Practical Cryptography for a Peer-to-Peer Web Browsing System

by

Alexander Pokluda

CS758
Cryptography and Network Security
Research Project

Waterloo, Ontario, Canada, August 2012

Abstract

This report investigates how cryptography is used in real-world peer-to-peer systems. The cryptographic protocols and schemes Skype and BitTorrent are examined and the security and privacy guarantees that these systems attempt to give their users is analyzed. Several weaknesses in Message Stream Encryption (MSE), the custom protocol used by BitTorrent, are identified. Three specific attacks on MSE are described in detail. One of the attacks describes how a trivial intruder-in-the-middle attack can recover the identity of content being shared in an encrypted BitTorrent session when the torrent info hash is publicly available. An example C++ program is provided that can help execute such an attack.

The security requirements of a new peer-to-peer Web browsing system called pWeb are also identified and and several cryptographic protocols and schemes were selected as suitable for pWeb. Well studied and widely used protocols were chosen in light of the problems identified with Message Stream Encryption. Finally, the performance of the AES block cipher and SHA-256 and BLAKE-256 hash functions is analyzed and SHA-256 is recommended for use in pWeb over BLAKE-256.

Preface

I started graduate studies part-time at Waterloo in pursuit of my Master of Mathematics degree in January 2012. Shortly thereafter I started working with a group of researchers including Raouf Boutaba, Reaz Ahmed, Faizul Bari, Rakib Rakibul and Shihabur Chowdhury on a Peer-to-Peer Web Browsing System called pWeb. The group had published a number of papers on various aspects of the system before I joined covering topics such as distributed hash table routing [5]; content naming [7]; guaranteed content availability [24]; and efficient distributed search and indexing [1], [4], [3], [2]. These papers discuss high level design and theories of operation. We have a lot of good ideas but no working system. One of the tasks that I am working on now is helping to integrate the many ideas and bring together a working implementation.

One thing that the group has not discussed up to this point is *cryptography and security*. Clearly any publicly accessible peer-to-peer system must provide some level of security to prevent abuse by malicious users and this can be achieved in part by the judicious use of cryptography. One of the many advantages of peer-to-peer systems is that their distributed architecture makes them more resistant to censorship; however, architecture alone cannot provide strong guarantees of security. Here too cryptography can be used to enhance the security of the system.

I have been interested in cryptography and network security for quite some time, but this course has been by first formal exposure to the subject. I was excited and glad that I had an opportunity to immediately apply what I had learnt in a meaningful way. Everything presented in this report is new work that I have done personally. I hope to be able to incorporate at least some of the ideas presented here into my master's thesis that will focus on implementing a working system and dynamic page generation in a peer-to-peer web hosting environment.

Alexander Pokluda
August 2012

Table of Contents

List of Tables

List of Figures

List of Protocols

1 Introduction

The aim of this project is to identify all of the instances where cryptography would be desirable or necessary to prevent abuse in a peer-to-peer Web browsing system, secure a basic level of privacy for the system's users, and evaluate the merits of various solutions in the pWeb context. This project makes a number of contributions to the field of peer-to-peer network security in pursuit of this goal.

The first part of this report examines the protocols and schemes used in arguably the most successful peer-to-peer systems ever created: Skype and BitTorrent. The design of Message Stream Encryption (MSE), which is used to protect the BitTorrent Protocol, is reverse engineered to discover the factors that likely lead to its final form. This section also looks at the privacy guarantees that these systems attempt provide to their users and flaws in their design and implementation. Several specific attacks on Message Stream Encryption are presented and a method of obtaining the identity of content being shared after the third flow of the MSE handshake is demonstrated.

The second part of this report will determine the general security related requirements for a peer-to-peer Web browsing system and identify the cryptographic protocols that meet those security requirements. One or two schemes implementing each cryptographic protocol will be selected and each scheme will be evaluated based on its level of security and performance. Finally, the best implementation for each protocol will be chosen and recommended for use in pWeb.

2 Cryptography in Real-World Peer-to-Peer Systems

In this section we look at two real-world peer-to-peer systems: Skype and BitTorrent. Skype is a popular Internet telephony and video conferencing application. In 2010, Skype reported that they had 663 million registered users worldwide [26], and currently claim that there are as many as 40 million users online simultaneously at peak times [27]. BitTorrent is a content distribution or file sharing application that enables individuals or organizations with limited computing and network resources to quickly distribute large files to virtually limitless numbers of end users. As of February 2009, BitTorrent accounted for approximately 43 to 70 percent of all Internet traffic worldwide depending on geographic location [23]. The Pirate Bay, one of many BitTorrent directories, is among the most visited sites on the World Wide Web [29]. Both Skype and BitTorrent provide protection from a passive adversary but do not provide protection from an active adversary.

2.1 Skype

Skype uses a hybrid peer-to-peer and client-server architecture. User names and passwords are are stored on the Skype owned and controlled login servers. The login server does not take part in the peer-to-peer network but is used to ensure that login names are unique across the Skype namespace and manage account information . Apart from the login server, there is no central server in the Skype network. User status information as well as user search queries are performed are distributed over the peer-to-peer network overlay [8].

Skype is proprietary, closed-source software and all network traffic is encrypted. There have been some efforts to document the Skype peer-to-peer architecture, but not much is known about the inner workings of the Skype software. Fortunately, the Skype FAQ reveals some information about how encryption is used:[1]

> Skype uses AES (Advanced Encryption Standard)–also known as Rijndel–which is also used by US Government organizations to protect sensitive information. Skype uses 256-bit encryption, which has a total of 1.1×10^{77} possible keys, in order to actively encrypt the data in each Skype call or instant message. Skype uses 1536 to 2048 bit RSA to negotiate symmetric AES keys. User public keys are certified by Skype server at login.

This description tells us that Skype is using a public key infrastructure with the Skype login server acting as a *TA*. A key agreement scheme is used to negotiate session keys for use with a block cipher to secure communications between nodes in the peer-to-peer network. The protocols used here are also used in many client-server applications. The protocols and schemes used in Skype are summarized in Table 1.

[1]The FAQ page is available at [28], but the wording has been updated and some details have been removed. The quote given here is from an older version of the webpage that is cited in [8].

Protocol	Scheme
Key Agreement	RSA with 1536- to 2048-bit key lengths
Block Cipher	256-bit AES
Public-Key Infrastructure	The Skype login server performs the role of *TA* and certifies user public keys

Table 1: Cryptographic protocols and schemes used in the Skype peer-to-peer client software.

2.2 BitTorrent

Message stream encryption (MSE), also known as Protocol Encryption (PE) or Protocol Header Encryption (PHE), is an encryption scheme implemented in most modern BitTorrent clients [35]. The first version of MSE was implemented unilaterally in Azureus, a BitTorrent client released under the GNU General Public License, on January 19 2006. This first version was heavily criticized because it lacked several key features. Over the next few days, the developers of different BitTorrent client applications designed and implemented an improved version of MSE.

The design of Message Stream Encryption was carried out informally between developers of the various BitTorrent clients in mailing lists and source code changes, many of which are no longer available. The Message Stream Encryption Protocol specification is available on the Vuze website [17] along with a short summary of the protocol objectives [32]. Unfortunately, a detailed justification for the design of the protocol and an analysis of the level of security provided is not given. The MSE developers' primary goal was to avoid passive protocol identification and traffic shaping by Internet Service Providers. The MSE protocol specification states:

> The [Message Stream Encryption] protocol describes a transparent wrapper for bidirectional data streams (e.g. TCP transports) that prevents passive eavesdropping and thus protocol or content identification.
>
> It is also designed to provide limited protection against active MITM attacks and port-scanning by requiring a weak shared secret to complete the handshake.

Protocol	**Scheme**
Key Agreement	Diffie-Hellman with 768-bit key lengths
Stream Cipher	RC4
Public-Key Infrastructure	None; New public keys are generated for each session
Hash Functions	Content is located using `.torrent` metainfo files containing an index of data chucks needed to reconstruct a file or set of files and their SHA-1 hash values; A metainfo file itself is identified by the SHA-1 hash of the index (known as an *info hash*)

Table 2: Most current BitTorrent clients use a custom encryption scheme known as *Message Stream Encryption* (MSE). Message stream encryption makes use of the above cryptographic protocols and schemes.

> You should note that the major design goal was payload and protocol obfuscation, not peer authentication and data integrity verification. Thus it does not offer protection against adversaries which already know the necessary data to establish connections (that is IP/Port/Shared Secret/Payload protocol).

Although the stated goal is "payload and protocol obfuscation not peer authentication and data integrity verification" most BitTorrent clients do not indicate the level of security offered by MSE. As mentioned previously, BitTorrent accounts for approximately 43 to 70 percent of Internet traffic worldwide and most of this traffic is protected using MSE. End users understandably expect data confidentiality and authentication from their client software; thus, MSE is an important protocol to study. In the next section we will look at some of the strengths and weaknesses of MSE, but first we examine the operation of the protocol.

Message Stream Encryption is presented in Protocol 1. (The *crypto_select* and *crypto_provide* option fields and additional optional padding have been omitted for clarity). MSE was intended for use with BitTorrent, but can also be used as a general encryption scheme for any application employing bidirectional data streams. It makes use of the Diffie-Hellman key agreement protocol with 768-bit key lengths to establish session keys; the RC4 stream

Protocol 1 MESSAGE STREAM ENCRYPTION

The public domain parameters consist of a published 768-bit safe prime p[a] and a generator element G defined to be 2.

1. A chooses a random integer between 128 and 180 bits long. Then she computes
$$Y_A = G^{r_A} \mod p$$
and sends Y_A and 0 to 512 bytes of random data denoted $PadA$ to B.

2. B chooses a random integer between 128 and 180 bits long. Then he computes
$$Y_B = G^{r_B} \mod p$$
and sends Y_B and 0 to 512 bytes of random data denoted $PadB$ to A.

3. A computes
$$S = Y_B^{r_A} \mod p,$$
and B computes
$$S = Y_A^{r_B} \mod p.$$
They each compute
$$K_A = H(\text{`keyA'} \mid\mid S \mid\mid T_{info\ hash})$$
and
$$K_B = H(\text{`keyB'} \mid\mid S \mid\mid T_{info\ hash})$$
where $H()$ is the *SHA-1* hash function and $T_{info\ hash}$ is a pre-shared secret. A then sends $H(\text{`req1'} \mid\mid S)$, $H(\text{`req2'} \mid\mid T_{info\ hash}) \oplus H(\text{`req3'} \mid\mid S)$ and $e_{K_A}(VC \mid\mid \textit{payload stream})$ to B where VC is defined to be eight bytes set to zero and e_{K_A} is the *RC4* stream cipher.

4. B sends $e_{K_B}(VC \mid\mid \textit{payload stream})$ to A.

[a] The prime p is defined to be 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A63A36210000000000090563.

cipher to protect the payload stream; and extensive use of the SHA-1 hash function for identification and verification of content in combination with the BitTorrent protocol. The protocols and schemes used by MSE are listed in Table 2 and the MSE protocol itself is illustrated in Figure 1.

Alice, the initiator of the protocol, sends a newly generated "public key," Y_A and some random data $PadA$ between 0 and 512 bytes in length to Bob in the first flow of the protocol. The random data is included in order to disguise the header, making the protocol harder to identify. Bob replies with his own "public key" and 0 to 512 bytes of random data. At this point, Alice and Bob are both able to compute a shared secret S using the Diffie-Hellman key agreement scheme. They also compute two additional shared secrets, K_A and K_B, using the ASCII text strings 'keyA' and 'keyB', and $T_{info\ hash}$, a pre-shared secret. When MSE is used with BitTorrent the *torrent info hash*, a unique value identifying the content Alice wishes to exchange with Bob, is used as the pre-shared secret. The torrent info hash is the SHA-1 hash of the *torrent metainfo file* that contains metainformation about the content, such as the directory structure and names of files making up the content, and an index of SHA-1 hashes of chunks of the content data. Torrent metainfo files are commonly retrieved from torrent directories over secure HTTP connections using SSL.

In the third flow of the protocol Alice sends several pieces of data to Bob, the first of which is $H(\text{'req1'} \parallel S)$. This hash value informs Bob that Alice has successfully computed the shared secret S, a form of key confirmation. The string 'req1' is included in the hash to destroy symmetry and prevent an adversary from reusing the value in a different place in the protocol. Next Alice sends $H(\text{'req2'} \parallel T_{info\ hash}) \oplus H(\text{'req3'} \parallel S)$ to Bob, which confirms that Alice knows the pre-shared secret $T_{info\ hash}$. The hash of the pre-shared secret is exclusive or'd with a hash of the string 'req3' concatenated with S to ensure that $H(\text{'req2'} \parallel T_{info\ hash})$ cannot be replayed from a previous session. When MSE is used with BitTorrent, the explicit confirmation of $T_{info\ hash}$ may also help Bob determine which content Alice wants to exchange if he is hosting a large number of different contents. The last piece of data Alice sends to Bob in the third flow is a verification constant VC, and optionally the start of the payload stream, encrypted using the RC4 stream cipher using K_A as the key. (The first 1024 keystream bytes from the RC4 cipher are discarded to defeat the attack by Fluhrer, Mantin and Shamir, known as the FMS-attack, described in [12] and [18]).

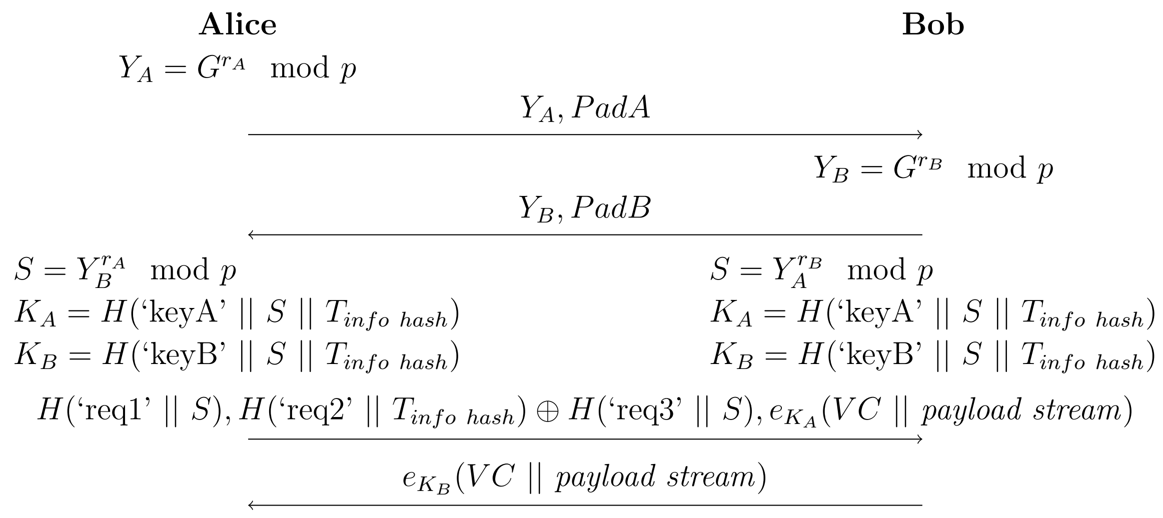
Alice
Bob
$Y_A = G^{r_A} \mod p$
$Y_A, PadA$
$Y_B = G^{r_B} \mod p$
$Y_B, PadB$
$S = Y_B^{r_A} \mod p$
$K_A = H(\text{‘keyA’} \mid\mid S \mid\mid T_{info\ hash})$
$K_B = H(\text{‘keyB’} \mid\mid S \mid\mid T_{info\ hash})$
$S = Y_A^{r_B} \mod p$
$K_A = H(\text{‘keyA’} \mid\mid S \mid\mid T_{info\ hash})$
$K_B = H(\text{‘keyB’} \mid\mid S \mid\mid T_{info\ hash})$
$H(\text{‘req1’} \mid\mid S), H(\text{‘req2’} \mid\mid T_{info\ hash}) \oplus H(\text{‘req3’} \mid\mid S), e_{K_A}(VC \mid\mid \textit{payload stream})$
$e_{K_B}(VC \mid\mid \textit{payload stream})$

Figure 1: The Message Stream Encryption protocol used by BitTorrent peer-to-peer software. The “crypto select” and “crypto provide” option fields have been omitted for clarity.

The keys K_A and K_B are constructed using both S, which is unique to this session, and the pre-shared secret $T_{info\ hash}$. The pre-shared secret is included in the key generation in an attempt to thwart a traditional intruder-in-the-middle attack as described in [30]; however, the next section describes how an adversary can still obtain useful information about the content being shared. Different keys are used for the RC4 cipher for data sent from Alice to Bob and Bob to Alice. Presumably this is to increase *semantic security*–if the same plaintext in sent in each direction, the use of distinct keys will ensure that the ciphertexts are different.

The MSE Protocol Specification states that the purpose of the verification constant is for one side to prove to the other that they know S and $T_{info\ hash}$. The encryption of the verification constant in the fourth flow also serves to mark the end of $PadB$. However, Alice has already confirmed to Bob that she knows S and $T_{info\ hash}$ in the third flow before she sends the encrypted verification constant, making this step redundant. Furthermore, the verification constant makes the protocol less secure as we will see in the next section.

2.3 Security and Privacy Guarantees

Both Skype and BitTorrent provide only very basic security guarantees. Even though secure links are established between Skype applications using a public key infrastructure combined with a robust block cipher, users are provided very little authentication guarantees at the application level. Skype users can only identify other Skype users based on their reported name, location, e-mail address and username. Skype provides a variety of ways for users to communicate with each other, including instant messages and voice and video conferencing. Conceivably it would be very easy for one user to impersonate another simply by reporting a false name, location and e-mail address unless the other user is able to verify physical attributes of the other person when using voice or video conferencing. Additionally, Biondi and Desclaux [9] briefly describe how an intruder-in-the-middle attack can be performed by Skype to decrypt sessions keys and intercept private communications.

The BitTorrent Protocol combined with Message Stream Encryption intend to guarantee that the content received using a particular torrent metainfo file is reconstructed exactly as it was published; however, Message Stream encryption and the BitTorrent protocol do not provide any way to securely verify the identity of a publisher. Message Stream Encryption and the BitTorrent protocol also do not provide any means of authenticating other peers from which content is downloaded. A number of organizations similar to US Copyright Group, which has been accused of developing a "lucrative trade in monetizing copyright infringement allegations," have emerged in recent years that exploit this vulnerability: they use specialized software to join BitTorrent "swarms" on behalf of copyright holders as if they were sharing a particular content. Then they record the Internet Protocol addresses of other peers in the swarm and use legal threats to coerce Internet service account holders to pay damages in the range of thousands of dollars [25] [6].

On the other hand, Message Stream Encryption is intended to protect individuals from targeted attacks. The security of Message Stream Encryption depends largely on the security of the weak pre-shared secret $T_{info\ hash}$. The $T_{info\ hash}$ is commonly obtained from a Web server using HTTP over SSL. Thus, it is not unreasonable to assume that an attacker targeting a specific user does not know the pre-shared secret(s) being used. This scenario may occur if a law or copyright enforcement agency wishes to target a specific user in order to determine what content they are sharing. A BitTorrent user may sensibly expect Message Stream Encryption to protect against such a scenario; however, MSE contains a number of weaknesses that may enable an attacker to gain information about what content is being shared.

Event though Message Stream Encryption was developed in 2006, it uses relatively short 160-bit keys. Furthermore, several weaknesses in the RC4 stream cipher were already known at the time and refined further in 2008, further weakening the scheme [15]. Weaknesses in SHA-1 were also known [33]. Even with disregard to these deficiencies, the Message Stream Encryption protocol has weaknesses of its own. An analysis by Brumley and Valkonen [11] identifies three specific weaknesses.

The lack of a keyed message authentication code or digital signature in the MSE handshake makes the protocol vulnerable to a bit-flipping attack. Message Stream Encryption has a *crypto_provide* option filed before the start of the payload stream in the third flow of Figure 1 and a *crypto_select* option filed before the start of the payload stream in the fourth flow, which were omitted for clarity. The crypto provide field is used to indicate the ciphers Alice supports for encrypting the payload stream. In the current version of MSE, Alice must indicate that she supports the RC4 cipher, a "null" cipher that results in the payload stream being sent in plaintext, or both. (The verification constant, crypto provide and crypto select fields are always encrypted using RC4). Bob selects the cipher for the payload stream using the crypto select field. If Alice does not support the cipher chosen by Bob, then she terminates the connection. If an attacker can locate the position of the crypto provide field in the ciphertext, it is possible that he can alter the ciphertext an undetectable manner such that the bit corresponding to the RC4 cipher in the crypto provide option field is cleared. This will result in either Alice or Bob terminating the connection, or proceeding with the payload stream being sent in plaintext.

The Message Stream Encryption Protocol contains a number of zero-valued fields. The verification constant, which is defined to be eight bytes set to zero, is sent in both directions encrypted using the RC4 stream cipher. The verification constant will appear in different offsets relative to the start of an MSE handshake due to the variable length padding that proceeds it. The MSE protocol also specifies that additional variable length, zero valued padding may be sent after the crypto select and crypto provide fields. (These were also omitted from Protocol 1 and Figure 1 for clarity). The RC4 stream cipher generates a pseudo-random stream of bits that are combined with the plaintext using and exclusive or operation that results in the ciphertext. These zero values fields result in the leakage of a significant number of keystream bytes. This is not necessarily a problem per se; however, an attack is presented in [11] that can lead to the instant recovery of $T_{info\ hash}$, and thus the identity of the the content being shared, when the same keystream is reused in separate sessions for the same torrent. The authors also present two methods an attacker can use to force keystream reuse.

In an active Diffie-Hellman intruder-in-the-middle attack, it is assumed that an attacker knows the negotiated key S. The only piece of information an attacker does not know in the second item of the third flow of Figure 1 is the pre-shared secret, $T_{info\ hash}$, but this can easily be recovered using a trivial dictionary or brute-force attack. Let X be the second item in the third flow of the protocol, that is, $X = H(\text{‘req2’} \mid\mid T_{info\ hash}) \oplus H(\text{‘req3’} \mid\mid S)$. The attacker can easily compute $X \oplus H(\text{‘req3’} \mid\mid S)$ to recover $H(\text{‘req2’} \mid\mid T_{info\ hash})$, which we will call the *salted info hash.* Most torrent info hashes are publicly available on the World Wide Web. In fact, an index of over 1.6 million popular torrent info hashes can be downloaded from The Pirate Bay in a matter of seconds over a broadband Internet connection [19]. An attacker can compute the salted info hash for every info hash in the dictionary, or every possible info hash in the case of a brute force attack, to recover the unsalted info hash value and thus the identity of the content being shared. The attacker could proceed further and compute K_A and K_B to decrypt the payload stream if desired. Even without recovering $T_{info\ hash}$, this technique enables an attacker to determine if the same content is being shared in multiple sessions by comparing the the session's salted info hashes.

Appendix 1 presents `mse_attack`, a simple C++ program that can be used to process the torrent index file obtained from [19] and compute the salted info hash for every info hash in the file and map it back to the unsalted info hash. The following output is produced when `mse_attack` is run against the index file on a modern (circa 2011) desktop computer:

```
Pre-processing file...
Error parsing line 499: info hash is not 40 characters long or contains invalid chars
Error parsing line 1041: info hash is not 40 characters long or contains invalid chars
Error parsing line 1407: info hash is not 40 characters long or contains invalid chars
[...]
Identified 1642674 torrent hashes. (took 0.717723 seconds)
Computing H('req2' || T_{info hash}) for each T_{info_hash} in index...
Done. (took 0.249529 seconds; 6.5831e+06 hashes per second)
The first 10 H('req2' || T_{info hash}) -> T_{info hash} pairs are:
20de8d028e7d33d89d3abe25e55995d5405fba1e -> 6e43d748d9446e3cddec69ce9b2ababb51bbf827
b7cf4c4107996248be4ee99f45c4bca8dfc0771a -> 170acce3275c7f21660821c4c7bbfe36ec8c45f2
ed621b0d0c3624d0cba4b086b2ef5003c3db70f7 -> cf829fb1516a6a07bf48336d531f0a44448a897a
```

```
247071aecfc762b3c36b033dd10f6795bbff0970 -> 49201f7320f42da6bb142e78b8c1168ef24c2b4b
ca29a4e3244532d6b7e4958b8365c5fdf5c4d0cd -> 690ea81cd15f957380b2f1704e947ded8eed631f
e6f4cfc2565dc9820764d223f82e79f89752c701 -> 986ceef49170db532d47314127769375 8e52ee87
ff05d11459a64101b68e02b048e9275580a6ba4d -> ba13cce5e2583e9c8f61f384a6387bfd967d883f
85a0ea06cb87a6cf4ac75b2f6623539209baa3bb -> a9b28e1f000527e2b18143379fe03dfe9e85166c
c189cd26e7a4a5307d331274b4ccbee54de1b755 -> 8393af5372eb5b43bf7ac89adb82b061cf055cb7
f6325cfa916f9ea8906444d66b1abafa9fd8745a -> 423399faf33b14e533f094007bb3681aa0f39ff1
```

As we can see from the output, computing the salted info hashes for over 1.6 million torrents took less than one quarter second without using any form of parallelization. The salted info hashes can be computed once offline and reused for every attack because the salt, ‘req2’ is a constant value defined in the MSE protocol specification. This demonstration shows that it would be very reasonable and practical for any Internet service provider, copyright, or law enforcement agency to perform an active intruder-in-the-middle and dictionary attack to identify the content being shared between two BitTorrent peers provided the info hash is publicly available on the World Wide Web. Once an adversary has obtained the unsalted info hash, they can download the torrent metainfo file from the Web or the BitTorrent distributed hash table. He can also continue to compute K_A and K_B and decrypt the payload stream.

A brute force attack, on the other hand, is theoretically possible, but unlikely to be practical. Based on a rate of 6.5831×10^6 hashes per second, computing the salted info hash for each of the 2^{160} possible torrent info hash values would take approximately 7×10^{33} years. Computing all 2^{160} salted and unsalted info hash pairs in advance would require at least 5×10^{37} tebibytes of storage.

Remember that one of the stated design goals of Message Stream Encryption is payload and protocol obfuscation. Unfortunately, Message Stream Encryption performs poorly here too. A 2010 report by Hjelmvik and John [14] describes how BitTorrent and Skype network traffic can be accurately identified by fingerprinting statistically measurable properties of TCP and UDP sessions. Their experiment demonstrated that it is possible to identify BitTorrent Protocol sessions protected with Message Stream Encryption with 96.3% accuracy by applying statistical analysis to the first 100 network packets exchanged.

3 Cryptography in a Peer-to-Peer Web Browsing System

In the previous section we discussed the security and privacy guarantees provided by BitTorrent and Skype: both of these systems aim to secure communication between peers in the network from eavesdropping by passive adversaries. Other peer-to-peer systems such as Bitcoin[2] offer fundamentally different privacy guarantees. Bitcoin is a new electronic cash protocol and unit of currency that makes use of a network of computing power to complete instant online payments. Message confidentiality is not a goal of Bitcoin as all transactions are public; however, non-repudiation is one of its primary goals.

Freenet[3] is a system that enables users to share files, and browse and publish "freesites" that are websites accessible only through Freenet. Freenet's primary security goal is anonymity for its users. Communications between Freenet nodes are encrypted and routed through other nodes to provide anonymity to those publishing and accessing content. Freenet has some goals in common with pWeb; however the architecture of the two systems is substantially different. We have chosen to emphasize speed, scalability and the ability to provide advanced features such as efficient distributed search capabilities over anonymity for the system's users. In locations where it may be necessary to protect user's identity from governments that do not fully respect human rights, pWeb network traffic can be tunnelled through the Tor anonymity network[4].

Nevertheless, it is clear that pWeb should support the basic guarantee of secure communication between network peers. The example of Message Stream Encryption has demonstrated that it is difficult to design and implement a secure encryption scheme–a task that is best performed as part of an open, peer-reviewed process. Furthermore, a mechanism should be built-in to allow encryption schemes to be upgraded as weaknesses are found and increased computing power becomes available.

Figure 2 shows the basic architecture of pWeb. At the outer level, users will use their web browser to communicate with the peer-to-peer Web software using JavaScript and HTML5. More than one user may communicate with an instance of the pWeb which may or may not run on the same machine as the web browser. Connections between the web

[2]http://bitcoin.org/
[3]https://freenetproject.org/
[4]https://torproject.org/

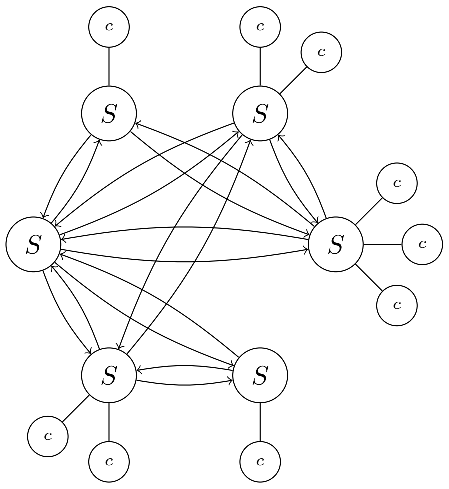

Figure 2: The basic architecture of pWeb. At the outer level, users will use their web browser to communicate with the peer-to-peer Web software using JavaScript and HTML5. At the inner level, instances of the peer-to-peer Web software will communicate with each other using a peer-to-peer network overlay.

browser and web server will use HTTP or a more modern protocol such as SPDY[5] over SSL. At the inner level, instances of the peer-to-peer Web software will communicate with each other using a peer-to-peer network overlay. These are the connections that we must protect using cryptography. Like BitTorrent, we can secure the peer to peer communication from eavesdropping by using a key agreement scheme and block or stream cipher and content can be located using using a hash value.

However, unlike BitTorrent, we wish to bind the identity of an author to content. We can achieve this using a signature scheme based on public key cryptography. Public keys can be stored in the pWeb distributed hash table and retrieved using hash values, like any other content in pWeb. One of our our design goals is to make pWeb completely distributed. This precludes the use of a certification authority to issue digital certificates. Instead the authenticity of the binding between a public key and its owner will be established using a Web of Trust. A Web of Trust provides a distributed trust model that is an alternative to the centralized trust model of a public key infrastructure with a certification authority. Table 3 lists the schemes that are being considered for use in pWeb.

[5]http://www.chromium.org/spdy/spdy-whitepaper/

Protocol	**Scheme**
Key Agreement	Diffie-Hellman
Block Cipher	AES
Hash Functions	SHA-256 or BLAKE-256
Signature Scheme	ElGamal or DSA
Public Key Infrastructure	None; A web of trust will be used to authenticate public keys

Table 3: Cryptographic protocols and schemes being considered for use in pWeb. The Diffie-Hellman, ElGamal and DSA schemes can be implemented in a finite multiplicative group or on an elliptic curve over a finite field. Implementations in both domains will be evaluated.

The Diffie-Hellman key agreement scheme and ElGamal and DSA signature schemes are in the process of being evaluated in both the finite multiplicative group and elliptic curve domains. The primary benefit of implementing schemes in the elliptic curve domain as opposed to the finite multiplicative group domain is smaller key sizes for comparable levels of security. For example, a 160-bit RSA key in the elliptic curve domain provides a computational and cost-equivalent level of security to to a 1024-bit public key in the finite multiplicative group domain [16]. The reduced key size also results in reduced storage and computational requirements. These features will likely be beneficial to pWeb.

AES is the current de facto standard for encrypting streams of data and the only block cipher selected for further investigation. Modern computer processors provide specialized instructions for accelerating encryption and decryption. In 2003, the United States National Security Agency approved the use of AES to protect top secret information [20]. The only successful attacks against AES were side channel attacks until 2009. The first key-recovery attacks against the full AES were published in 2011 and are faster than a brute force attack by a factor of about four [10].

SHA-256 is a member of the SHA-2 family of functions. The United States National Institute of Standards and Technology (NIST) currently endorses the use of the SHA-2 family of functions for all federal agencies [21]. Some weaknesses have been found in these hash functions in the face of preimage and collision attacks [33], but they still remain secure. Nevertheless, NIST in currently holding a competition for a successor to SHA-2.

The competition started in 2007 and is scheduled to conclude later this year.

BLAKE is one of five final candidates for the NIST Cryptographic Hash Algorithm Competition. The winner of the competition will be be named “SHA-3.” No significant weaknesses have been found during the the five year competition. The Status Report on the second Round of the SHA-3 competition authored by NIST that summarizes feedback from the cryptographic community states:

> BLAKE is among the top performers in software across most platforms for long messages. BLAKE-32 is the best performer on software platforms for very short messages.
>
> ...
>
> BLAKE was selected as a finalist, due to its high security margin, good performance in software, and its simple and clear design. [31]

The performance of AES, SHA-256 and BLAKE-256 are analyzed in the next section. The performance of the key agreement and signature schemes are not analyzed because they are less likely to impact the performance of an application. Key agreement is performed only once per session, while a stream or block cipher is used to encrypt and decrypt every byte transferred. Similarly, a signature scheme is typically used with a hash function to produce a short digest of a large volume of data that is signed.

4 Performance of Protocol Implementations in C/C++

In this section we briefly analyze the performance of block cipher and hash function implementations that are being evaluated for use in pWeb.

4.1 Block Cipher

The OpenSSL Project’s implementation of the AES block was selected because it is actively maintained and uses specialized features in modern processors for hardware acceleration. Listing 2 in Appendix B shows the C source code for `aes_time`, a program that times how long it takes to encrypt and decrypt a text file and video file on a modern desktop or laptop computer in a single thread. The output from one run of `aes_time` is shown below.

```
Timing OpenSSL's implementation of 256-bit AES...
Starting string:              "Lorem ipsu..."
Encrypted string data:        dce35707724c633c1717...
Time to encrypt 2 KiB string: 20052 nanoseconds
Decrypted string data:        4c6f72656d2069707375...
Ending String:                "Lorem ipsu..."
Time to decrypt 2 KiB string: 27780 nanoseconds

Encrypted video data:         127ab36854d0ee8193d4...
Time to encrypt 45 MiB video: 0.437203 seconds (834.723 Mbit/sec)
Decrypted video data:         4f676753000200000000...
Time to decrypt 45 MiB video: 0.59905 seconds (609.202 Mbit/sec)

Timing direct memory copy...
Time to copy 2 KiB string:    52 nanoseconds

Time to copy 45 MiB video:    0.00449201 seconds (81242.6 Mbit/sec)
```

This output shows that encrypting a large file is about ten times slower than simply copying it from one region of memory to another; however, AES encryption is still very fast. The 45 mebibyte file was encrypted at a rate of over 800 megabits per second using a single thread. This is nearly fast enough to saturate a one gigabit network connection. We expect that the pWeb software will be run primarily on end user laptop and desktop computers, therefore this throughput is more than adequate.

4.2 Hash Functions

The OpenSSL project's implementation of SHA-256 was also selected for analysis because it is widely used and actively maintained. The reference implementation of BLAKE-256 submitted to the final round of NIST Cryptographic Hash Algorithm competition was used. Listing 4 in Appendix C shows the C source code for `hash_compare`, a program that compares the relative speeds of the SHA-256 and BLAKE-256 algorithms. The output from one run of `hash_compare` is shown below:

```
Timing OpenSSL's implementation of SHA-256...
2 KiB string:         8162 nanoseconds
45 MiB file:     180005380 nanoseconds
Timing reference implementation of BLAKE-256...
2 KiB string:        35798 nanoseconds
45 MiB file:     796317758 nanoseconds
```

This output shows that it takes only 8 microseconds to hash a 2 kibibyte text file or 180 milliseconds to hash a 45 mebibyte video file using SHA-256; or 36 microseconds to hash a 2 kibibyte text file or 796 microseconds to hash a 45 mebibyte video file using BLAKE-256. These two file represent a small file and large file in pWeb. Although BLAKE-256 appears to be about four times slower that SHA-256, we expect most files to be small files, therefore the difference in running time between SHA-256 and BLAKE-256 is negligible. However, Gueron, Johnson and Walker describe how SHA-2 can be sped up by using SHA-512 and truncating the result in place of using SHA-256 [13]. Furthermore, a winner of the Cryptographic Hash Algorithm Competition has not been selected yet. Based on these results, SHA-256 is being recommended for use in pWeb.

5 Conclusion

The first part of this report examined the protocols and schemes used in two of the most successful peer-to-peer systems ever created: Skype and BitTorrent. Each of these systems provides only very basic privacy guarantees to their users. The protocols used by BitTorrent in particular contain several weaknesses that could enable an attacker to discover significant information about the content being shared.

The security requirements of a new peer-to-peer Web browsing system were identified and several cryptographic protocols and schemes were selected as suitable for pWeb. Well studied and widely used protocols were chosen in light of the problems identified with Message Stream Encryption. Finally, the performance of the AES block cipher and SHA-256 and BLAKE-256 hash functions was analyzed and SHA-256 was recommended for use in pWeb.

Appendices

A Source code for `mse_attack`

```
//==========================================================================
// Name        : mse_attack.cpp
// Author      : Alexander Pokluda
// Description : Computes SHA-1('req2', T_{info hash}) for all T_{info hash}
//               in the input file
//==========================================================================

#include <iostream>
#include <cstdio>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <boost/chrono.hpp>
#include <openssl/sha.h>

using namespace std;
using namespace boost;
using namespace boost::chrono;

typedef high_resolution_clock clock_type;
typedef clock_type::time_point time_pt;
inline clock_type::time_point now() { return clock_type::now(); }
typedef duration<double, ratio<1>> seconds_t;

struct infohash
{
  unsigned char bytes[SHA_DIGEST_LENGTH];
};

void hash_string_to_bytes(unsigned char* bytes, const char* str)
{
  const char* const end = str + 40;

  for (; str != end; ++str, ++bytes)
  {
    // Convert one nibble at a time (str and bytes array are big endian)
    if (*str - '0' < 10)
      *bytes = (*str - '0') << 4;
    else if (*str - 'A' < 26)
      *bytes = (*str - 'A' + 10) << 4;
    else if (*str - 'a' < 26)
      *bytes = (*str - 'a' + 10) << 4;
    else
      throw "invalid hash";

    ++str;
    if (*str - '0' < 10)
      *bytes |= (*str - '0');
    else if (*str - 'A' < 26)
      *bytes |= (*str - 'A' + 10);
    else if (*str - 'a' < 26)
      *bytes |= (*str - 'a' + 10);
    else
      throw "invalid hash";
  }
}

void hash_bytes_to_string(char *str, const unsigned char* bytes)
{
  const char* end = str + 40;

  for (; str != end; ++str, ++bytes)
  {
    // Convert one nibble at a time (str and bytes array are big endian)
    unsigned char nibble = *bytes >> 4;
    if (nibble < 10)
      *str = '0' + nibble;
    else
      *str = 'a' - 10 + nibble;

    ++str;
    nibble = *bytes & 0x0F;
    if (nibble < 10)
      *str = '0' + nibble;
    else
      *str = 'a' - 10 + nibble;
  }
```

```
}

void read_hashes(const char* c,  const char* end, vector<infohash>& hashes)
{
  cout << "Pre-processing file..." << endl;
  const time_pt time_start = now();

  size_t lnum = 0;
  for (; c != end; ++c)
  {
    if (*c == '\n')
    {
      ++lnum;

      // Sanity check
      if (*(c - 1) == '|')
      {
        // this line does not have an info hash
        continue;
      }
      else if (*(c - 41) != '|')
      {
        cerr << "Error parsing line " << lnum
             << ": invalid format" << endl;
        continue;
      }
      infohash hash;
      try {hash_string_to_bytes(hash.bytes, c - 40);}
      catch (...)
      {
        cerr << "Error parsing line " << lnum
             << ": info hash is not 40 characters long or "
             "contains invalid characters" << endl;
        continue;
      }
      hashes.push_back(hash);
    }
  }

  const time_pt time_end = now();
  cout << "Identified " << hashes.size() << " torrent hashes. (took "
      << duration_cast<seconds_t>(time_end-time_start) << ")" << endl;
}

infohash compute_hash(const infohash& h_in)
{
  SHA_CTX ctx;
  SHA1_Init(&ctx);
  SHA1_Update(&ctx, "req2", 4);
  SHA1_Update(&ctx, h_in.bytes, 20);
  infohash h_out;
  SHA1_Final(h_out.bytes, &ctx);
  return h_out;
}

int main(const int argc, const char* argv[])
{
  if (argc != 2)
  {
    cerr << "Usage: " << argv[0] << " torrent_index_file" << endl;
  }

  int retcode = EXIT_SUCCESS;
  int fd = -1;
  off_t fsize;
  void* addr = MAP_FAILED;

  try
  {
    // Open the file
    fd = open(argv[1], O_RDONLY);
    if (fd == -1) throw "Unable to open file.";

    // Determine the file size
    struct stat sb;
    if (fstat(fd, &sb) == -1) throw "Unable to determine file size.";
    fsize = sb.st_size;

    // Map the file into memory
    addr = mmap(NULL, fsize, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED) throw "Unable to mmap file.";

    // Read the hashes from the file
    vector<infohash> h_in;
    vector<infohash> h_out;
    h_in.reserve(2000000);
    read_hashes(static_cast<char*>(addr),
        static_cast<char*>(addr)+fsize, h_in);
    h_out.reserve(h_in.size());

    cout << "Computing H('req2' || T_{info hash}) "
        "for each T_{info_hash} in index..." << endl;
    const time_pt start = now();
    transform(h_in.begin(), h_in.end(), back_inserter(h_out),
        compute_hash);
    const time_pt end = now();
    const seconds_t elapsed = duration_cast<seconds_t>(end - start);
    cout << "Done. (took " << elapsed << "; "
        << h_out.size()/elapsed.count() << " hashes per second)"
        << endl;

    // This program was designed as a demonstration, so we simply print
    // a few of the computed hashes here. The following code could be
    // easily modified to output the computed hashes to a file.
    cout << "The first 10 H('req2' || T_{info hash}) -> T_{info hash} "
        "pairs are:" << endl;
    char h_str[41];
    for (vector<infohash>::const_iterator h_in_iter = h_in.begin(),
        h_out_iter = h_out.begin();
        h_in_iter != h_in.begin() + 10; ++h_in_iter, ++h_out_iter)
    {
      hash_bytes_to_string(h_str, h_out_iter->bytes);
      h_str[40] = '\0';
      cout << h_str << " -> ";
      hash_bytes_to_string(h_str, h_in_iter->bytes);
      h_str[40] = '\0';
      cout << h_str << '\n';
    }
    cout << endl;
  }
  catch (const char* err)
  {
    cerr << err << endl;
    retcode = EXIT_FAILURE;
  }
  catch (const std::bad_alloc&)
  {
    cerr << "Unable to allocate file buffer." << endl;
    retcode = EXIT_FAILURE;
  }
  catch (...)
  {
    cerr << "An unknown error occurred." << endl;
    retcode = EXIT_FAILURE;
  }

  if (fd != -1) close(fd);
  if (addr != MAP_FAILED) munmap(addr, fsize);

  return retcode;
}
```

Listing 1: Source code of the program `mse_attack` that demonstrates how a dictionary attack can be performed against Message Stream Encryption.

B Source code for `aes_time`

```
//============================================================================
// Name        : aes_time.cpp
// Author      : Alexander Pokluda
// Description : Measures how long it takes to encrypt a 2 KiB text string
//               and 45 MiB video file using 256-bit AES in CBC mode
//============================================================================

#include <cstring>
#include <cassert>
#include <openssl/aes.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <vector>
#include <iostream>
#include <iomanip>
#include <boost/chrono.hpp>
```

```

const int STRING_ITERATIONS = 1000;
const int FILE_ITERATIONS = 10;

using namespace std;
using namespace boost;
using namespace boost::chrono;

typedef high_resolution_clock clock_type;
typedef clock_type::time_point time_pt;
inline clock_type::time_point now() { return clock_type::now(); }
typedef duration<double, ratio<1>> seconds_t;

extern const char* const str;
unsigned long const len = strlen(str);

void exit(const char* const msg)
{
  fputs(msg, stderr);
  exit(EXIT_FAILURE);
}

const int AES_256_KEY_LEN = 32;

typedef unsigned char byte;
typedef vector<byte> buf_t;

inline const byte* to_bytes(const void* ptr)
{
  return reinterpret_cast<const byte*>(ptr);
}

inline byte* to_bytes(void* ptr)
{
  return reinterpret_cast<byte*>(ptr);
}

inline const byte* to_bytes(const buf_t& buf)
{
  return buf.data();
}
```

```

inline byte* to_bytes(buf_t& buf)
{
  return buf.data();
}

inline const char* to_str(const buf_t& buf)
{
  return reinterpret_cast<const char*>(buf.data());
}

// Print the first 20 bytes of buffer
std::ostream &operator<<(std::ostream &out, const buf_t& buf)
{
  out << hex << setfill('0');
  buf_t::const_iterator i = buf.begin();
  const buf_t::const_iterator end = (buf.end()-i>10 ? i+10 : buf.end());
  for (; i != end; ++i)
  {
    out << setw(2) << static_cast<unsigned>(*i);
  }
  if (end != buf.end())
  {
    out << "...";
  }
  return out << dec;
}

const byte* key_data = to_bytes("Really simple and insecure key.");
const byte ivec_data[AES_BLOCK_SIZE] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
    0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};

template <size_t multiple>
inline size_t round_up(const size_t len)
{
  if (len % multiple == 0) return len;
  else return ((len / multiple) + 1) * multiple;
}

void OpenSSL_AES_256_CBC_Encrypt(const byte* plaintext, byte* ciphertext,
    const size_t len, const byte* ivec_data, const byte* key_data)
```

```
{
  byte ivec[AES_BLOCK_SIZE];
  memcpy(ivec, ivec_data, AES_BLOCK_SIZE);

  AES_KEY e_key;
  AES_set_encrypt_key(key_data, 256, &e_key);
  AES_cbc_encrypt(plaintext, ciphertext, len, &e_key, ivec, AES_ENCRYPT);
}

void OpenSSL_AES_256_CBC_Decrypt(byte* plaintext, const byte* ciphertext,
    const size_t len, const byte* ivec_data, const byte* key_data)
{
  assert(len % AES_BLOCK_SIZE == 0);

  byte ivec[AES_BLOCK_SIZE];
  memcpy(ivec, ivec_data, AES_BLOCK_SIZE);

  AES_KEY d_key;
  AES_set_decrypt_key(key_data, 256, &d_key);
  AES_cbc_encrypt(ciphertext, plaintext, len, &d_key, ivec, AES_DECRYPT);
}

int main()
{
  try
  {
    // Open the video file
    int fd = open("ed_1024.ogv", O_RDONLY);
    if (fd == -1) exit("Unable to open file.");

    // Determine the file size
    struct stat sb;
    if (fstat(fd, &sb) == -1) exit("Unable to determine file size.");
    ssize_t fsize = sb.st_size;

    // Read the file into memory
    buf_t video;
    video.resize(fsize);
    if (read(fd, to_bytes(video), fsize) != fsize)
      exit("Error while reading file.");

    // Close the file
    close(fd);

    // ** Time OpenSSL's implementation of 256-bit AES **
    cout << "Timing OpenSSL's implementation of 256-bit AES..." << endl;

    assert(fsize >= static_cast<ssize_t>(len));
    const size_t encrypted_video_len = round_up<AES_BLOCK_SIZE>(fsize);
    const size_t encrypted_str_len = round_up<AES_BLOCK_SIZE>(len);
    buf_t ciphertext, plaintext;

    // Compute the amount of time required to encrypt and decrypt
    // a 2 KiB string by averaging 1000 trials
    ciphertext.resize(encrypted_str_len);
    time_pt start = now();
    for (int i = 0; i < STRING_ITERATIONS; ++i)
    {
      OpenSSL_AES_256_CBC_Encrypt(to_bytes(str), to_bytes(ciphertext),
          len, ivec_data, key_data);
    }
    time_pt end = now();
    string start_str(str, 10);
    start_str += "...";
    cout << "Starting string:\t\t\"" << start_str
        << "\"\nEncrypted string data:\t\t" << ciphertext
        << "\nTime to encrypt 2 KiB string:\t"
        << (end-start) / STRING_ITERATIONS << endl;

    plaintext.resize(encrypted_str_len);
    start = now();
    for (int i = 0; i < STRING_ITERATIONS; ++i)
    {
      OpenSSL_AES_256_CBC_Decrypt(to_bytes(plaintext),
          to_bytes(ciphertext), encrypted_str_len, ivec_data,
          key_data);
    }
    end = now();
    string end_str(to_str(plaintext), 10);
    end_str += "...";
    cout << "Decrypted string data:\t\t" << plaintext
        << "\nEnding String:\t\t\t\"" << end_str
```

```
        << "\"\nTime to decrypt 2 KiB string:\t"
        << (end-start) / STRING_ITERATIONS << endl;

    // Compute the amount of time required to encrypt and decrpyt
    // a 45 MiB video by averaging 10 trials
    ciphertext.resize(encrypted_video_len);
    start = now();
    for (int i = 0; i < FILE_ITERATIONS; ++i)
    {
      OpenSSL_AES_256_CBC_Encrypt(to_bytes(video),
          to_bytes(ciphertext), fsize, ivec_data, key_data);
    }
    end = now();
    double duration = duration_cast<seconds_t>(end - start).count() /
        FILE_ITERATIONS;
    double rate = (fsize / duration) * 8 / 1000000; // Mbit/sec
    cout << "\nEncrypted video data:\t\t" << ciphertext
        << "\nTime to encrypt 45 MiB video:\t" << duration
        << " seconds (" << rate << " Mbit/sec)" << endl;

    plaintext.resize(encrypted_video_len);
    start = now();
    for (int i = 0; i < FILE_ITERATIONS; ++i)
    {
      OpenSSL_AES_256_CBC_Decrypt(to_bytes(plaintext),
          to_bytes(ciphertext), encrypted_video_len, ivec_data,
          key_data);
    }
    end = now();
    duration = duration_cast<seconds_t>(end - start).count() /
        FILE_ITERATIONS;
    rate = (fsize / duration) * 8 / 1000000; // Mbit/sec
    cout << "Decrypted video data:\t\t" << plaintext
        << "\nTime to decrypt 45 MiB video:\t" << duration
        << " seconds (" << rate << " Mbit/sec)" << endl;

    // ** Time direct memory copy for comparison with AES **
    cout << "\nTiming direct memory copy..." << endl;

    // Compute the amount of time required to copy
    // a 2 KiB string by averaging 1000 trials
    start = now();
    for (int i = 0; i < STRING_ITERATIONS; ++i)
    {
      memcpy(to_bytes(ciphertext), to_bytes(str), len);
    }
    end = now();
    cout << "Time to copy 2 KiB string:\t"
        << (end-start) / STRING_ITERATIONS << endl;

    // Compute the amount of time required to copy
    // a 45 MiB video averaging 10 trials
    start = now();
    for (int i = 0; i < FILE_ITERATIONS; ++i)
    {
      memcpy(to_bytes(ciphertext), to_bytes(video), fsize);
    }
    end = now();
    duration = duration_cast<seconds_t>(end - start).count() /
        FILE_ITERATIONS;
    rate = (fsize / duration) * 8 / 1000000; // Mbit/sec
    cout << "\nTime to copy 45 MiB video:\t" << duration
        << " seconds (" << rate << " Mbit/sec)" << endl;

    return EXIT_SUCCESS;
  }
  catch (const std::bad_alloc&)
  {
    cerr << "Error allocating buffer." << endl;
    return EXIT_FAILURE;
  }
}
```

Listing 2: Source code of the program `aes_time` that measures the amount of time required to encrypt and decrypt a 2 KiB text string and 45 MiB video file using the OpenSSL Project's implementation of 256-bit AES in Cipher Block Chaining mode. The amount of time for a direct memory copy without encryption is also measured. This file must be compiled with `lorem.cpp` shown in Listing 3.

```
/*
 * lorem.cpp
 *
 * An except of the Lorem ipsum dolor Placeholder Text
 * obtained from http://desktoppub.about.com/cs/pagelayout/a/lorem.htm
 *
 */

extern const char* const str = "Lorem ipsum dolor sit amet, consectetuer "
"adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet "
"dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis "
"nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea "
"commodo consequat. Duis autem vel eum iriure dolor in hendrerit in "
"vulputate velit esse molestie consequat, vel illum dolore eu feugiat "
"nulla facilisis at vero eros et accumsan et iusto odio dignissim qui "
"blandit praesent luptatum zzril delenit augue duis dolore te feugait "
"nulla facilisi.\n"
""
"Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper "
"suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel "
"eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, "
"vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et "
"iusto odio dignissim qui blandit praesent luptatum zzril delenit augue "
"duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, "
"consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut "
"laoreet dolore magna aliquam erat volutpat.\n"
""
"Duis eget lorem ac odio lobortis suscipit nec et neque. Sed at quam ut "
"mauris scelerisque congue id eget dui. Quisque tellus lectus, tristique "
"eu posuere in, faucibus vitae urna. Duis vitae orci purus, quis euismod "
"augue. Ut wisi enim ad minim veniam, quis nostrud exerci tation "
"ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. "
"Morbi lorem arcu, aliquet sed feugiat et, posuere non elit fusce "
"sollicitudin.  Nunc at in hendrerit in vulputate, imperdiet fringilla.\n"
""
"Aliquam mauris felis, viverra in mattis vitae adipiscing elit, sed diam "
"nonummy nibh.  Euismod tincidunt ut laoreet dolore magna aliquam erat "
"volutpat. Ut wisi lobortis nisl ut aliquip ex ea commodo consequat. Duis "
"autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie "
"consequat, vel illum dolore vero eros et accumsan et iusto odio dignissim "
"qui blandit praesent luptatum zzril delenit augue. Proin dapibus "
"hendrerit purus sed dictum.\n";
```

Listing 3: Source code of `lorem.cpp` that is used by the `aes_time` and `hash_compare` programs.

C Source code for `hash_compare`

```
//============================================================================
// Name        : hash_compare.cpp
// Author      : Alexander Pokluda
// Description : Compares the relative speed of the SHA-256 and BLAKE-256
//               hash functions
//============================================================================

#include <cstdio>
#include <cstring>
#include <openssl/sha.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <new>
#include <boost/chrono.hpp>

const int STRING_ITERATIONS = 1000;
const int FILE_ITERATIONS = 10;

using namespace std;
using namespace boost;
using namespace boost::chrono;

typedef high_resolution_clock clock_type;
typedef clock_type::time_point time_pt;
inline clock_type::time_point now() { return clock_type::now(); }
typedef duration<double, ratio<1>> seconds_t;

extern "C"
{
  void blake256_hash(uint8_t *out, const uint8_t *in, uint64_t inlen);
}

extern const char* const str;
unsigned long const len = strlen(str);

void exit(const char* const msg)
```

```
{
  fputs(msg, stderr);
  exit(EXIT_FAILURE);
}

int main()
{
  // Open the video file
  int fd = open("ed_1024.ogv", O_RDONLY);
  if (fd == -1) exit("Unable to open file.");

  // Determine the file size
  struct stat sb;
  if (fstat(fd, &sb) == -1) exit("Unable to determine file size.");
  ssize_t fsize = sb.st_size;

  // Read the file into memory
  unsigned char* video = new(nothrow) unsigned char[fsize];
  if (!video) exit("Unable to allocate buffer.");
  if (read(fd, video, fsize) != fsize) exit("Error while reading file.");

  // Close the file
  close(fd);

  // ** Time OpenSSL's implementation of SHA-256 **
  cout << "Timing OpenSSL's implementation of SHA-256..." << endl;

  // Compute the amount of time required to hash a 2 KiB string
  // by hashing the string 1000 times and computing the average
  unsigned char hash[SHA256_DIGEST_LENGTH];
  time_pt start = now();
  for (int i = 0; i < STRING_ITERATIONS; ++i)
  {
    SHA256((const unsigned char* const)str, len, hash);
  }
  time_pt end = now();
  cout << "2 KiB string:\t" << (end-start) / STRING_ITERATIONS << endl;

  // Compute the amount of time required to hash a 45 MiB video
  // file by hashing the file 10 times and computing the average
  start = now();
  for (int i = 0; i < FILE_ITERATIONS; ++i)
  {
    SHA256(video, fsize, hash);
  }
  end = now();
  cout << "45 MiB file:\t" << (end - start) / FILE_ITERATIONS << endl;

  // ** Time the reference implementation of BLAKE-256 **
  cout << "Timing reference implementation of BLAKE-256..." << endl;

  // Compute the amount of time required to hash a 2 KiB string
  // by hashing the string 1000 times and computing the average
  start = now();
  for (int i = 0; i < STRING_ITERATIONS; ++i)
  {
    blake256_hash(hash, (const unsigned char* const)str, len);
  }
  end = now();
  cout << "2 KiB string:\t" << (end - start) / STRING_ITERATIONS << endl;

  // Compute the amount of time required to hash a 45 MiB video
  // file by hashing the file 10 times and computing the average
  start = now();
  for (int i = 0; i < FILE_ITERATIONS; ++i)
  {
    blake256_hash(hash, video, fsize);
  }
  end = now();
  cout << "45 MiB file:\t" << (end - start) / FILE_ITERATIONS << endl;

  delete [] video;
  return EXIT_SUCCESS;
}
```

Listing 4: Source code of the program `hash_compare` that measures the amount of time required to run SHA-256 and BLAKE-256 on a 2 KiB text string and 45 MiB video file. This file must be compiled with `lorem.cpp` shown in Listing 3 in Appendix 4

References

[1] Reaz Ahmed. *Efficient and Flexible Search in Large Scale Distributed Systems*. PhD thesis, University of Waterloo, 2007.

[2] Reaz Ahmed and Raouf Boutaba. Distributed pattern matching for p2p systems. In *Network Operations and Management Symposium*, pages 198–208. IEEE, 2006.

[3] Reaz Ahmed and Raouf Boutaba. A scalable peer-to-peer protocol enabling efficient and flexible search. Technical report, David R. Cheriton School of Computer Science, May 2006.

[4] Reaz Ahmed and Raouf Boutaba. Distributed pattern matching: A key to flexible and efficient p2p search. *IEEE Journal on Selected Areas in Communications*, 25(1):73–83, Jan 2007.

[5] Reaz Ahmed and Raouf Boutaba. Plexus: A scalable peer-to-peer protocol enabling efficient subset search. In *IEEE/ACM Transactions on Networking*, volume 17, pages 130–143. IEEE, 2009.

[6] Nate Anderson. P2p settlement lawyers lied, committed fraud says new lawsuit. http://arstechnica.com/tech-policy/2010/11/p2p-settlement-lawyers-lied-committed-fraud-says-new-lawsuit/, Nov 2010.

[7] Faizul Bari, Rakibul Haque, Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. Persistent naming for p2p web hosting. In *IEEE International Conference on Peer-to-Peer Computing*, pages 270–279. IEEE, Aug 2011.

[8] S. A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *eprint arXiv:cs/0412017*, December 2004.

[9] P. Biondi and F. Desclaux. Silver needle in the Skype. In *Black Hat Europe '06*, Amsterdam, Mar 2006.

[10] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full aes. In Dong Lee and Xiaoyun Wang, editors, *Advances in Cryptology ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer Berlin / Heidelberg, 2011.

[11] Billy Bob Brumley and Jukka Valkonen. Attacks on message stream encryption. In Hanne Riis Nielson and Christian W. Probst, editors, *Proceedings of the 13th Nordic Workshop on Secure IT Systems—NordSec '08*, pages 163–173, October 2008.

[12] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, SAC '01, pages 1–24, London, UK, UK, 2001. Springer-Verlag.

[13] S. Gueron, S. Johnson, and J. Walker. Sha-512/256. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 354 –358, april 2011.

[14] Erik Hjelmvik and Wolfgang John. Breaking and improving protocol obfuscation. Technical report, Chalmers University of Technology, Göteborg, 2010.

[15] Andreas Klein. Attacks on the rc4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008. 10.1007/s10623-008-9206-6.

[16] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[17] Ludde, uau, The_8472, Parg, and Nolar. Message stream encryption protocol specification. http://wiki.vuze.com/mediawiki/index.php?title=Message_Stream_Encryption&oldid=10408, 2011.

[18] Itsik Mantin and Adi Shamir. A practical attack on broadcast rc4. In *Proc. of FSE01*, pages 152–164. Springer-Verlag, 2001.

[19] Alfonso Maruccia. Backup of the pirate bay? a bunch of megabytes. http://www.neowin.net/news/backup-of-the-pirate-bay-a-bunch-of-megabytes, Feb 2012.

[20] National Security Agency, Ft. Meade, MD. *National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information*, June 2003.

[21] National Institute on Standards and Technology Computer Security Resource Center. Nist's policy on hash functions. http://csrc.nist.gov/groups/ST/hash/policy.html, Mar 2006.

[22] Yu Sasaki, Lei Wang, and Kazumaro Aoki. Preimage attacks on 41-step sha-256 and 46-step sha-512. *ASIACRYPT2009*, 2009.

[23] Hendrik Schulze and Klaus Mochalski. Internet study 2008/2009. www.ipoque.com/resources/internet-studies, Leipzig, Germany, 2009.

[24] Nashid Shahriar, Mahfuza Sharmin, Reaz Ahmed, and Raouf Boutaba. Diurnal availability for peer-to-peer systems. *Computing Research Repository*, 2011.

[25] Dmitriy Shirokov. Class action complaint and jury demand. Case 1:10-cv-12043 Document 1, United States District Court, District of Massachusetts, Nov 2010.

[26] Skype grows fy revenues 20%, reaches 663m users. http://www.telecompaper.com/news/skype-grows-fy-revenues-20-reaches-663-mln-users, Mar 2011.

[27] About skype. http://about.skype.com/, Aug 2012.

[28] Does Skype use encryption? https://support.skype.com/en/faq/FA31/does-skype-use-encryption, Aug 2012.

[29] Statistics summary for thepiratebay.se. http://www.alexa.com/siteinfo/thepiratebay.se, Aug 2012.

[30] Douglas R. Stinson. *Cryptography: Theory and Practice*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, Florida, USA, 2006.

[31] Meltem Sönmez Turan, Ray Perlner, Lawrence E. Bassham, William Burr, Donghoon Chang, Shu jen Chang, Morris J. Dworkin, John M. Kelsey, Souradyuti Paul, and Rene Peralta. Status report on the second round of the sha-3 cryptographic hash algorithm competition. Technical report, National Institute of Standards and Technology, 2011.

[32] Vuze. Message stream encryption — protocol objectives. http://wiki.vuze.com/w/Message_Stream_Encryption#Protocol_Objectives, Mar 2011.

[33] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full sha-1. In Victor Shoup, editor, *Advances in Cryptology CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin / Heidelberg, 2005.

[34] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman and Hall/CRC, 2008.

[35] Wikipedia. Comparison of bittorrent clients — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Comparison_of_BitTorrent_clients&oldid=507155133, Aug 2012.