IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

PACK: Speculative TCP Traffic Redundancy Elimination

Eyal Zohar, Osnat (Ossi) Mokryn and Israel Cidon

CCIT Report #770
July 2010

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #770 July 2010

PACK: Speculative TCP Traffic Redundancy Elimination

Eyal Zohar
Electrical Engineering Department
Technion
eyalzo@tx.technion.ac.il

Osnat (Ossi) Mokryn
School of Computer Science
Tel Aviv College
ossi@mta.ac.il

Israel Cidon
Electrical Engineering Department
Technion
cidon@ee.technion.ac.il

***Abstract*—Eliminating redundant network traffic became an important task with the volume increase of large files and rich media content. Consequently, many commercial traffic redundancy elimination (TRE) middleboxes, were placed at WAN and Internet access points.**

However, recent studies have shown that the majority of traffic redundancy results from end-to-end exchanges. Moreover, the penetration of laptops and smart phones has detached clients from specific access middleboxes. Consequently, there is a rising need for a universal, software based, end-to-end transport level TRE.

As many central services such as email and streaming video may use these new capabilities, it is important to minimize the overhead and latency increase associated with the TRE operations.

We present a novel low latency, low overhead, universal TRE mechanism, termed PACK. PACK is designed as a TCP extension and supports all applications built over TCP.

The main idea for reducing the latency and server overheads is PACK's receiver based speculative operation replacing the common sender based approach. The receiver sends data predictions to the sender which in turn moves to computational expensive actions only when these predictions are correct.

Consequently, PACK is best suited for highly loaded servers. Other benefits of PACK are its low latency and buffering requirements.

I. Introduction

TCP/IP traffic at both intranets and the Internet exhibits a significant amount of redundancy and replication. Traffic redundancy results from common users activities such as repeatedly accessing and modifying the same information (documents, data, web and video), and from communicating and sharing of information among multiple users.

Traffic redundancy at enterprise offices was found to vary between 20% to 60% [1][2][3], exacerbating the need for traffic redundancy elimination (TRE), as a mean to reduce network loads and costs and to speed up communication and applications.

A major progress in handling redundant data, followed several seminal papers that set the foundation of modern TRE [4], [1], [5]. In such techniques, the sender and the receiver compare signatures of data chunks, parsed according to the data content, prior to their transmission, possibly eliminating their transmission.

In [5] files are divided into chunks using the method described in [4], by computing a 48-byte Rabin fingerprint [6] for each byte and defining as the beginning of the chunk the byte for which the 12 least significant bits take a predefined value. A SHA-1 signature is calculated for each chunk. These TRE algorithms require that both sides perform file indexing and signature calculations.

Subsequently, TRE techniques have been explored at both the industry and the academic community to eliminate the redundant content and to significantly improve the network speed and efficiency. Most commercial TRE solutions involve the deployment of two or more middleboxes at both the intranet entry points of data centers and branch offices, eliminating repetitive traffic due to sharing, and preventing the exchange of redundant data between clients and data centers. Examples of commercial TRE solutions vendors are Cisco [7], Riverbed [8], Quantum [9], Juniper [10], Bluecoat [11], Expand Networks [12] and F5 [13].

A recent work [2] reports that in an enterprise environment 75%-90% of the redundancy captured by the shared middleboxes stemmed from an end-to-end redundancy (i.e. data redundancy in the traffic exchanged by a single server and a single client). Consequently, it is claimed that an end-to-end (software based) TRE solution can achieve most of the bandwidth savings of the TRE middleboxes, and offer higher savings for small to medium organizations, home offices and end users. The main advantages of an end-to-end solution are the reduction of cost, space and maintenance due to the elimination of the middlebox hardware. It also enables the freedom to move from incompatible proprietary vendor solutions to a standard protocol stack, and to cope with end-to-end encryption methods (e.g., IPsec). Finally, the vast use of mobile Wi-Fi laptops and smart phones eliminate the association between clients and middleboxes. Therefore, it becomes evident that an application transparent, end-to-end, standard TRE can benefit the emerging data intensive services and applications.

We believe that an end to end, software based TRE should meet the following desirable properties:

1) Standard: The TRE standard needs to work across all server and client platforms and operating systems. This will enable servers to reduce redundant traffic, regardless of the client nature and location.
2) Application independent: The TRE should support most applications that transmit redundant information. Similar data may be observed across different applications, e.g., mail attachments may repeat data transmitted by the file system, FTP or web browsing. This calls for the

implementation of a standard TRE at the transport layer.

3) Stateless: Servers should be able to perform equally well for both persistent and casual clients. Note that most middlebox solutions assume that the server side middlebox is aware of the state of the client side middleboxes.
4) High server performance: The additional TRE computations should minimize the performance impair of servers. In particular, it should limit the size of a TRE specific buffering and the amount of processing overheads due to expensive lookups and data computations. For smooth integration of TRE standards, the TRE solution itself should not become the server side bottleneck [3].
5) Minimum impact on end-to-end latency: TRE protocols introduce additional traffic latencies even when TRE is not in effect. The standard TRE should minimize additional latencies.

In this paper we introduce a low latency, low overhead universal TRE mechanism, termed PACK. PACK operates at the transport layer and is designed as a TCP extension. Therefore, PACK supports, on the fly, all applications built over TCP, such as web, mail and video streaming.

The PACK unique characteristic is that it is designed to reduce the additional computational effort placed on servers by TRE, leveraging the untapped computational power, buffering and relative low utilization of clients. Since most data is sent from the server to the clients, PACK is designed as the first *receiver driven* TRE. In normal operation, the sender does not buffer, index or calculate signatures and in contrast to existing TRE schemes, is not required to wait or negotiate prior to data sending. In turn, the receiver, upon receiving, parsing and signing the data, determines if it might be a preamble to an already known byte stream. If so, the receiver sends to the sender a sequence of expected chunk signatures, accompanied with an easy to verify hint for each chunk in the predictive chain. The sender moves into a computational intensive TRE mode only after it validates, using the hint, that in high probability the data already exist at the receiver.

PACK is best suited for loaded servers as it does not require them to buffer or perform calculations over the data until the receiver indicates that it identifies a clear case for TRE. In addition, unless it is desired for extra traffic saving, PACK does not force a negotiation prior to transmission and therefore keeps the end-to-end latency intact. Finally, PACK reduces the amount of the required TCP buffers at the sender compared to a sender based TRE. In order to allow for a maximum redundancy elimination, our design also enables the ”server driven” operation that can be operated at the choice of the sender.

The speculative operation of PACK can be very beneficial when the receiver based predictions identify the majority of redundant chunks. This happen when relative long sequences of redundant chunks (e.g. repeated data) exist. Our experimental study shows that such a phenomenon is evident in email data. Hence, PACK can decrease significantly user upload time, and improve the end user experience for web based mail services such as Google and Yahoo. Clearly, rich content such as streamed video, movies and audio also show similar sequence characteristics.

The paper is structured as follows. Related work is described in Section II. PACK is described in details in Section III. In Section IV we describe several PACK enhancements that better utilize its capabilities. Evaluations and simulation results are presented in Section V.

II. Related Work

Several TRE techniques have been explored in recent years. The first sender based TRE was proposed by Spring and Wetherall in [1]. They introduced a packet-level TRE, utilizing algorithms first presented by [4].

Several commercial TRE solutions described in [7] and [8], have combined the sender-based TRE ideas of [1] with the algorithmic and implementation approach of [5] along with protocol specific optimizations.

[14] presents a redundancy-aware routing algorithm. It assumes that the routers have a cache mechanism, and that they systematically search for routes that can make a better use of the cached data. The implementation is based on [1] with several adaptations for the speed and limited memory size of routers.

A subsequent redundancy-aware routing work [15] considers the hop by hop approach and offers an alternative network layer approach that identifies a caching location along a path for each packet.

A large scale study of real-life traffic redundancy is presented in [16] and [2]. In the latter, packet-level TRE techniques are compared [4], [17]. One of their main finding is that *”an end to end redundancy elimination solution, could obtain most of the middlebox’s bandwidth savings”*, motivating the benefit of low cost software end-to-end solutions.

To the best of our knowledge none of the previous works have addressed the requirements for a server friendly, end-to-end TRE solved by PACK.

III. The PACK Algorithm

For the sake of clarity, we first describe the basic version of the PACK protocol. Several optimizations and improvements are introduced in IV.

PACK is a receiver oriented TRE solution implemented as a TCP extension. The stream of data received at the PACK receiver is parsed to a sequence of variable size, content based signed chunks according to [4][5].

The chunks are then compared to the receiver local storage. If a matching chunk is found, the receiver identifies the local data sequence, termed a *chain* of chunks, to which the local matched chunk belongs, and sends a prediction to the sender for the following data in that sequence. The prediction consists of a byte by byte XOR checksum over the predicted data, termed a *hint*, and the SHA-1 signature of the data. The sender identifies the predicted range in its buffered data, and also calculates a checksum over that range. If the result matches the received hint, it continues to perform the more computational intensive SHA-1 operation. Upon a signature match, the sender sends a

confirmation message to the receiver, enabling it to copy the matched data from its local storage.

The PACK receiver maintains a large persistent chunk and chunk signature caches, employing caching and indexing techniques to efficiently manage and access the stored chunks. In addition, it maintains a chain store to keep the order by which the various chunks were last received. For example, consecutive chunks that belong to a particular video will be indexed at that order in the chain store. When new data is received and parsed to chunks, the receiver derives the chunk's signature using SHA-1. At this point the chunk and chunk signature caches and the chain stores are updated.

To enable PACK, both TCP parties need to set the PACK enable option during the TCP 3-way initialization handshake. For details, see Section III-D.

A. From Anchors to Chains

PACK parsing procedure is similar to [4][18] and its chunk store structure is similar to [5]. The chunk store holds pointers to the data chunks[1], and a list of per chunk properties such as the chunk size, a chunk hint (a simple byte by byte checksum) and the chunk signature (SHA-1). In addition, PACK uses a new chunk *chains* scheme, described in Fig. 1, in which data indexes in the disk are kept in a chain store. The chain store includes a linked list of indexes to previously observed chunks according to their original order. When a duplicate chunk is received, PACK identifies the chain to which a received chunk belongs, and uses this particular chain to predict future chunks.

B. The PACK Receiver Algorithm

Upon the arrival of new data, the receiver computes the chunks and their respective SHA-1 signatures. Then it looks for a match in the chunk signature cache. If one is found, it examines the chain store to find the following chunks in its database. If such a chain is found, the receiver sends a prediction, consisting of the following chunks in the chain, to the sender. Each prediction message carries the prediction starting point in the byte stream (i.e., offset) and several following chunks as explained later.

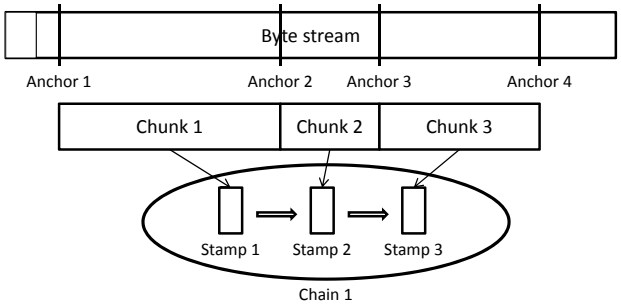


Fig. 1: From anchors to chunks and chain

[1]To prevent excessive disk accesses, deduplication is implemented and used for sharing data between the TCP receiver and its applications [19][20].

Upon a successful prediction, the sender responds with a PRED-ACK confirmation message. The PRED-ACK is carried within an empty (no payload) TCP segment. Once the PRED-ACK message is received and processed, the receiver copies the data from the chunk store to its TCP input buffers, placing it according to the given sequence numbers. At this point, the receiver sends a normal TCP ACK with the next expected TCP sequence number. In the case the prediction is incorrect, the sender does not send a PRED-ACK message but continues with a normal TCP operation, sending the raw data. The arriving data is treated as a new data, i.e., the receiver looks for a matching chunk in its chunk signature cache.

Proc. 1 Receiver Segment Processing

```
1. if segment carries payload data then
2.    calculate chunk
3.    if reached chunk boundary then
4.       activate predAttempt()
5.    end if
6. else if PRED-ACK segment then
7.    processPredAck()
8.    activate predAttempt()
9. end if
```

Proc. 2 predAttempt()

```
1. if received chunk matches one in local storage then
2.    if foundChain(chunk) then
3.       calculate prediction PRED
4.       send TCP ACK with PRED
5.       exit
6.    end if
7. else
8.    store chunk
9.    append chunk to current chain
10. end if
11. send TCP ACK only
```

Proc. 3 processPredAck()

```
1. for all  offset ∈ PRED-ACK  do
2.    read data from disk
3.    put data in TCP input buffer
4. end for
```

C. The Sender Algorithm

When a sender receives a PRED message from the receiver, it tries to match each of the chunk predictions to its buffered (yet to be sent) data. Each PRED segment usually contains several prediction offsets. The sender separates the predictions, finds the corresponding data's absolute TCP sequence number, and calculates the single byte XOR checksum over the corresponding data. If it matches the checksum of the prediction, the sender

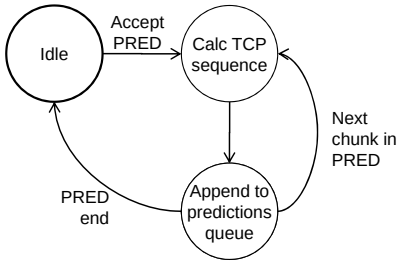


Fig. 2: Sender algorithm part 1/2: filling prediction queue

calculates the more computational intensive SHA-1 signature of the same chunk, and compares the result to the receiver's signature.

One of the key features of PACK is that the sender needs to perform SHA-1 operations only when data redundancy is very likely. When the receiver prediction is incorrect, a hint mismatch occurs with a 0.996 probability. This guarantees that the sender "false positive" (e.g. unnecessarily computing the SHA-1 signature) probability is less than 0.4%. When a signature prediction match occurs, the sender replaces the buffered data with a PRED-ACK signal. Since the receiver holds a list of the predictions it sent, the sender sends in its reply the byte range (denoted by pairs of sequence numbers) for which the prediction is correct. When an acknowledgement is received, the sender frees up the retransmission buffers.

Fig. 2 and Fig. 3 illustrate the sender operation using simple state machines. The state machine in Fig. 2 describes the parsing of a received PRED command. The predictions are processed to calculate each range sequence numbers, and the range boundaries, denoted by the corresponding sequence numbers, are put in a queue. Fig. 3 describes how the sender tries to match its predicted range to its own data. First, it finds out if this data was already sent or not. In case the data was already acknowledged, the corresponding prediction is discarded. Otherwise, it tries to match the data to the data in its outgoing TCP buffers, as described before.

D. The PACK Wired Protocol

Fig. 4 illustrates the way the PACK wired protocol operates, assuming the data is redundant. First, both sides enable the PACK option during the initial TCP handshake by adding a *PACK permitted* flag (denoted by a bold line) to the TCP options field. Then, the sender sends the (redundant) data (in one or more TCP segments) and the receiver identifies that a currently received chunk is identical to an indexed chunk at its chain store. The receiver, in turn, piggybacks a TCP ACK message and sends the prediction in the ACK message option fields. At the last step, the sender sends a confirmation message, PRED-ACK, replacing the actual data.

We suggest to add two new TCP options, similarly to the ones defined in SACK [21]. The first is an enabling option *PACK permitted* sent in a SYN segment to indicate that the PACK option can be used after the connection is established. The

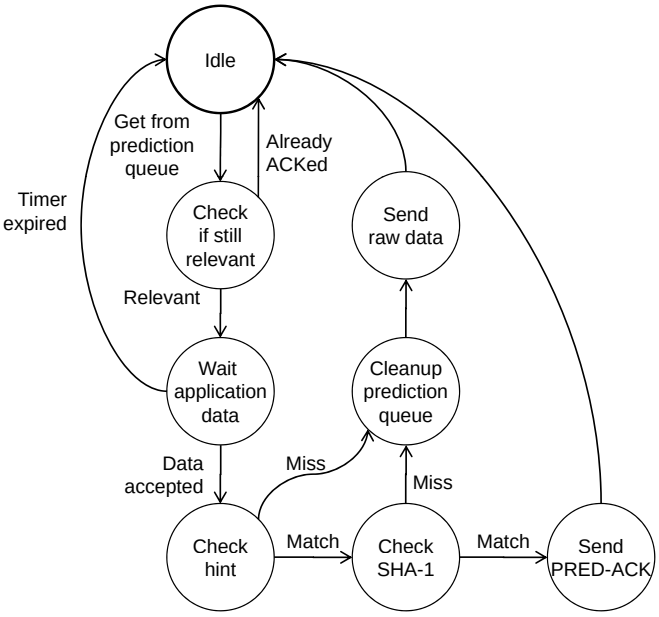


Fig. 3: Sender algorithm part 2/2: processing prediction queue and sending PRED-ACK and/or raw data

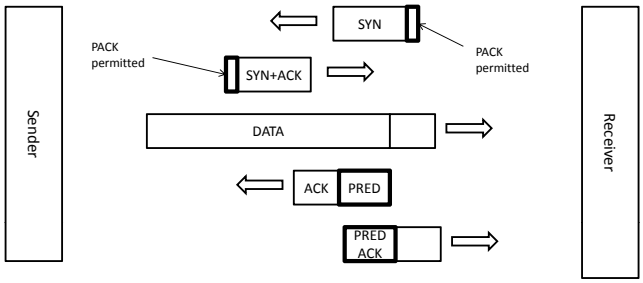


Fig. 4: PACK wired protocol in a nutshell

other is the *PACK message* that may be sent over an established connection once permission has been granted by both parties.

To allow a full backward compatibility to legacy TCP clients, PACK uses exclusively the TCP Options field for its new signaling. The 4-bit Data Offset field that determines the total length of the TCP header yields a usable TCP Options field of up to 40 bytes. Since an unknown option must include a length octet in addition to the option code octet, there are up to 38 bytes left for PACK control information.

A single PACK message is designed to wrap and carry multiple PACK commands. This not only saves message overhead, but also copes with security network devices (e.g. firewall) that tend to change TCP options order [22]. Note, that most TCP options are only used at the TCP initialization period. Several exceptions such as SACK [21] and timestamps [23] exist [22].

We use of the following option codes for PACK initiation during TCP SYN: option 29 (decimal) for PACK permitted (two hex bytes $1D02$) and option 30 for the PACK message. Fig. 6 presents the *PACK message* header format as it appears in the TCP options field.

The receiver always sends together the following two commands. The first command is OFFSET (Fig. 7) that informs the sender where the next prediction starts, relative to the acknowledgement offset of the piggybacked segment[2].

The second command is PRED (Fig. 8), which includes the predictive ACK parameters. Multiple PRED commands can be carried in one TCP segment. The offsets of the chunks predicted by the PRED commands start from the value of OFFSET and then follow each other. Each PRED message carries a set of parameters: the predicted chunk length, a hint per this chunk, and its SHA-1 signature.

The sender uses a single command termed PRED-ACK (Fig. 9). Since the receiver holds a list of sent predictions, the sender only needs to acknowledge a range of TCP sequence to approve some or all of them.

E. PACK Impact on TCP Mechanisms

Some of the built-in TCP mechanisms should be reviewed in light of the PACK protocol and should be slightly modified. In general, we found that it is most convenient to treat the delivery of virtual data (redundant chunks delivered by a reference to the chunk cache) as an independent process that does not influence the mechanisms that are tightly connected with the sending of the real data. Below we explain the key mechanisms that are reviewed in light of PACK.

1) TCP Retransmission: Retransmissions work normally except for one small modification. The data itself may be replaced at any given time with a PRED-ACK, and the transmission contains from that point on only the short virtual data, and not the data itself. As an option, in rare cases of a PRED time race, when the first transmission that contained real data is lost, the sender may replace the data with a PRED-ACK at the following retransmissions since a PRED command for this data was accepted after the initial transmission. This option also helps the sender to free faster its corresponding outgoing memory buffers.

2) Delayed ACKs: Delayed ACKs were introduced to reduce the number of ACKs sent from a receiver by sending accumulated ACKs for a bigger number of data segments. PACK may interfere with this mechanism as the receiver sends a PRED message as soon as possible, maybe before the delayed ACK is normally triggered. However, in practice the delayed ACK mechanism is not relevant to PACK, as it is designed for slow and low rate applications where PACK is less relevant.

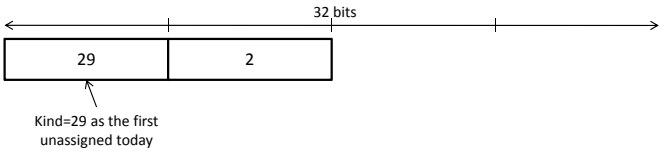


Fig. 5: PACK permitted TCP Option

[2]Another command, OFFSET-NEG, allows overlapping predictions of different sizes (see section IV).

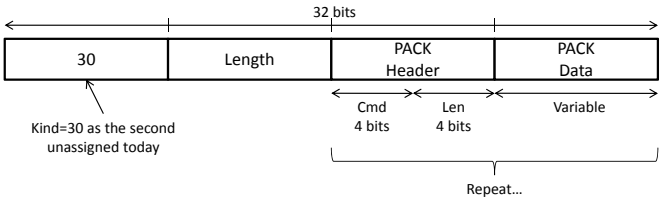


Fig. 6: PACK message header format as a single TCP Option

3) Round-Trip Time Measurements: RTT estimators continue to work normally by considering virtual data as real data. This is because PACK messages travel at the same path as other segments, and both sides do not hold data or acknowledgments more than the normal operation.

4) The Receiver Window: The receiver window value determines the amount of unacknowledged data the sender can send. When the application is fast enough this value reflects the size of the incoming buffer at the receiver. With PACK's virtual data, the sender should ignore the published window restriction and send virtual data that is much larger than the published window size. Therefore, even with the most basic PACK version, the sender treat a received PRED as a permission to virtually send at least the specified amount of virtual data contained in the prediction.

5) Duplicate ACK: In some cases the receiver may want to send independent PRED messages since there is not enough free space in the option field of the ACK messages. Since TCP gives a special treatment to duplicate ACKs, the sender should be able to distinguish between a real duplicated ACK and an ACK that was created to carry new PRED messages, similar to

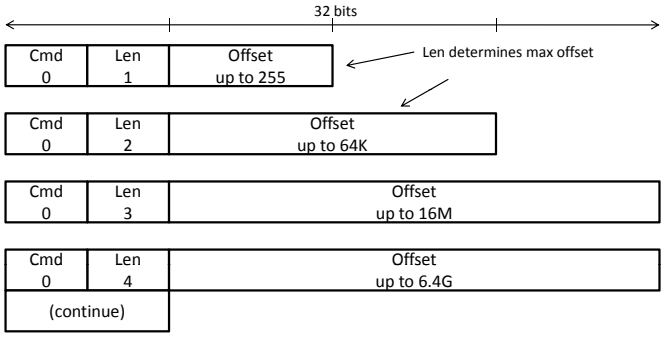


Fig. 7: PACK command, receiver offset: OFFSET

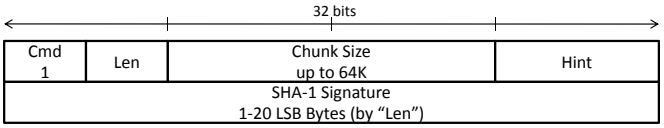


Fig. 8: PACK command, receiver prediction: PRED

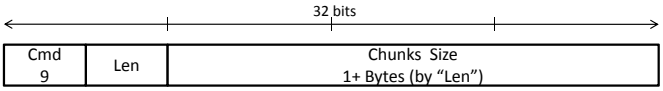


Fig. 9: PACK command, sender ACK: PRED-ACK

the way it treats TCP window updates.

IV. Optimizations

For simplicity and clarity we described in the previous section the most basic version of the PACK protocol. In the following, we describe several enhancements and optimizations that are likely to be included in a TCP based TRE standard.

A. Adaptive Receiver Virtual Window

PACK enables the receiver to locally obtain sender data when such a local copy is available, eliminating the need to send such data over the network. We term the receiver fetching of local data following the reception of sender confirmations to the previously sent receiver predictions as the reception of *virtual data*.

When a large amount of local data is fetched, the receiver expects to receive virtual data at high rates that are limited by the receiver capabilities (storage, CPU, etc.) and not by the network throughput. This in turn means that the receiver predictions and the sender confirmations regarding future virtual data transfers should be expedite as much as possible. For example, in the case of a repetitive success in PACK predictions, it is reasonable for the receiver side PACK algorithm to become greedy and increase the ranges of its predictions.

PACK enables a large prediction size by either sending several successive PACK predictions and/or by using an additional command (PRED-COMB) presented in Fig. 10. In the case of a several successive PACK prediction messages, the sender should recognize the different predictions and refrain from mistakenly recognizing the messages as duplicate acknowledgments. For that end we use a similar mechanism to the one used for handling successive different window advertisements.

Alternatively, The new PRED-COMB command can be used. PRED-COMB enables the receiver to combine several chunks into a single range, as the sender is not bounded to the anchors originally used by the receiver's algorithm. The combined range has a new hint that is a XOR of the chunks' hints, and a new signature that is a SHA-1 of the concatenated content of the chunks. To clarify how all these commands can be put together in practice, we present here an example of such a PACK message in Fig. 11.

Enabling a variable prediction size introduces the notion of a *virtual window*, which is the current receiver's window for virtual data. The virtual window, defined by each prediction, is the number of bytes in the prediction, derived from the specified offset till the end of the predicted range. The virtual window is first set to a minimal value, which is the receiver's flow control

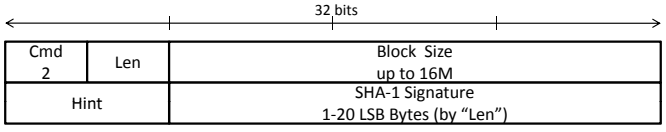


Fig. 10: PACK command, prediction with combined chunks: PRED-COMB

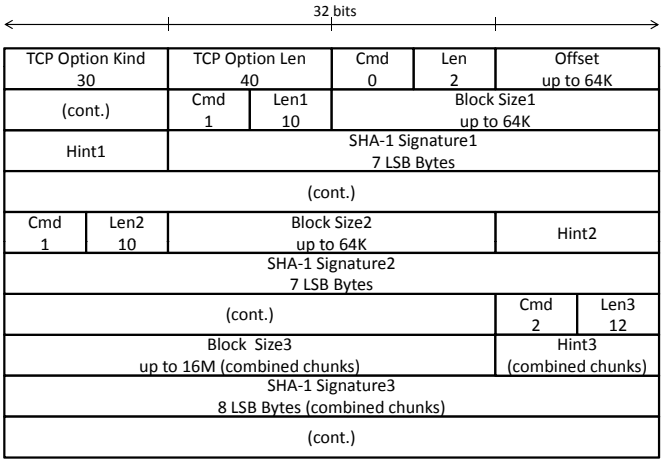


Fig. 11: PACK full message example of a large range prediction, using combined chunks command

window. The receiver increases the virtual window with each prediction success, according to the following description.

Upon a first chunk match, the receiver sends its initial size prediction. Before receiving a PRED-ACK, it is likely that some of the following real data segments that were already sent, arrived at the receiver (the sender may have sent a full window). At this point, the receiver can partially confirm its prediction, and increase its prediction range along the matched chain. For this phase, when the prediction is partially confirmed, PACK increases its prediction range by a moderate linear factor. Upon the reception of successive PRED-ACK confirmations from the sender, PACK becomes more aggressive and increases its prediction range exponentially. This is similar to the slow start part of TCP. When a mismatch occurs, the receiver returns to its initial virtual window size.

Proc. 1 describes the basic algorithm performed at the receiver side. The code starting at line 2 describes PACK behavior when a data segment arrives after a prediction is sent. Each time data that reinforces the prediction is received, the prediction range is increased linearly. Proc. 5 describes the receipt of a successful acknowledgement message (PRED-ACK) from the sender. The receiver reads the data from the local disk. It then modifies the next byte sequence number to the last byte of the redundant data just read plus one, and sends the next TCP ACK, piggybacked with its new prediction. This time, the prediction range is increased to twice the range of the last prediction.

The increase in the prediction range introduces a trade-off in case the predicted chain data is not completely identical to the real data, and the sender sends its original raw data. The code in Proc. 4 line 3 describes the receiver's behavior when new data arrives that does not match its recently sent predictions. This new chunk may of course start a new match. Following the reception of the data, the receiver reverts to a normal TCP mode until a new match is found in the chunk store, and resets the virtual data window range back to the regular window size. Note that even a minor change at the sender data, compared with former saved chain, causes the entire prediction block

range to be sent to the receiver, even if the majority of the data is identical. Hence, using large block sizes at the prediction introduces a tradeoff between the potential rate gain and the recovery effort in the case of a miss-prediction.

Proc. 4 predAttemptAdaptive()

```
1. {obsoletes Proc. 2}
2. if received chunk overlaps recently sent prediction and they
   do not match then
3.    predSizeReset()
4. end if
5. if received chunk matches one in local storage then
6.    predSizeLinear()
7.    if foundChain(chunk) then
8.       calculate prediction PRED according to predSize
9.       send TCP ACK with PRED
10.      exit
11.   end if
12. else
13.   store chunk
14.   append chunk to current chain
15. end if
16. send TCP ACK only
```

Proc. 5 processPredAckAdaptive()

```
1. {obsoletes Proc. 3}
2. for all offset ∈ PRED-ACK do
3.    read data from disk
4.    put data in TCP input buffer
5. end for
6. predSizeExponent()
```

B. A Hybrid Approach

PACK is less efficient for data with the same amount of redundancy if the changes are scattered along the data. This happen since the prediction sequences are interrupted frequently by misses. This in turn forces the sender to revert to raw data transmission until a new match is found at the receiver and reported back to the sender. To that end, we suggest a hybrid PACK approach. When PACK recognizes a pattern of dispersed changes, it may select to trigger a sender driven approach in the spirit of [1][7][8][15].

However, as was explained earlier, we would like to revert to the sender driven mode with a minimal computational and buffering overhead at the server in the steady state. Therefore, our approach is to first evaluate at the receiver the need for a sender driven operation, and report it back to the sender. At this point the sender can decide if it has enough resources to process a sender based TRE for some of its receivers. To support this enhancement, an additional command (DISPER) is introduced (Fig.12). Using this command the receiver sends periodically its estimated level of dispersement, ranging from 0 for long smooth chains, up to 255.

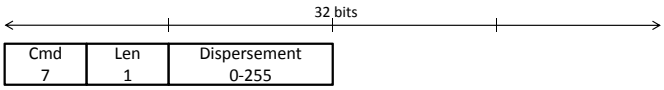


Fig. 12: PACK command, level of dispersement: DISPER

A good way to compute the data dispersement parameter over time is by an exponential smoothing function, as follows:

$$D \leftarrow \alpha D + (1-\alpha)M \tag{1}$$

Where α is a smoothing factor with recommended value of 0.9 or more. The value M equals 255 when a chain break is detected and 0 otherwise. We evaluate the data dispersement value of several data sets in Section V-E.

Proc. 6 Receiver Segment Processing Hybrid

```
1. {obsoletes Proc. 1}
2. if segment carries payload data then
3.    calculate chunk
4.    if reached chunk boundary then
5.       activate predAttempt()
6.       {new code for Hybrid}
7.       if detected broken chain then
8.          calcDispersement(255)
9.       else
10.         calcDispersement(0)
11.      end if
12.   end if
13. else if PRED-ACK segment then
14.   processPredAck()
15.   activate predAttempt()
16. end if
```

Proc. 7 processPredAckHybrid()

```
1. {obsoletes Proc. 3}
2. for all offset ∈ PRED-ACK do
3.    read data from disk
4.    put data in TCP input buffer
5.    {new code for Hybrid}
6.    for all chunk ∈ offset do
7.       calcDispersement(0)
8.    end for
9. end for
```

V. Experiments

In this section we present TRE experiments using the algorithms and protocols described above. We focus on several content examples that demonstrate the ideas behind PACK and the different trade-offs involved in speculative vs. sender based implementations. Our experiments are based on the three following data collections:

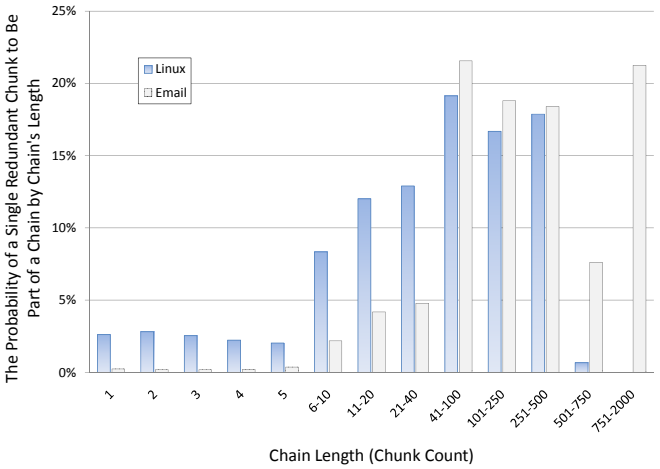


Fig. 13: Chain length histogram in two 1 GB collections: Linux (40 kernel sources) and Email1 (1,140 email messages)

1) Linux - HTTP download of the entire Linux 2.0.x kernel source code collection (the entire 40 files 1 GB). Contains 77% chunk redundancy.
2) Email1 - Download of 1,140 email messages (1.1 GB) one at a time, over IMAP. Contains 46% redundancy.
3) Email2 - A different email account with 2.5 GB and 26% redundancy.

A. Redundant Chain Length

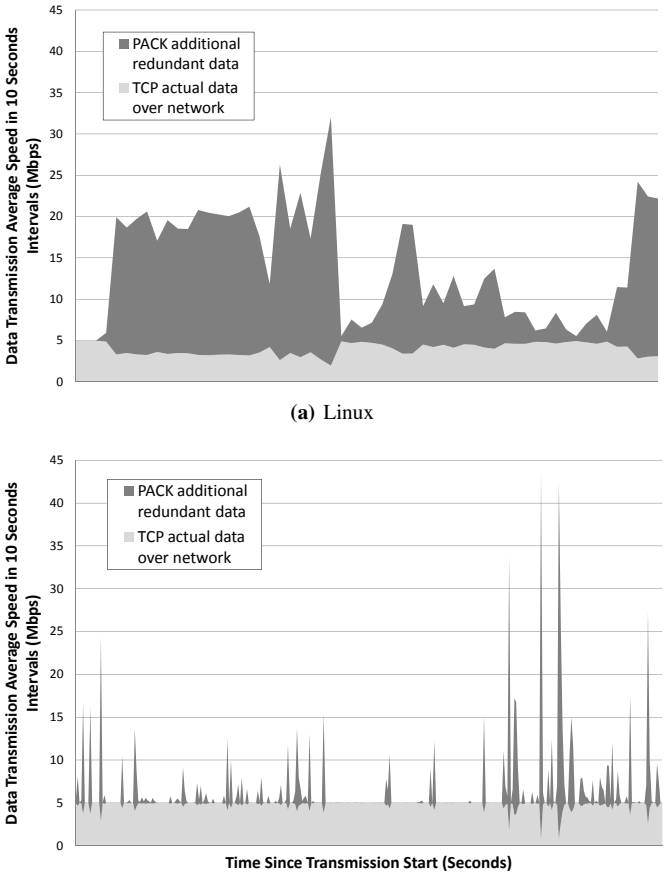
PACK works better when redundancy is correlated and chains are long, especially when end-to-end communication delays are not negligible. We expect to find long chains in the retransmissions of unmodified or slightly modified large files. Fig. 13 presents the chain length distribution of the Linux and Email1 data collections. Indeed, in Email1 about 88% of all redundant chunks are found in chains of length 40 chunks and over, compared to only 54% of the redundant chunks in Linux. It is considerable to assume that PACK will yield better results for Email1 than for Linux, and it may also suggest that mail servers would benefit greatly from a TRE solution.

B. Download Speed Over Time

In the following experiments we assume that the communication speed between the sender and the receiver is limited by both the network and the end host local copy operations speed limit (we assume it is faster than the network speed). If TRE is not utilized, the communication rate is governed only by the network bandwidth. However, TRE virtually ”increases” the network speed and thus the machines speed becomes a bottleneck.

Our experiments assume 50 Mbps machines connected over 5 Mbps line. According to our measurements, PACK is capable of performing at much more than x10 speedup by using TRE, but given that in practice one server serves many receivers we assume in this experiment a maximal speed that the server is willing to grant for a given receiver.

Fig. 14a describes the PACK transmission performance for Linux over time. The graph represents the average sampled



(a) Linux

(b) Email2

Fig. 14: Amount of data delivered to the application at the receiver side during 10 seconds intervals

speed in which data is delivered to the application at the receiver side during 10 seconds intervals. Therefore the shadowed area under the graph represents the accumulated amount of data delivered to the application at any given time. The lower (light gray) area is the actual data that was transmitted over the network and the upper (dark gray) area is the additional virtual data loaded to the application from the receiver's local chunk store. Clearly, the high redundancy of Linux (over 76%), exploited by PACK, speeds up the communication. It is interesting to see that in all the intervals the speed rarely goes beyond 20 Mbps, despite the ability of the receiver to operate at 50 Mbps. This results from the fact that matching chains are too short to increase the rate of ”virtual” transmissions for a long time.

Fig. 14b demonstrates the operation of PACK with less redundancy but longer chains. This experiment delivers Email2 (2.5 GB Email account with 26% redundancy). The differences between the two cases (a) and (b), stem from the differences between the data redundancy and chain lengths.

C. Sender Effort: Receiver-Driven vs. Sender-Driven

PACK is a receiver driven TRE algorithm that places most of the computational burden on the receiver. This approach is optimized for the typical Web scenario of a heavily loaded

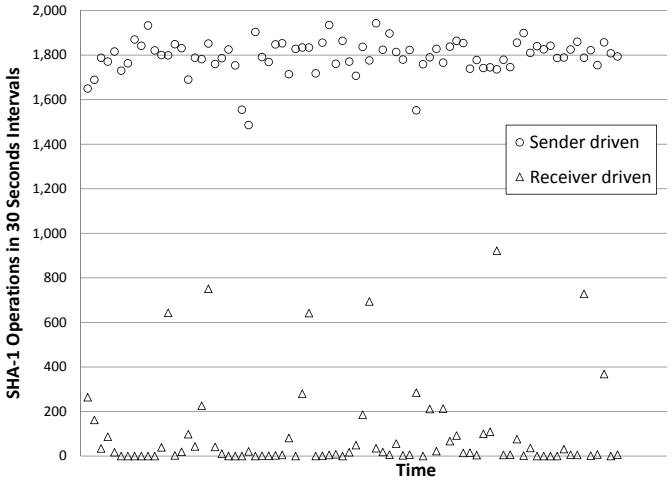


Fig. 15: Difference in computation efforts between receiver and sender driven modes for the transmission of Email1

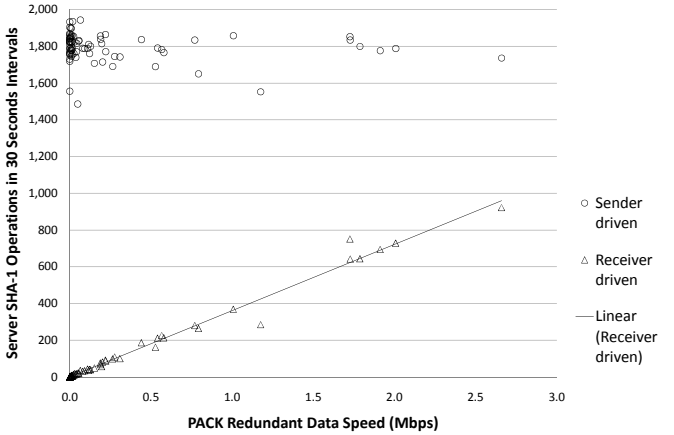


Fig. 16: Rearrangement of Fig. 15 using a redundancy speed axis that expresses the virtual speed

server that needs to send traffic concurrently to a large number of clients.

In the following experiment we repeat the experiment of section V-B only now we account for the computational effort of the SHA-1 operations at the sender's side. In the receiver driven mode the sender is required to perform a SHA-1 operation over a defined range of bytes (the prediction determines a starting point, i.e., offset, and the size of the prediction) only after it verifies the hint sent in the prediction matches its data (a 1-byte XOR over the predicted range). In a sender driven TRE solutions the sender is required to first compute Rabin fingerprint to slice the stream into chunks and then to compute a SHA-1 signature for each chunk prior to sending it.

Fig. 15 compares the difference in computation efforts between these two modes for the transmission of Email1, accumulating operations over time intervals of 30 seconds. The sender-driven mode constantly consumes processing power while the receiver-driven mode has a low computational demand with occasional peaks. Fig. 16 reveals the nature of the receiver-driven mode computational peaks. The figure draws the results of the same experiment using a redundancy speed axis that expresses the virtual speed achieved with a TRE of x10 speedup.

This clearly demonstrates that the sender-driven solution requires a high computational effort irrespective of the redundancy while the receiver-based solution triggers the sender's operations only when it is beneficial. Note, that these results are very dramatic for all sender-receiver sessions that contain no redundancy at all. Such sessions may be the majority of the server-client sessions and therefore may significantly impair the server throughput for the sender-driven mode.

D. RTT Impact: Receiver-Driven vs. Sender-Driven

Sender-driven TREs ([5][8][7]) are designed to optimize network bandwidth. They eliminate the transmission of any redundant chunk by sending the receiver a list of all future chunk signatures and waiting for acknowledgements before the data itself can be sent. PACK on the other hand does not delay sender transmissions and therefore is expected to send more data than sender-driven TREs but achieve a shorter response time to data requests. Since the virtual data speed is higher than the network bandwidth, the total download time presents a tradeoff between the amount of redundancy elimination and the sender response time. This tradeoff depends on network and virtual transmission speeds, round-trip time, number of transmitted file, their size, the redundancy level and PACK redundant chunk chain lengths.

The experiment presented in Fig.17 delivers Email1 (1,140 emails and 46.6% redundancy), over a single TCP connection, between two machines similar to the experiments presented above. The horizontal axis presents the amount of data that arrives to the receiver, assuming that initially the receiver has an empty chunk store. At the beginning PACK has a clear download time advantage over the sender-driven TRE, as the sender-driven spends at least one extra RTT for each downloaded email while there is hardly any redundancy at this stage. When the chunk store fills up the redundancy level gets higher. With large RTT and short chains PACK misses some redundant chunks that are already sent before the receiver predication reached the sender. This lowers the download time difference, and in some of the sampled intervals the sender-driven TRE performs faster.

E. Dispersement Level

Fig. 18 compares the difference in dispersement levels for the transmission of two collections. It plots the values sent by the receiver every 500 chunks, using the DISPER command.

Clearly Linux has higher dispersement levels most of the time because it has more frequent chain breaks than Email2. For example, assume that a single sender serves concurrently two receivers that report two dispersement levels similar to Fig. 18. In case the server can afford only one concurrent sender-based TRE operation (due to limited computational resources or buffers), it should prefer the receiver with the Linux download profile.

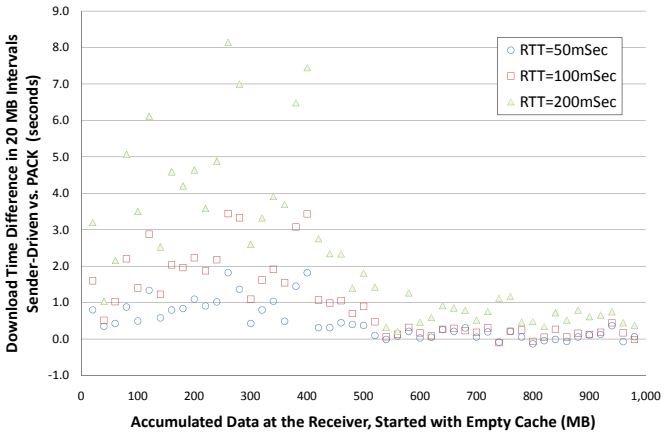


Fig. 17: Sender-driven vs. PACK download time of Email1 (1 GB 1,140 emails with 46.6% redundancy)

VI. Summary

In the last years, Internet and intranet traffic has been evolved to the shipment of large application files and rich data content. This shift has motivated the development and deployment of proprietary, middlebox based TRE solutions that address the need of large corporations. Clearly, the same traffic characteristic trends continue to dominate the new generation of mobile and wireless networks. As most of the data redundancy already happens at the end-to-end exchange, the need for a universal, standard, software based TRE is evident.

In this work we have presented PACK, a universal solution for a server friendly, end-to-end TRE which is based on novel speculative principles that place the majority of the TRE computational burden on the receiving client. We have chosen to implement it as a TCP extension, making it both easy to adopt as a standard and transparent to the majority of applications and to IPsec encryption. Our evaluation has shown that PACK meets the expected design goals and has clear advantages over sender based TRE when the server effort and buffering requirements are important. Several additional simple enhancements of PACK makes it the best of both worlds solution for an end-to-end TRE standard.

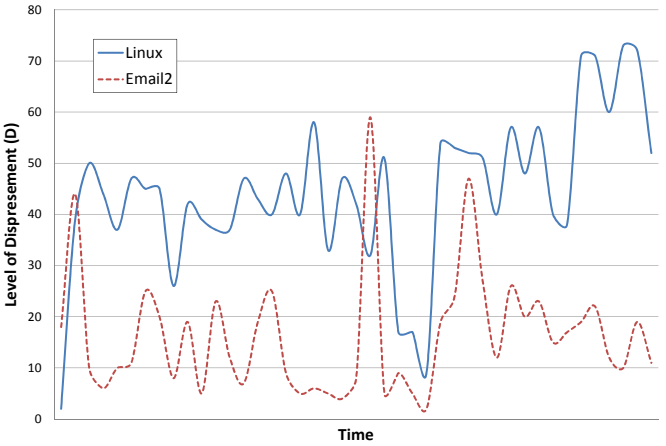


Fig. 18: Dispersement level in two collections: Linux (40 kernel sources) and Email2 (2.5 GB email messages)

References

[1] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 87–95, 2000.

[2] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: Findings and implications," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. ACM New York, NY, USA, 2009, pp. 37–48.

[3] M. Martynov, "Challenges for high-speed protocol-independent redundancy eliminating systems," *Computer Communications and Networks, International Conference on*, pp. 1–6, 2009.

[4] U. Manber, "Finding similar files in a large file system," in *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1994, pp. 2–2.

[5] A. Muthitacharoen, B. Chen, and D. Mazi'eres, "A low-bandwidth network file system," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2001, pp. 174–187.

[6] M. O. Rabin, "Fingerprinting by random polynomials," Harvard University, Tech. Rep. TR-15-81, 1981.

[7] E. Lev-Ran, I. Cidon, and I. Z. Ben-Shaul, "Method and Apparatus for Reducing Network Traffic over Low Bandwidth Links," Patent US 7 636 767, November 2009, filed: November 2005.

[8] S. Mccanne and M. Demmer, "Content-Based Segmentation Scheme for Data Compression in Storage and Transmission Including Hierarchical Segment Representation," Patent US 6 828 925, December 2004, filed: December 2003.

[9] R. Williams, "Method for Partitioning a Block of Data Into Subblocks and for Storing and Communcating Such Subblocks," Patent US 5 990 810, November 1999, filed: August 1996.

[10] Juniper Networks, "http://www.juniper.net/."

[11] BlueCoat, "http://www.bluecoat.com/."

[12] Expand Networks, "http://www.expand.com/."

[13] F5: WAN Optimization, "http://www.f5.com/solutions/acceleration/wan-optimization/."

[14] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: The implications of universal redundant traffic elimination," in *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. New York, NY, USA: ACM, 2008, pp. 219–230.

[15] A. Anand, V. Sekar, and A. Akella, "SmartRE: an Architecture for Coordinated Network-Wide Redundancy Elimination," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 87–98, 2009.

[16] A. Gupta, A. Akella, S. Seshan, S. Shenker, and J. Wang, "Understanding and Exploiting Network Traffic Redundancy," UW-Madison CS technical report 1592, Tech. Rep., 2007.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2003, pp. 76–85.

[18] A. Z. Broder, "Some Applications of Rabins Fingerprinting Method," in *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag, 1993, pp. 143–152.

[19] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur, "Single Instance Storage in Windows® 2000," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium*. USENIX Association, 2000, pp. 13–24.

[20] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey, "Redundancy Elimination Within Large Collections of Files," in *USENIX Annual Technical Conference, General Track*, 2004, pp. 59–72.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," *RFC 2018*, 1996.

[22] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, 2005.

[23] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," *RFC 1323*, 1992.