University of Twente

Master Thesis

Flow-based SSH Dictionary Attack Detection: the Effects of Aggregation

Mattijs Jonker

Date:	August 27, 2014
Committee:	prof. dr. ir. Aiko Pras
	Rick Hofstede, M.Sc.
	dr. Anna Sperotto
Faculty:	Electrical Engineering, Mathematics and Computer Science (EEMCS)
Chair:	Design and Analysis of Communication Systems (DACS)

Contents

Chapter 1

Introduction

While this document is titled *Master Thesis*, it is not a thesis in the traditional sense. This is because, in close agreement with the graduation committee, the research conducted for this Master assignment is reported on in a paper. This paper is to be submitted to an established and well-known conference in the field of computer science, namely *the 14th IFIP/IEEE Symposium on Integrated Network and Service Management* (IM 2015)[1]. *IEEE IM* is a conference to which the DACS chair frequently submits work. It has an average *SCImago Journal Rank* (SJR) of *0.596* over the course of the last three years[2]. This puts the conference well in line with other frequently chosen conferences in terms of quality and reputation.

A paper is more of a collaborative effort than would be a traditional Master Thesis when it comes to structure, wording, and the way in which results are presented and emphasized. Notwithstanding, I have been closely involved in every step of the research – that is, to put it broadly, its formulation, planning, and execution. This includes the planning of dataset collection and analysis, the evaluation of results and validation of work, as well as many cycles of paper writing and reviewing. Furthermore, independent efforts have been put in development of software that was an absolute necessity for data collection, which adds an industry-oriented element to the Master assignment. The choice for a paper thus does not in any way negatively impact the process in which all requirements for the Master Degree are met, but rather offered me the opportunity to do so in the common academic setting in which research is shared with other researchers all over the world.

This thesis outlines the Master assignment and explains how all requirements pertaining to quality are met, and briefly discusses the pending conference submission. As the to be published paper works out in detail the research, discussion related to its contents will, by design, be brief. How-

[1]`http://im2015.ieee-im.org/`.

[2]Information obtained from `http://scopus.com/source/eval.url` for *2011-2013*.

ever, as not all work is included in the paper, yet has yielded specific and promising results, part of this thesis will be dedicated to describing the work that did not make it into the publication. The remainder of this chapter will discuss the assignment, outline the research topics, and furthermore provide an overview as to how the rest of this thesis is organized.

1.1 Assignment

Getting a Master Degree involves gaining advanced knowledge, and showing a mastery of a specific field of study. The final part of fulfilling a Master programme typically comes as a final project, in which a student is expected to demonstrate advanced knowledge, analytical skills, the ability to think critically, and to be able to work independently in solving complex problems. Very often one shows that all requirements to obtain the degree have been met in a Master Thesis. The *EEMCS* faculty of the University of Twente has published in [1] a list of criteria for the Computer Science Master programmes. These criteria are key in earning a satisfactory result, and thus in successfully completing a programme. As such, they would come forward in a typical Master Thesis. Since the traditional thesis has in my case, in close agreement with the graduation committee, made way for a paper, these criteria must be discussed to ensure an equal level of quality is met by the paper. This section describes how every aspect of coming to the paper offered ample opportunity to meet the key criteria, and serves to show how this opportunity was seized accordingly.

The criteria are listed next, each with a justification of how the particular criterium has been met:

Clearly formulate a problem statement

Before starting the Master assigmnent, a separate *Research Topics* course has been completed. This undertaking resulted in identifying a few areas based on literature in which contributions to the field could be made. After brainstorming with, and receiving constructive feedback from supervisors, an initial direction for the Master assignment was chosen. This choice was followed by the formalization of several research questions, a research approach and the projected contributions. These efforts shaped and directed the research from the get-go, and as such are to be considered a concise formulation of the problem statement.

Identify relevant literature

After the problem statement had been clearly formulated, directions for further literature research became apparent that had not already been linked to a chosen problematic area as part of the *Research Topics*. This included not only related work of research that contains approaches from which lessons could be derived, but also literature in the form of protocol standards, and reports the operation of protocols in practice. The theoretic background fundamental to parts of the research has been addressed accordingly. Also, while the paper does not contain a dedicated *related work* section as is often the case [2], relevant related works are mentioned as part of its background discussion. Finally, over the course of the Master assignment, the identified literature helped in making adjustments based on intermediary results, when necessary.

Draw up a work plan

From the get-go, the end of the school year formed a deadline for the research. This is in part because this ending is just two weeks ahead of the submission deadline of the chosen conference. Key tasks were apparent in the early stages. This includes, but is not limited to: further exploration of literature; software development; extended periods of data collection, monitoring and sampling to identify the possible need for adjustments; and the projected time required for analysis, evaluation and going through cycles of writing and reviewing the paper. Considering these tasks and the projected deadline, a preliminary work plan was drawnn up in agreement with the supervisors.

Adjust problem statement and work schedule in accordance with interim evaluations

Several large datasets had to be collected in support of the research. Preliminary analysis was possible as data became available. This led to adjustments of the problem statement and work schedule for two reasons. Firstly, based on early analysis, changes needed in the data collection process became apparent. This was followed by modifications to the data collection software, as well as alterations to the planned collection periods. Secondly, some early results stood out in terms of being contributive. Based on constructive feedback from the graduation committe, the research questions and approach were adjusted to work even more in-depth towards utilization of these results. Beyond regular planned evaluations with the daily supervisor, an interim colloquium of the ongoing work allowed other members of the DACS chair to provide input, which too allowed for valuable fine-tuning of parts of the research. Unforeseen factors such as a power outage, architecture differences with a third-party data collection point, and bugs in third-party tools also played their part in making required adjustments.

Analyse different possible solutions and motivate a choice between them

During the early stages of research, literature studies and review of related work revealed several possible solutions to the (initial) research questions. Analysis of these solutions led to a narrowing of possibilities and adjustment of the problem statement. Further analysis was made possible by intermediate data and results coming in. This allowed well-thought-out choices to be made over time. Constructive brainstorming sessions with the daily supervisor were seamless in this process. Not all solutions that were passed on or later deemed of lower priority are unpractical per se. Rather, some were found to be unfitting within the research approach, or not suited to be presented alongside other results.

Communicate the research and design activities both written and in presentations

The choice for a paper publication rather than traditional Master thesis is a clear effort to communicate the research and design activities to a broad, world-wide audience of researchers. In a handful of informal evaluation sessions with members of the graduation committee, ongoing research efforts were presented so as to optimize the research process. Furthermore, during many cycles of paper writing and reviewing, research activities were communicated to others at the university. An interim colloquium was also held to communicate the ongoing efforts to other members of the DACS chair, and peer Master students, in the form of a presentation. Finally, should the paper be accepted to *IEEE IM*, a presentation opportunity at the conference arises.

Show the ability of reflection on the problem, on the research/design approach, on the solution and on oneâĂŹs own performance

As has been explained for preceding criteria, interim evaluations were held as preliminary results had become available. This allowed for reflection on the problem and on the research design and approach, and was naturally followed by making changes to the problem statement. Constructive feedback from the daily supervisor during the ongoing research allowed for me to reflect on performance. I set soft deadlines for milestones that lined up with the work plan. Whether or not goals were met in time, allowed for me to reflect on my efficiency. Last but not least, during the many cycles of paper writing and reviewing, iterations of feedback allowed for me to judge whether I interpreted results from all possible angles. Especially for the validation part of the publication, this feedback proved invaluable.

Demonstrate creativity and the ability to work independently

The research was performed full-time at the DACS chair, in an office shared with the daily supervisor. This allowed for the frequent provision of valuable feedback, as well as new ideas and insights. However, during all stages of the research, a lot of independent work and creativity was required. Particular efforts related to dataset creation and analysis. This included background research and design of dataset composition, cycles of small-scale measurements and analysis, and evaluation of preliminary results. Moreover, software development was necessary to be able to gather data to begin with. With the exception of a small feature request for deployment at a third-party, development of this extensive project was done independently. Finally, creativity was involved in independently coming up with ideas for analysis and evaluation.

1.2 Research Topics

A variety of topics had to be researched in support of the work reported on in the paper and this thesis. What follows is a list that outlines these topics. Each topic is briefly described alongside an explanation as to how the topic fits within the research context. Per topic, its relation to other topics is also given.

- **Intrusion Detection**
  This work, in the broadest sense, was performed within the field of intrusion detection – particularly, flow-based intrusion detection. Literature on intrusion detection approaches, systems and more had to be explored as part of this work. This went beyond – yet built on – the *Research Topics* course that was performed prior to starting the Master assignment, and involved specific studies of existing detection algorithms and experiences. All in all, exploring literature on intrusion detection was an essential part of working towards possible solutions and making choices between them. Furthermore, while in the paper the application of SSH comes forward as 'just' a case study, the SSH context was clear from the beginning, as the Master assignment builds on work performed in [3, 4].

- **IP Flow Information Export (IPFIX)**
  Flow monitoring has become the prevalent method approach for monitoring larger-scale, high-speed networks [5]. Example protocols for flow export are IP Flow Information Export (IPFIX) [6] and NetFlow [7]. Intrusion detection is explicitly mentioned as a possibile application of IPFIX [8], and state-of-the-art IDSs have shifted from taking a payload-based approach to working with flows. During early stages of this work, it had already become obvious that IPFIX would be an integral part of the research. The main reason IPFIX was chosen over NetFlow, is that IPFIX allows one to easily extend the set of fields exported during flow export. It was used for dataset creation, collection and analysis, and thereby also allowed the technology to be validated as a solution to the problem statement of the research.

- **Transmission Control Protocol (TCP)**
  A large amount of traffic sent daily over the Internet uses TCP [9], and many networked applications rely on it for data transport. In taking a network-based approach to detecting attacks on such applications, knowledge of TCP is required. The same can be said for flow-based intrusion detection. TCP had to be thoroughly studied for this work. This includes extensions such as congestion control mechanisms [10], *Selective Acknowledgements* (SACK) [11] and the *Eifel algorithm* [12].

- **Flow export configuration, extension and deployment**
  Several large datasets with flow data were fundamental to the research. In order to obtain these datasets, a flow exporter had to be extended, configured and deployed. The elaborate extension had to be designed, developed and repeatedly tested. This required becoming familiar with INVEA-TECH's FlowMon probe architecture [13], because the extension comes as several plugins that work within this architecture. Furthermore, the ins and outs of probe configuration and deployment had to be learned. In addition, detailed knowledge of the inner workings of the *Virtual Appliance* product from INVEA-TECH was required. With this product, development of extensions is made possible in a virtualized environment. All this added an industry-oriented aspect to the research, and led to a thorough understanding of all parts involved.

- **Flow collector configuration and deployment**
  In order to collect large datasets consisting of flow data, a state-of-the-art IPFIX collector had to be configured and deployed in a production setting. *IPFIXcol* [14] was the tool of choice, which stores data in a format specifically useful for academic purposes. Hands-on experience was required with *IPFIXcol*, and additionally others tools used for reading and parsing flow data after storage. This contributed towards the industry-oriented part of the Master assignment.

- **Secure Shell (SSH)**
  Secure Shell (SSH) [15] is one of the most widely used network protocols for remote system administration to networked computers. Remote access is provided by means of an SSH server. Threats of SSH intrusion have increased as of late [16, 17], and the consequences of an intrusion can be severe [18]. In the to be published paper, dictionary attacks against SSH are taken as a case study. Namely, SSH is considered in terms of network traffic behavior of dictionary attack activity at the transport-layer. Earlier on during the Master assignment, the application-layer behavior of SSH was also thoroughly analyzed. Not only behavior of dictionary attacks, but also of authenticated (interactive) sessions. Several protocol layers within the SSH architecture [19, 20, 21] had to be studied in support of this work. Furthermore, practical experiments and measurements had to be performed. The following coarse subdivision of research can be identified within this topic:

  1. **Packet and flow-level network traffic measurements** – SSH network traffic was studied in packet captures with full payload, as well as flows. The full captures were required to analyze plaintext parts of the connection, i.e., the data exchanged before encryption is negotiated. Measurements were performed in

an artificial setting at first, using network traffic created by several popular brute-force tools and regular OpenSSH clients. Examples of included tools are *Hydra* [3], *Ncrack* [4], *Medusa* [5], and *Brutessh* [6]. Analysis of packet captures made while operating these tools proved invaluable in reflecting on observations made based on literature, because certain theorized phenomena of both TCP and SSH coudl be validated this way. These efforts helped in designing later data collection, and steps for data analysis and validation.

2. **Source code analysis** – The source code of a small number of brute-force tools and an SSH library was analyzed to study the application-level behavior of said tools. For example, how the 'client banner' is chosen was studied, as well as how choices are made for the negotiated algorithms during the SSH connection setup. This analysis combined with observations made based on artificial measurements allowed for solutions to be proposed and reflected on.
3. **Encryption methods and more** – The SSH standard includes support for a variety encryption ciphers, compression algorithms, message authentication code algorithms, key exchange algorithms, authentication methods and more. Background research and practical experiments were performed to study how the network-level behavior of several protocol layers within the SSH architecture depends on these parts.

[3] `http://www.thc.org/thc-hydra/`
[4] `http://nmap.org/ncrack/`
[5] `http://foofus.net/goons/jmk/medusa/medusa.html`
[6] `http://www.edge-security.com/`

1.3 Organisation

The organization of this thesis is as follows. Performed research that yielded results, but was too divergent to include in the paper is discussed in Chapter 2. Chapter 3 provides details on the software that was developed for both the work included in the paper, and the additional work presented in this thesis. Next, in Chapter 4, I reflect on the research process Master assignment as a whole. Following some acknowledging words, the paper written as part of this work can be found in Appendix A.

Chapter 2

Additional work

Over the course of the Master assignment, the research questions and approach were subjected to change. Choices were made that led to the shelving of partially completed research. Making choices is an intrinsic part of research, and there are several reasons for them during the assignment. Firstly, some preliminary results were already very promising, and the associated approach was given priority over other approaches. Secondly, not all approaches, despite their potential, were fit for bundling in the same paper because of their divergence. Thirdly, personal preference weighed in. Due to the choices made, some additional work is not reported on in the paper, while still yielding results. This work is covered in this chapter. Section 2.1 explains the relation between the additional work and the paper. Next, in Sections 2.2-2.4, the work itself is discussed.

2.1 Bridging the gap

The relation between the research reported in the paper and the additional work that was done best be clarified before going into details on the additional work itself. This section explains the relation. It is assumed that the reader has read the paper (see Appendix A).

While studying literature, three disadvantages of using flows for intrusion detection were identified [22, 23, 24, 25, 26]. These are as follows:

1. *Information loss* – Flow exporters aggregate packets into flows. Due to this aggregate nature, information about individual packets is, by design, 'lost' during export. This loss can impede flow-based intrusion detection.

2. *Delay* – Due to the flow cache expiration logic on flow exporters, there is an inherent delay before flows are exported. Furthermore, flow collectors typically rotate data to disk in intervals, introducing another delay for tools that await interval completion before data is accessed.

For these reasons, flow data is usually not directly accessible for analysis ‘post-collection’.

3. *Attack effects on the flow data infrastructure* – The flow collector and exporter can themselves become victims of an attack, either as direct targets, or due to overload as attacks are ongoing.

From these three disadvantages, the first is addressed in the paper, while the other two are not. Working towards solutions for all these disadvantages was part of earlier research efforts. To understand how early research efforts were guided, consider the following research questions, which were formalized early on during the Master assignment to propose a direction:

1. Which information loss during the metering and export of IP flows affects the flow-based detection of brute-force attacks?
2. Which custom information elements can we add to IP flows during metering and export to overcome said information loss?
3. By how much can we improve the accuracy of existing flow-based brute-force detection systems with custom information elements, and at what cost?
4. How can we do detection of (SSH) brute-force attacks in (near) real-time?

The fourth research question directed research of detection algorithms that could work in real-time. Not only would such algorithms overcome the delay disadvantage, but a real-time solution can provide immediate feedback to the flow exporter and collector as well, thereby mitigating the effects of attacks as soon as possible. These efforts, which relate to the second and third disadvantages above, are not reported on in the paper, in any shape or form.

Furthermore, as the Master assignment builds on work in [3, 4], the context of flow-based detection of SSH dictionary attacks was a given from the beginning. In the paper, though, the application of SSH is less profound, as it ‘merely’ forms the case study of a more generic problem with flow-level detection of dictionary attacks. This more generic problem is that network traffic of dictionary attacks does not always exhibit ‘flat’ characteristics, meaning that not every connection is similar in terms of bytes, packets and duration, while that is expected. The datasets used in [4] confirm this, and put forth TCP retransmissions as the suspected cause. The contributions of retransmissions to features such as the number of *Packets-Per-Flow* (PPF) can typically not be discriminated in flow data. This should be considered a disadvantage of the aggregate nature of flow export, and thus is linked to the first disadvantage that we have identified earlier on.

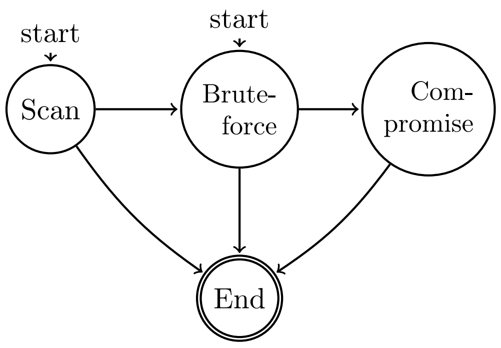

Figure 2.1: Brute-force attack phases.

Discussing the inability to discriminate retransmissions in flow data may sound like rehashing of the paper, but this is not the case. In the paper, only the detection of the 'attempts' part of dictionary attacks are evaluated as part of the SSH case study. The detection of scans or compromises (a successful brute-force authentication attempt) are not. Fig. 2.1 shows these three different phases of dictionary attacks according to [3, 4]. The *brute-force* phase in this model is only evaluated in the paper's case study. Earlier on during the assignment, however, other phases were considered as well, both with real-time detection in mind, as well as the other disadvantages.

Moreover, from discussions with the authors of [3], it was learned that the problem with network traffic flatness negatively impacted one of their approaches to *compromise* detection (from Fig. 2.1). Namely, it impeded detection of the transition moment from the *brute-force* to the *compromise* phase, which was detected based on a deviation from previously flat behavior, i.e., after brute-force activity had already been flagged. This knowledge directed early research efforts not only towards addressing the network traffic related issues, which would improve detection of the transition, but also towards alternative methods of SSH compromise detection. Finally, while TCP retransmissions are not typically deliberate from a malicious intent point of view, the deliberate introduction of variability into flows, so-called *flow stretching*, was also considered. From this point on, whenever 'variability', 'variable change', 'flatness effects' or any similar terms are mentioned, it relates to either unintentional or intentional effects on network traffic flatness in terms of bytes or packets.

The remaining sections of this chapter describe research into the previously explained parts. In Section 2.2, research relating to an alternative SSH compromise detection method is discussed. Then, in Section 2.3, research into detectable and undetectable SSH application-level flow stretching or unintentional effects on flatness is explained. Lastly, in Section 2.4, research efforts towards real-time detection of all three attack phases is discussed.

2.2 SSH compromise detection

Research efforts earlier during the assignment focused on SSH compromise detection. Not only in terms of real-time detection of compromises, but also assisting existing flow-based compromise detection approaches by adding information to flow data during export. Adding information during flow export using a customizable flow exporter seemed feasible considering [24, 6], and work such as [27] show that this can assist, or make possible to begin with, flow-based monitoring solutions, such as intrusion detection.

With a flow exporter extension in mind, methods were considered that could (partially) depend on payload-based information. Since payload is still available on a flow exporter. Alternative methods were considered would not necessarily rely on the flow-level number of PPF [3], or strictly operate post flow data collection [4]. Extensive background research for alternative methods was done. Eventually, progression has been made towards an algorithm that is promising in terms of both real-time detection, as well as 'post-collector' assistance. This algorithm depends on the ability to detect sessions within encrypted connections based on interactive keystroke recognition.

In the next sections, all works towards a real-time compromise detection algorithm is explained in layers. Section 2.2.1 details keystroke recognition in encrypted connections. In Section 2.2.2, interactive session detection, which is based on keystroke recognition, is explained. Afterwards, the compromise detection considerations are discussed in Section 2.2.3.

2.2.1 Encrypted keystroke recognition

During the background research, work in [28, 29] was encountered. These works show that keystrokes, i.e., interactive keys that a user sends, can be recognized within encrypted SSH connections based on the payload size of a packet. In these works, keystroke recognition is used for purposes such as statistical analysis of inter-arrival time of keystrokes. This allows for timing attacks on passwords, modelling of typed commands for user profiling, etc. While modelling commands would be an approach to recognize sessions, this was found to be impractical on a flow exporter for reasons such as requiring a user profile database. A much simpler observation is that interactive sessions can potentially be recognized based merely on the presence of keystrokes, i.e., based on characteristic packet sizes.

Password keystrokes in [28] are recognized based on the fact that password characters are not 'echoed' back to the terminal, c.q., SSH client. In other words, a packet of a characteristic size is observed from client to server, with no response sent in return by the server. A fixed size of *20 B* (excl. TCP header) is used in [28]. Under the SSHv2 standard this approach is infeasible, as 'fake echoes' can be sent back by the server. The client does not echo these, but a packet does traverse the network. Typical SSHv2 server implementations send these fake echoes as passwords are typed to prevent leaking that a password is being typed. The SSHv2 standard is well-suited to protect against traffic analysis techniques. For example, it allows additional random padding of packets. With additional random padding, a few more cipher block sizes of random data are added to packets. Clearly, recognizing keystrokes based on size would be impossible if this is used in practice on keystroke packets. This prompted measurements, to investigate if popular SSHv2 implementations of clients and servers apply additional padding to keystroke packets.

Several packet captures of interactive sessions have been created and studied to see if commonly used SSH implementations add additional random padding to keystroke packets. Several versions of *OpenSSH*[1] client and server implementations have been tested. *OpenSSH* is by far the most widely used implementation[2]. For this reason we did not test other implementations at this stage, such as the closed-source, commercial *Tectia*[3] clients and servers.

Based on the interactive sessions measurements, two key observations can be made:

1. None of the tested client and server implementations apply additional padding to the keystroke packets sent, or its (fake) echo.
2. The characteristic size of keystroke packets are different, in comparison, for some of the clients and server tested.

With the positive first observation in mind, the cause for second observation was given a closer look. It has become clear that the difference in size is influenced by the used encryption cipher, and the used keyed-hash message authentication code (HMAC) algorithm. None of the tested clients and servers are set to use compression by default, but further measurements with alternative configurations have confirmed the understandable assumption that compression also affects the characteristic size of a keystroke packet. These results validate that keystroke packets are of a fixed size, and that this size can be determined based on knowledge of used algorithms. This prompted further research into this area.

As has been explained, keystroke recognition in client to server SSH connections was found to be feasible based on literature and measurements. As the used encryption cipher, HMAC algorithm, compression algorithm, etc. affect the characteristic size of a keystroke packet, a compromise detection algorithm requires these to be known for every connection. In the Transport Layer Protocol [20] part of the SSH standard, it is explained how algorithms

[1]`http://www.openssh.com/`

[2]Information obtained from `http://www.openssh.com/usage/graphs.html`.

[3]`http://www.ssh.com/products/tectia-ssh`.

```
uint32      packet_length
byte        padding_length
byte[n1]    payload; n1 = packet_length - padding_length - 1
byte[n2]    random padding; n2 = padding_length
byte[m]     Message Authentication Code (MAC); m = mac_length
```

Figure 2.2: Binary Packet Protocol.

are negotiated as an SSH connection is set up. This requires both sides to send a `SSH_KEX_INIT` packet. These packets are (initially) sent in plaintext. While later renegotiation can occur under encryption, negotiated algorithms are typically not changed in practice at all, or within a potential detection period. In other words, given that payload is available, a potential algorithm will have access to the information required to recognize interactive sessions. This notion prompted the design and development of a detection algorithm, to be implemented as flow exporter extension. At the heart of the detection algorithm is mapping knowledge of the negotiated algorithms to the size of a keystroke packet.

The design of the detection algorithm required a thorough understanding of the Binary Packet Protocol of the SSHv2 [15] standard. This protocol dictates the format under which all messages are sent between SSH endpoints, and thus factors into the size of keystroke packets. Fig. 2.2 shows the format of packets in this protocol. The first four `packet_length` bytes indicate the length of the SSH packet, excluding the four bytes of the `packet_length` field itself, and the Message Authentication Code (MAC) bytes. Payload contains the packet payload, for example the keystroke information. Should compression be negotiated, the compression algorithm is applied to the payload only, and this is done before encryption and MAC calculation. According to the standard, when encryption is in effect, it is applied to the `packet_length`, `padding_length` and `payload`. This untrue, however, for certain variants of MAC algorithms, which are nowadays configured as default in *OpenSSH* clients and servers. There are two types of MAC algorithms that can be used:

- **Mac-Then-Encrypt (MTE)** – Under MTE, the message authentication code is calculated first. The input for the MAC algorithm are the length fields, the payload and the random padding. The MAC value is appended to this data, and then everything is encrypted.

- **Encrypt-Then-Mac (ETM)** – ETM is used by default nowadays, and encryption is performed first, followed by MAC calculation and appending. A noteworthy difference is that the four `packet_length` bytes are not encrypted when ETM is used, because as the MAC is verified first, it must be known where the encrypted data ends, and the message authentication code starts.

Table 2.1: Keyed-hash message authentication codes (HMAC) & digest sizes

Algorithm	**Digest size (B)**	**Support**
none	0	optional
hmac-sha1	20	required
hmac-sha1-96	12	recommended
hmac-md5	16	optional
hmac-md5-96	12	optional
hmac-sha1-etm@open	20	optional*
hmac-sha1-96-etm@openssh.com	12	optional*
hmac-md5-etm@openssh.com	16	optional*
hmac-md5-96-etm@openssh.com	12	optional*
hmac-sha2-256	32	optional*
hmac-sha2-512	64	optional*
hmac-sha2-256-etm@openssh.com	32	optional*
hmac-sha2-512-etm@openssh.com	64	optional*
umac-64@openssh.com	8	optional*
umac-128@openssh.com	16	optional*
umac-64-etm@openssh.com	8	optional*
umac-128-etm@openssh.com	16	optional*
hmac-ripemd160	20	optional*
hmac-ripemd160@openssh.com	20	optional*
hmac-ripemd160-etm@openssh.com	20	optional*

In terms of keystroke packets, the keystroke payload is compressed if compression is in use. Then, depending on whether *ETM* or *MTE* is used, either encryption is applied first to the involved fields, or a MAC is calculated over them. One final observation is that the encryption ciphers used in SSH are block ciphers. Block ciphers transform full blocks of 'plaintext' to 'ciphertext'. This is why, even if no additional random padding is added to the binary SSH packet, there may still be some padding required. This pads the size of the data input to the block cipher to the nearest natural multiple of its block size.

With the background research of the Binary Packet Protocol in mind, further research on ciphers and algorithms, as well as packet capture measurements, could be performed to determine the influence of negotiated algorithms on the characteristic size of a keystroke packet. Table 2.1 shows keyed-hash message authentication code (HMAC) algorithms commonly used by SSH implementations, and those referenced in the SSH standard. The digest size of HMAC algorithm is given. This is the length in bytes of the message authentication code that is calculated by the algorithm. Moreover, it is shown if the SSH standard mandates support for the algorithm. Any optional algorithm not referenced in the SSHv2 standard, is marked with

Table 2.2: Encryption ciphers & resulting keystroke bytes for HMAC types

Cipher	**Encrypted bytes**		**Support**
	MTE	*ETM*	
3des-cbc	24	16	required
none	-	-	optional
aes128-ctr	32	16	optional*
aes192-ctr	32	16	optional*
aes256-ctr	32	16	optional*
aes128-cbc	32	16	recommended
aes192-cbc	32	16	optional
aes256-cbc	32	16	optional
arcfour	24	16	optional
arcfour128	24	16	optional*
arcfour256	24	16	optional*
blowfish-cbc	24	16	optional
cast128-cbc	24	16	optional
rijndael-cbc@lysator.liu.se	32	16	optional*
aes128-gcm@openssh.com	*gcm*		optional*
aes256-gcm@openssh.com	*gcm*		optional*

an asterix. It should be clear that the chosen HMAC algorithm affects the overall size of an SSH binary packet, and thus a keystroke packet in particular, due to the length of the MAC. For example, for the required `hmac-sha1`, the digest size is 20 bytes, while for the recommended `hmac-sha1-96`, it is 12 bytes.

Table 2.2 shows encryption ciphers commonly used in SSH implementations. For most ciphers in combination with *MTE* and *ETM*, the size in bytes of the encrypted part of an SSH binary packet that contains a keystroke is shown. Just to be clear, this does not include the MAC digest bytes, and for *ETM* the four `packet_length` bytes do not count as encrypted part. Two things should be noted. Firstly, the chosen encryption cipher has an effect on the encrypted part of the packet. This is because of the resulting ciphertext. Any difference between ciphers is due to a difference in block size. Secondly, the choice for *Mac-Then-Encrypt* results in more 'ciphertext' than *Encrypt-Then-Mac*. This is because for *MTE*, the `packet_length` bytes are also encrypted as previously explained, and these additional four bytes require an extra block size for encryption. *Galois/-Counter Mode* (GCM) has implicit authentication of data integrity, and as such is a special case for which no values can be listed that do not include the 'MAC bytes'. Moreover, few algorithms referenced in the SSH standard are not shown, as sizes could not be determined with measurements due to lack of support in popular SSH implementations. These are the ones in the

Table 2.3: Effects of compression on characteristic keystroke size

Cipher	**-Δblock size count**	**-Δbytes**
3des-cbc	1	8
none	0	0
aes128-ctr	1	16
aes192-ctr	1	16
aes256-ctr	1	16
aes128-cbc	1	16
aes192-cbc	1	16
aes256-cbc	1	16
arcfour	1	8
arcfour128	1	8
arcfour256	1	8
blowfish-cbc	1	8
cast128-cbc	1	8
rijndael-cbc@lysator.liu.se	1	16
aes128-gcm@openssh.com	0	0
aes256-gcm@openssh.com	0	0

`serpent` and `twofish` families, and `idea-cbc`. Since `none` is not recommended, and not supported in popular implementations, its values are left out.

Given the effects of encryption ciphers and HMAC algorithms on the size of a SSH binary packet containing a keystroke payload, one influential part that remains is the chosen compression algorithm. The SSH standard references two methods of compression. The first is `none`, i.e., no compression, and support for this is required. Secondly, `zlib` (LZ77) compression, support for which is optional. The commonly used *OpenSSH* server and client implementations use a ZLIB variant, `zlib@openssh.com`, that delays compression until after user authentication. This is of no influence on recognizing keystrokes of interactive sessions, as interactive sessions occur after authentication, and both compression variants are thus in effect at that time. As has been discussed previously, compression, if negotiated, is applied to the payload before MAC and encryption. This results in potentially reducing the number of plaintext block sizes that need to be encrypted. Based on packet captures and measurements, it has been determined that in all cases, compression reduces by one block size of bytes, with the exception of *GCM* ciphers. That means for `3des-cbc`, `arcfour` and so on, the size of a keystroke packet is reduced by 8 bytes. For `aes128-cbc` etc., it is 16 bytes. Table 2.3 shows for all ciphers, the compression reduction in block size count and bytes.

The result of all research discussed sofar, is that plausible keystroke ‘events’ can be recognized in encrypted connections when the chosen algorithms and ciphers are known. We speak of plausible rather than certain, because other SSH packets can end up with the same size. What that means is that SSH packets that contain a non-keystroke payload may end up the same size as a keystroke packet would. Preliminary measurements showed that this is not often the case, and with this in mind extensive practical work was done to recognize plausible keystrokes in network traffic automatically, which involves determining the negotiated ciphers and algorithms to calculate the characteristic keystroke packet size. The functional details of the flow exporter extension in terms of configurability and so on that has been developed will be discussed in Chapter 3. The interactive session detection, which builds on the ability to recognize keystrokes in encrypted connections, will be discussed next.

2.2.2 Encrypted interactive session detection

With the practical parts for automated keystroke recognition fully operational, further measurements were performed with many more combinatorial choices of ciphers and algorithms. These measurements support earlier observations that implementations typically do not send non-keystrokes of characteristic size often. Based on these findings, it was theorized that with proper mathematical techniques and thresholds, keystroke event-driven session detection can possibly be realized with reasonable accuracy. In other words, it was believed that proper techniques could separate incidental non-keystrokes packets from real keystrokes packets to determine ongoing interactive sessions accurately. This prompted background research into techniques such as those that involved the Exponential Weighed Moving Average (EWMA) of events, c.q. process observations [30].

After background research on EWMA and other techniques, further practical work was done to realize event-driven session detection using these techniques. All efforts discussed sofar led to not only the automated detection of keystroke events, but also the feeding of these events to an EWMA algorithm to evaluate threshold violations, ergo, determine ‘session detected’. Because other additional work relies on the same mathematical techniques, the theoretical background will not be discussed until later.

2.2.3 Real-time compromise detection

Full support for interactive session detection in real-time was now a given. A further though towards compromise detection was as follows. If knowledge of an ongoing session is combined with a blacklist of ‘bad hosts’, or knowledge of previously flagged brute-force behavior from one of the hosts involved in the detected session, it reasonable follows that this session is in fact a compromise. This thought was shelved to focus on the work reported on in

Table 2.4: SSH dataset

Duration	**Packets**	**Bytes**	**Flows**
17 hours	111.6 M packets	96.7 GiB	4.5 M

the paper, and never picked up again due to time limitations.

To showcase the research leading up to a potential compromise detection algorithm, a seventeen-hour experimental dataset of SSH network traffic was created. This dataset contains seventeen hours of SSH flow data, collected on the University of Twente (UT) network. The details of the UT network are in the paper. The dataset contains, among other things, information on the occurrence and usage of various encryption ciphers, MAC algorithms and compression algorithms. The information was added analogous to the approach reported on in the paper, but with additional IPFIX information elements that have been defined. Table 2.4 shows information on this dataset. As can be seen, the dataset contains *4.5 M* flows, *111.6 M* packets and *96.7 GiB*. The reason for the limited measurement period is of a practical nature. To be more specific, the parts of the flow exporter extension that add application-specific SSH information, which involves payload inspection, were not enabled on account of optimization as the main datasets for the paper were collected. Again, details on the functionality will be discussed in Chapter 3.

Table 2.5 shows for the dataset the occurrences of encryption ciphers, expressed in flows that contain the negotiated cipher. Table 2.5 shows the same for MAC algorithm. As shown, the encryption cipher `aes256-ctr` is atop as it is used in *44.78%* of all flows. For MAC algorithm, `hmac-sha1` wins with *81.68%*. Table 2.7 shows the top compression algorithm choices, which has less to choose from, obviously. Clearly, `none` compression is used within *99.99%* of all flows, which makes sense because that is typically negotiated by default.

Table 2.8 shows for combinations of the top threes of encryption ciphers and MAC algorithms the percentage of total flows in which this combination is negotiated. As can be seen, the combination of `hmac-sha1` with `aes256-ctr` occurs most, with *44.69%*. We will show later that *libsshlibssh*[4] uses this combination[5] (Section 2.4.3). We will also show that *libssh* variants are used a lot (Section 2.3). We suspect this explains why this combination is way ahead of others.

[4]`http://www.libssh.org/`

[5]The client's algorithm choices take precedende in the negotiation, as long as the server supports the choices.

Table 2.5: Encryption cipher usage

Encryption cipher	**Flows**
aes256-ctr	313378 (44.78%)
aes128-ctr	246067 (35.16%)
aes128-cbc	68794 (9.83%)
3des-cbc	57917 (8.28%)
aes256-cbc	9840 (1.41%)
aes192-cbc	3722 (0.53%)
aes192-ctr	108 (0.02%)
blowfish-cbc	60 (0.01%)

Table 2.6: MAC algorithm usage

MAC algorithm	**Flows**
hmac-sha1	571682 (81.68%)
hmac-md5	119458 (17.07%)
hmac-md5-etm@openssh.com	7380 (1.05%)
hmac-sha1-96	1232 (0.18%)
hmac-sha2-256	134 (0.02%)

Table 2.7: Top compression algorithms

Compression algorithm	**Flows**
none	698872 (99.99%)
zlib@openssh.com	1002 (0.14%)
zlib	12 (<0.01%)

Table 2.8: Top three encryption ciphers & MAC algorithm combinations

Encryption cipher	**MAC algorithm**		
	hmac-sha1	hmac-md5	hmac-md5-etm@openssh.com
aes256-ctr	44.69%	<0.01%	-
aes128-ctr	25.82%	8.17%	1.05%
aes128-cbc	9.16%	0.67%	-

This concludes showcasing some fruits of the practical work that has been done towards a real-time compromise algorithm. We have shown the ability to detect encryption ciphers, MAC algorithms and compression algorithms. Furthermore, by adding them to flow data not only can statistics be presented as in the showcase, but the information can be used post flow collection for any purpose deemed useful. As the preceding statistics on the detected parts show, a multitude of combinations is used in the experimental dataset. In order to make confident statements that this is representative for the Internet, a longer dataset may be needed.

2.3 Application-level effects on flatness

Early research efforts were directed at the recognition of application-level variability in terms of packets and bytes that may affect otherwise flat traffic. Adding custom IPFIX information elements to flow data on application-level variability would, alike adding information on TCP retransmissions as reported on in the paper, possibly improve flow-based intrusion detection. As part of this effort, theoretical opportunities for SSH application-level effects on flatness were investigated. Not just 'unintentional' effects, but also deliberate, i.e., flow stretching.

From previous explanations on the Binary Packet Protocol (Fig. 2.2), it becomes clear that with the SSHv2 standard, adding a variable number of bytes to traffic by means of padding is possible by design, and rather straightforward. Moreover, as we will show, the packet type used to send a fake echo when passwords are typed, provides a way to introduce additional packets into SSH traffic. Since these messages are encrypted, they typically cannot be discriminated from other network traffic part of the connection between client and server. Considering the SSHv2 standard, the following opportunities for application-level effects on flatness of network traffic have been identified:

Random padding
The maximum number of randomly padded bytes is 255 per binary packet. This means that without impeding standard protocol operation, one can technically stretch up to this many bytes, per packet. A sophisticated brute-force tool could, for example, randomly pad packets containing authentication attempts, to 'unflatten' network traffic.

SSH_MSG_IGNORE
The SSHv2 standard allows `SSH_MSG_IGNORE` binary packets to be sent, which must be ignored (or at least recognized as containing no useful data) by the receiving side. This can be used, for example, to send the fake keystroke echoes when passwords are typed. This packet type enables one to unidirectionally introduce additional packets and bytes in network traffic, ergo stretch both packets and bytes.

SSH_MSG_UNIMPLEMENTED
SSHv2 implementations must respond to unknown message codes with a `SSH_MSG_UNIMPLEMENTED` message. This allows one side to trigger the other side to send an additional packet by sending a bogus message code. In other words, one can bidirectionally introduce additional packets and bytes this way.

SSH_MSG_DEBUG
SSHv2 implementations must understand the debug message code, but may choose to ignore it. This, much like `SSH_MSG_IGNORE`, allows the introduction of additional packets and bytes unidirectionally.

Protocol Version Exchange (PVE)
After the TCP connection establishment between SSH client and server, a mandatory protocol version exchange takes place. Both client and server send, in no particular order, a packet that contains the SSH protocol version along with other information, such as the software used. This packet contains at least one line that is terminated by a `\cr\lf` at the very end of the packet. This line starts with the `SSH-` prefix. The format of the line is `SSH-protoversion-softwareversion SP comments CR LF`.

According to the SSHv2 standard, the PVE the very first binary packet that must be sent by both sides after the TCP handshake. The `SSH-` prefix is followed by the protocol version, a dash delimiter, the software version and, optionally, a comment. An example is: `SSH-2.0-OpenSSH_ 6.6.1p1 Ubuntu-2ubuntu2 \cr\lf`. The maximum length of the PVE line is 255 bytes, but it may be preceded by optional `\cr\lf` delimited lines within the same very first SSH binary packet. Therefore, the length of the protocol version exchange packet can vary somewhat depending on the sent software version and optional comment plus its length.

The presence of optional lines can affect the size of the binary packet greatly. If many optional text lines are included in the binary packet that carries the protocol exchange, its size could exceed the TCP Maximum Segment Size (MSS). This causes TCP to segment the data and results in additional packets in the network traffic. Many TCP segments for a MSS, which is commonly around 1460 bytes, fit in a binary packet payload field of at least 32K bytes[6]. The result is that one can deliberately stretch network traffic this way. Considering the previously explained packet types, and how easy it is to use them for deliberate stretching, the PVE packet seems a cumbersome approach to take. A noteworthy difference is that, since the PVE packets are exchanged in plaintext, this type of stretching can be detected.

Key EXchange init
After PVE exchange, the negotiation of encryption ciphers, MAC algorithms and other information required for key exchange is initialized. This is done by both sides sending a `SSH_MSG_KEXINIT` packet, which

[6]The SSHv2 standard mandates that implementations can handle at least 32K bytes of payload, although support for a longer payload should be supported when required.

can be done in any order, but is usually done at about the same time. This packet type contains name-lists of all supported algorithms such as key exchange methods, encryption ciphers, MAC algorithms and compression algorithms. Protocol rules dictate the negotiation outcome.

Some of the name-lists of supported algorithms are given for the client-to-server and server-to-client directions in separate lists, which allows for different algorithms for these directions, should that be desired. There are ten name-lists in total, and they are used to negotiate the algorithms to use. The packet also contains some additional data with mostly fixed length which is not relevant to this discussion. Each name-list is preceded by a 32-bit integer that indicates the length of the list. This allows both SSH endpoints to parse the received name-lists, which are comma-delimited strings of supported algorithms that appear in a fixed order in the binary packet.

What is important to note, is that the supported algorithm name-lists can vary in length greatly, depending on the chosen algorithms. As such, the `SSH_MSG_KEXINIT` message may or may not require more than one TCP segment to be sent. Theoretically, one could use this packet to stretch network traffic by appending additional algorithms, the presence of which do not hamper protocol operation[7]. As with the PVE message, this is a cumbersome approach to take, but as the (initial) `SSH_MSG_KEXINIT` messages are sent in plaintext, this type of stretching too can be detected.

In conclusion, several binary packet types of SSHv2 can affect the flatness of network traffic from the application-level due to variability in size, and number of packets (TCP segments). To our knowledge, no research has investigated the SSHv2 standard from this angle. As the PVE and (initial) `SSH_MSG_KEXINIT` packets are sent in plaintext, effect on flatness caused by them can be recognized. Information on application-specific variability can be exported, analogously to transport-layer protocol phenomena as is done in the paper. This allows, to some extent, for application-level effects on flatness to be corrected, which may ultimately lead to improved intrusion detection, much like the case study in the paper. The practical component that has been implemented as part of this effort is discussed in Chapter 3, along further functional details of the flow exporter extension that has been developed.

To showcase the research discussed in this section, all protocol version exchange messages for the seventeen-hour dataset of SSH flow data detailed

[7] As long as they are appended behind valid choices offered by the server, a negotiation can take place.

Table 2.9: Top five client PVE strings

Client PVE	Occurrences
SSH-2.0-libssh-0.4.8	126108 (51.26%)
SSH-2.0-libssh2_1.4.2	71519 (29.07%)
SSH-2.0-libssh-0.1	28361 (11.53%)
SSH-2.0-libssh2_1.4.3	6861 (2.79%)
SSH-2.0-libssh-0.2	5887 (2.39%)
Other	7288 (2.96%)
Total	246024

Table 2.10: Top five server PVE strings

Server PVE	Occurrences
SSH-2.0-OpenSSH_6.0p1 Debian-4+deb7u1	44461 (17.00%)
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.4	21540 (8.24%)
SSH-2.0-OpenSSH_5.3	18272 (6.99%)
SSH-2.0-OpenSSH_5.3p1 Debian-3ubuntu7	14358 (5.49%)
SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2	12873 (4.92%)
Other	149927 (57.35%)
Total	261431

earlier have been inspected and recorded. Tables 2.9 and 2.10 show, respectively for clients and servers, the top five occurrences of protocol version exchange strings. As can be seen, *libssh* strings dominate the client top five. An *OpenSSH* string is in the sixth position, suspectedly of a regular client. Not many more *libssh* strings occur outside the top five. This leads to the conclusion that many scripts and tools work with *libssh*, and include only one of a small set of versions. For the server top five, *OpenSSH* implementations dominate. The first non-*OpenSSH* string is in the 24th position, and it is *SSH-2.0-dropbear_0.53.1*[8]. While *57.35%* is not in the top five, the absolute majority of the server to client flows have some version of *OpenSSH*.

In order to give a version-independent idea of occurrence of implementations, the version part of a PVE string should be ignored. Table 2.11 shows the top three occurring SSH server implementations based on PVE string, along with the position of its leading version. The third-placed *WeOnlyDo*[9] is a commercial SSH server. Table 2.12 show similar information, but for clients. The third-placed *check_ssh* is the PVE of a monitoring plugin[10], which allows one to monitor at intervals if an SSH server is still running.

[8]`https://matt.ucc.asn.au/dropbear/dropbear.html`.

[9]`http://www.weonlydo.com/SSHServer/ssh-telnet-server.asp`.

[10]`https://www.monitoring-plugins.org/doc/man/check_ssh.html`.

Table 2.11: Top three server implementations based on PVE strings

Server type	**Occurrence (all)**	**Top position (version)**
OpenSSH	250562 (95.83%)	1
Dropbear	5602 (2.14%)	24
WeOnlyDo	1897 (0.73%)	25

Table 2.12: Top three client implementations based on PVE strings

Client type	**Occurrence (all)**	**Top position (version)**
libssh & libssh2	240434 (97.71%)	1
OpenSSH	4559 (1.85%)	6
check_ssh	637 (0.26%)	10

2.4 Real-time detection

In attempt to overcome the delay disadvantage of using flow data for intrusion detection, and to be able to mitigate attack effects on the flow exporter and collector in the event of an ongoing attack, real-time detection methods were researched. Methods to detect all phases from Fig. 2.1 have been investigated. The research efforts towards SSH compromise detection in real-time have already been discussed in a preceding section, due to the application-specific nature of these efforts. This leaves real-time detection of scans and ongoing brute-force attempts to be discussed still. Research efforts into these areas led to, as will be shown, results that are not all specific to a given application.

To the end of finding suitable methods for real-time detection, much more literature was studied. As has been explained previously, using a customizable flow exporter to this end seemed feasible. Even though development of a flow expoter extension was a given, stressing the flow exporter in terms of resource usage was not. For this reason, all potential light-weight solutions encountered in literature were considered. Anomaly-based approaches such as in [31, 32] were considered among the best candidates after extensive background research. The power of the work in [31] is generating signatures for unknown attacks, at the flow-level. As stated by the authors, signatures that are generated for known attacks, come close to signatures that already exist for those known attacks. When focusing strictly on scans and dictionary attacks, ergo known attacks, the potential benefit of this algorith thus disappears and all that is left is extra overhead. Speaking of overhead, in terms of computational resources required, the work in [31] allows reasonable detection when required to process *2500 flows* in a 20 second interval. However, on the University of Twente (UT) network, a flow exporter that processes only a subset of all traffic on the campus network already exports *1700 flows/s*. Performance increases are realized by parallelization, but that would violate that requirement for a light-weight solution. The work in [32] was found to be suitable for scan detection, but no further research efforts were spent on it due to the low priority of scan detection.

One of the lessons learned from the literature studies was that event-driven detection of scans and brute-forcing would allow for a light-weight solution, as long as determination of the particular event could be performed at low cost. In other words, relatively easy to determine events such as the number of flow records created per time interval [24] are viable, but a costly machine learning algorithm requiring many cycles is not. It was believed that detection could be performed with suspected little overhead, as long as the proper events, mathematical techniques, and thresholds were used. This conclusion had been drawn before the keystroke event was identified for compromise detection, as discussed in Section 2.2. Mathematical techniques identified for use were Exponential Weighed Moving Average (EWMA) [30] techniques, and CUSUM algorithms [33, 34]. The work in [24] has shown not only that using CUSUM logic with little overhead is feasible on a flow exporter, but a CUSUM algorithm is very promising in terms of anomaly-based detection.

The following three sections, Sections 2.4.1 and 2.4.3, discuss several types of relatively easy to calculate events that have been identified to detect scans and brute-force attacks.

2.4.1 TCP handshake real-time detection & algorithm assist

Packet information that can be calculated with relative little overhead if packet headers are available are TCP header flags, such as the `SYN` bit. The `SYN` bit is set during the synchronization (of sequence numbers) of a TCP connection [9]. The sequence of packets involved in synchronization is typically referred to as the *three-way handshake*. The three-way handshake is required as a connection between SSH client and server is setup, and since a dictionary attack from attacker to target requires an SSH connection, it follows that a TCP handshake is a prerequisite for the *brute-force* phase in Fig. 2.1. Moreover, the most common forms of scans (from Fig. 2.1), namely the regular 'connect' and so-called 'half-open' (or stealth) scans[11] require (part of) the synchronize sequence as well.

As has been discussed, the `SYN` bit in a three-way TCP handshake is a prerequisite for the *scan* and *brute-force* phases in Fig. 2.1. Within INVEA-TECH's FlowMon probe architecture [13], TCP flags are accessible with little overhead. With this in mind, the `SYN` bit 'event' has potential for use in real-time detection for the two aforementioned phases. If the event frequency is fed to an algorithm that can separate normal from anomalous, perhaps scans or ongoing dictionary attacks can be recognized. The works

[11] `http://nmap.org/book/man-port-scanning-techniques.html`.

we have previously discussed on EWMA and CUSUM techniques to detect anomalies based on event, c.q. process observations, have shown that with the proper choice of events, such recognition is possible with fair accuracy.

After the previously discussed observations and literature review, it was believed that with an efficient and properly tuned CUSUM implementation, detecting anomalous `SYN` bits in real-time was possible. What else was theorized, is that adding information regarding a possible anomaly to flow data, may somehow be of assistance to detection algorithms that work with flow data, c.q., post collection. The following two uses of `SYN` bits were considered for the detection scans and brute-force attacks:

Scans – In order to detect the previously discussed forms of scans, the number of `SYN` events between a given source and multiple destinations could be subjected to EWMA/CUSUM techniques. It is unlikely that a single host will trigger thousands of such `SYN` events in a short period. Counting events this way and comparing the number to a threshold will allow for reasonably accurate scan detection. To detect SSH network scans, obviously the destination port filtered on would have to be 22.

Brute-force attempts – Counting `SYN` events between a given source and fixed destination allows for an anomalous number of connections in a short period to be detected between source and destination. This could be indicative of an ongoing brute-force attack, i.e., many connections which a few authentication attempts are made. The word used is could rather than is, as an anomaly is not synonymous with malicious behavior. In other words, even though the number of connections is anomalous, it does not mean the activity performed in those connections is malicious per se. Despite this, early research efforts were guided by the notion that knowledge of an anomaly could aid existing brute-force detection approaches rather than single-handedly realize detection. For example, given that flatness is assumed in works such as [3, 4], and that a threshold of N flows with the same number of PPF must be reached, it could be possible to improve accuracy. For example, by allowing minor PPF deviations in case flows are marked as probable anomalies, so that N is still reached even if traffic is not entirely flat. This would require adding an IPFIX information element which contains anomaly-related information to flow data, and then using it 'post-collection' to improve the algorithm. Another thought was that if the algorithm from [4] is implemented on the flow exporter to eliminate the delay disadvantage, and thus enable real-time detection, then knowledge of an anomaly can help in the same manner.

2.4.2 Active TCP close real-time detection & algorithm assist

A problem with SYN events is that regular connections follow the same three-way handshake. A not-so-practical script that recursively *secure copies* (scp) each file under a directory separately, would trigger an anomalous number of SYN packets in the same fashion as an SSH brute-force tool would. Because of this, further candidate events were investigated. The work in [33] notes that under normal operation, an SSH server never actively closes a TCP connection. Fig. 2.3 shows two possible sequences in which the server actively closes the connection. Further details on how connections are terminated can be found in the TCP standard [9].

Measurements with the attack tools and servers mentioned in Chapter 1.2 have confirmed that the server indeed never actively closes the connection during regular connections, secure copies, *sftp* etc. However, when too many authentication attempts fail, it does. For example, the *OpenSSH* server actively closes the connection, i.e., it is the first endpoint to send the FIN bit, in two cases. Firstly, when a client has reached MaxAuthTries failed authentication attempts. The MaxAuthTries server configuration directive typically has a default value of *6*. Secondly, when a client's connection violates the number of unauthentication connections that can be open, as defined by MaxStartups. So contrary to SYN events, the sever-initiated FIN event reveals more about the potential malice in a connection. The discriminating difference is of course, the order in which packets with the header flag set are sent.

The background research and practical measurements have confirmed that if the server actively closes the connection, this event allows for the detection of brute-force attempts. One important detail here is that only a subset of the evaluated tools tries the maximum number of authentications per connection to the SSH server. Tests have shown, for example, that *Medusa*, *Hydra* and *Ncrack* try as many authentication attempts per connection as possible, and trigger a server-side initiated FIN (and sometimes even RST) after MaxAuthTries attempts. Tools such as *Brutessh*, however, perform only one attempt per connection. Moreover, command-line arguments to *Ncrack* can configure the number of attempts per connection to something lower than MaxAuthTries. In these latter cases, the client attack tool actively closes the connection and thus, alike SYN events, the potential in terms of attack detection diminishes as the order is no longer a discriminating factor with respect to benign connections.

For the previously discussed event-driven detection approaches based on the SYN and FIN TCP header flags, the practical work for EWMA/CUSUM detection has been completed. To be more specific, server-side initiated FIN events and the brute-force variant of the SYN events are detected, sub-

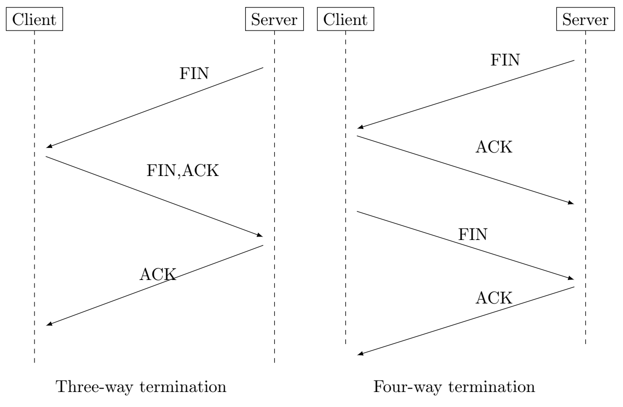

Figure 2.3: Connection termination sequences.

jected to EWMA/CUSUM housekeeping & heuristics, and can trigger an alarm when a configurable threshold is exceeded. While the functionality is present, no effort has been done to fine-tune any algorithmic parameters. This would have required making artificial datasets, or annotating datasets with ground truth information, and then applying techniques such as grid searches [35] to optimize detection results. This is where research efforts were shelved as other promising results with regard to what is reported in the paper had come in. More details on the implemented functionality is described in more details when the flow exporter extension is discussed as a whole, in Chapter 3.

2.4.3 Tool fingerprinting & signature-driven detection

One final brute-force detection approach was considered, which is more application-specific than those based on TCP header flags. Namely, the choice of supported algorithms in SSH's key exchange messages, and the sent banners in protocol version exchange messages, have been reviewed in an attempt to fingerprint attack tools. The idea is that a signature of a known attack tool can constitute an event for real-time detection.

The attack tools and regular clients that have been analyzed are the same as in Chapter 1.2. Measurements have shown that typical SSH clients communicate many supported algorithms, which lead up to two TCP segments per `SSH_SG_KEXINIT` message. On the other hand, some brute-foce tools only negotiate one or a limited number of algorithms in each category. For example, *Medusa* only indicates support for `none` compression, `aes256-ctr` encryption and `hmac-sha1` message authentication. As a second example, *Brutessh* only negotiates about four commonly supported ciphers and MAC algorithms, with no compression. In some cases the choices are hard-coded, but can be easily changed prior to recompilation. In other cases the choices cascade from included libraries, such as *libssh*[12]. Unfortunately, the chosen algorithms are not discriminating enough to fingerprint attack tools. One of the reasons is that *libssh*, for example, is included in a multitude of benign SSH clients, such as monitoring tools.

Source code review of all the aforementioned attack tools has shown that some of them have a clearly identifiable signature in the PVE message. For example, *Medusa* sends the `SSH-2.0-MEDUSA_1.x` banner, and the *Brutessh* tool sends `SSH-2.0-paramiko_1.5.x`. Despite these identifiable banners, this approach to detect attacks in real-time was shelved for several reasons. Firstly, *Medusa* allows an attacker to change the PVE with command-line options, and other tools that allow this may exist. Secondly, and alike algorithm choices, hard-coded parts can be changed. Thirdly, there are suspectedly many underground tools and custom scripts, that we cannot reasonably all fingerprint. Last, but definitely not least, a lot of malicious tools include *libssh*, and thus cannot be distinguished from benign clients based merely on the PVE string.

In conclusion, fingerprinting tools based on the above was considered not potential enough for futher research. Furthermore, it did not fit well with other research efforts. As shown in the section on flow stretching, Section 2.3, the practical work to recognize and record PVE banners has been completed nonetheless. Details on the functionality is in Chapter 3.

[12]`http://www.libssh.org/`.

Chapter 3

Assignment resultants

A lot of work has been performed during the master assignment. Considerable results are reported on in the paper. However, the paper does not do everything that has been done justice, as some of the assignment's are not, or only partially reported on in the paper. This chapter discusses these resultants. In Section 3.1, the flow exporter extension that has been designed and developed will be discussed in much more detail than in the paper. Next, in Section 3.1, the deliverable datasets are discussed.

3.1 Flow exporter extension

As discussed in the paper and in other chapters of this thesis, an elaborate flow exporter extension has been designed and developed. This added an industry-oriented aspect to the Master assignment. The extension comes as several plugins for INVEA-TECH's FlowMon Probe platform, as stated in the paper. Its implementation involved abundant amounts of background research on a multitude of topics, test cycles, tool-assisted debugging of stack traces, etc. According to the paper the extension realizes on-the-fly analysis of TCP conversations and adds IPFIX information elements to flow data on packet classifications, such as TCP retransmissions and confrol information. It does not end there. This chapter explains in more detail the extension, as it contains a lot of functionality that is not done justice in the paper. Because the extension has been developed as part of the INVEA-TECH Community Program, it will be released under the *BSD license*[1] after the paper has been published. It constitutes, therefore, a deliverable, and significant resultant of the Master assignment.

Functionality of the plugins will be explained in the following sections, Section 3.1.1-3.1.5.

[1]`http://opensource.org/licenses/BSD-3-Clause/`.

3.1.1 Source of input

In support of the dataset creation for the paper, as well as for the other research efforts discussed in this thesis, full access to packet payload is required. Within INVEA-TECH's FlowMon Probe architecture this is typically best accessed in an input plugin. For this reason, a custom input plugin had to be written, in addition to a custom process plugin.

The input plugin involved implementation of support for several packet sources, as follows.

> **rawnetcap** – To deploy the flow exporter extension on a live probe, packets must be readable from a monitoring interface. INVEA-TECH's proprietary *rawnetcap* format is supported and used on their products to capture at high speeds. Further benefits are nanosecond timestamp precision. While alternatives exist (see pcap below), support for rawnetcap was implemented on account of optimization. The custom input plugin supports command-line configuration of multiple (simultaneous) capture interfaces, sampling, packet snapping, and fine-tuning rawnetcap's queue size.
>
> **pcap (live)** – An alternative interface-based input for live capture that uses *libpcap*[2]. Command-line parameter support is available for configuration of the target interface, and setting packet snapping.
>
> **pcap (file)** – For both development purposes and reading offline data, support for reading packet captures, ergo *pcap* files, was necessary. To this end, there is support for *libpcap* files in offline mode. The source file is command-line configurable.

3.1.2 TCP analysis

In support of the work in the paper, the input plugin can analyze TCP connections on-the-fly and add IPFIX information on packet and byte counts of packet classes, such as TCP retransmissions. The TCP analysis module can be disabled with a command-line argument, but it defaults to enabled. Since the functionality of the module is reported on in the paper, it will not be explained here except for one part that has not been mentioned yet. This is that the input plugin can write TCP analysis statistics to disk, aggregated into intervals at the end of each interval. This functionality is useful when privacy (of flow data) is an issue, or when no IPFIX collector is available (or can be reached). This functionality can be separately enabled and configured with command-line parameters. Moreover, it requires the main TCP analysis module to be enabled. The command-line parameters are the time interval, and the path of the output file.

[2]`http://sourceforge.net/projects/libpcap/`.

3.1.3 Filtering

In order to prevent unwanted flows from being exported, the processing plugin part of the extension has command-line configurable support for two filters. First of all, filtering on TCP-only flows is possible, which was required to manage dataset sizes for the paper. Furthermore, allowing through only SSH flows is possible with a second filter. Technically speaking, the application-specific filter works based on port and as such can allow through non SSH traffic that runs over the same port.

3.1.4 SSH Deep Packet Inspection (DPI)

As discussed in preceding chapters, several SSH application-level packet types are sent in plaintext, and information can be extracted therefrom. The extension comes with fully functional DPI for these packets, that can be configured individually from the command-line. Moreover, as has been shown, for a subset of the inspected information, IPFIX information elements have already been defined and are being exported.

Given that SSH packets can be segmented at the transport-layer level, received out-of-order and so on, the involved program logic to perform DPI is quite sophisticated. Knowledge of negotiated encryption ciphers and other algorithms for a given connection is kept in cache so the information is retained in long-lived SSH flows, i.e., after the *active timeout* [5].

The packet types involved in the DPI related functionality are as follows.

> **The Key EXchange init packet** – The `SSH_KEX_INIT` packet type is recognized, its segments reassembled (if needs be), inspected, and parsed. Payload-derived information and number of TCP segments used to send the packet to begin with are known in program logic. This includes, in particular, knowledge of the supported encryption ciphers, compression algorithms, and MAC algorithms, as well as the negotiation outcome. This information is used for the calculation of the characteristic keystroke packet size, and the keystroke event is required for the alternative compromise detection approach, as discussed in Section 2.2. Furthermore, the negotiated algorithms are exported in IPFIX information elements that have been defined. Exporting the negotiated algorithms is not required for compromise detection, as that is done in program logic itself, in real-time. However, exporting this information allowed for the information in Tables 2.5-2.7 to be determined from flow data. The number of TCP segments used for the packet type, i.e., packets that factor in the number of PPF of a flow, are not exported. Should this be desired to compensate detectable application-level effects on flatness, as discussed in Section 2.3, then little additional development would be needed to realize this. DPI

and IPFIX export of the negotiated algorithms can be enabled and disabled with command-line parameters to the input plugin.

The Protocol Version Exchange packet – The PVE messages are recognized, their segments reassembled (if needs be), inspected, and parsed. Payload-derived information such as the communicated software version, comment, and protocol version is known in program logic. Information on the number of TCP segments of the PVE packet type is known in program logic also. The SSH protocol version has an IPFIX information element defined for it, which is exported by the extension. Furthermore, the input plugin allows for detected protocol version exchange messages to be written to disk, which was used to produce the results in Tables 2.9 and 2.10. Exporting this information using IPFIX for further research such as fingerprinting of attack tools (from Section 2.4.3) based on flow data is only a small step away in terms of development. Moreover, so is exporting information on the number of TCP segments used, for the further research suggested in Section 2.3. DPI and protocol version export in flow data is command-line configurable. Writing encountered PVE messages to disk can independently be configured, along with the target path.

With the DPI functionality explained, two peculiarities that were encountered during DPI research efforts can be revealed. Firstly, one of the attack tools was found to send SSH version 2 as part of its protocol version, but it did not include a `\cr` in the PVE string. The SSHv2 standard states that implementations may choose to not expect a carriage return, only for backwards compatibility with older SSH versions. This is thus strange behavior. Unfortunately, it was found not discriminating enough for fingerprinting purposes, because the tool in question, *Hydra v7.6*, includes the widely used *libssh* library. Secondly, in the SSH dataset from 2.4, an FTP server banner was found. It appears an FTP server was running on port 22. FTP terminates lines with a `\cr\lf` also, but these lines do not have the characteristic `SSH-` prefix. The encountered server was *Gene6 FTP Server v3.10.0 (Build 15)*[3].

3.1.5 EWMA/CUSUM based detection logic

The process plugin contains logic for EWMA and CUSUM techniques that is used for three of the event-driven detection algorithms discussed in this work. These are the characteristic keystroke packet size driven detection of interactive sessions, the `SYN` bit event-driven detection of connection anomalies, and the server-side initiated `FIN` bit event-driven detection of probably brute-force connections. Previous sections have explained why works such

[3] `http://www.g6ftpserver.com`.

as [33, 34, 24] have led to the implementation of this functionality. The purpose of this section is to explain the mathematical background of what has been implemented, and to provide functional details such as command-line parameters.

Each of the three detectors can be individually enabled and configured and has separate housekeeping using a cache with eviction mechanism based on last access time. Currently, only an alarm is output to the standard out when thresholds are violated, but further detection logic can be implemented and linked to violations of thresholds with ease. Obviously, the SSH session detector requires the DPI of algorithms to be enabled. Similarly, the `SYN` and `FIN` bit parts require the TCP analysis module to be enabled.

EWMA logic

The mathematical background for EWMA techniques can be found in [30]. Within the extension, the EWMA statistic is calculated as in Eq. 3.1. In this equation, x_i is the event intensity, t_x is the time at which the x_i event triggers the update, and λ is the smooting factor. Finally, $t_{z_{i-1}}$ is the time at which the previous EWMA statistic, z_{i-1}, was calculated. x_i is currently always set to 1, but if detection logic requires multi-event triggered calculation, that is possible.

$$z_i = \lambda x_i + (1-\lambda)^{(t_{x_i} - t_{z_{i-1}})} z_{i-1} \tag{3.1}$$

Using the previous smoothed event intensity, the one-step-ahead prediction error, e_i, can be calculated as in Eq. 3.2.

$$e_i = x_i - z_{i-1} \tag{3.2}$$

With this prediction error, the smoothed variance estimation is calculated as in Eq. 3.3. In this equation, θ_e is the variance smoothing factor, $\sigma_e^2(i-1)$ is previous smoothed variance, and the time values are analogous to the smoothed event intensity calculation.

$$\hat{\sigma}_e^2(i) = \theta_e e_i^2 + (1-\theta_e)^{(t_{x_i} - t_{z_{i-1}})} \sigma_e^2(i-1) \tag{3.3}$$

Furthermore, a reasonable upper limit on the observed events, $UCL_x(i)$, is calculated as in Eq. 3.4. If z_i violates this threshold, an alarm is triggered. L_e is a constant, as in [30].

$$UCL_x(i) = z_{i-1} + L_e \sigma_e(i-1) \tag{3.4}$$

For each of the three real-time detectors, values for the smoothing factor, variance smoothing factor, and L_e constant can be configured with command-line parameters.

The real-time interactive session detector works merely with an adaptive threshold based on the explained EWMA logic. This is by choice, because its event are suspected keystrokes. Currently the detector inspects packets of characteristic size only from client (potential attacker) to server. Checking bidirectionally if the server responds also with a characteristic keystroke-sized packet, may make more accurate the detection logic. This has not been implemented to focus on the paper.

CUSUM logic

A drawback of an EWMA-based detection algorithm is that only violations of the adaptive threshold are considered, and not the intensity of these violations. A CUSUM algorithm considers the extent by which a threshold is violated, and thus considers the intensity of violations [34]. Because the brute-force detectors have to deal with events quite differently from suspected keystrokes, they use CUSUM techniques. The implemented logic for this is based on [36] and is as follows.

The recursive version of non-parametric CUSUM algorithm is shown in Eq. 3.5. In this equation, $Y_0 = 0$ and Z_n is defined as $Z_n = X_n - \beta$. X_n is the event intensity (per time slot), and it is transformed to have a negative mean over time using the transformation constant, β. This transformation is done, so that during normal event intensities, they will not accumulate into the CUSUM value over time.

$$y_n = max(y_{n-1} + Z_n) \tag{3.5}$$

The event intensity, X_n, is measured per time interval, T_n, and then passed to the algorithm. The length of each time interval, ΔT, can be optimized for the operational setting of the algorithm. Quite often CUSUM implementations calculate the new value only after T_n has ended. The CUSUM implementation in the extension, however, predicts the y_n value during the interval T_n. This is possible with event-driven recalculations and some housekeeping. The predicted value will never be lower than the final value, thus allowing detection even before a time window has passed. The predicted value can be compared to a given threshold to see if an alarm must be triggered.

For both detectors that currently use CUSUM logic, the values of the β constants, length of the time window, ΔT, and CUSUM threshold can be configured with command-line parameters. In the current implementation, alarms are thus raised on an anomalous number of `SYN` connections, or server-side initiated `FIN` bits. That is where research efforts halted.

Optimization of any of the parameters above based on training data and other techniques such as grid searches [35] has not been done, due to prioritization of research efforts for the paper. The brute-force detectors depend on the TCP analysis module, which has proven to work in production settings. The interactive session detector depends on SSH DPI, and the performance thereof has not been evaluated in a production setting.

3.2 Flow datasets

The paper reports on the gathering and use of several datasets. These datasets have been collected on the UT network and on peering links between the Czech National Research and Education Network (NREN) CESNET and, among others, the Austrian NREN ACOnet. The datasets are used to measure the occurrence of TCP retransmissions and control information in overall traffic, as well as SSH traffic. Should future work require measurements for another specific application, such as HTTP, then a subset of the datasets can be taken. This would save months of gathering data. Furthermore, the UT datasets (the CESNET data is third-party) can be anonymized and released to others, if desired. It should be said that these plans currently do not exist, but are possible. For both these reasons, the datasets are considered a deliverable.

The authentication logs of SSH servers, which serve as ground truth for the research in the paper, can also be anonymized if desired. Therefore these logs can be considered a deliverable as well. It should be noted that the collection and processing was made much easier due to contributions by the authors of [4], but specific work had to be done for additional post-processing. One of the scripts used in [4] for ground truth processing has been improved. Since this script is being maintained and published, the contributions made to it are a deliverable.

Chapter 4

Lessons learned

In this chapter I reflect on the research process, and state the most important lessons I feel that I have learned during the Master assignment. Perhaps I will enjoy revisiting some personal annotations in some time to come, to see if the lessons stuck.

> **Set reasonable goals** – Looking back at early formalizations of research questions, and my notes and ideas on which approaches I envisioned in addressing every single one of them as part of my Master assignment, it becomes clear that I had considered taking on too much action. Having to let some parts go was not an efficiency-related matter, but the nominal time available simply would not have allowed for everything to be done. Moreover, some approaches were found to be more interesting in terms of publishing than others. I've learned that there are several angles to consider when setting goals, and that it is important to reflect on them often enough down the road. There is always the proverbial todo list that is the *future work* section of a paper to show that you have considered it.
>
> **Publications are hard work** – During my B.Sc and M.Sc curricula, not many courses involved writing a paper. For the courses that did, I cannot say I enjoyed the mandatory paper writing all that much. Admittedly, the idea of writing a paper as part of the Master assignment was not my own, and I was somewhat hesitant at first. However, it seemed a fitting challenge, and in hindsight I cannot say that I regret agreeing to go down this road. It has been a good learning experience in terms of collaborating with others who have different opinions at times, thinking critically about how best to report on work, and how to present results in a clear fashion.
>
> **Language barriers do not exist** – Over the course of the assignment I have learned to work with quite a few programming languages. Parts of the research used *Python* and *Ruby*, languages with which

I had absolutely no prior experience. Moreover, the FlowMon Probe extension was my first *C* project of significant size, which includes advanced and challenging topics. Lastly, I can now write baffling *awk* scripts in support of data analysis.

Debugging is painstaking – Having to develop in *C* provided me ample opportunity to create memory leaks, and to encounter a multitude of concurrency issues with memory access, etc. Even if everything works smoothly in a local test environment, a much higher data throughput in a production setting all of a sudden reveals race conditions that had not been identified yet. What started with me learning of *Valgrind*[1] to help a third-party acquire some debug information, ended with me gaining hands-on experience with it in terms of finding memory leaks, reads/writes of freed memory etc. It has been an educative journey.

Go with the flow – It would be odd for me to pick sides at this very moment and join either the payload-based or flow-based intrusion detection camp, since the Master assignment builds on strenghts of both. Though, when it comes to flow-based monitoring approaches, I have learned that they are fun to work with, and a good choice for scalable solutions. As has been shown in this work, adding payload-based information to flow data using IPFIX is a viable solution when retainment of said information is necessary. So for strictly payload-based advocates, I think the advantages of using flow technologies, combined with the ability to retain information in a standardized way, may come as a pleasant surprise.

[1]`http://valgrind.org/`

Acknowledgements

Before starting my Master assignment, my programme coordinator advised me to search for an external graduation assignment. He based his advise in part on his belief that, generally speaking for internal assignments, supervisors only have time to schedule meetings once every few weeks. I am glad that I did not follow his advise. At the DACS chair, I was given the opportunity to work full-time on my assignment on location, in an office shared with my daily supervisor. This proved incorrect the notion of students barely being given the chance to receive feedback. Moreover, it allowed me to be in the good company of Ph.D. candidates, peer M.Sc and B.Sc students, and interns. To top things off, group lunches with people from other offices, and celebratory events that involved cake (even though I barely eat cake) added to a pleasant working environment. I feel lucky for having been given the chance to spend the last seven months this way, while working towards achieving my own goals.

I would like to thank Rick, my daily supervisor, in particular. His guidance and insights have been a great help in achieving results. I have enjoyed working with him, and consider our collaboration a success. Aiko and Anna's feedback at a handful of planned occassions made key contributions to the work and for that I am thankful. I am also appreciative of Luuk's help with practical problems at times. Finally, it was a blast sharing the same office space, lunches and so on with Jair and Ricardo.

Several other parties made contributions to the research process. I would like to thank Václav Bartoš for his allowing of, and assisting with, the collection of the CESNET datasets. Thanks go out to the INVEA-TECH Community Program, for providing parts of the measurement infrastructure used during this Master assignment. Finally, I would like to thank Petr Velan for his valuable help with software he authored.

Mattijs

Bibliography

[1] University of Twente, “Master guide 2012/2013 (Study guide),” July 2012, accessed on 10 Aug, 2014. [Online]. Available: http://www.utwente.nl/ewi/onderwijs/voorzieningen/studiegidsen/2012-2013/programme_guide_master_csc.pdf

[2] A. Pras, “How to get your paper accepted,” in *Proc. of the First International Conference on Autonomous Infrastructure, Management and Security (AIMS'2007)*, 2007.

[3] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, and A. Pras, “SSHCure: A Flow-Based SSH Intrusion Detection System,” in *Proceedings of the 6th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS'12*, ser. Lecture Notes in Computer Science, vol. 7279. Springer Berlin Heidelberg, 2012, pp. 86–97.

[4] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, “SSH Compromise Detection using NetFlow/IPFIX,” *ACM Computer Communication Review*, vol. 44, no. 4, 2014, (to appear).

[5] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, “Flow Monitoring Explained: From Packet Capture to Data Analysis with Netflow and IPFIX,” *IEEE Communications Surveys & Tutorials*, 2014.

[6] B. Claise, B. Trammell, and P. Aitken, “Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information,” RFC 7011 (Internet Standard), Internet Engineering Task Force, September 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7011.txt

[7] B. Claise, “Cisco Systems NetFlow Services Export Version 9,” RFC 3954 (Memo), Internet Engineering Task Force, October 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[8] T. Zseby, E. Boschi, N. Brownlee, and B. Claise, “IP Flow Information Export (IPFIX) Applicability,” RFC 5472 (Memo),

Internet Engineering Task Force, March 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5472.txt

[9] J. Postel, “Transmission Control Protocol,” RFC 793 (Internet Standard), Internet Engineering Task Force, September 1981. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[10] M. Allman, V. Paxson, and E. Blanton, “TCP Congestion Control,” RFC 5681 (Internet Standard), Internet Engineering Task Force, September 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5681.txt

[11] M. Mathis, D. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgment Options,” RFC 2018 (Internet Standard), Internet Engineering Task Force, October 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt

[12] L. Ludwig and M. Meyer, “The Eifel Detection Algorithm for TCP,” RFC 3522 (Internet Standard), Internet Engineering Task Force, April 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3522.txt

[13] INVEA-TECH, “FlowMon Probe,” accessed on 10 Aug, 2014. [Online]. Available: https://www.invea.com/en/products-and-services/flowmon/flowmon-probes

[14] Velan, Petr, “IPFIXcol,” accessed on 10 Aug, 2014. [Online]. Available: https://www.liberouter.org/ipfixcol/

[15] T. Ylonen and C. Lonvick, “The Secure Shell (SSH) Protocol Architecture,” RFC 4251 (Internet Standard), Internet Engineering Task Force, January 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4251.txt

[16] SANS Institute, “SANS (SysAdmin, Audit, Network, Security) Institute,” 1989, accessed on 13 jun 2014. [Online]. Available: https://www.sans.org/

[17] Lechtermann, Michael, “The OpenBL.org project,” 2009, accessed on 14 jul 2014. [Online]. Available: http://www.openbl.org/

[18] Hill, Gavin, “YouâĂŹre Already Compromised: Exposing SSH as an Attack Vector,” February 2014, accessed on 13 jun 2014. [Online]. Available: http://www.venafi.com/blog/post/youre-already-compromised-exposing-ssh-as-an-attack-vector/

[19] T. Ylonen and C. Lonvick, “The Secure Shell (SSH) Authentication Protocol,” RFC 4252 (Internet Standard), Internet Engineering Task Force, January 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4252.txt

[20] ——, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253 (Internet Standard), Internet Engineering Task Force, January 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4253.txt

[21] ——, "The Secure Shell (SSH) Connection Protocol," RFC 4254 (Internet Standard), Internet Engineering Task Force, January 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4254.txt

[22] S. Y. Lim and A. Jones, "Network Anomaly Detection System: The State of Art of Network Behaviour Analysis," in *International Conference on Convergence and Hybrid Information Technology (ICHIT'08)*, 2008, pp. 459–465.

[23] R. Hofstede and A. Pras, "Real-Time and Resilient Intrusion Detection: A Flow-Based Approach," in *Proceedings of the 6th International Conference on Autonomous Infrastructure, Management and Security (AIMS'12)*, 2012, pp. 109–112.

[24] R. Hofstede, A. Sperotto, and A. Pras, "Towards Real-Time Intrusion Detection for NetFlow and IPFIX," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM'13)*, 2013, pp. 227–234.

[25] R. Sadre, A. Sperotto, and A. Pras, "The Effects of DDoS Attacks on Flow Monitoring Applications," in *IEEE Network Operations and Management Symposium (NOMS'12)*, April 2012, pp. 269–277.

[26] P. Barford, J. Kline, D. Plonka, and A. Ron, "A Signal Analysis of Network Traffic Anomalies," in *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment (IMW'02)*, 2002, pp. 71–82.

[27] P. Velan, T. Jirsík, and P. Čeleda, *Design and Evaluation of HTTP Protocol Parsers for IPFIX Measurement.* Springer, 2013.

[28] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *Proceedings of the 10th USENIX Security Symposium*, 2001.

[29] R. Koch and G. Dreo Rodosek, "User identification in encrypted network communications," in *Proceedings of the 6th IEEE/IFIP International Conference on Network and Service Management, CNSM'10*, October 2010, pp. 246–249.

[30] N. Ye, C. Borror, and Y. Zhang, "EWMA techniques for computer intrusion detection through anomalous changes in event intensity," *Quality and Reliability Engineering International*, vol. 18, no. 6, pp. 443–451, November 2002.

[31] P. Casas, J. Mazel, and P. Owezarski, “Steps Towards Autonomous Network Security: Unsupervised Detection of Network Attacks,” in *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS’11)*, 2011, pp. 1–5.

[32] G. Fernandes and P. Owezarski, “Automated Classification of Network Traffic Anomalies,” pp. 91–100, 2009.

[33] Z. M. Fadlullah, T. Taleb, N. Ansari, K. Hashimoto, Y. Miyake, Y. Nemoto, and N. Kato, “Combating Against Attacks on Encrypted Protocols,” in *Proceedings of the 2007 IEEE International Conference on Communications, ICC’07*, June 2007, pp. 1211–1216.

[34] V. a. Siris and F. Papagalou, “Application of anomaly detection algorithms for detecting SYN flooding attacks,” *Computer Communications*, vol. 29, no. 9, pp. 1433–1442, May 2006.

[35] B. Krishnamurthy, S. Sen, Y. Zhang, F. Park, and Y. Chen, “Sketch-based Change Detection: Methods, Evaluation, and Applications,” pp. 234–247, 2003.

[36] T. Peng, C. Leckie, and K. Ramamohanarao, “Proactively Detecting Distributed Denial of Service Attacks Using Source IP Address Monitoring,” pp. 771–782, 2004.

Appendices

Appendix A

IEEE IM 2015 paper

This appendix contains the paper that was written as part of the Master assignment. It is titled *Unveiling Flat Traffic on the Internet: An SSH Attack Case Study.* The included version is ready for the *IEEE IM 2015* conference submission due date of September 15th, 2014. The track the work will be submitted to is the *Technical Paper Sessions.* At the time of this writing, the included paper is **CONFIDENTIAL**. Furthermore, this version is subject to revision post-graduation, pending both further peer review process, as well as the becoming available of extra datasets.

Unveiling Flat Traffic on the Internet: An SSH Attack Case Study

Mattijs Jonker, Rick Hofstede, Anna Sperotto and Aiko Pras
Design and Analysis of Communication Systems (DACS)
Centre for Telematics and Information Technology (CTIT)
University of Twente, Enschede, The Netherlands
m.jonker-1@student.utwente.nl, {r.j.hofstede, a.sperotto, a.pras}@utwente.nl

***Abstract*—Many types of brute-force attacks are known to exhibit a characteristic flat behavior at the network-level, meaning that connections feature a similar number of packets and bytes, and duration. Flat traffic is usually caused by repeating similar application-layer actions, such as login attempts in a brute-force attack. This characteristic is used by many intrusion detection systems, both for identifying the presence of attacks and – once detected – for observing deviations, pointing out potential compromises, for example. However, the flatness of network traffic may become indistinct when TCP retransmissions and control information come into play. In this paper, we show exactly that, based on an SSH attack case study. More specifically, we show that our approach dramatically improves the number of true detections of a state-of-the-art detection algorithm up to 20 percentage points, as well as increasing its accuracy – at no cost for analysis applications.**

I. Introduction

Flow monitoring has become the prevalent approach for monitoring larger-scale, high-speed networks [1]. By aggregating packets into flows, which are defined as sets of packets that pass by an observation point in a network during a certain time interval [2], flow monitoring is typically said to be more scalable than monitoring approaches that rely on storing and analyzing individual packets. The fact that many high-end packet forwarding devices are already shipped with support for flow export protocols like NetFlow [3] and IPFIX [2], makes flow monitoring a cost-effective monitoring approach at strategic observation points in the network. The scalability advantages of flow monitoring however also come at a cost. For example, it is widely known that measurement artifacts may be present in flow data that affect the data's accuracy in a negative way [4]. Also, by design, all packets that belong to a particular flow are accounted in the same way, making it impossible to differentiate classes of packets within the same flow.

Flow-based intrusion detection has become increasingly popular during recent years, since its proven applicability for various types of attacks [5]. One of these types of attacks are *dictionary* attacks, which aim at compromising targets by using lists of frequently-used login credentials (referred to as *dictionaries*). These attacks are often launched against SSH daemons, and the flow-based detection of these has been researched in [6]–[10]. A consensus in these works is that network traffic of such attacks exhibits 'flat' characteristics, meaning that the number of packets and bytes, and the duration of involved flows are alike. The problem with this notion is that network traffic of dictionary is not always as flat as one would expect, an observation that is supported by the datasets used in [7]. Investigation of the network traffic at the packet-level has revealed that TCP retransmissions and other TCP control information are the suspected causes, which are typically not identifiable at the flow-level.

Only two works so far have studied behavior of TCP retransmissions and control information on the Internet. Both works, published in the 1990's, focus on the behavior of TCP variants in terms of congestion control when facing packet loss, reordering or unexpected delays [11], [12]. The retransmissions in these works are introduced by means of simulating the aforementioned events, rather than measuring their occurrence on the Internet. To the best of our knowledge, there are no recent Internet measurements on the number of retransmissions and other variable TCP control packets. This paper bridges this gap.

In this paper, we study the impact of TCP retransmissions and other control information packets on network traffic in general and SSH intrusion detection in particular, both in a campus and backbone network. To do so, we have extended the set of fields exported using the IPFIX protocol, commonly referred to as IPFIX Information Elements (IEs), to include per-flow statistics on retransmissions and control information. Measurements have shown that we see up to 1 TiB and 5 TiB of retransmissions per week on typical campus and backbone network links. This underlines the hypothesis that supposedly flat traffic may turn non-flat. We study the effects of retransmissions and control information packets on SSH dictionary attacks, and compare the detection results of a state-of-the-art dictionary attack detection algorithm when the added information is used for detection instead of only traditional packet and byte counters in flow records.

The organization of this paper is as follows. In Section II, we provide background information on TCP retransmissions and control information, and describe how TCP can affect the 'flatness' of network traffic. Then, in Section III, we discuss the defined IPFIX IEs and the implementation needed for measuring TCP retransmissions and control information. A description of the acquired datasets and measurement results are provided in Section IV. In Section V, we validate this work

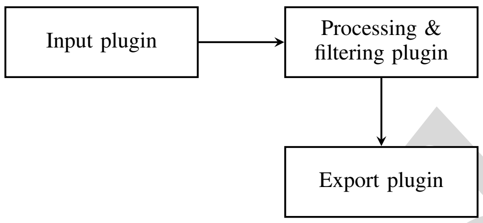

Fig. 1. INVEA-TECH FlowMon platform architecture, as of FlowMon Probe 6.x.

in the context of SSH intrusion detection, after which we draw our conclusions and state future work in Section VI.

II. Background

To facilitate reliable data delivery between endpoints, TCP uses a cumulative acknowledgement scheme in which sequence and acknowledgement numbers are used to signal the reception of data. In the absence of any feedback from the data receiver, a Retransmission TimeOut (RTO) is used to ensure delivery, which is based on the estimated Smoothed Round-Trip Time (SRTT). Due to unexpected delays or reordering of packets in the network, retransmissions can occur spuriously. For example, when a packet or its acknowledgement is delayed unexpectedly rather than lost, the RTO timer expires and the packet is retransmitted. Also, a fast retransmission may be sent when a certain number of consecutive duplicate acknowledgements is received, signalling the potential loss of a packet to the sender. Due to reordering of packets, duplicate acknowledgements may be sent even though no packet has gotten lost. These duplicate acknowledgements can trigger a spurious fast retransmission. In both examples, spurious retransmissions and their duplicate acknowledgements cause additional packets and bytes in network traffic and hence affect the potential flatness of a connection.

To optimize network throughput while avoiding congestion or overloading an endpoint, TCP uses several techniques, such as a flow control mechanism based on a sliding window, and the *delayed ACK* mechanism. The former requires the *receive window* to be signalled from receiver to sender, and under the latter data acknowledgements are held back for a brief delay to save overhead. If data or additional control information becomes available during the delay, the held back acknowledgement can be combined with this information. For some forms of control information, such as data acknowledgements and *receive window* changes, the *delayed ACK* mechanism and circumstances dictate whether a dedicated packet is sent to carry the control information to the endpoint. For example, if during a *delayed ACK* data is pushed down from the application-layer, the held back acknowledgement can be *piggybacked* with a data packet. This prevents sending a dedicated acknowledgement with no payload. Also, the *delayed ACK* mechanism allows for the cumulative acknowledgement of two data packets received in rapid success. This too saves sending a dedicated packet. If the *receive window* changes at the receiver, this information can be combined with a held back acknowledgement, again saving a dedicated packet. Sometimes, however, the *receive window* expands when there is no data to acknowledge, in which case a dedicated *window update* needs to be sent. Whether or not such dedicated packets are sent affects the flatness of network traffic.

There are also types of control information that are always sent in separate packets: *Zero Window probes* and responses, *KeepAlive Probes* and responses, and `RST` packets. Also, depending on the TCP implementation, a three-way `FIN` close sequence may not be supported, thereby potentially introducing an additional packet during the connection termination. Any of these additional packets obviously affects the flatness of network traffic as well.

It is important to note that the presence of the aforementioned situations mostly depends on network conditions, resource availability and scheduling on endpoints, whether or not there is data to send or acknowledge, timing, etc.

III. Implementation

To export information that allows for the discrimination of TCP retransmissions and control information in flow data, several IPFIX Information Elements (IEs) have been defined and implemented as part of a flow Metering Process. This section describes these IEs and the accompanying software that has been implemented.

We have defined IPFIX IEs for each of the TCP protocol characteristics that have been discussed in Section II. To facilitate the export of these IEs, we have developed an extension to INVEA-TECH's FlowMon flow exporter. This platform was chosen because of its highly customizable plugin architecture, and because we have full control over it in our networks. The complete architecture is shown in Fig. 1. It is based on plugins for data input, flow record processing & filtering, and export. Input plugins process data from a given source and are responsible for creating flow cache entries. The flow cache contains entries for each flow that is being metered. Process plugins allow for manipulations of these cache entries once they have been created, and are best suited for program logic that does not necessarily require packet payload anymore. The export plugin is responsible for exporting cache entries by sending flow records to a collector using NetFlow or IPFIX. From within these plugin types, actions can be hooked to when flow entries are added to, updated in or expired from the cache. Among these actions is the filtering of flow cache entries to prevent them from being exported.

Our extension comes in the form of an input plugin. The plugin measures TCP retransmissions and control information packets, and stores and maintains related counters in the flow cache. To recognize these particular packets, TCP conversations are analyzed in real-time by evaluating sequence and acknowledgement numbers, timestamps, flags, receive window sizes, and payload sizes. This implementation is heavily based on the TCP packet dissector used by *Wireshark*.[1]

[1]http://www.wireshark.org/

TABLE I
DATASETS

Dataset	Period	Duration	Packets	Bytes	Flows	Retransmissions		Control Information	
						Packets	*Bytes*	*Packets*	*Bytes*
UT1	June 2014	7 days	105.93 G	84.79 TiB	1.92 G	1.27 G (1.20%)	0.92 TiB (1.09%)	4.77 G (4.50%)	0.22 TiB (0.25%)
CESNET1	June 2014	7 days	119.72 G	105.95 TiB	3.46 G	4.63 G (3.87%)	4.82 TiB (4.55%)	8.42 G (7.03%)	0.38 TiB (0.36%)
UT2	July / August 2014	14 days	162.63 G	126.67 TiB	3.48 G	3.55 G (2.18%)	1.67 TiB (1.31%)	43.67 G (26.85%)	1.87 TiB (1.47%)
CESNET2	July / August 2014	14 days	113.39 G	107.21 TiB	13.01 G	2.17 G (2.08%)	0.84 TiB (0.87%)	10.52 G (10.12%)	0.43 TiB (0.44%)

For the TCP analysis to be accurate, it is crucial that packets in both directions of a TCP conversation pass the observation point. Otherwise, the housekeeping of sequence and acknowledgement numbers may be affected, which obviously impairs the analysis. The same is true when packets are lost downstream of the observation point. We are also aware that the TCP packet dissector used by *Wireshark* cannot but misclassify packets in its on-the-fly analysis in some cases, especially when packets are reordered. To optimize our plugins to work on high-speed links, e.g., of *10 Gbps* and higher, we accept these exceptional cases for the sake of performance.

IV. Measuring TCP Retransmissions & Control Information

Our first step towards understanding the impact of TCP retransmissions and control information is to measure them in two networks that are different in nature. Four datasets have been collected, as shown in Table I, consisting of only TCP flow data. Dataset *UT1* and *UT2* are collected on the campus network of the University of Twente (UT). This network features a publicly routable /16 network address block with connections to faculty buildings, student and staff residences, etc. Dataset *CESNET1* and *CESNET2* are collected on backbone links of the Czech National Research and Education Network (NREN); *CESNET1* is collected on a link between CESNET and the Austrian NREN ACOnet, while *CESNET2* is collected on a link between CESNET and the 'commercial Internet'. Due to the academic nature of these networks, the relative amount of traffic during summer holidays is considerably lower than during working days.

The remainder of this section is organized in two parts. First, in Section IV-A, we analyze retransmissions and control information in detail based on our measurements. After that, we perform a similar analysis only for SSH traffic in Section IV-B, given that the validation of this work (Section V) will be performed in the context of SSH intrusion detection.

A. *Overall Traffic*

Details on the number of retransmitted packets and bytes, and the amount of control information in terms of packets and bytes are shown in Table I. Several observations can be made. On the one hand, TCP control information is mostly visible in terms of packets. On the other hand, retransmissions contribute more towards the percentage of bytes, relatively speaking. Another observation is that there are many more packets with control information than there are retransmitted packets. This is mainly because many of the control information packet types, such as those that result from the *delayed ACK* mechanism, are sent under all network conditions, while retransmissions appear more frequently during network congestion, for example.

The distribution of retransmission types in terms of packets and bytes is shown in Table II. As can be observed, most retransmissions are regular retransmissions. Also, for each dataset, the fraction of the total number of bytes for the fast retransmission type is higher than the packet fraction. We believe this is because regular retransmissions can also contain no payload, e.g., retransmissions of TCP empty `SYN`

TABLE II
DISTRIBUTION OF RETRANSMITTED PACKETS AND BYTES

Dataset	Retransmissions		Fast Retransmissions	
	Packets	*Bytes*	*Packets*	*Bytes*
UT1	94.64%	90.83%	5.36%	9.17%
CESNET1	96.50%	95.46%	3.50%	4.54%
UT2	96.92%	91.78%	3.08%	8.22%
CESNET2	97.27%	90.41%	2.73%	9.59%

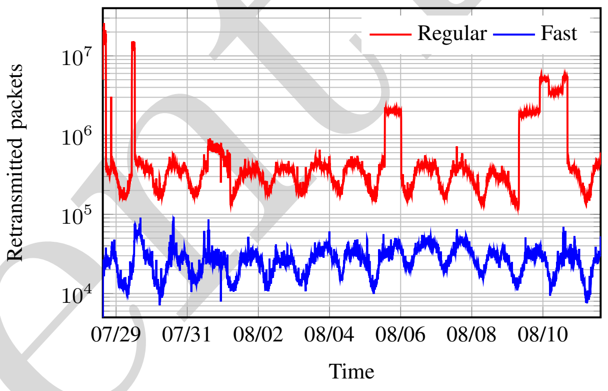

Fig. 2. Retransmissions over time.

TABLE III
DISTRIBUTION OF CONTROL INFORMATION PACKETS

Type	Dataset			
	UT1	*CESNET1*	*UT2*	*CESNET2*
Duplicate ACK	35.53%	37.12%	5.20%	10.42%
Non-piggybacked ACK	47.28%	41.16%	7.59%	35.01%
Consecutive empty ACK	–	–	83.23%	–
Window Update	12.50%	12.27%	1.96%	5.75%
Zero Window Probe	0.02%	$< 0.01\%$	$< 0.01\%$	7.84%
ZWP response	0.01%	$< 0.01\%$	$< 0.01\%$	7.78%
`RST`	–	–	0.91%	–
Four-way close packet	–	–	0.10%	0.58%
KeepAlive Probe	2.54%	6.64%	0.54%	31.50%
KeepAlive Response	2.12%	2.80%	0.46%	1.12%

TABLE IV
TCP RETRANSMISSIONS & CONTROL INFORMATION – SSH

Dataset	Retransmissions		Control Information	
	Packets	*Bytes*	*Packets*	*Bytes*
UT1	4.13 M (0.08%)	2.14 GiB (0.04%)	98.79 M (1.81%)	4.50 GiB (0.08%)
CESNET1	2.32 M (1.75%)	1.61 GiB (1.86%)	13.39 M (10.12%)	0.64 GiB (0.74%)
UT2	1466.33 M (16.79%)	157.11 GiB (2.95%)	1587.10 M (18.18%)	72.10 GiB (1.35%)
CESNET2	72.52 M (2.25%)	10.23 GiB (1.76%)	665.01 M (20.58%)	27.64 GiB (4.76%)

TABLE V
DISTRIBUTION OF RETRANSMITTED PACKETS AND BYTES – SSH

Dataset	Retransmissions		Fast Retransmissions	
	Packets	*Bytes*	*Packets*	*Bytes*
UT1	90.63%	75.38%	9.37%	24.62%
CESNET1	99.94%	99.89%	0.06%	0.11%
UT2	99.79%	97.25%	0.21%	2.75%
CESNET2	99.89%	99.14%	0.11%	0.86%

and `FIN` segments bring down the average number of bytes per retransmitted packet.

The number of retransmitted packets and fast retransmitted packets within every five-minute interval in the first two weeks of the *UT2* dataset is shown in Fig. 2. A diurnal pattern can be clearly identified, which follows the working hours at faculty buildings, and the presence of on-campus residents. While Table I provides absolute numbers, and as such is not specific about the points in time at which events occur, Fig. 2 shows that retransmissions occur at any time of the day. The two outlying groups of retransmitted packets around *5 Aug 18:00* and *10 Aug 18:00* coincide with severe SSH dictionary attacks from China that involve many retransmissions, which makes these anomalies visible in our measurements. These attacks will be discussed later, as part of the case study in Section V.

The distribution of the various types of control information packets is shown in Table III. As can be seen, packets relating to the *delayed ACK* mechanism, i.e., *non-piggybacked ACKs* and *consecutive empty ACKs*, account for large percentages of the total number of control information packets in each dataset. For example, *non-piggybacked ACK*s take up 47.28% and 41.16% in *UT1* and *CESNET1*, respectively. Another example is the *consecutive empty ACK*, with 83.23% in *UT2*. It should be noted that as some information was not exported in earlier datasets, the distributions can vary due to missing IEs.

Given the significant presence of TCP retransmissions and control information in our measurements in two networks that are different in nature, we conclude that these packets are omnipresent on the Internet. Also, we believe to have demonstrated that the flatness of originally flat network traffic on the Internet is likely affected by this omnipresence, as theorized in Section II.

B. SSH Traffic

The SSH traffic considered in this paper has been obtained by filtering the datasets presented in Table I for traffic on port 22. Details on the number of retransmissions and control information packets and bytes are shown in Table IV. Several observations can be made when comparing the results to the full datasets (listed in Table I). First, considering that the overall *CESNET1* dataset is significantly larger than *UT1*, the relative amount of SSH traffic in *UT1* is much larger than in *CESNET1*. For *UT2* and *CESNET2* the prior also contains much more SSH traffic, even though the dataset is marigi-nally larger in comparison. Second, the relative percentage of retransmissions is generally lower for SSH than in the full datasets for *UT1* and *CESNET1*. For *UT2*, however, it is much higher, namely 16.79% versus 2.55%. The reason behind this is that the *UT2* dataset contains several large-scale SSH attacks, as discussed previously alongside Fig. 2. Third, and similar to the full datasets, control information in the SSH datasets is more dominant than retransmissions in terms of packets and bytes.

As for retransmissions in SSH traffic, the distribution of these in terms of packets and bytes is shown in Table V. Compared to the distribution of retransmissions in the full dataset, it can be observed that a higher percentage in the SSH traffic is of the regular retransmission type. Taking the *UT2* dataset as an example, only 0.21% of retransmissions are classified as fast retransmissions, in contrast to a figure of 3.08% in the full dataset. Relative differences between other full datasets and their SSH-only traffic subset are following the same trend. We believe that this is the case because it is less common for SSH connections to have four or more packets with payload sent by one side within a short period. In other words, there are not enough consecutive data packets to trigger a fast retransmission.

The distribution of control information in SSH traffic is

TABLE VI
DISTRIBUTION OF CONTROL INFORMATION PACKETS – SSH

Type	Dataset			
	UT1	*CESNET1*	*UT2*	*CESNET2*
Duplicate ACK	33.53%	4.71%	2.17%	2.68%
Non-piggybacked ACK	64.34%	94.91%	7.82%	68.00%
Consecutive empty ACK	–	–	89.19%	–
Window Update	2.11%	0.25%	0.39%	0.31%
Zero Window probe	$<$ 0.01%	$<$ 0.01%	0.00%	3.14%
ZWP response	$<$ 0.01%	$<$ 0.01%	0.00%	3.14%
`RST`	–	–	0.34%	–
Four-way close packet	–	–	0.08%	1.87%
KeepAlive probe	0.02%	0.13%	0.00%	20.45%
KeepAlive response	0.02%	$<$ 0.01%	0.00%	0.41%

shown in Table VI. This distribution features several key differences compared to the full datasets (see Table III). A prime example is the significantly lower number of packets related to *KeepAlive*. A possible explanation for this is that the majority of SSH connections is short-lived, or otherwise active enough to not trigger the TCP *KeepAlive* timer, which is typically in the order of hours [13]. Another observation is that while the distribution of control information types is very similar within the full datasets collected with the same version of the extension, this is not the case anymore for the SSH datasets. For example, in the *UT1* dataset, roughly 2% of all SSH packets are *Window Updates*, while *Window Updates* account for only 0.25% in *CESNET1*. Also, for *CESNET1*, *non-piggybacked ACKs* are at a staggering 94.91%, whereas in *UT1* they account for 64.34%. We believe these differences stem from the fact that a lot more data is sent within SSH connections on the UT network. Furthermore, for the *UT2* and *CESNET2* datasets it can be seen that *four-way close* features only very small percentages of control information packets, namely 0.08% and 1.87%, respectively. This leads us to believe SSH network traffic is typically not affected much by this type of control information.

V. VALIDATION

In this section, we quantify and study the effects of TCP retransmissions and control information in the context of flow-based SSH intrusion detection. The case study, together with its motivation, is presented in Section V-A. The validation methodology is discussed in Section V-B. Finally, in Section V-C, we present the validation results.

A. Case Study: SSH Intrusion Detection

Flow-based detection of dictionary attacks is typically performed by comparing the characteristics of two or more flow records to identify possible attacks. In [14], it is shown that these attacks typically consist of three phases, as shown in Fig. 3, which feature specific flow-level characteristics. During

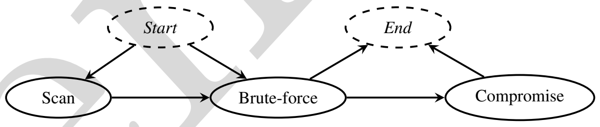

Fig. 3. Dictionary attack phases, from [1].

the *scan* phase, an attacker probes for the presence of specific services on one or more hosts in a network. During the *brute-force* phase, a high-intensity dictionary attack is performed on one or more targets on which the service is found active. The *brute-force* phase typically contains many flow records with an equal number of *Packets-Per-Flow* (PPF), since flows featuring an equivalent number of login attempts between the same client and server typically consist of the same number of packets. Should a compromise ensue, the *compromise* phase is reached.

In this work, we focus on the *brute-force* phase, since it is the only phase where 'flat' traffic should be predominant. The concept of an equal number of PPF, i.e., flat traffic, for brute-force attacks detection forms the basis of the state-of-the-art *brute-force* phase detection algorithm presented in [7], which considers the number of PPF in consecutive flow records. The algorithm starts with a preselection of source and destination IP address pairs for which flow records have a PPF value of $x \in [11, 51]$. For each of these preselected address pairs, the most frequently used PPF value is taken as the *baseline* for determining brute-force behavior. This *baseline* is then used for comparing consecutive flow records with identical PPF values to. If at least N consecutive flows feature the *baseline* number of PPF, a brute-force attack is recognized. We set the threshold N to 5[2], and the result of the detection algorithm is a list of attacks. In the remainder of this work, we define an attack as a set of one or more targets featuring brute-force behavior for a given attacker, i.e., where every target in the set has reached N. A tuple is defined as a pair of attacker and target, such that every attack consists of one or more tuples.

If more than N flow records feature the same number of PPF, a dictionary attack is detected. On the one hand, false negatives, i.e., undetected attacks, can occur in this context when dictionary attack flows end up with diverse PPF values, causing the threshold N to not be reached, even though the application-layer activity remains the same. On the other hand, false positives, i.e., false alarms, can occur when non-dictionary attack flows end up with equal PPF values, enough to reach the threshold N.

B. Methodology

We perform the validation of this work by executing the state-of-the-art detection algorithm presented in [7] on the datasets listed in Section I. Instead of only considering the regular number of PPF in the detection algorithm, as would

[2]Note that 5 consecutive flow records with the same number of PPF would represent 15 failed login attempts in a benign situation, as explained in [7], which we consider highly unlikely.

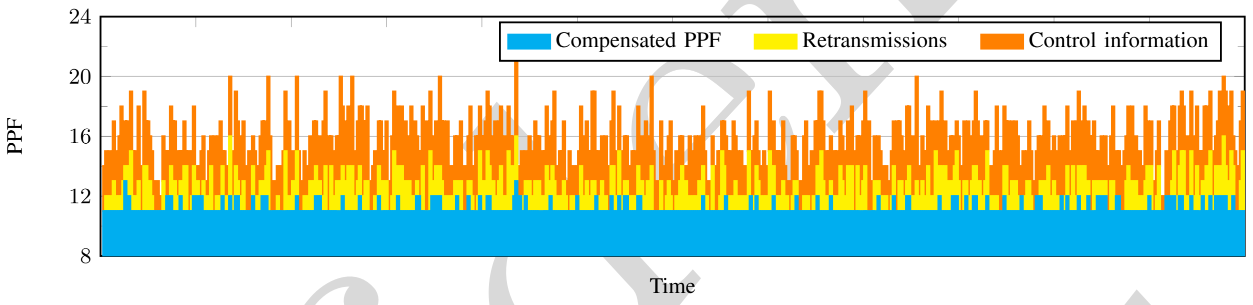

Fig. 4. Compensated brute-force flow records.

be the case in a regular flow monitoring setup, we also consider a compensated number of PPF. The compensated number of PPF consists of the number of the total number of packets metered for each flow minus the number of packets of all retransmissions and TCP control information fields. Ultimately, this should result in flat traffic.

By comparing the detection results when using non-compensated and compensated data, we can quantitatively evaluate the gain of 'flattening' traffic in the context of SSH intrusion detection. We perform the comparison in two dimensions – attacks and tuples – as this allows us to discover potential differences in the impact of compensation. Although comparing the number of detections in terms of attacks and tuples before and after compensation provides an indication of the detection improvements, it does not reveal anything about to accuracy of these detection outcomes. To assess these accuracies, we have collected the authentication logs of 58 machines within the campus network of the UT – 56 servers and 2 honeypots – to serve as the ground-truth for validation. These authentication logs are the only means of validating whether a machine has indeed been under attack. Since we only have the logs for UT hosts, we only consider the *UT1* and *UT2* datasets in this part of the validation.

In the authentication logs, a minimum number of failed attempts must be encountered for the behavior to be considered a dictionary attack. Since the detection algorithm considers at least N consecutive flow records, only N or more connections to the SSH server that contain at least one failed attempt are considered. This comes down to at least 5 sessions with one or more authentication failures each. A list of attacks featuring this property is used as the ground-truth for validation. We use this ground-truth for expressing the accuracy of the detection algorithm, both in terms of attacks and tuples, by comparing detection results to the ground-truth based on the following metrics:

- *True Positives* (TP) – Attacks/tuples correctly classified to feature a *brute-force* phase, for which 5 or more sessions with authentication failures are reported in the ground-truth.
- *False Positive* (FP) – Attacks/tuples incorrectly classified to feature a *brute-force* phase, for which less than 5 sessions with authentication failures are reported in the ground-truth.
- *True Negatives* (TN) – Attacks/tuples correctly classified to not feature a *brute-force* phase, for which less than 5 sessions with authentication failures are reported in the ground-truth.
- *False Negatives* (FN) – Attacks/tuples incorrectly classified to not feature a *brute-force* phase, for which 5 or more sessions with authentication failures are reported in the ground-truth.

Using these metrics, we can evaluate the differences in the detection algorithm for the non-compensated and compensated cases in terms of accuracy (Acc), which is defined as follows:

$$Acc = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \qquad (1)$$

In addition, to understand the relation between TCP control information and retransmissions, and geographical locations, we determine the physical origin of attacks and tuples based on a snapshot of the MaxMind GeoIP[3] database at the time of the measurements. The physical location can reveal why certain attacks or the majority of tuples are more likely to be detected only after compensation, as we hypothesize that retransmissions are strongly bound to the geographical distance between attackers and targets.

C. Results

The best way to visualize the achievements of this paper is by means of a plot, as shown in Fig. 4. This figure shows the traffic in terms of the number of PPF over time between a single tuple of attacker and target. Clearly, the original network traffic (i.e., the sum of the three series in the figure) is not flat, but after compensating for control information packets and retransmissions, traffic that is almost flat remains. Occasional variations in the remaining number of PPF after compensation are the result of the performance trade-off discussed in Section III. We accept these variations, considering that most attacks feature a large enough number of flows to reach the threshold N.

The results of operating the detection algorithm on the considered datasets, both with and without PPF compensation,

[3]We have used MaxMind's *GeoLite City* database, which can be retrieved from http://dev.maxmind.com/geoip/legacy/geolite/

TABLE VII
TOP FIVE ATTACK ORIGINS – ATTACKS

Dataset	Country	Non-compensated	Compensated
UT1	China	370	494 (+34%)
	Netherlands	63	72 (+14%)
	Russian Federation	42	45 (+7%)
	Other	142	159 (+12%)
	Total	**617**	**774 (+25%)**
CESNET1	Canada	5	49 (+880%)
	France	3	30 (+900%)
	Germany	4	5 (+25%)
	Other	14	19 (+36%)
	Total	**26**	**99 (+281%)**
UT2	China	851	1089 (+28%)
	Venezuela	190	222 (+17%)
	Netherlands	151	319 (+111%)
	Other	816	928 (+14%)
	Total	**2008**	**2558 (+27%)**

TABLE IX
TOP FIVE ATTACK ORIGINS – TUPLES

Dataset	Country	Non-compensated	Compensated
UT1	China	6137	10040 (+64%)
	Vietnam	1048	1056 (+1%)
	United States	638	658 (+3%)
	Other	2027	8346 (+311%)
	Total	**9850**	**14074 (+43%)**
CESNET1	Poland	1186	2365 (+99%)
	France	10	613 (+6030%)
	Canada	19	520 (+2637%)
	Other	369	487 (+32%)
	Total	**1584**	**3985 (+152%)**
UT2	China	12014	22888 (+91%)
	Netherlands	4536	4811 (+3%)
	United States	2029	2597 (+26%)
	Other	8877	10340 (+16%)
	Total	**27456**	**40636 (+48%)**

TABLE VIII
DETECTION PERFORMANCE – ATTACKS

Dataset	Logged attacks	TPR	FPR	TNR	FNR	*Acc*
UT1	125	0.696	0.041	0.959	0.304	0.839
compensated		0.880	0.061	0.939	0.120	0.912
UT2	423	0.671	0.081	0.919	0.329	0.803
compensated		0.811	0.085	0.915	0.189	0.866

TABLE X
DETECTION PERFORMANCE – TUPLES

Dataset	Logged attacks	TPR	FPR	TNR	FNR	*Acc*
UT1	794	0.623	0.021	0.979	0.377	0.810
compensated		0.786	0.033	0.967	0.214	0.881
UT2	2236	0.434	0.046	0.954	0.566	0.696
compensated		0.625	0.055	0.945	0.375	0.787

are shown in Table VII for attacks and Table IX for tuples. The number of detected attacks and tuples is considerably higher after compensation for all datasets. In *CESNET1*, the total number of detected attacks is almost four times higher after compensation, i.e., from 26 to 99. The improvement in terms of tuples is almost threefold, at 152%. The reason for this high improvement in terms of attacks is that in many cases, the effects of retransmissions and control information hinder the detection for all tuples of an attack and, as such, the attack itself is also not detected. The improvement in detecting attacks and tuples after compensation is inverted for *UT1*, where we see a gain of 25% in terms of attacks and 43% in terms of tuples. Similarly, for *UT2*, the improvements are 27% and 48%. The reason for this inversion in *UT* datasets is that for many attacks there is at least one target of the attack that triggers detection of both its tuple and the attack as a whole, even without compensation.

Since we assume that retransmissions depend in part on the geographical location of and route between attacker and target, we show for each dataset the three countries from which most attacks originate, both in terms of attacks (Table VII) and tuples (Table IX). The total number of countries involved in attacks is 38 for *UT1*, 11 for *CESNET1*, and 51 for *UT2*. Furthermore, we show the number of attacks and tuples reported only after compensation for those countries.

Several observations can be made from the results. First, regarding *UT1* and *UT2*, many attacks that are detected only after compensation have the attacking host located in China, with a figure of 124 attacks and 3903 tuples for *UT1*, and 238 attacks and 10874 tuples for *UT2*. While China easily outperforms the other countries in terms of attacks and tuples, the relative increase of the number of attacks and tuples not reported until after compensation from China is also relatively high in the *UT* datasets. These increases are 34% and 64% for attacks and tuples in *UT1*, respectively. For *UT2*, the increase in the number of attacks from China is 28%, and tuples is at a staggering 91%. Second, we can observe even more extreme patterns for *CESNET1*, where the overall number of attacks and tuples reported only after compensation has increased by 281% and 152%, respectively. Canada is most dominant here in terms of both attacks and targets, and China is missing completely; since the peering link from which the *CESNET1* dataset has been collected has no direct connection to China, it carries only little Chinese traffic. All these observations make us conclude that TCP control information and retransmissions are indeed strongly bound to the distance in geographical location between attacker and target.

Thus far we have shown the different detection results when using compensated and non-compensated data. However, we have yet to compare these detection results to our ground-truth, authentication logs from 58 machines on the campus network of the UT. Since the ground-truth covers only a subset of the machines considered before, the number of attacks and tuples reported in the remainder of this section is lower than reported in Table VII and Table IX.

The detection performance of the detection algorithm in terms of attacks is shown in Table VIII, where we again divide the results in both compensated and non-compensated. Analogously, the detection performance in terms of tuples in shown in Table X. In both tables, we use the percentages of the previously introduced evaluation metrics. For example, the *True Positive Rate* (TPR) is the percentage of correctly identified attacks/tuples for which 5 or more sessions with authentication failures are reported in the ground-truth. The overall conclusion of the results is that compensation of the number of PPF yields a significantly improved TPR for both attacks and tuples. The TPR for attacks has improved from 70% to 88% for *UT1*, and 67% to 81% for *UT2*. For tuples, the figures are from 62% to 79%, and from 43% to 62%. These major improvements come at a minor cost in terms of false detections of roughly 1%. Also the accuracies for both attacks and tuples have improved significantly, from 84% to 91%, and 81% to 88%, respectively, for *UT1*. For *UT2*, this is from 80% to 87%, and 70% to 79%, respectively. We believe that the slight increase of false detection is the case because of misclassified packets, which in some cases cause benign network traffic to mimic dictionary attacks by becoming flat. These false positives are thus coupled to the performance trade-offs made in the plugin, as explained in Section III.

VI. Conclusions

In this paper, we have measured the impact of TCP control information and retransmissions both on networks in general, and on flow-based intrusion detection in particular. We have started by hypothesizing that these types of traffic may negatively affect analysis applications that assume traffic to be flat in terms of packets, bytes and duration. And yes, we can confirm that such applications are indeed impaired by retransmissions and control information, especially when the application-layer activity is assumed to remain the same, such as during dictionary attacks. This is also confirmed by our measurements, which have shown that these types of traffic are omnipresent in any network: In terms of bytes, retransmissions can account for 1-5% of the total number of packets, while control information contributes significantly more: 4-27%.

Besides measuring the presence of retransmissions and control information, we have analyzed the impact of using a compensated number of PPF in a state-of-the-art dictionary attack detection algorithm for SSH. The results unmistakably demonstrate that flow-based intrusion detection benefits from a compensated number of PPF; the TPRs of the detection algorithm have improved for attacks and tuples from 70% to 88%, and from 62% to 79%, respectively. Moreover, the accuracies increased from 84% to 91%, and 81% to 88%. These improvements come at no cost for the analysis application.

As future work, we consider analyzing variability in application-layer protocols. This variability can be performed both intentionally and unintentionally. In the former case, protocols like SSH use padding, for example, to make sure that all packets have a size required for the negotiated encryption algorithms. In the latter case, these protocols also allow for inserting additional packets, such that hardly two flows in an attack appear similar. Detecting and using the described forms of variability may aid in obtaining even flatter traffic patterns.

Acknowledgments

Special thanks go to Václav Bartoš (CESNET) for his valuable contributions to the research process, and the INVEA-TECH Community Program for providing parts of the measurement infrastructure used in this paper. This work was partly funded by FLAMINGO, a Network of Excellence project (ICT-318488), and SALUS, a STREP project (ICT-313296), both supported by the European Commission under its Seventh Framework Programme.

References

[1] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis with Netflow and IPFIX," *IEEE Communications Surveys & Tutorials*, 2014.

[2] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," RFC 7011 (Internet Standard), Internet Engineering Task Force, September 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7011.txt

[3] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Memo), Internet Engineering Task Force, October 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[4] R. Hofstede, I. Drago, A. Sperotto, R. Sadre, and A. Pras, "Measurement Artifacts in NetFlow Data," in *Proceedings of the 14th International Conference on Passive and Active Measurement, PAM'13*, ser. Lecture Notes in Computer Science, vol. 7799. Springer Berlin Heidelberg, 2013, pp. 1–10.

[5] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-Based Intrusion Detection," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[6] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, and A. Pras, "SSHCure: A Flow-Based SSH Intrusion Detection System," in *Proceedings of the 6th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS'12*, ser. Lecture Notes in Computer Science, vol. 7279. Springer Berlin Heidelberg, 2012, pp. 86–97.

[7] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, "SSH Compromise Detection using NetFlow/IPFIX," *ACM Computer Communication Review*, vol. 44, no. 5, 2014, (to appear).

[8] J. Vykopal, "Flow-based Brute-force Attack Detection in Large and High-speed Networks," Ph.D. dissertation, Masaryk University, 2013.

[9] M. Drašar, "Protocol-Independent Detection of Dictionary Attacks," in *Proceedings of the 19th EUNICE Open European Summer School, EUNICE'13*, 2013, pp. 304–309.

[10] M. Vizváry and J. Vykopal, "Flow-based detection of RDP brute-force attacks," in *Proceedings of 7th International Conference on Security and Protection of Information, SPI'13*, 2013, pp. 131–138.

[11] J.-C. Bolot, "End-to-End Packet Delay and Loss Behavior in the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 23, no. 4, pp. 289–298, 1993.

[12] T. Lakshman and U. Madhow, "The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss," *IEEE/ACM Transactions on Networking*, vol. 5, no. 3, pp. 336–350, 1997.

[13] R. Braden, "Requirements for Internet Hosts – Communication Layers," RFC 1122 (Internet Standard), Internet Engineering Task Force, October 1989. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[14] A. Sperotto, "Flow-Based Intrusion Detection," Ph.D. dissertation, University of Twente, 2010.