela Guimaraes. Vol. 2189. Lecture Notes in Computer Science. 10.1007/3-540-44816-0_11. Springer Berlin / Heidelberg, 2001, pp. 105–114. URL: http://dx.doi.org/10.1007/3-540-44816-0%5C_11.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. "Design and Evaluation of a Wide-area Event Notification Service". In: *ACM Trans. Comput. Syst.* 19.3 (Aug. 2001), pp. 332–383. ISSN: 0734-2071. DOI: 10.1145/380749.380767. URL: http://doi.acm.org/10.1145/380749.380767.

[CW03] Antonio Carzaniga and Alexander L. Wolf. "Forwarding in a Content-based Network". In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: ACM, 2003, pp. 163–174. ISBN: 1-58113-735-4. DOI: 10.1145/863955.863975. URL: http://doi.acm.org/10.1145/863955.863975.

[DAm+07] Anita D. D'Amico, John R. Goodall, Daniel R. Tesone, and Jason K. Kopylec. "Visual Discovery in Computer Network Defense". In: *IEEE Comput. Graph. Appl.* 27.5 (Sept. 2007), pp. 20–27. ISSN: 0272-1716. DOI: 10.1109/MCG.2007.137. URL: http://dx.doi.org/10.1109/MCG.2007.137.

[Den01] Dorothy E Denning. "Activism, hacktivism, and cyberterrorism: the Internet as a tool for influencing foreign policy". In: *Networks and netwars: The future of terror, crime, and militancy* 239 (2001), p. 288.

[Den87] D.E. Denning. "An Intrusion-Detection Model". In: *Software Engineering, IEEE Transactions on* SE-13.2 (Feb. 1987), pp. 222–232. ISSN: 0098-5589. DOI: 10.1109/TSE.1987.232894.

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[Dha+03] Sarang Dharmapurikar, Praveen Krishnamurthy, T. Sproull, and J. Lockwood. "Deep packet inspection using parallel Bloom filters". In: *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*. Aug. 2003, pp. 44–51. DOI: 10.1109/CONECT.2003.1231477.

[Din+12] Qi Ding, Natallia Katenka, Paul Barford, Eric Kolaczyk, and Mark Crovella. "Intrusion As (Anti)Social Communication: Characterization and Detection". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 886–894. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339670. URL: http://doi.acm.org/10.1145/2339530.2339670.

[Don10] Debora Donato. "Graph Structures and Algorithms for Query-Log Analysis". In: *Programs, Proofs, Processes*. Ed. by Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes. Vol. 6158. Lecture Notes in Computer Science. 10.1007/978-3-642-13962-8_14. Springer Berlin / Heidelberg, 2010, pp. 126–131. URL: http://dx.doi.org/10.1007/978-3-642-13962-8_14.

[Ell+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. "Graphviz–Open Source Graph Drawing Tools". In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. 10.1007/3-540-45848-4_57. Springer Berlin / Heidelberg, 2002, pp. 594–597. URL: http://dx.doi.org/10.1007/3-540-45848-4_57.

[Eri+03] Kasper Astrup Eriksen, Ingve Simonsen, Sergei Maslov, and Kim Sneppen. "Modularity and Extreme Edges of the Internet". In: *Phys. Rev. Lett.* 90 (14 Apr. 2003), p. 148701. DOI: 10.1103/PhysRevLett.90.148701. URL: http://link.aps.org/doi/10.1103/PhysRevLett.90.148701.

[Eug+03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[Fab+01] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 115–126. ISBN: 1-58113-332-4. DOI: 10.1145/375663.375677. URL: http://doi.acm.org/10.1145/375663.375677.

[Far+09] Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen. "Parallel Event Processing for Content-based Publish/Subscribe Systems". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. Nashville, Tennessee: ACM, 2009, 8:1–8:4. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619269. URL: http://doi.acm.org/10.1145/1619258.1619269.

[FBO09] Alexandre Francisco, Ricardo Baeza-Yates, and Arlindo Oliveira. "Clique Analysis of Query Log Graphs". In: *String Processing and Information Retrieval*. Ed. by Amihood Amir, Andrew Turpin, and Alistair Moffat. Vol. 5280. Lecture Notes in Computer Science. 10.1007/978-3-540-89097-3_19. Springer Berlin / Heidelberg, 2009, pp. 188–199. URL: http://dx.doi.org/10.1007/978-3-540-89097-3_19.

[Fin+14] Jim Finkle, Mark Hosenball, Andrea Shalal, and Christian Plumb. *Exclusive: Iran hackers may target U.S. energy, defense firms, FBI warns.* http://www.reuters.com/article/2014/12/13/us-cybersecurity-iran-fbi-idUSKBN0JQ28Z20141213. Dec. 2014.

[Fis+08] Fabian Fischer, Florian Mansmann, Daniel A. Keim, Stephan Pietzko, and Marcel Waldvogel. "Large-Scale Network Monitoring for Visual Analysis of Attacks". In: *Proceedings of the 5th International Workshop on Visualization for Computer Security*. VizSec '08. Cambridge, MA, USA: Springer-Verlag, 2008, pp. 111–118. ISBN: 978-3-540-85931-4. DOI: 10.1007/978-3-540-85933-8_11. URL: http://dx.doi.org/10.1007/978-3-540-85933-8_11.

[Fis12] Dennis Fisher. *Attacks Targeting US Defense Contractors and Universities Tied to China.* threatpost.com/attacks-targeting-us-defense-contractors-and-universities-tied-china-061312/76685. June 2012.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. *W32.Stuxnet Dossier.* www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf. Feb. 2011.

[Fon+10] Marcus Fontoura, Suhas Sadanandan, Jayavel Shanmugasundaram, Sergei Vassilvitski, Erik Vee, Srihari Venkatesan, and Jason Zien. "Efficiently Evaluating Complex Boolean Expressions". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 3–14. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807171. URL: http://doi.acm.org/10.1145/1807167.1807171.

[Fou] Apache Software Foundation. *Storm: Distributed and fault-tolerant realtime computation*. http://storm.apache.org.

[Fou10] Apache Software Foundation. *Hadoop*. http://hadoop.apache.org/. Jan. 2010.

[Gat+04] Carrie Gates, Michael Collins, Michael Duggan, Andrew Kompanek, and Mark Thomas. "More Netflow Tools for Performance and Security". In: *Proceedings of the 18th USENIX Conference on System Administration*. LISA '04. Atlanta, GA: USENIX Association, 2004, pp. 121–132. URL: http://dl.acm.org/citation.cfm?id=1052676.1052691.

[Gon+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387883.

[Gon+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. "Graphx: Graph processing in a distributed dataflow framework". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[GS12] David F. Gleich and C. Seshadhri. "Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 597–605. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339628. URL: http://doi.acm.org/10.1145/2339530.2339628.

[Gu+08] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-independent Botnet Detection". In: *Proceedings of the 17th Conference on Security Symposium*. SS'08. San Jose, CA: USENIX Association, 2008, pp. 139–154. URL: http://dl.acm.org/citation.cfm?id=1496711.1496721.

[Gup+04] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. "Meghdoot: Content-based Publish/Subscribe over P2P Networks". In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 254–273. ISBN: 3-540-23428-4. URL: http://dl.acm.org/citation.cfm?id=1045658.1045677.

[Gup+12] Manish Gupta, Jing Gao, Yizhou Sun, and Jiawei Han. "Integrating Community Matching and Outlier Detection for Mining Evolutionary Community Outliers". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 859–867. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339667. URL: http://doi.acm.org/10.1145/2339530.2339667.

[Haa05] P. Haag. "Watch your Flows with NfSen and NFDUMP". In: *50th RIPE Meeting*. 2005.

[Hae+10] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. "Accountable Virtual Machines". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–16. URL: http://dl.acm.org/citation.cfm?id=1924943.1924952.

[Hal95] Lawrence R. Halme. "AIN'T Misbehaving – A Taxonomy of Anti-Intrusion Techniques". eng. In: *Computers & Security* 14.7 (1995), p. 606.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. "PeerReview: Practical Accountability for Distributed Systems". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 175–188. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294279. URL: http://doi.acm.org/10.1145/1294261.1294279.

[Ho+08] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. "Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, pp. 175–188. ISBN: 111-999-5555-22-1. URL: http://dl.acm.org/citation.cfm?id=1387589.1387602.

[Hon12] Jason Hong. "The State of Phishing Attacks". In: *Commun. ACM* 55.1 (Jan. 2012), pp. 74–81. ISSN: 0001-0782. DOI: 10.1145/2063176.2063197. URL: http://doi.acm.org/10.1145/2063176.2063197.

[III+05] William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. "Finding strongly connected components in distributed graphs". In: *Journal of Parallel and Distributed Computing* 65.8 (2005), pp. 901–910. ISSN: 0743-7315. DOI: DOI:10.1016/j.jpdc.2005.03.007. URL: http://www.sciencedirect.com/science/article/pii/S0743731505000535.

[ILW06] Vinay M. Igure, Sean A. Laughter, and Ronald D. Williams. "Security Issues in SCADA Networks". In: *Comput. Secur.* 25.7 (Oct. 2006), pp. 498–506. ISSN: 0167-4048. DOI: 10.1016/j.cose.2006.03.001. URL: http://dx.doi.org/10.1016/j.cose.2006.03.001.

[IR05] Milena Ivanova and Tore Risch. "Customizable Parallel Execution of Scientific Stream Queries". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 157–168. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083614.

[IRC10a] Mihaela Ion, Giovanni Russello, and Bruno Crispo. "Supporting Publication and Subscription Confidentiality in Pub/Sub Networks". English. In: *Security and Privacy in Communication Networks*. Ed. by Sushil Jajodia and Jianying Zhou. Vol. 50. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2010, pp. 272–289. ISBN: 978-3-642-16160-5. DOI: 10.1007/978-3-642-16161-2_16. URL: http://dx.doi.org/10.1007/978-3-642-16161-2_16.

[IRC10b] Mihalea Ion, Giovanni Russello, and Bruno Crispo. "An Implementation of Event and Filter Confidentiality in Pub/Sub Systems and Its Application to e-Health". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 696–698. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866401. URL: http://doi.acm.org/10.1145/1866307.1866401.

[Isa+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: http://doi.acm.org/10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[Jac+09] H.-A. Jacobsen, A. Cheung, G. Lia, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. "The PADRES Publish/Subscribe System". In: *Handbook of Research on Adv. Dist. Event-Based Sys., Pub./Sub. and Message Filtering Tech.* 2009.

[JE11] K. R. Jayaram and Patrick Eugster. "Split and Subsume: Subscription Normalization for Effective Content-Based Messaging". In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 824–835. ISBN: 978-0-7695-4364-2. DOI: 10.1109/ICDCS.2011.85. URL: http://dx.doi.org/10.1109/ICDCS.2011.85.

[JF08] Zbigniew Jerzak and Christof Fetzer. "Bloom Filter Based Routing for Content-based Publish/Subscribe". In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS '08. Rome, Italy: ACM, 2008, pp. 71–81. ISBN: 978-1-60558-090-6. DOI: 10.1145/1385989.1385999. URL: http://doi.acm.org/10.1145/1385989.1385999.

[JY12] Ari Juels and Ting-Fang Yen. "Sherlock Holmes and the Case of the Advanced Persistent Threat". In: *Proceedings of the 5th USENIX Conference on Large-Scale Exploits and Emergent Threats*. LEET'12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228340.2228343.

[Kal+05] Satyen Kale, Elad Hazan, Fengyun Cao, and Jaswinder Pal Singh. "Analysis and Algorithms for Content-Based Event Matching". In: *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05) - Volume 04*. ICDCSW '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 363–369. ISBN: 0-7695-2328-5-04. DOI: 10.1109/ICDCSW.2005.40. URL: http://dx.doi.org/10.1109/ICDCSW.2005.40.

[KJ12] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. "Opportunistic Multipath Forwarding in Content-based Publish/Subscribe Overlays". In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag New York, Inc., 2012, pp. 249–270. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442643.

[Lab13] Kaspersky Labs. *Winnti — More than just a game*. kasperskycontenthub.com/wp-content/uploads/sites/43/vlpdfs/winnti-more-than-just-a-game-130410.pdf. Apr. 2013.

[Lab14] Kaspersky Labs. *Unveiling Careto — The Masked APT*. www.securelist.com/en/downloads/vlpdfs/unveilingthemask_v1.0.pdf. Feb. 2014.

[Lab15] Kaspersky Labs. *Carbanak APT The Great Bank Robbery*. https://securelist.com/files/2015/02/Carbanak_APT_eng.pdf. Feb. 2015.

[Les+08] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. "Statistical Properties of Community Structure in Large Social and Information Networks". In: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. Beijing, China: ACM, 2008, pp. 695–704. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367591. URL: http://doi.acm.org/10.1145/1367497.1367591.

[Leu+09] Ian X. Y. Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. "Towards real-time community detection in large networks". In: *Phys. Rev. E* 79 (6 June 2009), p. 066107. DOI: 10.1103/PhysRevE.79.066107. URL: http://link.aps.org/doi/10.1103/PhysRevE.79.066107.

[Lev+09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 1–14. URL: http://dl.acm.org/citation.cfm?id=1558977.1558978.

[Li+07] G. Li, A. Cheung, Sh. Hou, S. Hu, V. Muthusamy, R. Sherafat, A. Wun, H.-A. Jacobsen, and S. Manovski. "Historic Data Access in Publish/Subscribe". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 80–84. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266908. URL: http://doi.acm.org/10.1145/1266894.1266908.

[Li+10] Zhichun Li, Gao Xia, Hongyu Gao, Yi Tang, Yan Chen, Bin Liu, Junchen Jiang, and Yuezhou Lv. "NetShield: Massive Semantics-based Vulnerability Signature Matching for High-speed Networks". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 279–290. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851216. URL: http://doi.acm.org/10.1145/1851182.1851216.

[Li+12] Wei Li, Songlin Hu, Jintao Li, and Hans-Arno Jacobsen. "Community Clustering for Distributed Publish/Subscribe Systems". In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing*. CLUSTER '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 81–89. ISBN: 978-0-7695-4807-4. DOI: 10.1109/CLUSTER.2012.67. URL: http://dx.doi.org/10.1109/CLUSTER.2012.67.

[Lib07] Ben Liblit. *Cooperative bug isolation: winning thesis of the 2005 ACM doctoral dissertation competition*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 978-3-540-71877-2.

[Log+10] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. "Stateful bulk processing for incremental analytics". In: *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*. Indianapolis, Indiana, USA: ACM, 2010, pp. 51–62. ISBN: 978-1-4503-0036-0. DOI: http://doi.acm.org/10.1145/1807128.1807138.

[Low+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354. URL: http://dx.doi.org/10.14778/2212351.2212354.

[Mal+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184.

[Mar+11] André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, Andrey Brito, and Christof Fetzer. "Low-overhead fault tolerance for high-throughput data processing systems". In: *ICDCS '11: Proceedings of the 2011 31st IEEE International Conference on Distributed Computing Systems*. Minneapolis, Minnesota, USA: IEEE Computer Society, June 2011, pp. 689–699. DOI: 10.1109/ICDCS.2011.29.

[McD+06] Patrick Drew McDaniel, Subhabrata Sen, Oliver Spatscheck, Jacobus E. van der Merwe, William Aiello, and Charles R. Kalmanek. "Enterprise Security: A Community of Interest Based Approach". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*. 2006. URL: http://www.isoc.org/isoc/conferences/ndss/06/proceedings/papers/enterprise_security.pdf.

[Mis+07a] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert E. Tarjan. "Clustering Social Networks". In: *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*. WAW'07. San Diego, CA, USA: Springer-Verlag, 2007, pp. 56–67. ISBN: 3-540-77003-8, 978-3-540-77003-9. URL: http://dl.acm.org/citation.cfm?id=1777879.1777884.

[Mis+07b] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. "Measurement and Analysis of Online Social Networks". In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. IMC '07. San Diego, California, USA: ACM, 2007, pp. 29–42. ISBN: 978-1-59593-908-1. DOI: 10.1145/1298306.1298311. URL: http://doi.acm.org/10.1145/1298306.1298311.

[MS09] Andrew Mehler and Steven Skiena. "Expanding Network Communities from Representative Examples". In: *ACM Trans. Knowl. Discov. Data* 3.2 (Apr. 2009), 7:1–7:27. ISSN: 1556-4681. DOI: 10.1145/1514888.1514890. URL: http://doi.acm.org/10.1145/1514888.1514890.

[Neu+10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed Stream Computing Platform". In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177. ISBN: 978-0-7695-4257-7. DOI: 10.1109/ICDMW.2010.172. URL: http://dx.doi.org/10.1109/ICDMW.2010.172.

[OKA10] AdamJ. Oliner, AshutoshV. Kulkarni, and Alex Aiken. "Community Epidemic Detection Using Time-Correlated Anomalies". In: ed. by Somesh Jha, Robin Sommer, and Christian Kreibich. Vol. 6307. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 360–381.

[Pag+99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Tech. rep. Stanford InfoLab, 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768.

[PD10] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–15. URL: http://dl.acm.org/citation.cfm?id=1924943.1924961.

[Pen+03] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. "Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251353.1251363.

[Per+09] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. Ernst, and M. Rinard. "Self-defending software: Automatically patching security vulnerabilities". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[Plo00] Dave Plonka. "FlowScan: A Network Traffic Flow Reporting and Visualization Tool". In: *Proceedings of the 14th USENIX Conference on System Administration*. LISA '00. New Orleans, Louisiana: USENIX Association, 2000, pp. 305–318. URL: http://dl.acm.org/citation.cfm?id=1045502.1045522.

[PPD13] Andreas Papadopoulos, George Pallis, and Marios D. Dikaiakos. "Identifying Clusters with Attribute Homogeneity and Similar Connectivity in Information Networks". In: *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 01*. WI-IAT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 343–350. ISBN: 978-0-7695-5145-6. DOI: 10.1109/WI-IAT.2013.49. URL: http://dx.doi.org/10.1109/WI-IAT.2013.49.

[Qia+09] Feng Qian, Zhiyun Qian, Z.Morley Mao, and Atul Prakash. "Ensemble: Community-Based Anomaly Detection for Popular Applications". English. In: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 19 (2009). Ed. by Yan Chen, TassosD. Dimitriou, and Jianying Zhou, pp. 163–184. DOI: 10.1007/978-3-642-05284-2_10. URL: http://dx.doi.org/10.1007/978-3-642-05284-2_10.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Phys. Rev. E* 76 (3 Sept. 2007), p. 036106. DOI: 10.1103/PhysRevE.76.036106. URL: http://link.aps.org/doi/10.1103/PhysRevE.76.036106.

[Rau+11] Matthieu Pélissié du Rausas, James Manyika, Eric Hazan, Jacques Bughin, Michael Chui, and Rémi Said. *Internet matters: The Net's sweeping impact on growth, jobs, and prosperity*. McKinsey & Company, 2011.

[Reu09] J. Reumann. *Pub/Sub at Google*. CANOE and EuroSys Summer School. 2009.

[Roe99] Martin Roesch. "Snort - Lightweight Intrusion Detection for Networks". In: *Proceedings of the 13th USENIX Conference on System Administration*. LISA '99. Seattle, Washington: USENIX Association, 1999, pp. 229–238. URL: http://dl.acm.org/citation.cfm?id=1039834.1039864.

[Rom+11] Luigi Romano, Danilo De Mari, Zbigniew Jerzak, and Christof Fetzer. "A Novel Approach to QoS Monitoring in the Cloud". In: *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*. CCP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 45–51. ISBN: 978-0-7695-4528-8. DOI: 10.1109/CCP.2011.49. URL: http://dx.doi.org/10.1109/CCP.2011.49.

[Rom00] Steve Romig. "The OSU Flow-tools Package and CISCO NetFlow Logs". In: *Proceedings of the 14th USENIX Conference on System Administration*. LISA '00. New Orleans, Louisiana: USENIX Association, 2000, pp. 291–304. URL: http://dl.acm.org/citation.cfm?id=1045502.1045521.

[RR06] C. Raiciu and D.S. Rosenblum. "Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures". In: *Securecomm and Workshops, 2006*. Aug. 2006, pp. 1–11. DOI: 10.1109/SECCOMW.2006.359552.

[RS98] M. Reilly and M. Stillman. "Open infrastructure for scalable intrusion detection". In: *Information Technology Conference, 1998. IEEE*. Sept. 1998, pp. 129–133. DOI: 10.1109/IT.1998.713398.

[Sah+98] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. "A Bayesian Approach to Filtering Junk E-Mail". In: *Learning for Text Categorization: Papers from the 1998 Workshop*. Madison, Wisconsin: AAAI Technical Report WS-98-05, 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.1254.

[SE12] Chris Strohm and Eric Engleman. *Cyber Attacks on US Banks Expose Computer Vulnerability*. www.bloomberg.com/news/2012-09-28/cyber-attacks-on-u-s-banks-expose-computer-vulnerability.html. Sept. 2012.

[Sol09] C. So-In. *A Survey of Network Traffic Monitoring and Analysis Tools*. CSE 576M Computer System Analysis Project. Washington University in St. Louis, 2009.

[SP10] Robin Sommer and Vern Paxson. "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 305–316. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.25. URL: http://dx.doi.org/10.1109/SP.2010.25.

[Tae99] Gabriele Taentzer. "Distributed Graphs and Graph Transformation". In: *Applied Categorical Structures* 7 (4 1999). 10.1023/A:1008683005045, pp. 431–462. ISSN: 0927-2852. URL: http://dx.doi.org/10.1023/A:1008683005045.

[Tay+09] Teryl Taylor, Diana Paterson, Joel Glanfield, Carrie Gates, Stephen Brooks, and John McHugh. "FloVis: Flow Visualization System". In: *Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security*. CATCH '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 186–198. ISBN: 978-0-7695-3568-5. DOI: 10.1109/CATCH.2009.18. URL: http://dx.doi.org/10.1109/CATCH.2009.18.

[TÇZ07] N. Tatbul, U. Çetintemel, and S. Zdonik. "Staying FIT: efficient load shedding techniques for distributed stream processing". In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. Vienna, Austria: VLDB Endowment, 2007, pp. 159–170. ISBN: 978-1-59593-649-3.

[Tsi+13] Alexander Tsiatas, Iraj Saniee, Onuttom Narayan, and Matthew Andrews. "Spectral Analysis of Communication Networks Using Dirichlet Eigenvalues". In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW '13. Rio de Janeiro, Brazil: International World Wide Web Conferences Steering Committee, 2013, pp. 1297–1306. ISBN: 978-1-4503-2035-1. URL: http://dl.acm.org/citation.cfm?id=2488388.2488501.

[Ver+06] Patrick Verkaik, Oliver Spatscheck, Jacobus Van der Merwe, and Alex C. Snoeren. "PRIMED: Community-of-interest-based DDoS Mitigation". In: *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense*. LSAD '06. Pisa, Italy: ACM, 2006, pp. 147–154. ISBN: 1-59593-571-1. DOI: 10.1145/1162666.1162673. URL: http://doi.acm.org/10.1145/1162666.1162673.

[VG13] N. Virvilis and D. Gritzalis. "The Big Four - What We Did Wrong in Advanced Persistent Threat Detection?" In: *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. Sept. 2013, pp. 248–254. DOI: 10.1109/ARES.2013.32.

[Wan+02] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. "Subscription Partitioning and Routing in Content-based Publish/Subscribe Networks". In: *DISC '02: 16th International Symposium on Distributed Computing*. Springer-Verlag, Oct. 2002.

[Wan+04a] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. "Shield: Vulnerability-driven Network Filters for Preventing Known Vulnerability Exploits". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '04. Portland, Oregon, USA: ACM, 2004, pp. 193–204. ISBN: 1-58113-862-8. DOI: 10.1145/1015467.1015489. URL: http://doi.acm.org/10.1145/1015467.1015489.

[Wan+04b] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. "Automatic Misconfiguration Troubleshooting with Peerpressure". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 17–17. URL: http://dl.acm.org/citation.cfm?id=1251254.1251271.

[WCR09] Benjamin Wun, Patrick Crowley, and Arun Raghunth. "Parallelization of Snort on a Multi-core Platform". In: *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '09. Princeton, New Jersey: ACM, 2009, pp. 173–174. ISBN: 978-1-60558-630-4. DOI: 10.1145/1882486.1882528. URL: http://doi.acm.org/10.1145/1882486.1882528.

[WHF11a] Stefan Weigert, Matti A. Hiltunen, and Christof Fetzer. "Community-based Analysis of Netflow for Early Detection of Security Incidents". In: *Proceedings of the 25th International Conference on Large Installation System Administration*. LISA'11. Boston, MA: USENIX Association, 2011, pp. 20–20. URL: http://dl.acm.org/citation.cfm?id=2208488.2208508.

[WHF11b] Stefan Weigert, Matti A. Hiltunen, and Christof Fetzer. "Mining Large Distributed Log Data in Near Real Time". In: *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. SLAML '11. Cascais, Portugal: ACM, 2011, 5:1–5:8. ISBN: 978-1-4503-0978-3. DOI: 10.1145/2038633.2038638. URL: http://doi.acm.org/10.1145/2038633.2038638.

[WHF14] Stefan Weigert, Matti A. Hiltunen, and Christof Fetzer. "Finding the Needle in the Haystack: Identifying Business Communities in Internet Traffic". In: *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*. Vol. 1. Aug. 2014, pp. 175–182. DOI: 10.1109/WI-IAT.2014.31.

[WHW07] D.W. Williams, Jun Huan, and Wei Wang. "Graph Database Indexing Using Structured Graph Decomposition". In: *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. Apr. 2007, pp. 976–985. DOI: 10.1109/ICDE.2007.368956.

[Wil13] Gregory C. Wilshusen. *A Better Defined and Implemented National Strategy Is Needed to Address Persistent Challenges*. U.S. Government Accountability Office, GAO-13-462T, Mar. 2013. URL: http://www.gao.gov/assets/660/652817.pdf.

[WKM00] T. Wong, R. Katz, and S. McCanne. "An evaluation of preference clustering in large-scale multicast applications". In: *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 2. 2000, 451–460 vol.2. DOI: 10.1109/INFCOM.2000.832218.

[Wue14] Candid Wueest. *Targeted Attacks Against the Energy Sector*. www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/targeted_attacks_against_the_energy_sector.pdf. Jan. 2014.

[WW13] Jons-Tobias Wamhoff and Stefan Weigert. "DREAM: Dresden Streaming Transactional Memory Benchmark". In: *TRANSACT 13*. Houston, TX, USA: ACM New York, NY, USA, Mar. 2013.

[WY85] Andrew K. C. Wong and Manlai You. "Entropy and Distance of Random Graphs with Application to Structural Pattern Recognition". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-7.5 (Sept. 1985), pp. 599–609. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1985.4767707.

[Yah] Yahoo. http://finance.yahoo.com/.

[YD03] Dit-Yan Yeung and Yuxin Ding. "Host-based intrusion detection using dynamic and static behavioral models". In: *Pattern Recognition* 36.1 (2003), pp. 229–243. ISSN: 0031-3203. DOI: http://dx.doi.org/10.1016/S0031-3203(02)00026-2. URL: http://www.sciencedirect.com/science/article/pii/S0031320302000262.

[YL15] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth". English. In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213. ISSN: 0219-1377. DOI: 10.1007/s10115-013-0693-z. URL: http://dx.doi.org/10.1007/s10115-013-0693-z.

[YMJ11] Young Yoon, Vinod Muthusamy, and Hans-Arno Jacobsen. "Foundations for Highly Available Content-Based Publish/Subscribe Overlays". In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 800–811. ISBN: 978-0-7695-4364-2. DOI: 10.1109/ICDCS.2011.93. URL: http://dx.doi.org/10.1109/ICDCS.2011.93.

[Yur06] W. Yurcik. "VisFlowConnect-IP: a link-based visualization of Netflows for security monitoring". In: *18th Annual FIRST Conf. on Computer Security Incident Handling*. 2006.

[ZHS10] Yuanyuan Zeng, Xin Hu, and K.G. Shin. "Detection of botnets using combined host- and network-level information". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. June 2010, pp. 291–300. DOI: 10.1109/DSN.2010.5544306.

[ZLK10] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. "A survey of coordinated attacks and collaborative intrusion detection". In: *Computers & Security* 29.1 (2010), pp. 124–140. ISSN: 0167-4048. DOI: http://dx.doi.org/10.1016/j.cose.2009.06.008. URL: http://www.sciencedirect.com/science/article/pii/S016740480900073X.

[ZPU08] Jian Zhang, Phillip Porras, and Johannes Ullrich. "Highly Predictive Blacklisting". In: *Proceedings of the 17th Conference on Security Symposium*. SS'08. San Jose, CA: USENIX Association, 2008, pp. 107–122. URL: http://dl.acm.org/citation.cfm?id=1496711.1496719.

[ZW11] Yaxiong Zhao and Jie Wu. "Towards Approximate Event Processing in a Large-Scale Content-Based Network". In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 790–799. ISBN: 978-0-7695-4364-2. DOI: 10.1109/ICDCS.2011.67. URL: http://dx.doi.org/10.1109/ICDCS.2011.67.

LISTS OF FIGURES, TABLES AND ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS