Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze

Master Thesis

Invasion of Privacy Using Fingerprinting Attacks

David Sauter
<dave.sauter@gmail.com>

Advisors

Martin Burkhart
Dominik Schatzmann

Supervisor

Prof. Dr. Bernhard Plattner

Communication Systems Group (CSG)
Computer Engineering and Networks Laboratory (TIK)
Department of Information Technology and Electrical Engineering

16.09.2008 - 15.03.2009

Acknowledgments

I would like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to write this thesis at the Communication Systems Group and for the supervision of my work.
I would like to thank both my advisors Martin Burkhart and Dominik Schatzmann for their continuous support and the productive meetings and discussions during the progression of my thesis. I'm especially grateful for the many useful suggestions and the constructive feedback that guided me through my work. It was a very enjoyable cooperation with them.
Thanks go to my colleagues and lab mates for the interesting discussions and suggestions during the last half year.
Special thanks go to my girlfriend and family for the unconditional support and patience.

Abstract

The Internet of today has grown to a large scale infrastructure for business operations and different sorts of communities. Unfortunately attacks on web sites and web services increase every year and the Internet security or the lack thereof has a direct impact on business operations of many companies. The network security research community still lacks network activity logs from independent sources to make reliable estimates and forecasts about possible threats to the Internet or individual companies.
Many companies are reluctant to share information because they want to protect the users of the network on one hand and the network itself on the other hand. Even anonymization of the logs could only solve this problem partially, since attacks on particular anonymization schemes were discovered quickly. This makes most companies feel uneasy about releasing network traces because the security of the anonymization schemes remains unclear.
The goal of this thesis is to show that anonymization is not safe to apply in the context of network activity logs by using an approach called active fingerprinting. While traffic is captured, we inject packets with special attributes (the fingerprint) into the monitored network. The fingerprints are only known to us and should make those packets recognizable even through the following anonymization of the traffic. We show that simple anonymization approaches fail to counter this attack technique completely and even the most elaborate anonymization schemes have a hard time ensuring the security of the captured data.
To this end we have developed a framework for injecting packets and comparing injected with captured packets. We demonstrate the effectiveness of the framework on live data by successfully deanonymizing individual hosts. Finally we provide some recommendations for countermeasures against active fingerprinting and explain why existing countermeasures don't work in this setting.

Zusammenfassung

Das Internet ist zu einer grossflächig verteilten Infrastruktur für Unternehmen und einer Plattform für diverse andere Gemeinschaften geworden. Leider mehren sich die Angriffe auf Webseiten und Internet-Dienste von Jahr zu Jahr, und viele Unternehmen spüren einen direkten Einfluss auf ihre Geschäfte. Was den Netzwerk-Sicherheitforschern immer noch fehlt, sind Netzwerk-Daten von unabhängigen, unterschiedlichen Quellen, um Gefahren für das Internet und einzelne Firmen besser und früher vorherzusagen oder abzuschätzen.
Viele Firmen zögern jedoch solche Daten bereitzustellen, weil sie einerseits die Endnutzer des Netzwerkes und andererseits das Netzwerk an sich schützen wollen. Die Daten zu anonymisieren half auch nur teilweise, weil man rasch Angriffe auf diese jeweiligen Anonymisierungstechniken fand, und als Folge davon fühlten sich viele Firmen immernoch unsicher, ob sie Netzwerk-Daten öffentlich zugänglich machen sollten.

Das Ziel dieser Arbeit ist zu zeigen, dass das Konzept der Anonymisierung generell nicht sicher ist im Bezug auf Netzwerk-Daten, anhand von einer Technik, die "aktives fingerprinting" genannt wird. Noch während die Netzwerk-Aktivität aufgezeichnet wird, schicken wir speziell präparierte Pakete mit Eigenschaften, welche nur uns bekannt sind (der Fingerprint), in das Netzwerk. Dies soll später ermöglichen, dass die geschickten Pakete trotz einer Anonymisierung erkannt werden können. Unsere Absicht ist zu zeigen, dass einfache Anonymisierungen kläglich scheitern, sich gegen diese Technik zu verteidigen, und auch komplizierte Anonymisierungen keinen leichten Stand haben, die Originaldaten zu schützen.
Um dies zu realisieren haben wir ein Framework geschrieben, um Pakete zu verschicken und um verschickte Pakete mit aufgezeichneten Daten zu vergleichen. Wir werden im Laufe der Arbeit aufzeigen, wie effektiv das Framework auf realen Daten arbeitet und wieviele Rechner es erfolgreich deanonymisieren konnte. Zum Schluss werden wir Empfehlungen für Gegenmassnahmen gegen das aktive fingerprinting besprechen und aufzeigen, warum existierende Gegenmassnahmen in diesen Falle versagen.

Contents

1 Introduction

The Internet has become a mass phenomenon today and is used by an estimated 6 billion people [13] world wide. Most users are familiar with the World Wide Web (WWW), but the Internet is much larger and more diverse than that. It has its own dynamics and grows by the second. Although the concepts and techniques behind the Internet are human inventions and generally very well understood, the ongoing dynamics and complex interactions between hosts are a constant topic of research nowadays.
Over the last years, the Internet has suffered an increasing amount of attacks on large infrastructures and companies. Since the Internet lacks an organization responsible for its security and protection, each company and organization tries to defend against attacks on its own. Methods to fend off attackers are publicly discussed and shared, and there are even efforts to improve the overall security of the Internet through wide spread detection mechanisms. Clearly missing are detailed network activity logs from distributed and independent sources. Assuming each company would release logs of their captured network traffic on a regular basis, researchers would be able to combine those records to get an impression of most of the traffic on the Internet at any particular time. Those records could help identify threats to a single company but also to the Internet in general [24].
The reasons for such reports not being publicly available are obvious. Companies are reluctant to share their private network activity records because they need to protect the privacy of their users as well as the integrity of the network itself. Activity logs typically include internal IP (Internet Protocol) addresses of a network allowing insights into the network architecture, and possibly even application level information, which allows to trace each user in the network and profile his activities in detail. Knowledge about the internal structure of a network might simplify attacks and even provide completely new ways of penetrating it. In the wrong hands, a full network trace of a company could therefore prove disastrous. So we have companies unwilling to share internal network activity logs due to anonymity and security constraints on one hand, and network security researchers that would love to analyze such records to enhance general security on the other hand.
A solution to this dilemma was proposed some years ago and involves anonymization of the security logs. The idea is to anonymize the logs to a state where nobody would be able to correctly correlate single users to their masked pseudonyms in the log. Several unanswered questions are raised when anonymizing a record. Such as which fields should be anonymized and which fields are better left untouched? Which anonymization technique is applied to which field? Are the applied techniques secure and to what degree or in which circumstances do they leak information? While anonymization is still an active topic of research, there are many techniques already available to us [23, 1]. Which fields

one can or even should anonymize is still not clear today, apart from the obvious fields like source- and destination IP.
One of the reasons for this is that we simply do not have concrete means to measure the strength of an anonymization scheme and there is no theory about what anonymization can achieve exactly in terms of security. Another reason is that anonymization is a trade-off between security and utility of the data. As a consequence, there are no hard security proofs of anonymization schemes and every time someone comes up with a new scheme, people try to break this particular approach and attacks on a scheme are discovered rather quickly. Today we're in a state where some anonymization schemes look promising, but for most schemes we either already know or at least suspect possible attacks.

1.1 Goal of this Thesis

This thesis is looking the opposite way than what has been discussed so far. We don't want to prove the security of any particular anonymization scheme, but rather display the potential insecurity of most or all of the existing schemes known so far and of anonymization in general in context of network activity logs. This is achieved by means of a technique called *fingerprinting*.
There has been a great deal of research going on in the field of "passive fingerprinting" in the last years [20, 2, 14] with emphasis on breaking permutations [4, 21]. This method assumes an attacker to observe the characteristics of the obscured data closely, and then correlate it with known traffic patterns in the monitored network. This can be achieved by consulting public information sources like search engines or DNS (Domain Name System) records. The other approach, that has been discussed or at least mentioned in [1, 4, 2], is called "active fingerprinting". In contrast to passive fingerprinting, this technique allows the attacker to manipulate the traffic that is being captured and anonymized. The malicious user injects traffic while it is being captured, which changes the resulting traffic patterns slightly. In this scenario, an attacker can actively influence the distribution of the traffic and is therefore much more powerful than a passive attacker.
The actual task is to develop a framework for active fingerprinting attacks and test its effectiveness on live traces of network data. The framework should be tested against various existing anonymization schemes and reveal how effective they are in countering the fingerprinting. Our intention is then to demonstrate that the framework can deal with even the most aggressive of anonymization methods, at the cost of revealing more and more about the attackers identity. During the tests we will focus on deanonymizing IP addresses e.g. those addresses that belong to the victims. To conclude we will present and analyze new ways of defending against active attacks.

1.2 Setup

The task of capturing the network traffic will be performed by the five border routers of the Swiss Education and Research Network SWITCH [26]. In the tests we will inject packets from outside into the SWITCH network, where the packets will be captured. Everything inside the SWITCH network is referred to as the "monitored segment" and all traffic that passes through the border routers is captured and exported. The captured data is stored in Cisco IOS NetFlow [6] format, so we will limit our research and analysis to this format.
As a next step, the data is stored locally and then anonymized according to some policy. With a detailed log of our injected packets in one hand, and the anonymized data from the network in the other hand, we are ready to analyze the traffic. Now it is a matching problem to find the fingerprinted flows withing the captured flows. The framework will work on two different hosts, namely a host outside the SWITCH network which will perform the injection, and a host inside[1] the network which will analyze and compare the two flow sets.

1.3 Assumptions

There are some crucial assumptions that we used in the implementation of the framework. For once, the border routers are expected to work nearly flawlessly and capture packets with a high reliability and accuracy down to single packets. We did not compensate for possible router errors and for missing packets. We will also use standard cryptographic assumptions that the attacker knows all public values of an anonymization scheme and furthermore that he has access to meta data about the capturing, meaning he knows time and place where the logging took place. The attacker is assumed to know the anonymization beforehand and is therefore able to inject packets which will work best with the current method of anonymization. We also make the assumption that packets are captured in real time and no packet sampling is used.

1.4 Outline

In the upcoming Section 2 we will give a detailed overview about the current state of research and give an introduction into anonymization and pseudonymization. This section reviews state-of-the-art techniques and also gives a short summary about active fingerprinting attacks and possible countermeasures against fingerprinting in general. Section 3 will subsequently introduce the fingerprinting framework PIFF (**P**acket **I**njection and

[1]it does not need to be inside, but it is in our setup due to the flows being available there

Fingerprinting **F**ramework) and offer a quick primer into the code. Design methodologies and practices as well as overall framework design and file structures will be discussed. The following Chapter 4 presents the analysis of the tests that were performed with the framework and shows the results in various plots, whereas the last section, Chapter 5, will conclude with a discussion of the results and give an outlook for future work in this direction of research. This chapter will discuss the effectiveness of active fingerprinting in general and present working countermeasures against it.
In the Appendix is the original task description along with the schedule of the thesis. There you will also find a user manual which explains the basic steps with the framework and instructions on how to compile and run the framework on your machine. Furthermore the Appendix contains the original attack scripts used in the tests.

2 Literature Review and Related Work

Ideally each network administrator would capture all the traffic he observes and then release it for further study. Unfortunately such network activity logs contain confidential information, such as IP addresses or payloads, which users as well as administrators don't want to become public. As a consequence such information is either stripped from the logs or kept and obscured before publishing. We will give a short summary of anonymization methods currently in use and briefly discuss their applications.

2.1 Anonymization schemes

When obscuring data, one can take two separate approaches: One can try to *anonymize* a value, effectively trying to make it indistinguishable from other values in that field or one can assign it a *pseudonym* by replacing the actual identity with an alternate one. Both may be achieved in several manners and have separate applications and security concerns.

2.1.1 Anonymization

A subject which is not identifiable within a set of subjects ("anonymity set") is called anonymous. Of course the larger the set, the better for the subjects anonymity. In our case, a value in a field of exported data should not be identifiable from other values in that particular field.
The first method is to *remove* the contents of a field or replace the value with a constant (black marker approach), which is quick and simple to perform. This leaves absolutely no information about that field and is equivalent to removing the field from the record altogether. A very similar approach is to *randomize* sensitive data within a field, which makes the anonymized contents also completely unlinkable to the original values.
Another idea is to *generalize* data in order to protect single individuals. Several entities are grouped together and the entities' identifiers are replaced by the group identifier. This can prove problematic as soon as the grouping does not always produce equal sized parts. In the cases where a single individual or a small group can be identified with a group name, the anonymity of the users in question could be violated. As a special type of generalization we will add *truncation* of values, since it has many applications and is used often. A fixed-length prefix is normally left untouched, whereas the rest of the field is discarded. This could for example be used to preserve the subnet structure of a network while protecting the individual users by applying truncation to the IP address field.

2.1.2 Pseudonymization

Pseudonymization replaces the actual identity of an individual by a pseudonym through a bijection and is as such a reversible process by definition.
Often data is *permuted*, meaning original identities and pseudonyms are drawn from the same set. Anyone who knows the permutation that was used can easily reverse the mapping, since permutation is a bijection. A special form of permutation, which is used frequently because of its convenient properties, is called *prefix-preserving permutation.* This is often applied to IP addresses, because network topology is preserved in a way that addresses that are next or close to each other initially are still next or close to each other after the permutation, but all addresses have been remapped. More formally, two plain text IP addresses which share a n-bit prefix will still share a n-bit prefix in their anonymized form. The advantage of preserving the topology of a network is at the same time a disadvantage: if someone managed to compute the mapping from an IP to its anonymized form, he will also get information about the machines nearby.
There are a number of cryptographic methods that can be applied to record fields, like *hashing*, *encrypting* or *keyed hashing.* Applications for hashing could be hash functions which preserve some prefix information, *e.g.* topology information in the case of IP addresses. Keyed hashing is used when the values for the hashing are too short and as a consequence dictionary attacks on the hashing scheme are possible. This is important since computing *e.g.* MD5 hashes for an entire IP address space is a matter of hours.

2.2 Setup

In out scenario, there's a network segment of arbitrary size, which is monitored through some sort of packet capturing device. Captured data can be recorded as single packets, flows or be sampled. Once captured, a trace of all the traffic is exported and anonymized according to some predefined policy in order to protect sensitive user and network data from malicious users. The only assumptions are that the payload of packets is unavailable and certain header information, like IP addresses or port numbers, remain intact[2]. An additional assumption is that TCP and UDP port numbers map to their according services as stated by Coull et al. in [7]. After the trace has been anonymized, it is released to the public for further research and analysis. Early detection of worm outbreaks, network congestion or detection of distributed denial of service (DDoS) attacks are just some of the motivations for inspecting large amounts of network data.
The motivation of an attacker in this setup is straightforward. As soon as he will be

[2]meaning they are included in the log, but could be anonymized

able to deanonymize the final trace, he will have a complete log of activities for the entire network. In order to deanonymize a trace, the adversary has to somehow mark a machine with a (unique) signature he recognizes later in the anonymized trace. This marking of a physical machine over the network is called *fingerprinting*. There exist two different methods to fingerprint a host. In the first approach, the attacker only inspects anonymized traces and can deduce information about the real IP addresses from them. The other method involves the attacker being active by injecting packets that he can identify later in the anonymized trace. The ultimate goal of the malicious user is to reveal the binding between real and anonymized IP addresses of some or all hosts inside the monitored network.

2.3 Passive fingerprinting

2.3.1 Description

As mentioned above, with passive fingerprinting the malicious user does not inject packets or influence the distribution of the trace in any way. He gains information about the victims strictly by inspecting the anonymized trace and from public information sources like search engines or web statistics.
The general approach with passive fingerprinting is to try some sort of matching algorithm on the anonymized data set. Patterns that are either very similar to each other or well-known can be observed and then compared. Mostly the attacks work by means of behavioural profiling. Actions or attributes of hosts are analyzed and later used to recognize them.

2.3.2 Attacks

As an example each web site has a unique structure (”signature”) and request/response-pairs will look similar in terms of size and response time (the time the web server takes to compute a dynamic script). Koukis et al. [14] obtained a signature for each web page to be identified and created a database of signatures. Those signatures were matched with information extracted from the trace and a similarity score for potential matches was computed. As a consequence they were able to reconstruct about 8% of the requests, which shows that matching can at least partially be successful. Once several web sites are discovered that way, it is possible to profile the web browsing behaviour of users through the anonymized logs.
The goal of an adversary could also be to identify the so called ”heavy-hitters”. These are the servers (typically web servers, DNS servers), which are frequently visited and

therefore appear more often in the log. Coull et al. [7] showed that they were able to deanonymize from 66% up to 100% of the targeted SMTP servers and 28% to 50% of the significant HTTP servers using only behavioural and public information. By measuring the entropy of distinct IP addresses and taking the lowest values they could correctly filter out the heavy-hitters and compare them to known heavy-load servers in the network. Popularity-based search engines like `http://www.alexa.com` and other public information sources like DNS records or Google search helped greatly to identify the servers. They were also able to correctly deanonymize a /24 subnet by means of the "Subnet Clustering" technique.
Each network has a special distribution of services. If a subnet, lets call it a.b.c, has a server with File Transfer Protocol (FTP) port and Secure Shell (SSH) port open at a.b.c.1, a heavy-load web server at a.b.c.2, another server with FTP and SSH port open at a.b.c.3 and a machine with telnet, SSH, FTP and HTTP ports open at a.b.c.4 (and no other servers), this subnet may well be the only one with these exact characteristics (depending largely on the size of the monitored network). Assuming one wants to find a particular server which interacts with other servers, one can draw a relationship map with all the servers and the services they offer. While this takes a lot of manual matching, there's a chance of finding the server one wants to identify through neighbour-relations. In their worst-case scenario, Ribeiro et al. [21] managed to identify up to 44% of the hosts uniquely in a class B network with full prefix-preserving IP address permutation, while only about 4% could be identified when a partial prefix-preserving scheme was chosen.
Yet another idea could be to actually attack the anonymization scheme protecting the data[3]. Data truncation for example reveals a certain amount of information, since the prefix of the value remains intact. If truncation is applied to IP addresses, one has some information depending subnet structure and matching the hosts becomes easier, but the utility of data for anomaly detection is decreased [5]. Permutations as a second example are often applied in the form of prefix-preserving permutations. Fan et al. [9] studied different attacks on prefix-preserving permutations like frequency analysis, active attacks and port scanning in depth and compared them to each other. In a related work Brekne et al. [4] also suggested that existing permutation techniques are not completely free of information leakage by finding attacks on two prefix-preserving anonymization schemes.
Another well-known approach of passive fingerprinting is to make use of port scanning activities. Often port scans will sweep over a whole subnet, scanning one or several ports in the process. Such sequential scans can be used to reconstruct the original host portion and perhaps also the subnet-prefix, as the scan will hit the hosts in ascending order. Exploiting the fact that port scans already exist in an anonymized log can thus reveal the structure of whole /24 subnets [14]. Still this method probably requires some

[3]more precise: the IP address field of the anonymization scheme

manual matching but can greatly reduce the number of possible combinations.

2.4 Active fingerprinting

2.4.1 Description and Motivation

A serious drawback of passive fingerprinting is that although heavy-hitters are accurately identifiable, machines that "stay below the radar" cannot reliably be uncovered in the trace or there is at least no guarantee that one particular target host is identifiable. What we would like to have is that an attacker can pick targets at will and extract these IP addresses only. Since a possible victim has to somehow draw attention in the anonymized log and because the activity of a host is nothing a passive attacker can change without interfering directly, a stronger attacker model is needed. The adversary should now be able to actively inject packets at will in order to recognize his packets in the final trace without relying on the attacked machine in any way[4]. This change gives the adversary much more power, since he is now able to profile the behaviour of any host he wants.
The attacker model for active fingerprinting is not restrictive and the assumptions are reasonable for an attacker with moderate available bandwidth. The attacker is assumed to know about the address space of the monitored network and can direct traffic there at will [1]. It is also reasonable to assume that an adversary has one or several machines outside the network under his control and could potentially compromise hosts inside. It is imperative to mention that an attacker does not need to have excessive computing power or a large data transfer rate.
The intentions of the adversary in this model are to attack a set of physical machines by revealing their addresses in the anonymized data. Such an active malicious user can potentially do anything to the network, but since he wants to profile selected hosts, it is in his own interest not to raise too much suspicion (*e.g.* by flooding the network with lots of traffic). On the other hand he must ensure that he'll be able to recognize his traffic later on. The attacked host *will* notice the attack with high probability if appropriate intrusion detection mechanisms are in place, because the whole idea is to generate traffic that is atypical for this host.

2.4.2 Attacks

An active probing attack could look like that: An attacker from outside sends specially crafted packets to a potential victim, which may or may not reply, inside the monitored network. Those packets have attributes that the attacker will recognize later in

[4]The machine could even be offline during the attack

the anonymized trace, *e.g.* an unusual port number, seldom used Transmission Control Protocol (TCP) flags or IP ToS, such that each attack will get its own unique signature as explained by Foukarakis in [1]. As an example, a victim could receive ten packets with a size of `77 Bytes`, starting at port number `33'777` in ascending order and each packet having all 8 TCP control bits set to `ON`. The combination of those values for one particular victim is the fingerprint, and should be unique for every machine in the same timeframe. The attacker restricts himself to those attributes that are not going to be anonymized as suggested in [1]. They also suggested that in the unlikely event where all attributes are anonymized, temporal patterns should be used. As a trivial way to initiate recognizable flows they proposed a SYN scan on the target network, making it detectable in the trace.
Brekne et al. propose in [4] to forge packet headers or traffic patterns such that they are recognizable in their anonymized form. This way an attacker should be able to find an exact match between plaintext and anonymized data.
In their work about prefix-preserving IP address anonymization Fan et al. [9] state the possibility to encode the victim IP address into fields like port numbers or packet length. They also remark that active attacks could span long periods of time and could thus be made arbitrarily hard to detect.
In order to actively fingerprint a machine, one has to take advantage of a covert channel, such as time or packet size [2]. What the malicious user effectively chooses as fingerprint will largely depend on the format that the captured data is exported to and the anonymization technique used to obscure the fields, since only fields that are left intact can be used [1]. Still there are certain attributes of a connection which are better suited than others to build a fingerprint. The attacker in this model is assumed to know about the anonymization scheme that is used and can therefore prepare his attack accordingly.

2.5 Countermeasures

How to defend against fingerprinting attacks is still an open debate, because with anonymization there is always some kind of trade-off involved. The two major concerns are privacy of the users versus information value. The key is to find a balance between usefulness of the information to the researcher ("utility of an anonymization algorithm" as introduced in [23]) and privacy of the users inside the monitored network as well as security of the network itself ("strength of an anonymization algorithm" as introduced in [23]). To anonymize only the IP addresses for example is perhaps beneficial to network researchers, since all kind of studies can be made on the data, but may not be appropriate from a network administrators point of view. As a result no perfect solution exists, but only compromises between the two parties involved.
Also not all countermeasures are equally applicable against passive and active attacks.

A countermeasure which will prevent passive attacks may not do much good when faced with an active adversary and vice versa. We can divide the countermeasures into two broad categories: one that is aiming at improving and reinforcing the anonymization scheme and one that focuses on non-technical measures. The countermeasures are ordered by their "utility", starting with those that give the analysts the most information.

Reinforcement of the Anonymization Lets first have a look at a weak method which tries to tamper with the available data as few as possible. Adding a small random amount to all fields can help countering attacks which work with the similarity of web server responses, which is called "Random value shifting" by Foukarakis et al. in [1]. The interval of the random number could be dependent on the original number n to jitter, making it for example $[n - \frac{n}{100}, n + \frac{n}{100}]$ or it could be the smallest value s observed[5], *e.g.* $[n - \frac{s}{2}, n + \frac{s}{2}]$. The success of the approach depends largely on the type of field. Whereas this looks promising with timestamps and the like, this just doesn't work at all with port numbers or IP addresses.
A bit stronger an approach would be to remove ARP and port scan traffic entirely from the log [7]. While this would in fact obfuscate some statistics, it is very effective against the attacks mentioned, since routers and scanned subnets can no longer easily be identified. One could argue that port scans are rarely of importance and their removal would not considerably warp any statistics, but on the other hand scans might be used to prepare an attack and could contain valuable information regarding the attackers.
A very interesting way of countering an active attack could be to hide details about the capturing of network packets itself. While security through obscurity is generally not considered a good methodology, keeping date and time as well as the exact place of the capturing hidden from attackers could be a good idea [9, 14]. The vision is that a large amount of organizations release network data sporadically without saying who took the trace and where it was taken. Timestamps would need to be excluded from the trace and at least IP addresses should be anonymized. Behavioural profiling would still work in this case, but active attacks would become very hard to execute, whereas this would have no serious impact on the research value of the data.
Coull et al. [7] discussed the idea to remap the TCP and UDP port numbers to counter the mapping of neighbouring relations. This would preserve certain metrics, like certain ports are used heavily while others aren't at all, but it would be hard to guess which ports they are. But then again, this approach might destroy important information since there's often a corresponding service behind a port and without application data (which could get filtered out by the capturing device) and ports, there is no way of determining which service it was.
Another approach is to obscure bidirectional connections [1]. The idea is that whereas

[5]would require that the whole trace is traversed at least twice

a connection from A to B (denoted $A \rightarrow B$) is mapped to $A' \rightarrow B'$ during the anonymization, the connection in the other direction $B \rightarrow A$ is mapped to $C \rightarrow D$. This mapping makes any attempt to match bidirectional connections in the anonymized data to real connections very tricky, because initiating and receiving host both were randomly remapped. Of course this is also information which would greatly interest researchers, but all they see now are unidirectional connections without correlation, which renders most analyses on the data useless.
In order to protect the data, one could also apply a combination of methods and use a dedicated method for each field, like prefix-preserving pseudonymization for IP addresses and remapping for TCP or UDP ports. The loss of research value would have to be analyzed for each field separately and may vary with the method used. Still, as Coull et al. [7] point out, an attacker might be able to bypass the protection by using an easily detectable temporal pattern.
As a very strong countermeasure (in a sense that much of the contained information is no longer accessible) I will add top-lists. Instead of publishing whole traces, only top-list traces of selected fields are released, meaning only the top 10 of the most frequently used ports for example. As most of the value of the information is clearly destroyed that way, this approach isn't meant to encourage extensive research, but only to give an overall impression about the state of the network. This idea was discussed by Bethencourt at al. in [2].

Non-technical measures The first idea here would be to impose legal requirements on the published data [7]. This might not scare off many attackers, because law is tricky to apply across countries and to proof that someone has broken the agreement could mean a great deal of work.
Another approach might be to give only remote access to the data. This way the owner of the data could control and restrict applications on his traces. The third and most extreme non-technical solution is to restrict access to a group of trusted individuals, which performs requested operations on behalf of the clients [7]. These solutions require the publisher to have a good infrastructure and possibly to dedicate a considerable amount of resources.
Most of the discussed non-technical solutions will in fact not work against active fingerprinting. To restrict on remote analysis would not counter the attack, since requests from an attacker which has performed an active fingerprinting attack can look legitimate[6], depending on the attack strategy. This is also the reason why it is very hard if not impossible to filter out active attacks from the anonymized trace without damaging ordinary traffic as well.

[6]as long as the packets are valid and reasonably sized there is no reason to suspect an active attack

2.6 Conclusions

There certainly is a lot of research going on in the field of anonymization and pseudonymization. Much of the research literature is about passive fingerprinting in a sense that it concentrates on breaking an anonymization without direct interference. Many clever ways have been suggested to use information that is already available in a trace or which can be acquired through legitimate channels in order to break an anonymization scheme. All these approaches have in common that they require some kind of *context information*, which can link an IP address to its obscured counterpart.
Context can be public information which is freely available on the web, like DNS replies or results from search engines. Also context could mean that information is extracted from the obscured trace itself, like port scanning patterns or neighbour relationships. This additional information is essential since it allows to draw a connection between anonymized and original data.
Basically each server can influence his "presence" on the Internet at will and could for example decide not to interact with the environment, in which case the host would be very hard to find by a passive adversary. In the end, what really makes a server linkable and identifiable is the traffic he initiates or receives. So an active attacker does not need to anticipate the behaviour of the victim, since he can direct traffic there and as a consequence bias the victims traffic pattern. Active fingerprinting gives the malicious user the ability to *manufacture the context* himself instead of blindly rely on it to be there in a trace. The difference between passive and active fingerprinting is that while the former relies on existing traffic patterns, the latter can generate context information at will for any host in the network.
Not only can a malicious user in the active setting decide on its own *which* victims he wants to profile but also *how* they appear in the anonymized log. This leaves the active user with two distinct advantages over the passive guy: he can say exactly which hosts he wants to target and he is able to control how much attention these victims should attract in the log (to those who know the fingerprint).
Many research papers only hint in the direction of active attacks. The motivation for developing a framework for active fingerprinting attacks is to investigate and demonstrate the capabilities of an active attacker directly on live data. Also it should show how effective existing countermeasures are against active fingerprinting and to what degree they can still be applied. The expectations are that most countermeasures against passive attacks will in fact not work against active injections and that a new set of countermeasures will have to be devised.

3 Framework Architecture and Implementation

PIFF (**P**acket **I**njection and **F**ingerprinting **F**ramework) is written nearly entirely in objective ANSI C++ and is tested under Linux 2.6.27 and Linux 2.6.20, but should work fine on any other Linux system with the proper libraries installed (see Section 3.2 for more information on required libraries). Each class has a source and header file with the extensions `.cpp` and `.hpp` that contains exactly one class definition and declaration respectively. The framework produces two executables: `InjectorStart` is the binary that handles the generation and injection, and `AnalyzerStart` is the program handling the final matching. There are also additional shell scripts that handle minor work and are to be understood as small helper programs. We also provided a small shell script to ease the process of large scale injections.

3.1 Framework Design Overview

The framework is divided into four packages, while some depend on each other and others are free of internal dependencies. See Figure 1 for a visual illustration of PIFF. There are three main components which work independent of each other. In order to generate and inject packages we use the package called **Generator/Injector**. This component is actually a merge of two separate components, the Generator and the Injector, which were bound together for simplicity because their invocation is sequential i.e. what the Generator creates is fed into the Injector and injected into the network from there. The second component is the **Converter** package which is a bridge between the Generator/Injector and the last component, called the **Analyzer**. A typical run of an injection is Generator/Injector → Converter → Analyzer, while those three components normally don't run on the same machine.

The packages are logically separated and perform separate tasks. Whereas the Generator/Injector solely works on packet level and does not understand flows, the Analyzer package processes NetFlow data exclusively. To bridge the gap between the two components, the Converter creates flows out of packets, thus simulating the behaviour of a capturing device exporting NetFlow data.

An attack is specified as input file to the Generator/Injector component and is called *generator file*. Once the Generator has read and processed the file, it produces a list of packets, which in turn are passed to the Injector, where they are injected into a raw socket and into the network. This component writes a detailed protocol of the injection, dumping the packets into one large XML (Extensible Markup Language) file, storing meta data about each packet (*e.g.* timestamps) and packet data itself. Being a list of stored packets, this file called *injector file*, is read and transformed by the Converter.

The purpose of the Converter is to emulate router behaviour by converting packets into

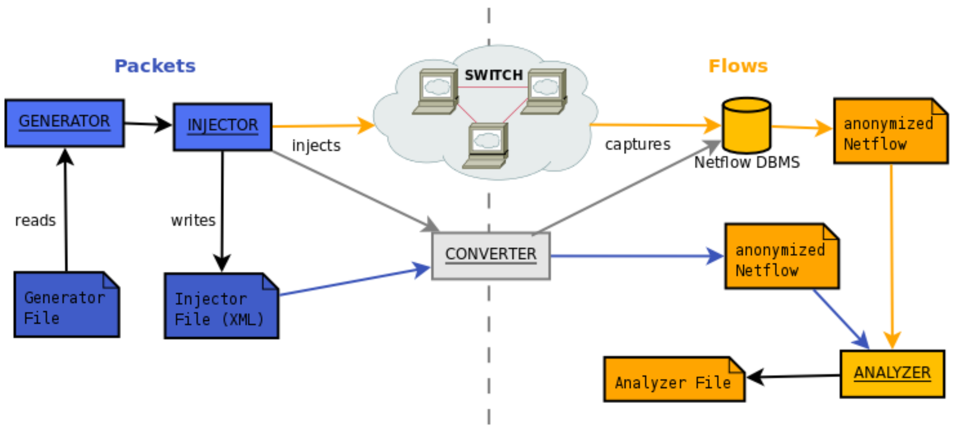

Figure 1: Framework Design

flows. We included active and passive flow timers, which are set to standard values as in [6]. The timers are the only values that need to be adjusted in case the framework was used on a network other than SWITCH.
The role of this program is crucial, since cleverly injected packets are worthless when they're incorrectly transformed into flows and the Analyzer is unable to recognize them later on. As clearly visible in Figure 1, all components on the right side work on flow level. On one hand, the border routers of the SWITCH network are creating NetFlow records out of real traffic, on the other hand the Converter simulates this behaviour with the injected packets from the injector file. The flows from the border routers are exported and stored in a NetFlow database, where they can be retrieved and anonymized according to some predefined policy. Flows leaving the Converter are sent to the Analyzer, where they may also be anonymized to some degree to match the anonymization of the captured real flows. The Analyzer is then provided with two sets of flow records: anonymized flows that came over the border routers into the local NetFlow database, and partly anonymized flows from the Converter, initially coming from the Generator/Injector component. There are processes that are **common to both paths** and the ability to **simulate injections** without going over the network is also shown in figure 1.
The final component in the chain is the Analyzer. Its job is to compare the two received flow sets with each other and recognize the **injected flows** within the **captured flows**. This is done with a similarity score, that is calculated over the attributes of a flow. The results are finally written to a file, called the *analyzer file*. This file lists the injected flows one by one and assigns the best match from the captured flows to each of them. As a result, we know the correlation between the two flow sets and how the flows we have injected look like in the anonymized version. Reading out the destination IP of the

flows, we can undo the anonymization of those hosts we have fingerprinted and profile their activities within the anonymized flow records. At the end of each analyzer file, there is a short summary about the comparison, where the destination IP that occurred most can be seen.

3.2 Libraries

In addition to the standard C++ library (STL), we used some public libraries and third party tools to improve extendibility and compatibility.
In order to parse the generator file, we chose to make use of the *Spirit parser framework* [15] in version 1.83. Spirit is object-oriented and part of Boost [3], which is a collection of freely available, peer-reviewed and portable C++ libraries. The Spirit framework has a unique approach to parsing grammars by allowing the user to approximate the syntax of EBNF (Extended Backus Normal Form) with ordinary C++ syntax. As a result, grammars can be written exclusively in C++ and mix freely with other code. This framework was the logical choice to use because it supports everything we wanted (and much more) in terms of expressiveness and simplicity. With very few lines of code we were able to create a parser which recognizes an arbitrary arithmetic expression and assigns semantic actions to the single elements. The Spirit Framework is also written very modularly such that extending the EBNF for the generator file is very easy and straightforward, in case someone wanted to improve or extend the syntax. For more details about the structure of the generator file please refer to chapter 3.4.2.
For reading and writing the XML file (injector file) we have used the *Xerces-C++ XML Parser 2.8.0* [10] toolchain, which is a validating XML parser providing a reliable and stable API (Application Programming Interface) for both W3C [30] standards SAX [29] and DOM [28]. Since PIFF should provide an interface for single packet injections and not for large scale packet generation, we decided that DOM suffices, although it is more space and execution time consuming, but has an easier interface and requires less code to be written. We chose the Xerces parser because it is a mature project, available for Perl, C++ and Java, and is freely available under the Apache License in version 2.0. Furthermore the design of the parser is very well structured and the whole project is written with solid object-oriented concepts. The Xerces parser is used in the packages Injector and Converter to read and write the injector file. For a detailed description of the injector file syntax please consult chapter 3.4.2.
To process flows we used the TIK internal library called *ProcessingNG* to read, write and modify NetFlow data. The library has a modular design and provides a hook with a base module. Custom modules can be written as derived classes from the base module, implementing a public method to process flows and send them to the next module of the chain. Unfortunately the library contains few documentation, but is well structured and

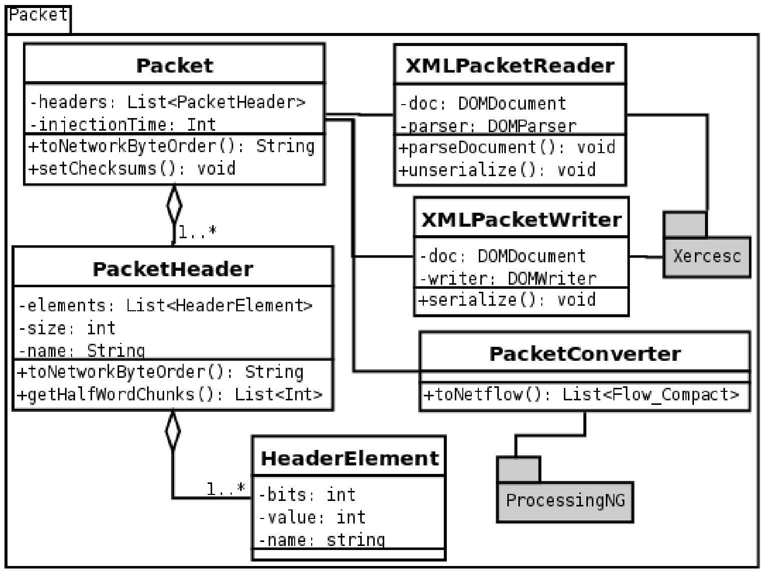

Figure 2: Design of the Packet Package

intuitive to use. All flow input and output as well as anonymization is performed with this library.

3.3 Patterns and Best Practices

Before we started the implementation, we made UML (Unified Modelling Language) diagrams of the packages and modules. The UML diagrams were adapted continually and changed slightly to reflect the current state of the code. The final version of those diagrams can be viewed in Figures 2 to 5. Attribute names and methods have been simplified, but the data flow and relationships between classes are modelled accurately.

The code of the framework is nearly entirely written in C++ with object-oriented concepts. We tried to keep classes short and clear, and split large procedures to make them manageable. The source code is documented in a style similar to javadoc and follows the sun recommendations for writing code comments [25]. This way the code can be exported into HTML or PDF (Portable Document Format) such that the interfaces for all classes are displayed nicely using tools like Doxygen [27] or DoxyS [11].
We particularly wanted to make the code look nice and easily understandable. In order to avoid bugs, we employed some C++ best practices, such as the *rule of three*. The rule demands that if one of

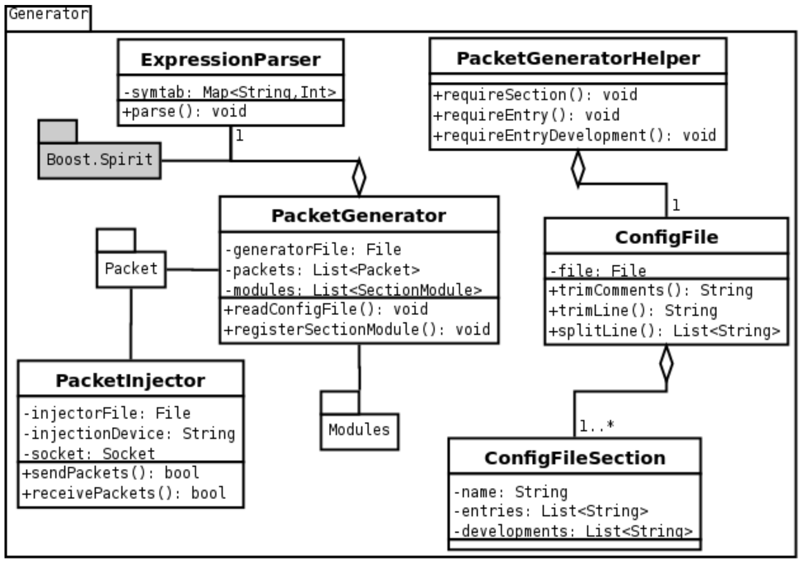

Figure 3: Design of the Generator/Initiator Package

(i) Copy assignment operator

(ii) Destructor

(iii) Copy constructor

is explicitly written in a class, the others probably should be written as well. This avoids that an object is somehow incorrectly copied while containing non-trivial members as references or lists. We enforced this rule on all our classes since object of type Packet are copied a lot inside the Generator, for example.
We did not want to give the Generator a static list of packets that it can process, instead we made the Generator have modules that understand their own section in the generator file. This eases extendibility of the Generator a lot, since all that is needed to allow the Generator to understand an additional packet header is to write a module where the abstract method of the base module is implemented.
Since the design of the whole framework is not overly complex, we did not want to blow up the code with unnecessary patterns, as the code is very manageable like this. With the Injector however we felt that a singleton pattern is needed, since the Injector is accessing the network device directly on a very low level[7] through raw sockets. The class has a static private member `INSTANCE_` that is set once on calling `getInstance()`

[7]The Injector basically provides everything for a packet, down to the link layer

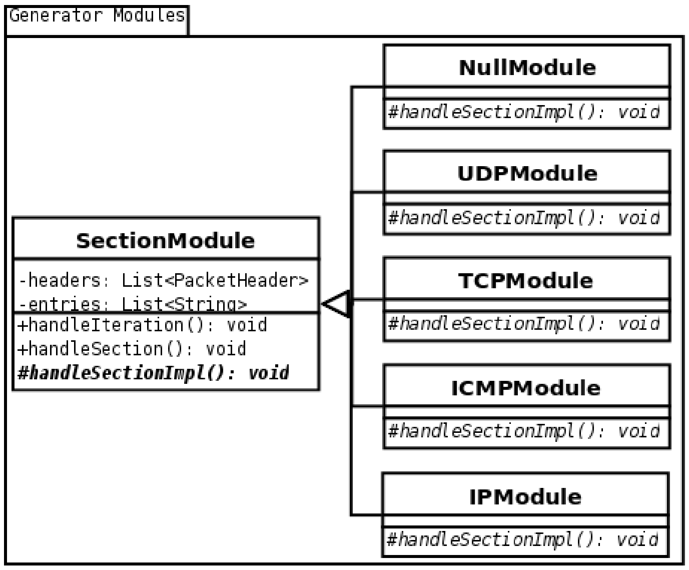
Generator Modules
SectionModule
-headers: List<PacketHeader>
-entries: List<String>
+handleIteration(): void
+handleSection(): void
#handleSectionImpl(): void
NullModule
#handleSectionImpl(): void
UDPModule
#handleSectionImpl(): void
TCPModule
#handleSectionImpl(): void
ICMPModule
#handleSectionImpl(): void
IPModule
#handleSectionImpl(): void

Figure 4: Design of the Generator Modules

the first time, and is just returned on subsequent calls. The class itself does not have a (public) constructor and an instance can only be obtained through `getInstance()`. The instance is destroyed with a call to `destroyInstance()` which frees the allocated memory again.

3.4 Packages

As mentioned the main packages are the **Generator**, the **Injector**, the **Converter** and the **Analyzer**. There is also a package which does not represent a part of the framework, but rather is an internal library or shared package which is used by most of the other packages. The library is called **Packet**, since it is all about network packets.
We present the most important classes of PIFF now, explaining how they go together and what their role is in the framework. Methods will only be mentioned, but not their full signature given.

3.4.1 Packet Design

There are 8 classes in this package, 3 of them deal with XML processing. All classes with their respective files are summarized in Table 1 and the design details can be seen

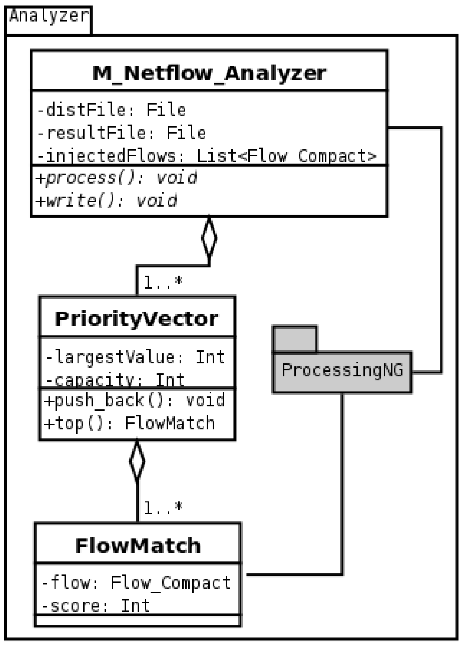

Figure 5: Design of the Analyzer Package

in Figure 2. The whole Packet library has no executable class, but only provides the `Packet` class as the most important storage class. Most components using this library will only include this one class and work with it.
The most basic storage unit is the class `HeaderElement`. A `HeaderElement` consists of only three values: a name, a value and a size for that value. What is generally stored in such an object is for example one single value of a arbitrary header. This class is derived from `Element` to allow other types than integers as values, which could happen when taking a completely different set of protocols other than the standard IP/TCP/UDP/ICMP that are supported by default.
Taking several of these `HeaderElement`s and aligning them sequentially, we can model an arbitrary header. The programming representation of a header is the class `PacketHeader`. This class has an ordered list of `HeaderElement`s and appropriate query and manipulation methods. The most interesting methods from this class are `toNetworkByteOrder()` and `getHalfWordChunks()`, which help in converting an object of this class into a network packet. While the former converts the entire header into a representation suitable for piping into a raw socket[8], the latter splits the header elements in 16 bit chunks, which is needed for checksum calculations.

[8]An array of type char, where all integers are in network byte order

Class	**Description**
`Packet`	Represents a network packet
`PacketHeader`	Represents a header in a packet
`HeaderElement`	Represents an element in a header
`Element`	Represents an element with a value and a length
`XMLPacketWriter`	Writes the injector file, serializes `Packet` objects
`XMLPacketReader`	Reads the injector file, unserializes `Packet` objects
`XMLPacketConstants`	Constant needed to parse the injector file
`PacketConverter`	Converts from `Packets` to NetFlow

Table 1: Package Packet

The programming representation of a whole network packet is the previously mentioned

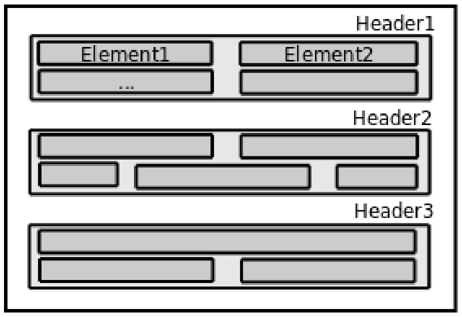

Figure 6: Schema of a Packet

class `Packet`. Very much like a `PacketHeader` has an ordered list of `HeaderElements`, a `Packet` contains a list of `PacketHeaders`. Please see Figure 6 for a visual illustration of the packet concept. As an example, a packet might contain an IP header, a TCP header and some application layer protocol. One would build up these headers by means of the `HeaderElement` class, and then add the headers in order to a `Packet` object with the available setter methods.
A `Packet` also provides the method `toNetworkByteOrder()`. Another special function this class has is `setChecksums()`, which calculates the checksum value of several headers, namely TCP, UDP and IP. We're not extremely happy with the procedure being there, because a `Packet` should be independent of the `PacketHeaders` it contains. But due to the way checksums are calculated in the TCP/IP model, we were forced to include this method directly into the `Packet` class. Checksums are calculated over multiple headers and operate on the network byte order representation, so there are dependencies between

single `PacketHeaders` by design.
The class `XMLPacketWriter` is used in the Injector to write the injector file. Next to several setter and getter methods, the class provides two public functions. The first one is called `buildDocument()` and is used to build up the DOM document. Using the internal list of `Packet` objects, this procedure populates the internal `DOMDocument`, which is a representation of an XML document. The `Packet` objects are processed one by one by appending child elements to the document root element. Once the build up is complete, the DOM document can be serialized into a file with a call to `serialize()`.
The Converter uses an `XMLPacketReader` to read the document serialized by the Injector. Since the Generator/Injector and finally the Analyzer component run at different times and probably on different hosts, the use of persistent storage, in this case an XML document, is needed. The reader just reverses the process of the writer, thus reading from an XML document, building up the internal DOM tree and then create `Packet` objects out of it. The invocation of methods is also very similar: after having called `parseDocument()` to read a file and represent the contents as a tree, a subsequent call to `unserialize()` creates a list of `Packet` objects, that can then be retrieved via the query methods of the class.
The class `PacketConverter` is in fact the only class in the Converter component, and even a relatively small one. This class could have been put in a separate package, but since it will likely stay the only class providing the conversion feature, it has been put into this package. The only method it currently supports is the static `toNetflow()`, which converts from the packet library class to a flow class made available by the ProcessingNG library. This class could easily be extended with further static methods that convert between packets and anything else that comes to mind. In case the final analysis is about to be made on packet level, the Converter would even fall away completely.

3.4.2 Generator/Injector Design

This package contains 5 classes, including one executable. All classes of the package are summarized in Table 2. The main task of the package is parsing and interpreting the generator file, which is the starting point of an attack. The file contains a description of a sequence of attacks on one or multiple hosts with all kinds of specially crafted packets. After writing an attack script, the user has to register modules with the Generator, that parses and understands what has been written to the generator file. There are only few things in the generator file that are fixed and have to be there, much of it is modular and could be extended to support protocols other than TCP/IP, UDP/IP or ICMP.
Processing begins with the configuration file for an attack, called the generator file, whose exact structure is explained below. The file has a one-to-one mapping to a class, the `ConfigFile`. This class represents the generator file as a whole, since structure and

Class	**Description**
`ConfigFile`	Represents a generator file
`ConfigFileSection`	Represents a section in a config file
`ExpressionParser`	Parses evolutions in the config file
`PacketGenerator`	Parses the config file altogether, defines structure of file
`PacketGeneratorHelper`	Helper functions to parse the config file
`PacketInjector`	Injects packets, writes injector file
`InjectorStart`	Starting class that launches the injections

Table 2: Package Generator/Injector

semantics of the file are somewhat complex. `ConfigFile` is a list of `ConfigFileSection` objects and declares several constants needed for parsing. It also offers some typical file parsing functions, like methods to strip comments, trim values or tokenize a string. The generator file is organized like a normal configuration file with sections and key-value pairs belonging to those sections. The class handling the actual parsing is the `ExpressionParser`, whereas the two classes mentioned before store the contents of the generator file.

While the `ExpressionParser` deals with the syntactical correctness of the expressions in the key-value pairs, both the `PacketGenerator` and `PacketGeneratorHelper` classes define the overall structure of the file. Unlike the contents of a section, the series of sections is well defined and induced by those two classes. We will subsequently discuss the structure of the generator file and state which parts are fixed and which of them are modular in design.

Generator File To describe an attack, at least 5 sections have to be specified. Those sections are mandatory and are present in every generator file. The sections include *"sequence"*, *"packet"* followed by three times a *"header"* section. The "sequence" section describes attributes common to all injected packets, normally this is only the amount of packets assigned to this sequence. In the section "packet" we've put metadata about each of the packets, like their size or their injection time.

The three header sections represent the application-, transport- and network layer headers common to most packet protocols. We realize that it is not possible to simulate arbitrary packets due to this restriction, but to allow an arbitrary set of headers would blow up the design of the file massively and make parsing cumbersome. Each header section has mandatory entries for type and type name, and additionally the optional and mandatory entries from the respective module. Each module is responsible for one

Program 1 Example Configuration File

```
[section]
key = initial value / evolution
key = initial value

[section]
key = initial value
key = initial value / evolution
key = initial value / evolution
```

Keyword	**Meaning**
$rand(x, y)$	Random value between x and y inclusive.
$rands(x_1, x_2, ..., x_n)$	Random value from the set $(x_1, x_2, ..., x_n)$.
@	Value from the last iteration.

Table 3: Generator File Keywords

header section and understands the key-value pairs it specifies. Modules are explained in detail in the following paragraph.

An example structure is given in Program 1. Each `ConfigFileSection` represents a section in the configuration file, with a name and a list of key-value pairs. To enhance to expressiveness of the generator file, the notion of *evolutions* was added such that there were no longer key-value pairs, but key-value-evolution triples (separated with "=" and "/" respectively). Evolutions describe the shift of a value over iterations. As an example the entry

```
TCP_Destination_Port=80 / @ + 10
```

describes a series of values like

```
80 -> 90 -> 100 -> 110 -> 120 -> ...
```

for the destination port of TCP. This syntax is quite powerful and parsed with the Spirit library of Boost, as noted in section 3.2.

A generator file can list an arbitrary number of those 5-tuples mentioned above, which are processed in order of appearance. How a working generator file describing TCP packets to random ports using the TCP- and the IPv4 modules might look like, is shown in Program Listing 2.

As stated already, each key-value pair can additionally specify an evolution, which is

Program 2 Example of a Working Generator File

```
[sequence]

Packets=10

[packet]
Size=128 / @ + rands(0,8)
Time=0 / @ + rand(150,450)

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=rand(1,65535) / rand(1,65535)
TCP_Destination_Port=rand(1,65535) / rand(1,65535)
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP=155.155.155.155 / @
IPv4_Source_IP=144.144.144.144 / @
IPv4_Protocol=6 / @
```

Class	Description
`SectionModule`	Module that defines a general section in the config file
`IPModule`	Module that understands IPv4 syntax
`ICMPModule`	Module that understands ICMP syntax
`TCPModule`	Module that understands TCP syntax
`UDPModule`	Module that understands UDP syntax
`NullModule`	Module that is invoked when a section is left empty

Table 4: Package Generator/Injector (Modules)

applied to the value in each iteration for as long as there are more packets to be generated. Keywords that can be used are summarized in Table 3. It is up to the modules to specify when a key-value pair needs an evolution and when it doesn't.

Generator Modules There are five modules that are built-in into PIFF, because they are used for our injection tests and they represent commonly used protocols. There are four common modules which have a direct representation in the OSI model, and one that is there for programming convenience. This module is called the `NullModule` and is invoked when the user decides *not* to model a particular header. If there's no application header needed, the user won't write anything in this section and the `NullModule` will automatically handle this case. All the built-in modules are listed in Table 4 and their names should be self-explanatory.
The module base class is called `SectionModule` and defines the entry point for additional modules. A module typically inherits from the base module and is then forced by the base class to implement the abstract method `handleSectionImpl()`. This method receives one section from the configuration file which (hopefully) contains the key-value pairs that this particular modules does understand. The module then manipulates a `PacketHeader` based on the information gathered from the file, which is finally added to the list of other headers. So each module that is called adds one header to the packet, resulting in three headers (one for each of the layers in the OSI model). The base module provides several methods to check the generator file section for mandatory entries and to retrieve the values in the section comfortably.

It is the Generator class `PacketGenerator` that allows the user to register and unregister modules. A module is generally responsible for one section inside the configuration file. Although there are built in modules, the concept allows for arbitrary modules to be written and included into the framework, depending on what packets the user wants to

craft.
Once the generator file is parsed, found to be valid and the contents are stored in the appropriate objects, the `PacketGenerator` creates a list of `Packet` objects by means of the Packet library introduced above. At this point the part that actually "generates" anything is done, and the "injection" starts. An instance of the `PacketInjector` is created and initialized with needed information such as the injecting device and the list of generated packets. After having sorted all the packets by ascending injection time, the Injector instantiates a raw socket operating on the link layer level. First thing the Injector does is to find out the MAC (Media Access Control) addresses of both the injecting device and the router. Upon completion, it crafts a valid Ethernet frame with the available information and prepends it to all packets that are to be injected. All that is left now is to send the packets over the wire according to the timing information of the packets. The Injector writes a detailed log file about the injection in XML.

Injector File The injector file is designed to be a machine readable[9] log file of the injection. The file is written in XML and the tags are chosen to be very similar to the C++ classes. The whole file is a list of all the packets that were injected. Each packet is given an injection time and a list of headers, much like in the implementation. A header consists of a unique (for this file) identifier and a collection of header elements, which in turn have a name, a value and a length in bits. To see a short example of such a file, please have a look at Code Listing 3.
The `PacketInjector` appends to this file after each injected packet, thus making the injection more error resilient. In case the injection aborts, the file is still correctly terminated and contains the packets that were successfully injected. The `PacketConverter` class reads this file and converts the contents to a list of `Packet` objects, thus reverting the process of the Injector. The conversions are in fact just a serialization and deserialization of the `Packet` class to an XML file and vice versa.

Finally, `InjectorStart` is the "running" class that is called upon starting the framework. Its job is to register all the modules with the Generator, launch the Generator with a configuration file and then pass on the retrieved packets to the Injector. In order to provide feedback for long injections, the launcher continually updates the screen with information about injection times and whether errors occurred along the way. This program has to be launched as root, since raw sockets require root privileges.

[9]The file should not have to be inspected or modified in any way, it is simply fed into the converter afterwards

Program 3 Example Injector File

```
<?xml version="1.0" ?>
<packets>
  <packet time="1234567890">

    <header>
      <headerName>IPv4</headerName>
      <headerElement>
        <name>IPv4_Version</name>
        <value>4</value>
        <bits>4</bits>
      </headerElement>
      <headerElement>
        <name>IPv4_Header_Length</name>
        <value>5</value>
        <bits>4</bits>
      </headerElement>
    </header>

    <header>
      <headerName>TCP</headerName>
      <headerElement>
        <name>TCP_Source_Port</name>
        <value>b084</value>
        <bits>16</bits>
      </headerElement>
      <headerElement>
        <name>TCP_Destination_Port</name>
        <value>50</value>
        <bits>16</bits>
      </headerElement>
    </header>

  </packet>
</packets>
```

Class	**Description**
`FlowMatch`	Represents a flow and an associated score
`M_Netflow_Analyzer`	Module that compares the two flow sets and writes the analyzer file
`PriorityVector`	List of N best `FlowMatch`es
`AnalyzerStart`	Starting class that launches the analysis

Table 5: Package Analyzer

3.4.3 Analyzer Design

This package contains 4 classes, including one executable. All classes of the package are summarized in Table 5. The Analyzer is using the ProcessingNG library extensively. All modules that belong to that library are prefixed with a capital letter and an underscore (`M_`), and class names are separated by underscores, unlike other classes in PIFF that are written in camel case.

The main part of the Analyzer is a class named `M_Netflow_Analyzer`. This module receives two sets of flows: one that was captured over the network and one that the converter read from the analyzer file. Also this module has one input and one output file, which are called *distance file* and *analyzer file* respectively. As mentioned above, the idea is to simulate the behaviour of the capturing device with the `PacketConverter`, but similarly we also apply the same anonymization techniques to the injected flows as those that were applied to the captured flows. What we initially have is two nearly identical flow sets (or at least one is a subset of the other), then one of them is anonymized according to some policy. The logical way to deal with this is to apply the same policy also to the other set. The only problem is, that an attacker would not know the secret values of a pseudo random generator, if some sort of random modification was utilized in order to make fields unrecognizable. While deterministic functions like ”map all ports below 1024 to 0 and all above to 1” are very easy to simulate this way, problems arise when random elements are involved.

The real question here is what can an anonymizer do with a random generator? One must keep in mind that if an anonymizer introduces its own random numbers, original values are distorted inevitably by this, so we limited our analysis to a few basic algorithms that a random generator could produce. What is frequently used for example is to give a timestamp or a packet size a slightly distorted value, like randomly offset the value by 5% of his range. Another way random numbers might be introduced is to permute a value or to generate a completely random number for a field.

There’s really nothing we can do about randomization of a value, the field is useless in the anonymization afterwards. Similarly we don't want to attack a permutation directly, since our approach should work in any case. What we can counter is the random offset, and that's what the distance file is all about.

Distance File This file has a similar syntax to the generator file introduced above, but it is way simpler in design. The file is a list of the attributes of a flow (destination port, source port, packet count, ...). We assigned a number to each attribute, which represents the maximal expected offset for this particular attribute. So the line

```
Packet_Count=10
```

means that the amount of packets was distorted and can differ by 10 from the original value in each direction. Meaning if the original value was 17, numbers ranging from 7 to 27 could be observed.

Setting a value to 0 means exactly what it should, this field does not tolerate any

Program 4 Example Distance File

```
[Distance]

Destination_Port=0
Source_Port=0
#Destination_IP=0
#Source_IP=0
Bytes=10
Packets=0
Timestamp_Start=0
Timestamp_End=0
#TOS=0
#Protocol=0
```

deviations from the original value. If an attribute is not mentioned in this file, the attribute should not be considered at all in the matching algorithm. This covers the cases we mentioned above, where the anonymization includes randomization which we cannot simulate properly and where we just have to ignore this field. As a guideline, if a field was randomized, erased or permuted, you can safely omit this field in the distance file and if no offset was used in the anonymization, this field is set to 0 in the distance file[10]. An example of a distance file is given in Code Listing 4. In the example, the IP

[10]but needs to be present nevertheless

addresses, TOS flags and the protocol are not considered at all, whereas the byte count can tolerate an offset of 10. This is a typical setting when recognizing a flow that had only the IP addresses permuted and the byte count slightly modified , but everything else was left unchanged.

The distance file is needed when calculating matching scores between two flows. For each captured flow all injected flows are iterated over and a similarity score is calculated for each pair.

Similarity Score Calculation The main task of the Analyzer is to compare sets of flows. All possible combinations of injected and captured flows are compared and for each of the injected flows a minimal score and the related flow are stored.
The score itself is calculated over all the flow attributes and by considering the distances from the distance file as well. A value of a field (in a captured flow) $V \in \mathbb{N}$ is assumed to be normally distributed with mean μ and variance σ^2. In our case μ is the value of he field in the injected flow and σ is the range that this field can assume, so we can say $V \sim \mathcal{N}(\mu, \sigma^2)$. To receive a distribution which has mean 0 and variance 1, we warp the curve to a *standard normal distribution* by means of the z transformation. Once all values are normalized in the range $[0, 1]$, we can compare them safely. The overall score computations looks like

$$S_{i,j} = \sum_{k=0}^{k<K} w_k * s_k$$

where $S_{i,j}$ is the computed score for flows i and j, K is the amount of attributes in a flow, w_k is the weight for a particular flow attribute and

$$s_k = \frac{x_i - x_j}{r_X}$$

is the z transformed value where x is the value of a particular flow attribute and r_x is the range of this attribute. When the distance file is taken into consideration as well, the

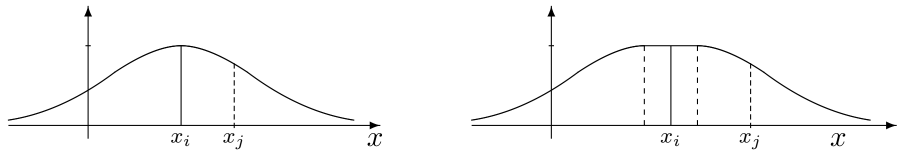

Figure 7: Gauss Curve with (right) and without Distance (left).

score computation is slightly more complicated as the curve for a single attribute is no longer a bell-shaped gauss curve, but a cut off version. A value V is no longer normally distributed, but also features a uniformly distributed component. For a visual clarification of the shape see Figure 7 with the two versions of gauss curves. As a consequence, the calculation for s_k is more elaborate as we cannot just take the difference between the two values x_i and x_j anymore. Instead the calculation is depending on the distance and is written in pseudo code in Program Listing 5. The `M_Netflow_Analyzer` class stores

Program 5 Score Computation for a Flow Attribute

```
if($x_i + distance \geq x_j$) then
  $s_k = 0$
else
  $s_k = (x_j - x_i - distance)\ /\ r_X$
```

the N best flow matches for each of the injected flows, that is those flows from the set of the captured flows that resemble the injected flows the most. In our case they store the flows that have the *minimal score*[11] with the flows from the injected packets. Those N best matches are stored in a data structure that is similar to a priority queue, but is in fact a list and allows to access any element currently stored.
To model this we wrote the class `PriorityVector`, which itself is a list of `FlowMatch` objects. `FlowMatch` is a very simple storage class that stores a flow and a score associated to it. The `PriorityVector` on the other hand contains more complex code. This class has an upper limit of elements it can store, called the capacity. It inserts elements until the capacity is reached, and then overwrite those elements with the worst score when inserting a new element. Whenever a new element is pushed into the vector, the `FlowMatch` with the highest score is overwritten and lost. Just like a priority queue, our `PriorityVector` has a `top()` method to retrieve the "best" element.

The result at the end of the comparison of the flow sets is stored in the analyzer file, which is the final product of the framework.

Analyzer File Once the N matches are stored for each injected flow, an algorithm chooses the best match for each flow by taking the order of flows and the matching score into consideration. If timing remains undistorted, the Analyzer tries to find a sequence of matches such that the ordering of matches is consistent with the injected flows. The result could be a sequence of matches where the match for the first flow has occurred before the match for the second flow, which has occurred before the match for the third

[11] scores range from 0 to 100, where 0 is the best score possible

flow and so on. If several such sequences exist, the one with the lowest scores is taken[12]. If timing is unavailable or only in modified form, the flows are matched one by one. The Analyzer however has an option to specify how large the timestamp difference should be at most for a flow to be considered in the matching.
The file lists all injected flows chronologically and displays the respective best match next to them along with the computed similarity score. At the end of the file you can see a short summary of the matching, indicating how many flows were probably guessed correctly and how many could not be matched at all. The Analyzer presents you with the destination IP that it thinks is the most likely target at the very bottom.

3.5 Error Handling

In general error handling was not seen as a major issue, since the programs are all batch programs and will not run for very long, except the Analyzer which runs for 2 hours or more on our machines for 1 hour of processing flow data. Also a failure means in many cases that the framework cannot continue at all, so error recovery was treated not as important as error detection.
The Generator/Injector part is very meticulous about errors. Both the generator file and the injector file have a strict syntax and the validating parsers simply fails if the syntax is incorrect. The internal classes in the Packet component have been thoroughly tested and should not contain severe bugs. In this component, what can be handled and thus is not a fatal error throws an STL exception and notify the user via the standard error channel.
Errors in the Injector part are almost always fatal and simply causes the program to exit with an error message. Exception messages from the included libraries are caught, handled and should thus never reach the user.
In the Analyzer, the behaviour is somewhat different. Since most of the processing of NetFlow data is handled by the ProcessingNG library, we don't deal with reading or writing errors ourselves. Large parts of the Analyzer include score computation and reading or writing files, so there's not much to do in terms of error handling on our part.
Clearly the most fragile part is the injection of packets, as this is very low level and many things can go wrong. For example the Injector determines the MAC address of the router by means of a shell script, which is not explicitly written to be compatible with all UNIX derivates, so this part is likely to fail if the the Injector is tested in an atypical environment and guaranteed to fail on anything else than a UNIX machine. Many problems about the Injector have been addressed, however there may still arise certain difficulties when trying to inject packets which we did not anticipate.

[12]the scores of the matches are summed to form an overall score

3.6 Limitations of the Framework

The Generator/Injector basically is a full-blown packet craft library which has a modular input file and a detailed logfile that suits our purpose. The logfile is built in a way that we can easily create flows out of the contents. With the built-in modules at the time, the framework can craft just about any valid TCP, UDP or ICMP packet you can imagine. Here PIFF could easily be extended by writing additional modules like an IPsec or ICMPv6 module. Also writing modules for the application layer will be straight forward, but would need some effort depending on the complexity of the protocol to implement. Of course implementing application layer protocol based on TCP is not such a good idea, since the framework is thought to be a packet injection machine. As a result, the framework is non-interactive. Nevertheless, implementing a application layer protocol can make sense, namely when the response is not important and injecting application layer information helps to identify the trace later. Additionally it could help an attacker to make his traffic look legitimate and to avoid being caught in a firewall or Intrusion Detection System (IDS).
Also the framework, although written as an extendable piece of software, currently has only limited functionality. As stated above, modules for the most common of protocols exist, but more modules would make PIFF more complete and powerful. Another limitation is the lack of additional converters and analyzers. The current converter offers only one way conversion from packets to NetFlows and nothing else. The Analyzer is purely based on NetFlow and does not currently support any other format. In order to allow the framework to analyze other formats than NetFlow, one would have to write the appropriate conversion between the class `Packet` and the available format, and write yet another Analyzer module that can read and process this format. The whole reading, writing and most of the processing functionality for the NetFlow format is wrapped inside the ProcessingNG library, and therefore we chose not to make this extendable at all, since it would have been too big an effort.
There were some simplifications and assumptions made in the packet generation process. We chose to follow the Internet model and stick to the layer architecture in every case. In fact we made it a requirement for a packet to have at most three layers. Those three layers are the network layer, the transport layer and the application layer. The link layer is handled by the Injector and cannot be interfered with. Due to this restriction, it might be impossible to implement complex protocol stacks like they are needed when dealing with IPv6 for example. Since IPv6 wraps parts of other protocols and adds additional headers before and after the original header, this would most likely screw up the simple three layer architecture of the generator file. However, the chosen simplifications work very well with most common protocols nowadays.

3.7 Related Work

The Generator/Injector component is basically a traffic generator. Traffic generators are usually used to stress test routers or servers under extreme load. Often such tools are stateful and interactive in a way that they can simulate client-server behaviour. Popular tools include for example the "Network Traffic Generator and Monitor" [17] or the "HttpTrafficGen" [16].
Our framework is more likely to be classified as a *packet craft engine* as it does not generate regular and valid traffic containing some application layer protocol. It crafts raw packets, operates mostly on the transport layer and above all is non-interactive. This makes PIFF unsuitable for penetration testing or even stress testing, where legal connections are desired.
A frequently used packet assembler and analyzer is "Hping" [22], which supports a wide range of protocols. Although initially intended as a security tool, Hping is nowadays also used to test networks. Another very good packet injection utility is "Nemesis" [18]. Unfortunately this tool is no longer maintained since late 2003, but contains an impressive list of features and runs on both UNIX and Windows systems. Last but not least, Nmap [19] is a very popular tool to perform port scans and penetration testing. The open source project is also available for many platforms, but is not on such a low level as the other tools discussed.
One of the main reasons we chose to implement a completely new packet crafting engine, is that we have special requirements for this project since the engine is only a part of a bigger project. For once we need very precise logging of the packets we have injected with exact timing information. Second we wanted an easy interface where the user is not distracted with unneeded options, but where a very simple configuration file with a powerful syntax is enough. This way the user does not have to learn any unnecessary details about the framework itself, but a fundamental knowledge of the injected packets suffices to craft complex sequences. This was particularly hard to find in the available tools, as most of them are just not on the right abstraction level. It turns out we needed a combination of things that are not yet available as a program. The strength of the Generator/Injector is also, that it uses the same internal libraries as the Converter and the Analyzer, which probably saved us some compatibility problems.
Another alternative would have been to wrap an existing project and provide an interface that suits our needs, but then the Converter and Analyzer would have gotten more complex. We finally decided to write the Generator/Injector without any existing code and used existing libraries for file handling instead.

4 Testing and Results

In order to receive a meaningful sample set, we conducted our injection tests on several machines with different amounts of traffic. Intuitively machines which produce a lot of traffic seem harder to fingerprint than machines that have few or no traffic at all. In addition, traffic can be more diverse or monotonous, depending on the amount and type of services a machine provides. IP addresses are assumed to be anonymized always. The primary goal of the test is to recover the original IP addresses of these hosts. Table 6 summarizes the three classes of hosts we used for our injections tests. We tested 10 machines of each class to make statements of some statistical relevance.

4.1 Testing Setup

In order to fingerprint machines with a large amount of traffic, we chose the most active[13] web servers in the SWITCH network, since web servers have interesting properties with respect to packet size and count. Both HTTP (Hypertext Transfer Protocol) requests and responses can vary considerably in size and the resulting flows can contain a variable amount of packets (depending on the amount of items on a web page), which makes recognizing injected patterns harder. We called hosts from this category "class A" machines and they represent those hosts in our test set that seem to be the hardest to fingerprint. Despite this fact, heavy-load servers are easily identified with passive fingerprinting already, so active injections might be superfluous.
To model hosts with medium traffic, we picked random student workstations at ETH Zürich. The diversity of traffic patterns those hosts generate is bigger than the diversity of web server traffic. A desktop computer is expected to use HTTP, SMTP (Simple Mail Transfer Protocol) and maybe some instant messenger protocol and possibly FTP (File Transfer Protocol) or SSH (Secure Shell). Note that desktop machines may actually be inactive during the fingerprinting period which could oversimplify the recognition process afterwards for this category of machines. Desktop computers belong to "Class B" according to our terminology.
The "Class C" hosts are machines that do not generate any traffic at all on their own. It is one of the major benefits of using active over passive fingerprinting to be able to deanonymize hosts that are inactive during the capturing period. Such a host would not even appear in the anonymized log if it wasn't for us. We took dark space[14] from the SWITCH network to test fingerprinting on inactive machines. Intuitively, it should not

[13]with the most replies going out the SWITCH network during one hour of measurement
[14]addresses that are not assigned to hosts, but routed nevertheless

Class	**Description**
A	Heavy-Load Web Servers
B	Average-Load Desktop Machines
C	Inactive Addresses

Table 6: Test Set of Machines

be hard to make those IP addresses stand out in the anonymized log since we are the only ones connecting to them (apart from random scans). Theoretically very few packets should already suffice to deanonymize those IP addresses successfully.
While this reasoning might sound consistent, in fact all machines are equally hard to fingerprint. It does not matter whether a host receives or generates traffic at all, but rather what's happening in the monitored network altogether. Say someone makes a connection that results in the exact same flow[15] as the one we're currently injecting, it does not matter what type of machine we are fingerprinting, this flow *will* interfere with ours since IP addresses are anonymized afterwards anyway. Everything we will see in the anonymized flow file is two connections with the same attributes to different (anonymized) addresses.

4.2 Methods of Testing

An attacker may be interested not only in successfully deanonymizing a target host, but also in masking his approach as good as possible. Why would an attacker risk detection when he can also look like an ordinary host? Additionally, when fingerprinting traffic becomes too easy to recognize, the straightforward thing to do will be to filter out this traffic before anonymizing the trace. So an attackers actually cares whether his traffic stands out or blends in well with other traffic on the Internet. He should inject traffic that is distinct enough to all other traffic that it stands out and is easily recognized later, but does not attract too much attention. An attackers strategy should also include the type of host he is attacking, as certain servers have very monotonous traffic while others allow for more diverse connections. We made several injections with different traffic patterns that should cover all aspects discussed. All of our traffic was injected at times of day with medium or high amounts of traffic (Table 7).

Parallel to the traffic patterns, we devised anonymization "policies" that we want to test against. Those policies span a wide range of possible anonymizations and ideally should cover most of what can be observed and is likely to be used. We paired each

[15]to another IP address but at the same time

Class	**Date / Time**
A	21.02.09 / 12:00 - 13:00
B	27.02.09 / 16:00 - 17:00
C	21.02.09 / 13:00 - 14:00

Table 7: Injection Times

policy with each pattern to find out which policy is effective against which pattern and vice versa.

Patterns Patterns in our settings are denoted with upper case letters as P_x. We wanted to have very different injection patterns in a way that when injected, they could be used to break various anonymization schemes simultaneously. It should particularly be hard to counter or find those patterns in the trace in an effort to decrease the chance of the injected traffic to be eliminated easily. The patterns we've used range from very simple patterns with one packet on a well-known port to large-scale injections made up of largely random attributes.
What makes a pattern successful? What makes it easily detectable by the defenders and therefore useless? How large does a pattern need to be to prevail? If we only consider NetFlow data, we deal with flows containing the following attributes: *Source IP*, *Destination IP*, *Source Port*, *Destination Port*, *Byte Count*, *Packet Count*, *Timestamps*, *TCP Flags*, *Type of Service* and the *IP Protocol Field.* Assuming we only consider those attributes, this spans a vector space with ten dimensions. What we would like to do is to find the vectors of the traffic we've injected and then project on the "Destination IP" field. Since a couple of base values will be changed somehow or omitted by an anonymization scheme, we can only hope that those bases that remain are unique enough (in combination) for us to recover. For the sake of simplicity, we will focus on the ports, the counters and the timing, assuming that IP addresses will never be available in anonymized records.
Lets first discuss how it is possible to deanonymize a single server with as few traffic as possible. If this is really an issue, one would analyze available traffic patterns from the server through public information sources and try to find ports that are very rarely used in combination with packet sizes that are very unusual for this particular server. If all this succeeds, there is already a great deal of information available on this server and deanonymizing the server won't be a big deal anyways (via passive fingerprinting for example). Sending one or a few packets to an unused port could already suffice in this case, but the chance that it doesn't is still high. If however one wants a more uni-

P	**Source Port**	**Dest. Port**	**Timing**	**Packets**	**Size**
P_1	Fixed	Service	Regular	1	160 / @
P_2	Random	Random	Regular	5	256 / @
P_3	Fixed	Service	Regular	10	480 / @+32
P_4	Random	Random	Regular	10	832 / @+32
P_5	Random	Random	Random	50	1208 / @+rand(0,8)

Table 8: Injection Patterns

versal approach, one could use the timing information in the packets to build sequences of packets, which minimizes the chance of duplicates in the trace. Introducing packet sequences with unique attributes increases the chance of recovery in the final trace exponentially. When using such timing information, attributes don't need to be that different from normal traffic, the sequence alone will make the pattern recognizable. One could even go further and spoof the source IP address with random addresses from widely used services on the Internet such as Akamai [12] or Doubleclick [8] to mask the attempt to fingerprint a machine.

We've now seen that there are two ways to deal with the deanonymization problem: using unique attributes in packets that really stand out in contrast to all other traffic and using sequences of traffic that look ordinary. Of course those two approaches can be combined to make up for a really strong packet detection mechanism. As seen in Table 8, we begin with a single packet to a well-known port that only has (hopefully) unique source port and packet size fields. While this single packet won't do us much good when it comes to deanonymizing heavy-load servers, pattern P_2 already uses a small sequence of packets with fixed size and random ports. P_3 is thought as a not-so-offending method to fingerprint servers as it connects on the service port of a server and thus looks legitimate. Like with P_1, this only relies on source ports and sizes, but additionally constitutes a sequence of ten packets and will be merged into one flow, since all flow attributes are fixed. This is the only multi packet flow we will generate, all other resulting flows will only contain one packet in most cases[16]. The last two patterns P_4 and P_5 extend this idea of sequences and random values even further. The last pattern should prove to be an irresistible combination of unique random attributes and sequences of packets that should give a hard time to even the most aggressive of anonymization schemes.
We chose to use random values over fixed ones that are known not to occur very often, because we want the patterns to be unpredictable, hard to counter and universal.

[16]they could contain more than one due to chance when a random source port is several times the same during a pattern

A	**IPs**	**Ports**	**Timing**	**Packets**	**Size**
A_0					
A_1	Permute				
A_2	Permute			Offset (5)	Offset (50)
A_3	Permute	Bucket (6)	Offset (30)		
A_{3A}	Permute	Bucket (2)	Offset (60)		
A_4	Permute	Bucket (6)	Offset (30)	Offset (5)	Offset (50)
A_{4A}	Permute	Bucket (2)	Offset (120)	Offset (10)	Offset (200)
A_5	Truncate (08)	Bucket (6)	Offset (30)		
A_6	Truncate (12)	Bucket (6)	Offset (30)	Offset (5)	Offset (50)

Table 9: Anonymization Test Policies

Anonymization Policies Policies in our settings are denoted with upper case letters as A_x. In a way our devised policies are closely related to the patterns we've injected. Just like the patterns, they describe what we do to each of the flow attribute. The patterns characterize what we do to flows *before* they are injected, the policies what's being done *after* the injection took place.
As stated above, when anonymizing there is always a trade-off involved. Too strong an anonymization results in the data being unusable but a too weak anonymization policy makes the data transparent and violates the privacy of the users. We tried to take this into account by offering a wide range of anonymization policies. Our intent is to use rather strong anonymization techniques and demonstrate that even when data is nearly useless for researchers, our fingerprinting still works reliably.

The policies with which we came up are listed in Table 9. The first policy, A_1, is kind of standard nowadays and involves masking the IP addresses only. In our case it does not matter what kind of masking is deployed, as long as it maintains a one-to-one mapping[17] between addresses. Our expectation is that this widely used anonymization technique provides no or very little defense against an active fingerprinting attack. The following policies A_2 to A_4 apply a technique to blur values by adding or subtracting small values, which is called "Offset" in the table. The number in parentheses denotes the offset that is used, for example if an offset of 30 is used and the original value is μ, then the anonymized value is uniformly distributed in the interval $[\mu - 30, \mu + 30]$.
Stronger policies also categorize values into buckets, which is a special form of generalization of data. The number in parentheses denotes the number of buckets. When using 2 buckets, we chose one bucket to contain all well-known ports and the other bucket

[17]unlike truncation used in later policies which creates a one-to-many mapping

Pattern	**Class A**	**Class B**	**Class C**
P_1	100.0	100.0	100.0
P_2	100.0	100.0	92.0
P_3	100.0	100.0	100.0
P_4	100.0	100.0	99.0
P_5	100.0	100.0	99.2
μ	100.0	100.0	98.04

Table 10: Recognition without Anonymization and Mean Recognition Values of Patterns [%]

all remaining ports. When there are 6 buckets available, the first bucket contains the well-known ports, the second the port numbers from $1'024$ to $10'000$, the third the ports ranging from $10'001$ to $20'000$ and so on. Note that A_4 is already a very strong mechanism as no field is left untouched during the anonymization. All the fields have some sort of modification, but yet the anonymization can be reversed due to the IP address field being one-to-one mapped. The last two policies try to remove exactly this mapping from the anonymized record. A_5 and A_6 truncate the IP addresses and as a consequence remove the one-to-one mapping between addresses. The number in parentheses represents the number of bits to truncate. When using 8 bits of truncation, what we logically see is the connections between subnets. What we expect here is not a complete IP address but an IP with the last 8 bits truncated, meaning a subnet, as final result. The last policy, A_6, should prove to be very strong as it uses 12 bits of truncation and thus removes nearly all important connection information completely. The ultimate result that we get here is at best the remaining 20 bits of the destination IP.

4.3 Injection without Anonymization

First we tested the border router's accuracy and our capability to recognize single-packet flows with a total of 30 injections[18] (10 for each class) using 76 packets per test, totalling in 2280 injected packets. The patterns that we used are visualized in Table 8 and the exact attack scripts used are available in Appendix E.

If the capturing device worked absolutely reliable, we would expect to recover all of the injected traffic without exception. Since the SWITCH border routers work in best effort mode, packet loss is expected especially at peak times. As a result some flows will contain less packets or even vanish completely in case the packet count was low to

[18]targeting 30 different hosts altogether

begin with. From the 76 packets we used per test only 10 are expected to be merged into a flow, the other 65 of them constitute single-packet flows and are therefore likely to disappear completely in case of an error.
The results from the recovery are shown in the tables of Table 10. The results are very promising indeed as 99.35% of packets were recovered successfully during the tests. As detector we used the Analyzer like in the following tests, but the IP addresses were considered as well in addition to all other attributes. Whereas the results are perfect for classes A and B, patterns P_2, P_4 and P_5 of class C leaked 4, 1 and 4 flows respectively. We suppose these packets were lost already during capturing and didn't make it into the exported NetFlow data. During the following anonymizations we did not compensate for those losses, although we know exactly which packets were not found without anonymization, because the capturing is an integral part of the whole chain and must be considered a possible source of error as well. Also this step is unavailable to attackers under normal circumstances since they do not have the clean version of a trace and are therefore unable to estimate the capturing error.

4.4 Injection with Anonymization

We will present now the results for each anonymization method. The injections were performed once and the resulting data was anonymized according to several different policies and then tested against our the Analyzer of our framework.

Policy 1

The first policy, A_1, involves only a permutation of the IP addresses and leaves all other flow attributes untouched. This means the addresses are reordered with a bijective function, but are still taken from the same set as before. As expected, we can observe that the percentages of the recognition matches the ones from A_0 (Table 10) exactly, as there are an abundance of attributes we can use to find the perfect match for each flow (Table 11). If we account for the capturing errors, we even have an overall match of 100%. The row labeled Σ is the overall recognition of the IP addresses when all patterns are taken into account. Combining the recognition of all patterns and making a frequency analysis of the occurring destination IP addresses, we can form an overall recognition that improves the detection process even further.

In the tests we did not actually permute the IP addresses, but we left them untouched, but did not consider them in the matchings. It is clear that the anonymization function does not need to be a permutation on the set of occurring addresses, it can be an arbitrary bijection of the IPv4 address space. In fact an injective function is already sufficient in

Pattern	**Class A**	**Class B**	**Class C**
P_1	100.0	100.0	100.0
P_2	100.0	100.0	92.0
P_3	100.0	100.0	100.0
P_4	100.0	100.0	99.0
P_5	100.0	100.0	99.2
Σ	100.0	100.0	100.0

Table 11: Recognition of Anonymization Policy A_1 [%]

Pattern	**Class A**	**Class B**	**Class C**
P_1	100.0	100.0	100.0
P_2	100.0	100.0	92.0
P_3	100.0	100.0	100.0
P_4	100.0	100.0	99.0
P_5	100.0	100.0	99.2
Σ	100.0	100.0	100.0

Table 12: Recognition of Anonymization Policy A_2 [%]

our case. Every injective function that maps from the IP address space to some other set is acceptable and will give the same results.

Policy 2

Since permuting only the addresses is clearly insufficient to counter the fingerprinting, we will now subsequently build up more and more elaborate anonymization schemes from A_2 to A_4. The current policy, A_2, also distorts the byte- and packet sizes slightly in addition to the IP permutation. This reduces the flow attributes that can be perfectly matched to only the ports and the timestamps. We used an offset of 50 for the byte count and an offset of 5 for the packet count in our tests.

As we suspected it is more than enough to have two independent attributes that remain unchanged. Table 12 demonstrates the overwhelming results when anonymizing with this policy. As with the two policies already discussed, we have an overall match of 99.35% and if we consider the capturing errors, even blatant 100% were recognized correctly. Again it does not matter which pattern was utilized as all patterns were recognized

Pattern	**Class A**	**Class B**	**Class C**
P_1	100.0	80.0	100.0
P_2	46.0	22.0	24.0
P_3	100.0	100.0	100.0
P_4	95.0	94.0	95.0
P_5	97.0	91.2	97.0
Σ	100.0	100.0	100.0

Table 13: Recognition of Anonymization Policy A_3 [%]

entirely.

Policy 3

As a next step, we chose to anonymize not the packet count and packet size, but the other two remaining attributes (ports and timestamps) instead. This method should prove to be more successful because packet size and packet count have somewhat limited variety. Very many flows contain exactly one packet and the size of a packet can only range from 1 to 1500 bytes due to Ethernet having a standard MTU (Maximum Transmission Unit) of 1500. Timestamps on the other hand are vital in recognizing flows and ports have a wide range of acceptable values (1 to 65535). Ports were categorized into 6 buckets, one for the well-known ports and then always one for the next $10'000$ ports whereas timestamps were given an offset of 30 seconds.

Still the injected flows were recognized very reliably during the tests. The results for this policy are summarized in Table 13 and feature an overall recognition of 82.75%, although pattern P_2 was detected very poorly[19]. The other patterns all were recovered very well with an average detection rate of over 90%, the most successful ones being P_3 and P_5. When considering all patterns P_1 to P_5 simultaneously, we can achieve a 100% deanonymization of all targeted IP addresses. As in the previous cases, all 30 addresses were guessed correctly by the Analyzer. Those stunning numbers seem to imply that when repeating patterns or taking combinations of patterns we can increase the detection rate step by step.

[19]without P_2 the number raises to 95.77% of recognized flows

Pattern	**Class A**	**Class B**	**Class C**
P_1	0.0	20.0	0.0
P_2	2.0	0.0	0.0
P_3	90.0	90.0	100.0
P_4	31.0	19.0	33.0
P_5	59.0	50.0	68.6
Σ	90.0	100.0	100.0

Table 14: Recognition of Anonymization Policy A_{3A} [%]

Policy 3A

Policy A_3 was the first to show a tiny effect on our patterns, so we decided to make a more aggressive form of this policy by reinforcing the existing anonymizations. We only classified ports into two buckets: one for the well-known ports and one for the rest. Additionally we gave the timestamps an even bigger offset from the original value with 60 seconds.

The results in Table 14 support our presumption that the anonymized fields were a great help in the tests before nevertheless. The fact that a stronger anonymization lowers the detection rate considerably raises the suspicion that a combination of attributes is way stronger than the sum of the single attributes.
The overall recognition is still very good with nearly 100% of flows being detected. We take this as a further hint that a combination of patterns is indeed very strong and sequentially injecting patterns can boost the recognition extremely although the single patterns are not recognized very well.

Policy 4

The policy A_4 is actually a combination of the two previous ones, A_2 and A_3. To every useful attribute in the flow we have applied some sort of anonymization, e.g. bucketizing for the ports and offsets for all other fields. We would like to emphasize that this anonymization leaves very little information intact. What are you going to see as a researcher when inspecting the data? "I've seen a flow that started somewhere from 13:36 to 13:37 from one IP to another on some high port to a well-known port with about 3 packets and a total size of about 230 bytes". While maybe still useful for certain statistical analyses, most studies on the data are just not possible anymore due to this extreme blurring.

Although this policy is very strong and gives us quite distorted data, the fingerprinting

Pattern	Class A	Class B	Class C
P_1	0.0	0.0	0.0
P_2	0.0	0.0	0.0
P_3	50.0	0.0	30.0
P_4	10.0	3.0	5.0
P_5	25.0	11.2	19.6
Σ	70.0	60.0	40.0

Table 15: Recognition of Anonymization Policy A_4 [%]

is not that heavily impaired as one might expect (Table 15). While patterns P_1, P_2 and P_4 achieve very low detection rates, the remaining ones are actually not that bad. Ranging from 0.0% to 50.0%, those two pattern together have an average recognition rate of 22.63%. Surprisingly high is the overall recognition rate which lies at over 56% and even at 70% for the web servers.

Policy 4A

Since A_4 seemed to work not so bad after all in "countering" our fingerprinting, we wanted to have an extreme case. The policy is ridiculously strong and would never be employed like this on live data, as most flows will look exactly the same in this setting. Ignoring the timestamps for once, you will see about 8 to 14 different flows depending on the traffic you observe in your network. This does not allow for much of an analysis, but we tested the policy nevertheless.
We classified the ports again into two buckets and chose an offset of 120 seconds for the timestamps. This does not allow to determine the exact starting point of a flow exactly with an uncertainty of 4 minutes. Additionally we chose an offset of 10 for the packet count and 200 for the byte size. Combining all those facts, you will see at most about 250 different flows during one hour. It is clear that the more traffic is captured, the more flows actually have the same attributes.

The outcome looks devastating at first sight (Table 16). There was no recognition at all, most flows were matched with the wrong counterpart. What was the problem with our fingerprinting? Does fingerprinting just not work anymore in this setup? If yes, where is the boundary when it stops working? When posing the questions like that, it becomes clear that the fingerprinting cannot just cease to work at some point when it worked well until now, since detection did not stop at some point but went down gradually. A

Pattern	Class A	Class B	Class C
P_1	0.0	0.0	0.0
P_2	0.0	0.0	0.0
P_3	0.0	0.0	0.0
P_4	0.0	0.0	0.0
P_5	1.0	0.0	0.0
Σ	0.0	0.0	0.0

Table 16: Recognition of Anonymization Policy A_{4A} [%]

more likely explanation is that it depends on the fingerprinting itself and the patterns employed by it.
The obvious thing to state is that our 5 patterns were not sufficient for the magnitude of blur this policy employs. Yet this anonymization scheme can be defeated, namely with more injections and possibly also with differently structured patterns. You might ask yourself why we didn't do this as well. The problem is that there is always a more aggressive policy. That is, until you reach the point where all flows have the same attributes, except the IP addresses.
That the information is completely useless when all attributes are the same, is out of the question, but still we can recognize our fingerprints in this extreme scenario by means of a *temporal pattern*. This technique is discussed in depth in Section 5.1.4.

Policy 5

Before analyzing the results in depth, we want to pursue another approach. We are now convinced that it is impossible to counter a fingerprinting when the IP addresses are only permuted and thus identifiable one by one in theory. The method we're using now is called *truncation* and in essence cuts off bits from addresses. Policy A_5 truncates the last 8 bits from the IP addresses whereas the last policy will erase the 12 least significant bits.
This is an entirely different way to make several flows look the same than we've used until now. IP addresses are (unique) identifiers of hosts in a network, so messing with addresses in this way is an irreversible modification to a network trace. Truncation is a one way function in contrast to permutation which is reversible by definition[20]. When anonymizing flow attributes such as size or ports, the uniqueness of the flows depend on the other flows that can currently be observed in the network, but when truncating

[20] see Section 2.1 about anonymization versus pseudonymization

Pattern	Class A	Class B	Class C
P_1	100.0	80.0	100.0
P_2	46.0	22.0	24.0
P_3	100.0	100.0	100.0
P_4	95.0	94.0	95.0
P_5	97.0	91.2	97.0
Σ	100.0	100.0	100.0

Table 17: Recognition of Anonymization Policy A_5 [%]

addresses you actively remove the uniqueness of flows.
Attributes are anonymized identically to A_3, with the important exception of the IP addresses, which are truncated. What can logically be observed now, is connections between subnets. In a real test these addresses would be permuted first and then truncated, but since we simulate permutations only, we skipped this step.

The numbers obtained (Table 17) are identical to those of policy 3 (Table 13), but the percentages have slightly different meaning here. We were able to correctly identify the subnets only, not single IP addresses. The fact that the numbers are exactly the same probably suggests that no other connections were captured between the two subnets[21] during the measurement period, that could decrease or increase the recognition. So the recognition worked as well as it could, but still we were unable to identify the target IP directly, because that information was erased from the clean log by the truncation.

Policy 6

To have another sample of truncation we also duplicated policy A_4 and employed truncation instead of permutation. This time we've cut off the least significant 12 bits in an attempt to reduce the remaining information from addresses even further.

As expected, the raw numbers (Table 18) are largely the same as when anonymizing with policy A_4 (Table 15). Instead of a target IP address `xxx.2.229.7`, we receive the truncated version of the address: `xxx.2.212.0`. Technically we obtained an arbitrary IP from the subnet `xxx.2.212.0/20`, which can be one of $2^{12} = 4'096$ possible target addresses. When truncating more and more bits, we increase the size of the *anonymity set* that contains the target address exponentially. The recognition of flows seems to be largely unaffected by truncation, but the interpretation of the results changes considerably.

[21] attacker subnet and victim subnet

Pattern	Class A	Class B	Class C
P_1	0.0	0.0	0.0
P_2	0.0	0.0	0.0
P_3	50.0	0.0	30.0
P_4	10.0	3.0	5.0
P_5	25.0	11.2	19.6
Σ	70.0	50.0	40.0

Table 18: Recognition of Anonymization Policy A_6 [%]

4.5 Useful Injection Patterns and Universal Patterns

A pattern is successful when it does not occur at this time in the whole network. If the pattern is sufficiently different from all other connections occurring at the time, it has a good chance to be successfully detected. The more diverse an injected packet is the more resilient it is against random blurring of values. An example for this is the development of patterns P_1 and P_2, which are perfectly recognized up to some point and after that the detection drops to 0%. Clearly those patterns were diverse enough as long as no anonymization occurred, but as soon as two or more fields were blurred, they just weren't different enough anymore to be recognized.
From the results we have seen that certain patterns are better suited for fingerprinting while others didn't work so well. We realize we had a somewhat limited amount of patterns to inject and we created them "out of the blue", but we can deduce certain tendencies from them nevertheless. In order to get more successful patterns, a thorough analysis of a trace should be conducted. For instance, perform a frequency analysis for each of the flow fields, take the most rarely used values for each field and shape your fingerprints accordingly. The fingerprints should even be crafted in a way that they emit packet fields with the reverse frequency of the capturing device, i.e. the most rarely occurring values are generated the most and vice versa. This analysis of the trace should be conducted at different times and then the injection time should be chosen when the probability distribution resembles a uniform distribution the most or when traffic is generally very low.

We had no knowledge about the distribution of traffic in our network, so we decided to create patterns that are generic in a way that they should perform well in most environments. Most importantly we chose to use random attributes to make the pattern unpredictable and more impervious to errors occurring due to chance. During the tests we noticed that patterns P_3 and P_5 seem to be overly successful in comparison to other patterns. This is particularly interesting since P_3 used fixed values on nearly all fields

	A_0	A_1	A_2	A_3	A_{3A}	A_4	A_5	A_6
μ_A	100.0	100.0	100.0	87.6	36.4	17.0	87.6	17.0
σ^2_A	0.0	0.0	0.0	436.2	1184.2	356.0	436.2	356.0
μ_B	100.0	100.0	100.0	77.4	35.8	2.8	77.4	2.8
σ^2_B	0.0	0.0	0.0	810.5	990.6	18.8	810.6	18.8
μ_C	98.0	98.0	98.0	83.2	40.3	10.9	83.2	10.9
σ^2_C	9.3	9.3	9.3	879.8	1533.3	142.6	879.8	142.6
μ_{ABC}	99.4	99.4	99.4	82.7	37.6	10.3	82.7	10.3
σ^2_{ABC}	3.9	3.9	3.9	726.2	1240.0	206.1	726.2	206.1
F	1.6	1.6	1.6	1.8	0.1	1.2	0.1	1.2

Table 19: Analysis of Variance and F-Test

whereas P_5 chooses its values randomly. We suspect that the success of P_3 is largely due to the fact that the values used do not occur often in the SWITCH network, but they could appear frequently in other networks.
Our policy P_5 seems to be the closest one can get to a universal pattern, which is choosing all values randomly. According to the analysis above, the distribution of the random experiments should not be uniform as in our case, but match the reversed distribution of the network.

4.6 Further Considerations

We now pursue our theory about the three classes being equally hard to fingerprint. We fingerprinted web servers, workstations and dark space. This gave us three values per pattern and policy. In order to demonstrate that those tables are in fact similarly distributed, we conducted an Analysis of Variance (ANOVA) and an F-Test. Our null hypothesis reads

$$\mu_{i,A} = \mu_{i,B} = \mu_{i,C}$$

where i denotes the i-th policy. The variances and means are listed in Table 19 along with the F-Test in the last column. According to the F-table[22] the significance value for our test is 3.83. Everything below this value is not significant and as a consequence we keep the null hypothesis for all columns as the values are way below the threshold. Figure 8 also confirms the conjecture visually. We've only plotted the "interesting" policies here, as policies A_0, A_1, A_2 are equal and A_{4A} has nearly 0% recognition. The graph shows

[22]with degrees of freedom $df1 = 12.0$, $df2 = 2.0$

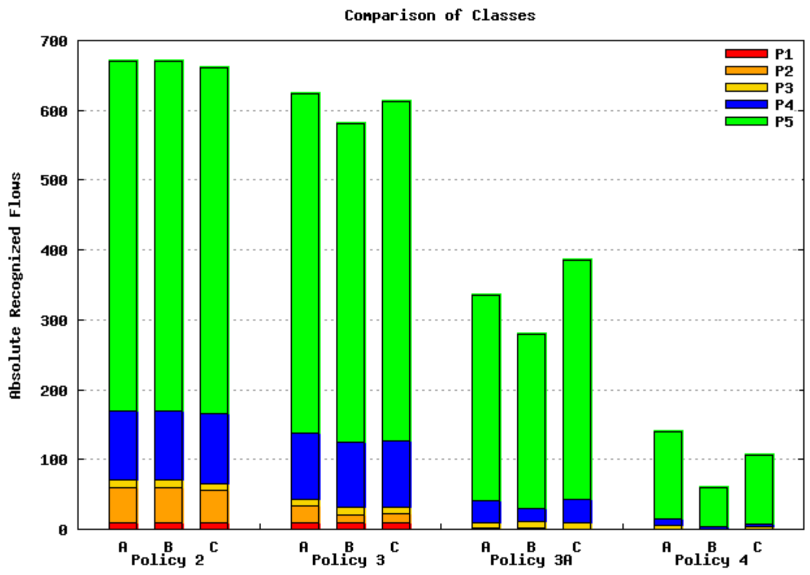

Figure 8: Class Comparison with Absolute Flows Values

policies A_2 to A_4 and the *absolute* amount of flows that was recognized when utilizing a policy. The figure illustrates very well that the differences between the three classes is minimal, considering the injections were performed at different times.

To visualize the results, we plotted them from several viewpoints and with various axes. Since we have established that there is no significant difference between the three classes, we took mean values over the classes in all the graphs. The following two histograms basically plot the same data, but with different emphasis. The first histogram in Figure 9 emphasizes on the patterns P_1 - P_5. It shows by how many patterns each policy was recognized. This figure is somewhat counterintuitive to read, as large bars denote weak policies and vice versa. The rows are also stacked on each other which shows us which patterns contributed decisively to the overall recognition and which didn't. The second histogram (Figure 10) shows the opposite direction. The main statement of the graph is how many policies each pattern was able to deanonymize. On second sight the plot also visualizes which policies were easily deanonymized via the stacked rows.

Combining those two plots we get a histogram that shows single bars for each pattern (Figure 11), separated by policies. This graph contains all the information, but does not get it to the point as well as the stacked ones. Also this is the only plot where relative values are displayed, the other graphs show absolute recognition values.

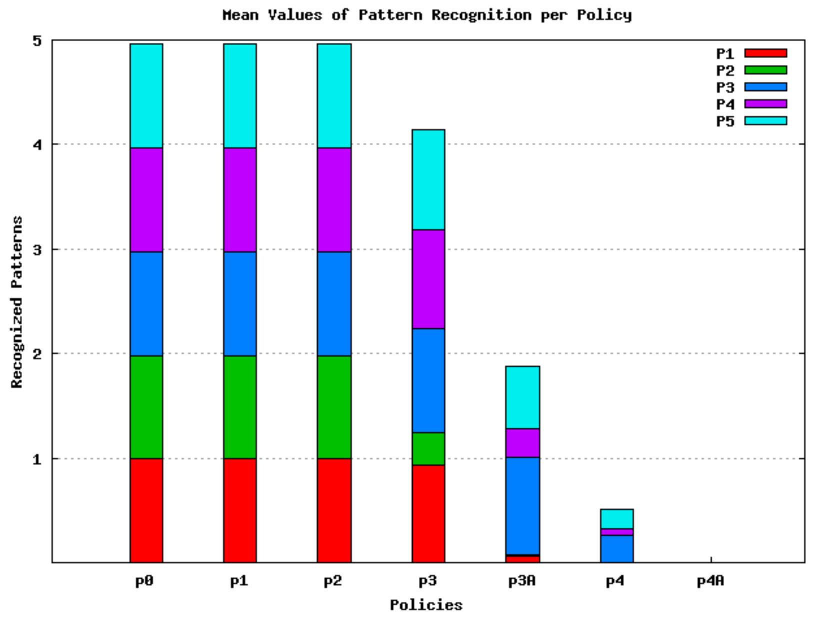

Figure 9: Absolute Pattern Recognition Values per Anonymization Policy

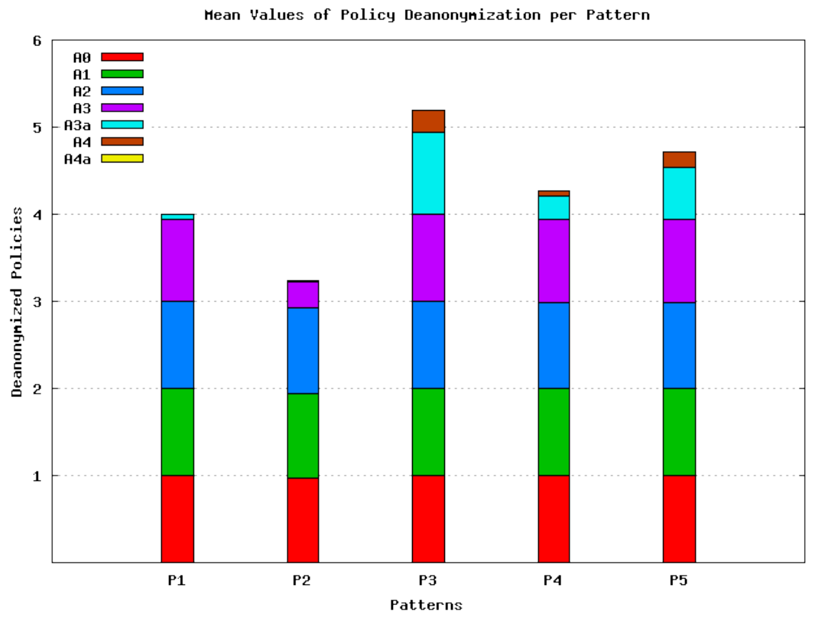

Figure 10: Amount of Deanonymized Policies per Injection Pattern

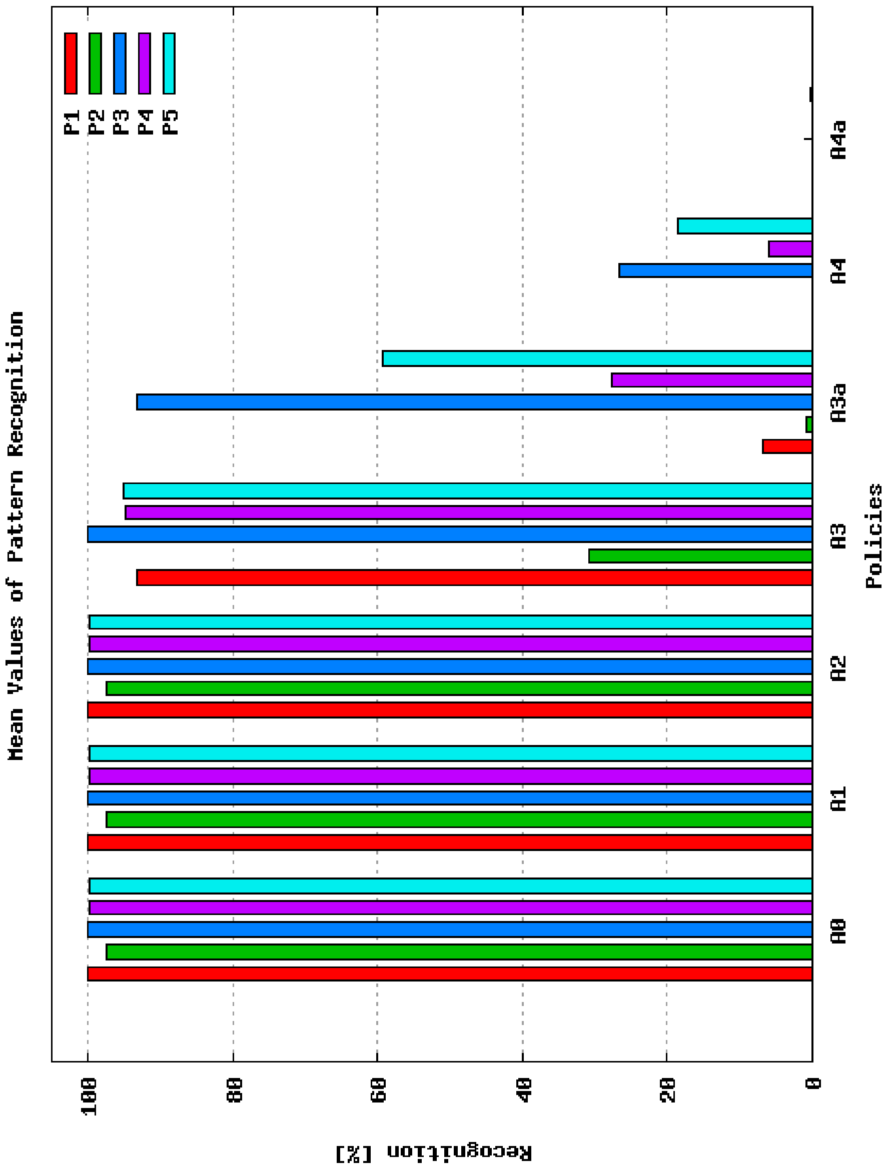

Figure 11: Relative Recognition of Patterns per Policy

5 Conclusions and Outlook

5.1 Discussion

In the last section we have stated that the three classes we fingerprinted are similarly distributed and the F-Test we used could not find a significant difference between the mean values of the table columns. All the columns had a value way below the significance threshold, so the three tables seem to follow the same distribution and are likely to be drawn from the same sample space. This seems to support our conjecture that in fact all hosts in a network are equally hard to fingerprint, independent of the traffic a host actively generates on its own. As suspected earlier, it does only depend on the traffic that is generated in the entire network at any particular time, not on the traffic of individual machines.

5.1.1 Permutation

We have seen that anonymizing only the IP addresses (A_0) is very easy to circumvent via active fingerprinting. As mentioned in Section 4.4 a very weak pattern already suffices to break this anonymization reliably. An attacker would not choose to inject some large pattern, as a single packet (P_1) was enough to demask a host in all test cases. It would be easy to fingerprint all machines in a network this way and reverse the permutation completely. Everything that is needed is a detailed log of the injections with timestamps and the important flow attributes. To summarize, this anonymization scheme cannot be employed in any serious setting as it doesn't offer the needed protection by far.
Basically the same considerations apply to policy A_2, where choosing moderate offsets for packet and byte count did not help at all against our attacks. The injected patterns were all too strong for this policy and therefore A_2 does not seem to be a viable option either.
It does get more interesting with the introduction of A_3. As we can see P_2 suffers a massive drop and is detected very poorly. We suspect this is due to an unfortunate choice of packet count and byte size on our part. The size was chosen to be 256 for all packets in this pattern which is apparently a size occurring frequently in the network. To have fixed size and fixed packet count is obviously a bad choice for this anonymization policy so the result is more than comprehensible. Conversely with P_1, we probably made a rather lucky choice of packet size since the pattern is detected in almost all cases.
When inspecting policy A_{3A} we are not very surprised. Noteworthy is the drop in detection of P_4, which was recognized very well before. Since the pattern is found when anonymizing with policy A_3, the bad detection rate seems to be correlated with the

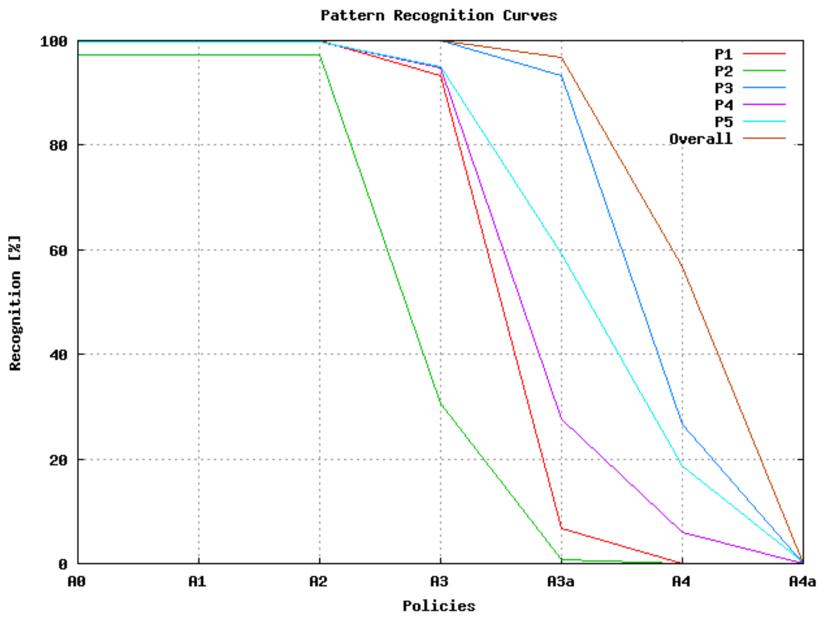

Figure 12: Recognition Curves for the Patterns

ports. This would however affect P_5 the same way as the ports are chosen exactly the same way and the bad percentage does not seem to be a coincidence as it affects all three classes equally. When inspecting the attack scripts in Appendix E, we find the only notable differences between the two patterns are the packet size and the timing. We suspect that the random timing in combination with the large size prevents this problem from occurring in the last pattern.
With A_4, the individual success of the patterns is easily understandable, as the first two are expected to fail in this setup whereas the last three perform quite well under those circumstances. P_4 is again lower than expected, the reasons being the same as in the last policy.

The trends we have discussed so far can be very well visualized with a function plot (Figure 12). We see that the more fixed attributes a pattern has the steeper the slope and vice versa.

5.1.2 Truncation

We were able to establish that truncation does not affect the recognition process as addresses are not even considered in our score computation. Policies A_5 and A_6 have identical results as A_3 and A_4, that employ the same methods but permutation instead of truncation.
In contrast to permutation, the truncation used in those policies had a strong effect on the results nevertheless. The bits that are truncated are in fact *deleted* and are not recoverable with the fingerprinting. Truncation seems to be an good countermeasure as it effectively prevents active fingerprinting of single hosts. However the consequences are substantial because there is no way a researcher can associate IP addresses to single hosts either, so objectively stated much of the information in the trace is destroyed by this method. One could argue that it does not make a lot of sense to use additional anonymization methods next to truncation, like employ port bucketizing or offsetting, since the individual hosts are not identifiable anymore.
Truncation is effective, but unlike anonymization techniques like blurring or permutation, this method deletes information, which makes it an irreversible modification to the trace.

5.1.3 Other Techniques

Having discussed permutation and truncation to addresses, we will now focus on the techniques that are applicable to other flow attributes. We have demonstrated that some anonymization policies are very easy to break and the blazing 100% recognition rates look very promising. But what if you have an unknown anonymization scheme that you do not know how successful your pattern will be with? How can you determine the target IP through the anonymization with a high certainty nevertheless?
The overall detection rate in Figure 12 seems to be well above the other curves, which might indicate that one can indeed increase identification of flows through *sequential application of patterns*. If we utilize a pattern that is recognized in 40% of the cases during Run 1, this pattern will also be recognized in about 40% during the following run. When combining those two sample sets, we have one large set where we can identify 40% of the flows we have injected. The point is that this number does never decrease if you combine more and more such samples, but the number of flows that *didn't match* is dropping from set to set. If you perform this test many times and determine the IP occurring the most, you're very likely to get the target IP you're looking for. Starting with only one sample set, what you will observe, when taking more and more samples into consideration, that one IP will remain at the same percentage while all others will

probably decrease very fast[23].
In essence what you want is not 100% detectability of a pattern, it suffices if the IP you're looking for is the destination IP occurring the most in the trace. Detection rates of 50% to 100% are thus guaranteed to give the correct IP, values below that are not. But with the method discussed you can improve the rate over and over, until it is the one occurring with the highest frequency. As soon as the relative detection rate of the target IP is higher compared to the detection rate of the second best IP, we can retrieve the correct address. When utilizing several sequential patterns, one should plot a frequency diagram of the target addresses occurring the most after each step, that is each time a pattern is added to the evaluation. While most target IP addresses will drop over time, one will remain more or less constant and represents the one we are looking for.

5.1.4 Minimal Requirements

Lets assume all fields have been erased except the timestamps, which have been heavily anonymized. Lets further assume for simplicity that all timestamps have been truncated to 5 minutes, so we see a new block of flows every 5 minutes. The idea here is simple, but deadly effective nevertheless. We make buckets of 5 minutes, since during this time we see identical flows anyway. In the first bucket we inject some[24] packets towards our target host. In the next bucket we *don't inject* at all. In the next we inject the same amount of packets as in the first bucket to the same target host and so on. Now we try to recognize this pattern through the extreme anonymization. After the first bucket we have a huge list of connections that are potential candidates. The second bucket will eliminate all connection that occurred during both buckets and each following bucket will decrease the size of potential matches until we only have one connection pair that must be the one we're looking for. The size will decrease exponentially during this algorithm and after a certain amount of iterations it should contain exactly one pair.
As stated earlier, the detection rate can in principle be increased arbitrarily. We would like to extend this idea and show that completely anonymized records can be recognized as well. As explained in the previous paragraph, when all flow attributes are anonymized, there is still the possibility to work with temporal patterns. Making 5 minutes buckets and injecting every 10 minutes[25] gives us a temporal pattern which can be recognized later on.
Speaking in terms of sets, we perform the relative complement ($\setminus$) and then the intersection ($\cap$) alternately. The former operation will filter out those connections that occur too often while the latter will rule out connection pairs that communicate too rarely.

[23]they will decrease exponentially
[24]enough that we can be sure they won't be lost due to an error in the capturing device
[25]resulting in two 5 minute buckets with one having an injection and one having none

Having saved all connection pairs from the very first bucket in a set S_1, we must eliminate duplicate pairs which gives us the set U_1. The set then develops as follows:

$$U_1 \backslash U_2 \cap U_3 \backslash U_4 \cap U_5 \backslash U_6 ... = U$$

As those two operations are commutative we can rewrite this to

$$U_1 \cap U_3 \cap U_5 \cap U_7 ... = U_\cap$$

and finally

$$U_\cap \backslash U_2 \backslash U_4 \backslash U_6 \backslash U_8 ... = U$$

and obtain the set U which contains all those connection pairs that satisfied the requirements. The size of this set will either stay the same or shrink during the set operations. Assuming that connections are normally distributed, the matching follows a negative exponential curve. If we denote the number of unique connection pairs in the first bucket with X_1, the expected number of intervals N needed is

$$E[N] = \log_2 X_1$$

as the expected size of the set is 1 at this time. The value X_1 can be estimated from previous traces from the same network at the same capturing time, and from the mean value $E[N]$ we can estimate the injection time by multiplying it with the bucket size. Note that the assumptions for temporal patterns are as low as it gets. Timestamps must be available *to some degree*, that is they cannot be erased entirely, but everything else offers an attack vector. All other flow attributes can be erased or anonymized at will. Increasing the strength of the anonymization applied to the timestamps does not compromise the technique, but only increases the injection time. The method works best when the IP addresses are anonymized in some reversible way (*e.g.* permuted), but works just as good if information was deleted from the addresses (*e.g.* truncated) to the extent possible. As all fingerprinting attacks, temporal patterns use a covert channel [31] when using the timing information to later identify the injected packets. In summary it can be said that *one bit of information* is enough to execute a fingerprinting on a target host successfully, as we do not consider any attributes at all, but we only use the fact that a host shows up in the log at a certain time and doesn't at some other time.

5.2 Possible Countermeasures

We have discussed general countermeasures in Section 2.5. Most non-technical measures are not applicable to active attacks, as an active attack can look legitimate (*e.g.* Pattern P_1). For example, to only allow remote analysis of the data does not work for the

reason stated above. For the same reasons it is also very hard to filter out active attacks from a trace, as they can blend in with legal traffic very well. What would indeed work is that every researcher who comes in contact with anonymized data needs to sign an NDA (Non-Disclosure Agreement) explicitly prohibiting him from deanonymizing the data. This approach would shift the problem to law enforcement, as the deanonymization would probably not be difficult, but illegal.
A big difference between passive and active fingerprinting is that an active attacker can break any reversible function that was applied to anonymize the data, whereas a passive attacker can do this only to a limited degree. This forces to delete information from the trace in order to effectively counter an active fingerprinting attack.
The first idea that comes to mind is to delete timestamps completely and release large chunks of data, such that traffic is as diverse as possible. This however does not prevent fingerprinting in general, as the effort on the attacker side would not increase considerably. He might have to inject four identical fingerprinted packets instead of one, but generating traffic that is recognizable is still not hard.
To remove any other flow attributes by deleting them would serve no purpose as we've seen above. The goal of the anonymizer should be to remove the one-to-one mapping of the IP addresses from the trace. As long as each host is identifiable on its own, the trace is not secure from active attackers. Possible measures include truncation, remove bidirectional connections as discussed in Section 2.5 or randomization of the addresses. Basically this is the same trade-off as we encountered with passive fingerprinting, we must weight the utility against the strength of the anonymization algorithm.
When removing information from the IP addresses in the trace one must also consider whether it is save to leave all attributes clean or if they should be blurred to some degree. Depending on the amount of information deleted it could make sense to leave the rest of the trace unanonymized, as the linkability is removed by the previous step.

5.3 Future Work

We only had time to test a limited amount of injection patterns. We did not have enough time to pursue the effects of different patterns in detail. Further test should be conducted to make relevant statements about the universality and strengths of individual patterns with reference to anonymization policies. Furthermore our conjecture about improving the detection rate with sequential application of patterns should be verified in practice. It would be interesting to test our theories about how one can boost the effectiveness of random patterns when making detailed traffic statistics of a network before writing the patterns. The patterns should be tailored to the traffic distribution of the network which should improve the recognition considerably. This should also help greatly in making an attacker undetectable to network administrators. It should be interesting by how much

the detection rate can be improved with the methods mentioned in Section 4.5.
We made all our tests under the assumption that traffic is not being sampled in any way. Tests should be made without this requirement and tested how much more traffic is needed to achieve recognition rates comparable to ours.
It would also be interesting to test different anonymizations to IP addresses that destroy the linkability between hosts as mentioned in Section 5.2.
Our theory about the application of temporal patterns should be tested on live traffic data to verify the claims and test its effectiveness.
As future work the Analyzer part of our framework could be extended to allow also the recognition of other formats than Cisco NetFlow and to improve detection with more sophisticated algorithms.

5.4 Conclusion

In essence, we think that anonymization should not be applied to protect network activity log files, at the very least not the way it is done today. We have demonstrated that strong anonymization schemes can be broken successfully with less than 80 injected packets[26] overall and we've given a step by step guide how to strengthen the application of patterns in case the detection should prove insufficient. In the last chapter we have explained a mechanism to deanonymize the "ultimate" anonymization policy where nothing else than timing information is available to a limited degree, and everything else is not available to use in the fingerprint. *The success of deanonymization depends therefore completely on the anonymization technique used on the IP addresses.* The more bits are deleted from the addresses, the more inaccurate results are available to the attacker.
We have seen that, in most cases. permutation and offsets are useless against an active attacker. We basically see three options for a network administrator when faced with the question of releasing network activity logs. The first and safest option is not to release any logs. The second option is to permute the IP addresses or apply a similar pseudonymization, but anonymize everything else so heavily that the information is practically useless. This is the path we have chosen with patterns P_1 to P_{4A}. The problem here is that the IP addresses are recoverable in theory and single hosts can be identified through the IP addresses, so everything else has to be blurred so heavily that most flows will actually fall together. The third option is then to apply truncation or a similar anonymization to the IP addresses, and optionally distort other attributes if needed. This was covered in our tests with patterns P_5 and P_6. This approach does not leave the single hosts identifiable one by one, but groups them together in a large anonymity set. Pseudonymizing other attributes becomes less important now, as single

[26]for most schemes applied today even 1 packet was sufficient

individuals can no longer be accurately identified.
When inspecting the three choices above, we see that the first two can actually be merged, as releasing logs that are anonymized with an extremely strong policy are worthless and give the researcher no additional value. This leaves us with exactly two options: either don't release network logs or release them while truncating the IP addresses in the trace.

A Original Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Master Thesis
for
David Sauter

Supervisors: Martin Burkhart, Dominik Schatzmann

Issue Date:	16.09.2008
Submission Date:	15.03.2009

Invasion of Privacy Using Active Fingerprinting Attacks

1 Introduction

Over the last decade, the network security community has suffered from two fundamental and related problems: (i) a lack of "real" network data for research studies and method validation, and (ii) reluctance of organizations to share network data, which impedes cooperation in network defense; as attacks typically cross organizational boundaries, effective prevention requires defenders to look beyond their own perimeter in cooperation with other organizations.

Anonymization techniques are crucial for the safe sharing of network data. They are necessary to obscure certain identifying information (e.g., IP addresses) in order to protect the privacy of end users and the security of internal networks. While current network data anonymization techniques are generally acknowledged to be useful, it is difficult to ensure they are free from information leakage.

Recent work has shown that many state-of-the-art techniques for IP address anonymization are not as secure as expected [4, 6, 3, 8]. The reason for this weakness is rooted in the fact that random permutation and (partial) prefix-preserving permutation [5, 7] are reversible. Permutations, in general, are vulnerable to fingerprinting attacks and behavioral analysis, i.e., individual hosts can be profiled and mapped back to original entities.

The weakness of permutation-based approaches is obvious even with the passive adversaries assumed by previous work. Although the adversary is capable of probing the network in some cases, e.g. by scanning for open ports [8], he is not able to manipulate the anonymized traffic data directly.

This thesis should make the next logical step and assume a more powerful *active adversary* capable of injecting arbitrary traffic into the network at hands [2, 1]. This traffic is then captured, anonymized (e.g. using FLAIM [9]), and published. If the adversary successfully recognizes his fingerprinted traffic, he can step-by-step recover the secret mapping between original and anoynmized IP addresses. This attack bears analogy to the chosen-plaintext attack in cryptography. The adversary picks an IP address, injects a fingerprint to this address, recovers the fingerprint in anonymized data and learns the mapping for the chosen IP address.

The task of this thesis is to develop a framework for active fingerprinting attacks. That is, several methods of adding a fingerprint to a network packet or a sequence of packets (e.g. magic packet sizes, port numbers, timing between different packets) have to be implemented. The injected fingerprints have to be recognized in anonymized traces. The effectiveness of existing anonymization techniques to protect against fingerprinting attacks has to be studied and possible countermeasures need to be investigated.

For the evaluation of the framework, the student has access to live network traffic traces (Cisco NetFlow) from the five border routers of the Swiss Education and Research Network SWITCH.

2 The Task

The task of this thesis is to develop a framework for active fingerprinting attacks.

The task is split into four major subtasks: (i) literature study, (ii) design of a fingerprint injection framework, (iii) implementation of the framework, and (iv) evaluation of anonymization techniques and possible countermeasures with real data.

2.1 Literature study

David should actively search for and study secondary literature. A short survey on passive and active fingerprinting attacks on anonymization techniques should be written.

2.2 Design of the application

A framework for active fingerprint injection should be designed. The framework should meet the following requirements:

- It provides an easy interface to configure and run different types of injection attacks with all the necessary parameters.
- Attacks can be performed (i) online, by crafting and sending real network packets (ii) offline, by injecting fingerprints into captured flow traces
- A log of performed attacks should be written that serves to later identify the injected traffic in anonymized traces
- Given anonymized traces and a log file of an attack, the framework must try to identify as much of the injected traffic as possible

Preferably, the complexity of the application is increased step by step. Once the basic functionality is provided, more involved algorithms can be developed.

2.3 Implementation of the application

Implementation language is C++, the platform is Linux.

2.4 Evaluation of the application

The effectiveness of existing anonymization techniques to protect against the implemented types of active fingerprinting attacks has to be studied and possible countermeasures need to be evaluated using the implemented framework. For this purpose, captured flow traces from the SWITCH network can be used.

3 Deliverables

The following results are expected:

- Short survey on passive and active fingerprinting techniques
- The design of the fingerprinting framework
- The implementation of the framework
- Evaluation of existing anonymization techniques in the light of active fingerprinting. New countermeasures should be devised and also evaluated.
- A final report, i.e., a concise description of the work conducted in this project (motivation, related work, own approach, implementation, results and outlook). The abstract of the documentation has to be written in both English and German. The original task description is to be put in the appendix of the documentation. The documentation needs to be delivered at TIK electronically. The whole documentation, as well as the source code, slides of the talk etc., needs to be archived in a printable, respectively executable version on a CDROM.

4 Assessment Criteria

The work will be assessed along the following lines:

1. Knowledge and skills
2. Methodology and approach
3. Dedication
4. Quality of Results
5. Presentations
6. Report

5 Organizational Aspects

5.1 Documentation and presentation

A documentation that states the steps conducted, lessons learned, major results and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another developer within reasonable time. At the end of the project, a presentation will have to be given at TIK that states the core tasks and results of this project. If important new research results are found, a paper might be written as an extract of the project and submitted to a computer network and security conference.

5.2 Dates

This project starts on September 16th, 2008 and is finished on March 15th, 2009. It lasts 6 months in total. At the end of the second week David has to provide a schedule for the theses. It will be discussed with the supervisors.

Two intermediate presentations for Prof. Plattner and the supervisors will be scheduled 2 and 4 months into this project.

A final presentation at TIK will be scheduled close to the completion date of the project. The presentation consists of a 20 minutes talk and reserves 5 minutes for questions. Informal meetings with the supervisors will be announced and organized on demand.

5.3 Supervisors

Martin Burkhart, burkhart@tik.ee.ethz.ch, +41 44 632 56 63, ETZ G95
Dominik Schatzmann, schatzmann@tik.ee.ethz.ch, +41 44 632 54 47, ETZ G95

References

[1] S. Anotonatos, D. Antoniades, M. Foukarakis, and E. P. Markatos. On the anonymization and deanonymization of netflow traffic. In *FloCon 2008*, 2008.

[2] J. Bethencourt, J. Franklin, and M. Vernon. Mapping internet sensors with probe response attacks. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[3] T. Brekne, A. Arnes, and A. Øslebø. Anonymization of IP traffic data: Attacks on two prefix-preserving anonymization schemes and some proposed remedies. In *Workshop on Privacy Enhancing Technologies*, pages 179–196, 2005.

[4] S. Coull, C. Wright, F. Monrose, M. Collins, and M.K.Reiter. Playing devil's advocate: Inferring sensitive information from anonymized network traces. In *14th Annual Network and Distributed System Security Symposium*, February 2007.

[5] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon. Prefix-preserving IP address anonymization. *Comput. Networks*, 46(2):253–272, 2004.

[6] D. Koukis, S. Antonatos, and K. G. Anagnostakis. On the privacy risks of publishing anonymized IP network traces. In *Communications and Multimedia Security*, volume 4237 of *Lecture Notes in Computer Science*, pages 22–32. Springer, 2006.

[7] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *SIGCOMM Comput. Commun. Rev.*, 36(1):29–38, 2006.

[8] B. Ribeiro, W. Chen, G. Miklau, and D. Towsley. Analyzing privacy in enterprise packet trace anonymization. In *15th Annual Network and Distributed System Security Symposium (NDSS 08)*, February 2008.

[9] A. Slagell, K. Lakkaraju, and K. Luo. Flaim: A multi-level anonymization framework for computer and network logs. In *20th USENIX Large Installation System Administration Conference (LISA'06)*, 2006.

August 28, 2008

B User Manual and Installation

The framework is divided into two executable programs, namely the injection binary and the Analyzer binary. In order to compile the Injector, you should have the Xerces library installed (refer to Section 3.2 about libraries) and all the framework components in the same base directory. You also need the libraries ProcessingNG and NetflowVxPlusPlus. The directory listing will look something like

```
drwxr-xr-x  4 user user 4096 09-03-11 10:18 Packet/
drwxr-xr-x  7 user user 4096 09-02-27 18:51 Analyzer/
drwxr-xr-x  7 user user 4096 09-03-11 10:19 Injector/
drwxr-xr-x  9 user user 4096 09-01-12 15:03 ProcessingNG/
drwxr-xr-x  7 user user 4096 09-01-05 14:34 NetflowVxPlusPlus/
drwxr-xr-x  8 user user 4096 09-03-02 16:16 ../
drwxr-xr-x 12 user user 4096 09-03-10 22:34 ./
```

We will refer to the current directory as root directory. To create an executable for the Injector, you should change the directory to the `Injector/` directory and issue

```
user@host:/Injector$ make
```

which will create the binary `InjectorStart` by compiling the Packet library, the Generator modules and the Injector itself. This executable can only be launched as user `root`, since raw sockets are utilized. Launching the binary will output

```
usage: ./InjectorStart if_device input_file
```

The `input_file` is the generator file describing the attack, which was discussed in Section 3.4.2 about the Packet library. To see a sample generator file, have a look at `generatorFile.sample` in the root directory. The device is your network device, most commonly `eth0` or `wlan0`. If you choose just to simulate the injection or want to test the attack script, you can inject on the loopback device `lo`. When the executable has finished injecting packets into the network, it will conclude with

```
Packet log written to "trace.xml"
```

which is the logfile for the injection and should be renamed and stored for the analysis afterwards. If you have many sequences in a generator file, you should write the IP sections without the actual addresses. A small script named `fill.sh` can include them for you later. To use this script on you attack file issue

```
user@host:/Injector$./fill.sh attack.conf "12.12.12.12" "11.11.11.11" \
> attack.script
```

where `attack.conf` was the generator file without addresses, the next argument is the source IP, followed by the destination IP. The output of the script is redirected into the final attack script which can be used as input to the Injector.
To start the Analyzer, you should have the Xerces library, the ProcessingNG library as well as the NetflowVxPlusPlus library installed. The latter should reside in the root directory just like the framework itself. A listing should give something like

```
drwxr-xr-x  7 user user 4096 09-02-27 18:51 Analyzer/
drwxr-xr-x  4 user user 4096 09-03-11 10:18 Packet/
drwxr-xr-x  7 user user 4096 09-03-11 10:19 Injector/
drwxr-xr-x  9 user user 4096 09-01-12 15:03 ProcessingNG/
drwxr-xr-x  7 user user 4096 09-01-05 14:34 NetflowVxPlusPlus/
drwxr-xr-x  8 user user 4096 09-03-02 16:16 ../
drwxr-xr-x 12 user user 4096 09-03-10 22:34 ./
```

To create an executable for the Analyzer, you should change the directory to the `Analyzer/` directory and issue

```
user@host:/Analyzer$ make
```

which will create the binary `AnalyzerStart` by compiling the Packet library and the Analyzer. Starting the executable will output

```
usage: ./AnalyzerStart file1.dat.gz2 file2.dat.gz2 inputFile \
resultFile distFile
```

where the `*.dat.bz` files denote the files in NetFlow format, the `inputFile` is the log file obtained from the injection before (was called `trace.xml`), the `resultFile` is the file where the results will be written to and the `distFile` denotes a file with flow attributes to consider. To view a sample file for the distance file have a look at `distFile.sample` in the root directory. A detailed description of those files can be found in Section 3.4.3 about the Analyzer.

C Thesis Schedule

Start	**End**	**Description**
16/09/08	03/10/08	Reading, Design Ideas, Literature Survey
06/10/08	07/11/08	Design (Class Diagrams, Flow Diagrams, ...), File Formats, Libraries (XML-Parser, Socket-Library, NetFlow-Library)
10/11/08	02/01/09	Implementation of Framework, Testing of Implementation, Code Documentation
05/01/09	13/02/09	Implementation changes, Testing, Evaluation of Framework, Writing of Thesis
16/02/09	27/02/09	Further Evaluation, Writing
02/03/09	15/03/09	Additional Time

Table 20: Original Thesis Schedule

The thesis schedule was written at the beginning of the thesis and continually adjusted slightly (Table 20).

D EBNF for the Generator File

Here we give the original EBNF (Program 6) that was used in the program code, although it was rewritten to fit C++ syntax there.

Program 6 EBNF

```
alpha = ”a” - ”Z”
digit = ”0” - ”9”;
integer = digit+;
ipatom = digit {digit} {digit};
ip = ipatom ”.” ipatom ”.” ipatom ”.” ipatom;
var = (alpha | ”_” | ”@”) {alpha | ”_”};
rand = ”rand(” integer ”,” integer ”)”;
rands = ”rands(” integer {”,” integer} ”)”;
packet = rand | rands | ip | var;
factor = packet | integer | (”(” expr ”)”) | (”-” factor) | (”+” factor);
term = factor {(”*” factor) | (”/” factor)};
expr = term (’+’ term) | (’-’ term);
```

E Attack Scripts

The first attack (Code 7) is a legitimate looking connection attempt on a service port that should help to hide the attacker behind legal traffic. If this attack already succeeds, there is no use in employing more elaborate measures.

Program 7 Attack Script for Pattern P1

```
[sequence]
Packets=1

[packet]
Size=160 / @
Time=0 / @+200

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=45188 / @
TCP_Destination_Port=80 / @
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP= / @
IPv4_Source_IP= / @
IPv4_Protocol=6 / @
```

The second program (Program 8) is also a pattern that focuses on undetectability by injecting only 5 packets to random ports.

The third script (Program 9) describes 10 packets with fixed connection endpoints and variable sizes.

Program 8 Attack Script for Pattern P2

```
[sequence]
Packets=5

[packet]
Size=256 / @
Time=5000 / @+200

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=rand(1,65535) / rand(1,65535)
TCP_Destination_Port=rand(1,65535) / rand(1,65535)
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP= / @
IPv4_Source_IP= / @
IPv4_Protocol=6 / @
```

The fourth attack script (Code 10) combines the last two patterns by varying end-points and sizes at the same time to form a more recognizable pattern.

The last pattern (Code 11) is a more aggressive form of P_4. It combines all the methods and forms a highly random and variable pattern where sizes, times and ports are randomized which makes it an unpredictable and rather obvious attack.

Program 9 Attack Script for Pattern P3

```
[sequence]

Packets=10

[packet]
Size=408 / @ + 32
Time=10800 / @+200

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=34119 / @
TCP_Destination_Port=80 / @
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP= / @
IPv4_Source_IP= / @
IPv4_Protocol=6 / @
```

Program 10 Attack Script for Pattern P4

```
[sequence]

Packets=10

[packet]
Size=832 / @ + 32
Time=17600 / @+200

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=rand(1,65535) / rand(1,65535)
TCP_Destination_Port=rand(1,65535) / rand(1,65535)
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP= / @
IPv4_Source_IP= / @
IPv4_Protocol=6 / @
```

Program 11 Attack Script for Pattern P5

```
[sequence]

Packets=50

[packet]
Size=rands(1208, 1224, 1232) / @ + rands(0,8)
Time=29400 / @+rand(150,450)

[header]
Type=Application
Type_Name=none

[header]
Type=Transport
Type_Name=TCP

TCP_Source_Port=rand(1,65535) / rand(1,65535)
TCP_Destination_Port=rand(1,65535) / rand(1,65535)
TCP_Flags=16 / @

[header]
Type=Network
Type_Name=IPv4

IPv4_Destination_IP= / @
IPv4_Source_IP= / @
IPv4_Protocol=6 / @
```

References

[1] Spiros Anotonatos, Demetres Antoniades, Michalis Foukarakis, and Evangelos P. Markatos. On the anonymization and deanonymization of netflow traffic. In *FloCon*, 2008.

[2] John Bethencourt, Jason Franklin, and Mary Vernon. Mapping internet sensors with probe response attacks. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[3] Boost c++ libraries. http://www.boost.org/.

[4] T. Brekne, A. Arnes, and A. Øslebø. Anonymization of IP traffic data: Attacks on two prefix-preserving anonymization schemes and some proposed remedies. In *Workshop on Privacy Enhancing Technologies*, pages 179–196, 2005.

[5] Martin Burkhart, Daniela Brauckhoff, Martin May, and Elisa Boschi. The risk-utility tradeoff for ip address truncation. In *1st ACM Workshop on Network Data Anonymization (NDA)*, October 2008.

[6] Cisco Systems Inc. *NetFlow Services and Applications - White paper.*

[7] S.E. Coull, C.V. Wright, F. Monrose, M.P. Collins, and M.K.Reiter. Playing devil's advocate: Inferring sensitive information from anonymized network traces. In *14th Annual Network and Distributed System Security Symposium*, February 2007.

[8] DoubleClick. http://www.doubleclick.com/.

[9] Jinliang Fan, Jun Xu, Mostafa H. Ammar, and Sue B. Moon. Prefix-preserving IP address anonymization. *Comput. Networks*, 46(2):253–272, 2004.

[10] Apache Foundation. Xercesc xml parser. http://xerces.apache.org/xerces-c.

[11] Martin Harring and Martin Lutken. Doxys. http://www.doxys.dk.

[12] Akamai Technologies Inc. Akamai. http://www.akamai.com.

[13] World internet usage statistics. http://www.internetworldstats.com/stats.htm, Feb 2009.

[14] D. Koukis, Spyros Antonatos, and Kostas G. Anagnostakis. On the privacy risks of publishing anonymized IP network traces. In *Communications and Multimedia Security*, volume 4237 of *Lecture Notes in Computer Science*, pages 22–32. Springer, 2006.

[15] Boost Libraries. Spirit framework. http://spirit.sourceforge.net.

[16] Nsasoft LLC. Http traffic generator for testing web applications. http://www.nsauditor.com/web_tools_utilities/http_traffic_generator.html.

[17] PB Software LLC. Network traffic generator and monitor. http://www.pbsoftware.org/id17.html.

[18] Jeff Nathan. Nemesis. http://www.packetfactory.net/projects/nemesis/.

[19] Network mapper. http://nmap.org/.

[20] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. *SIGCOMM Comput. Commun. Rev.*, 36(1):29–38, 2006.

[21] B. Ribeiro, W. Chen, G. Miklau, and D. Towsley. Analyzing privacy in enterprise packet trace anonymization. In *15th Annual Network and Distributed System Security Symposium (NDSS 08)*, February 2008.

[22] Salvatore Sanfilippo. Hping - active network security tool. http://www.hping.org/.

[23] Adam Slagell, Kiran Lakkaraju, and Katherine Luo. Flaim: A multi-level anonymization framework for computer and network logs. In *20th USENIX Large Installation System Administration Conference (LISA'06)*, 2006.

[24] Adam J. Slagell and William Yurcik. Sharing computer network logs for security and privacy: A motivation for new methodologies of anonymization. *CoRR*, cs.CR/0409005, 2004.

[25] Sun. Javadoc reference. http://java.sun.com/j2se/javadoc/writingdoccomments.

[26] SWITCH. The swiss education and research network. http://www.switch.ch.

[27] Dimitri van Heesch. Doxygen. http://www.stack.nl/ dimitri/doxygen/.

[28] W3C. Document object model. http://www.w3.org/DOM.

[29] W3C. Simple api for xml. http://www.saxproject.org.

[30] W3C. World wide web consortium. http://www.w3.org.

[31] Wikipedia: Covert channels. http://en.wikipedia.org/wiki/Covert_channel.