Improved Forensic Recovery of PKZIP Stream Cipher Passwords

Sein Coray, Iwen Coisel and Ignacio Sanchez
European Commission, Joint Research Centre (DG JRC) - Via Enrico Fermi 2749, 21027 Ispra (VA), Italy

Keywords: Stream Cipher, High Performance Computing, Forensics, Passwords, Cybersecurity, Cryptanalysis.

Abstract: Data archives are often compressed following the PKZIP format and can optionally be encrypted with either the PKZIP stream cipher or the AES block cipher. In this article, we present new implementations of two attacks against the PKZIP stream cipher. To our knowledge, this is the first time those attacks have been demonstrated on Graphical Processing Unit (GPU). Our first implementation is retrieving archive passwords using the internal state of the PKZIP stream cipher obtained through the known-plaintext attack of Biham and Kocher. Passwords up to length 14 can be recovered within a month considering a single Nvidia 1080 Ti GPU. If one hundred of those cards are available, passwords up to length 15 would be recovered in less than 27 days. The second implementation is a more direct attack designed to retrieve an archive's password without requiring any additional knowledge than the ciphertext. Experimental results show that our two implementations are at least ten times faster than the state of the art. This is an undeniable asset for investigators who may be particularly interested in further deepening their forensic analysis on an encrypted archive.

1 INTRODUCTION

Digital forensic investigations often come across the need to access content protected by a password based authentication mechanism. Forensic investigators have a variety of techniques at their disposal to access such content, usually by retrieving the associated password. The most common password guessing approaches are an exhaustive search trying all possible combinations of characters from a specific alphabet (e.g. all numbers, letter and special characters) or a more targeted search exploring most common passwords with or without classical modification (e.g. dictionary attack with mangling rules). The available computational power as well as the speed of the algorithm used to test password candidates heavily limit the success rate of those attacks. Therefore only short passwords or predictable ones are generally recovered in practice. If the algorithm used by the target of the forensic analysis has a known weakness or vulnerability, it can be sometimes exploited to recover the target's password in a more efficient way than an exhaustive search over all possible candidates.

The PKZIP standard falls into such category of vulnerable algorithms as Biham and Kocher (Biham and Kocher, 1994) have exhibit a known-plaintext attack against the PKZIP stream cipher used to encrypt the compressed archive. Such attack can retrieve the internal state of the cipher using the knowledge of at least 12 bytes of the plaintext. The PKZIP standard supports nowadays AES encryption, yet, an analysis of the main ZIP encryption software available in the market reveals that in up to 50% of the cases, the PKZIP stream cipher remains the default encryption method for ZIP archives. There is no statistics available about the usage of one encryption method or the other, however, we assume that in those tools where it is set as a default method, the original PKZIP stream cipher encryption is still heavily used.

In this paper, we present a new implementation of the known-plaintext attack of Biham and Kocher (Biham and Kocher, 1994) that takes advantage of Graphical Processing Units (GPUs) hardware allowing the retrieval of even long passwords (up to 15 characters depending on the available hardware resources) of ZIP archives encrypted using the PKZIP stream cipher. Furthermore, we also present a novel GPU implementation that can recover the PKZIP stream cipher password when no plaintext is known.

It is the first time that such attacks against the PKZIP steam cipher are implemented for GPU hardware. The obtained benchmark results invalidate the existing hypothesis[1] that GPU implementations

[1] As discussed in https://hashcat.net/forum/thread-5709.html and https://github.com/magnumripper/JohnTheRipper/issues/718

Coray, S., Coisel, I. and Sanchez, I.
Improved Forensic Recovery of PKZIP Stream Cipher Passwords.
DOI: 10.5220/0007360503280335
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 328-335
ISBN: 978-989-758-359-9

would not make a meaningful performance difference compared to existing CPU implementations due to the nature of the encryption algorithm. Our experimental results show that our two implementations are at least ten times faster than the current implementations.

The rest of the paper is structured as follows. In Section 2, we describe the PKZIP stream cipher and review the existing attacks and respective implementations. Our GPU based implementation to retrieve the PKZIP stream cipher password from a known internal state is described in Section 3. In Section 4, we describe our more generic attack to retrieve the encryption password without any plaintext knowledge. Finally, we present our conclusions in Section 5.

2 BACKGROUND

PKZIP is a data compression program designed in 1989 by Phil Katz and his company PKWARE mostly famous for the introduction of the .zip format. In 1993, PKZIP v2.X was released with the introduction of the DEFLATE algorithm, an efficient lossless data compression algorithm. Whereas the main objective of the software is to archive file(s) in a compressed manner, the archive can optionally be encrypted to secure the data at rest. As mentioned earlier, several encryption algorithms are available, yet in this paper we will only focus on the PKZIP stream cipher that is described in what follows.

2.1 PKZIP Encryption Format

The PKZIP stream cipher is a symmetric encryption scheme, where the secret key is required for both encryption and decryption. This key is used to produce a keystream of bytes for the one-time pad algorithm.

The stream cipher has a 96-bit internal state split into three 32-bit values denoted key_0, key_1 and key_2. This internal state is updated every round using a plaintext byte to produce a single byte of keystream. Twelve bytes, denoted $p_1, \ldots, p_{12}$, are prepended to the plaintext in a matter of randomization of the produced keystream. The last (or two lasts for some versions of PKZIP) of those bytes, called the *checksum* byte(s), are dependent of the plaintext and are used as control bytes to detect wrong decryption (e.g. decryption process carried out with an incorrect password).

The algorithm producing the keystream byte is described in Algorithm 1. The notations used in this algorithm are the following. p_i and c_i denote respectively a byte of plaintext and a byte of ciphertext. P is the complete plaintext (including the twelve prepended bytes). $\oplus$ is the exclusive or, $|$ the inclusive or, $>>$ is a right shift and LSB and MSB denotes respectively the least and the most significant byte of a value. The CRC32 used in this algorithm is the classical cyclic redundancy check code used in this specific case with the reversed polynomial representations *0xEDB88320*.

Algorithm 1: Encryption(P, pwd).

initialize_internal_state(pwd);
for $p_i \in P$ **do**
  $c_i = p_i \oplus k_i$;
  $key_0^{(i+1)} = CRC32(key_0^{(i)}, p_i)$;
  $key_1^{(i+1)} = (key_1^{(i)} + LSB(key_0^{(i+1)}))$
    $*0x08088405 + 1$;
  $key_2^{(i+1)} = CRC32(key_2^{(i)}, MSB(key_1^{(i+1)}))$;
  $temp = key_2^{(i+1)} | 3$;
  $k_{(i+1)} = LSB(temp * (temp \oplus 1) >> 8)$;

The initialization phase corresponds to the derivation of the password into the initial state. The three keys key_0, key_1, and key_2 are respectively set to *0x12345678*, *0x23456789* and *0x34567890*. The internal state is then updated with each byte of the password as it is done with each plaintext byte except that the keystream byte is not computed. We denote $(key_0^{(1)}, key_1^{(1)}, key_2^{(1)})$ the obtained initial internal state considered to be the secret key of the stream cipher.

2.2 Related Attacks

Biham and Kocher (Biham and Kocher, 1994) have designed a known plaintext attack aiming at retrieving the secret key of the stream cipher. Once in possession of the internal state, the encrypted archive can be fully decrypted, as well as any other archive encrypted with the same password. It requires the knowledge of some consecutive bytes, at least twelve, of the plaintext that is encrypted in the archive.

This strong requirement is not that inconceivable in practice. The filenames contained in the archive are always accessible even when the archive is encrypted. If headers of known file types can be extracted or if known files (e.g. system files) are included into the archive, this knowledge can be used to perform the attack. Michael Stay later on softened this knowledge requirement in his article (Stay, 2001) at a price of a higher complexity of the attack and/or a minimum number of files. Jeong et al (Jeong et al., 2011) have also exploited the presence of several files to reduce the complexity of the attack of Biham and Kocher.

In all cases, the retrieved internal state does not allow the direct recovery of the password that was used to encrypt the archive. An additional process needs to

be undertaken to retrieve this password with a complexity of $2^{8(l-6)}$ where l is the length of the password. This means that if the password has a maximum of six characters[2], it is retrieved instantly.

In case there is no knowledge about the plaintext inside a zip archive, the only way left to retrieve the password is by carrying out an exhaustive search over all possible password candidates. This is done by trying a password, decrypting the data and checking if the CRC32 checksum for the file matches.

There are multiple tools currently supporting PKZIP attacks in CPU, for example John The Ripper (JohntheRipper, 2018) and also proprietary tools like Passware (Passware, 2017) or ARCHPR (Elcomsoft, 2018). Having only CPU implementations available puts a limit in the length of the password that is feasible to recover[3]. So far, no efforts were done in implementing the algorithm in GPU, as it was hypothesized that a GPU would not offer a meaningful gain in performance compared to existing implementations[4].

3 RETRIEVING THE PASSWORD FROM THE INTERNAL STATE

This section introduces our high-performance GPU implementation to recover the password from the internal state resulting of the known-plaintext attack of Biham and Kocher (Biham and Kocher, 1994).

3.1 Determining the Internal State

The known-plaintext attack of Biham and Kocher requires the knowledge of $n > 12$ consecutive bytes of the plaintext, together with the corresponding ciphertext. This position does not matter as long as they are consecutive. Indeed, once the internal state of a given round is known, it is possible to go backward or forward to decrypt the full archive or retrieve the initial internal state.We briefly describe the attack below and refer interested readers to the original article (Biham and Kocher, 1994) for further details.

The strategy of the attack is to determine a reduced list of sequences of internal states using the knowledge of the plaintext and exploiting the linearity of most of the functions used in the key update. This list must be composed so that: i) the correct sequence, that is the one produced by the correct password, belongs to the list, ii) there exists a discriminator allowing to identify it, iii) the list is small enough to be explored in reasonable time. We will use the following notations to describe the attack. We will denote the known plaintext bytes as $p_1, \ldots, p_n$, the corresponding ciphertext as $c_1, \ldots, c_n$ and the keystream used to produce the ciphertext as $k_1, \ldots, k_n$. By construction, the relation $c_i = k_i \oplus p_i$ always hold.

The $temp$ value used in Algorithm 1 is made of the 16 least significant bits of key_2 where the two least significant ones are forced to one. Consequently, the keystream byte k_n is only determined by 14 bits of $key_2^{(n)}$. The function producing this keystream byte is therefore mapping a space of size 2^{14} to a space of size 2^8 as a single byte of keystream is produced by this function. This function is balanced so each keystream byte will have exactly $2^6 = 64$ preimages in the space 2^{14}. The 16 most significant bits of key_2 do not influence at all the byte k_n, therefore no information can be guessed on those bits and, consequently, all possible combinations for those bits should be considered at this stage. The two least significant bits are useless, and are ignored for the moment. To summarize, we have defined a list $\mathcal{C}$ containing $2^{22} (= 2^6 \times 2^{16})$ candidates for $key_2^{(n)}$.

The $CRC32$ function can be inverted, thus we can express $key_2^{(n-1)}$ as follows.

$$key_2^{(n-1)} = CRC32^{-1}(key_2^{(n)}, MSB(key_1^{(n-1)})) \quad (1)$$

A list of 64 possible values for 14 bits of $key_2^{(n-1)}$ can be determined following the previously described approach. The 22 most significant bits of the right part of the equation can be determined for each candidate $key_2^{(n)}$ in $\mathcal{C}$ as the $MSB(key_1^{(n-1)})$ only influences the least significant byte. A single value $key_2^{(n-1)}$ out of the 64 possible values will match the corresponding bits of the right part of the equation. Consequently, we can associate to each candidate in $\mathcal{C}$ a value $key_2^{(n-1)}$, for which we know 30 bits (22 bits from the right parts plus the 14 from the keystream minus the 6 that are in common). We can repeat this operation for each round until we reach the first one. We therefore know a list of 2^{22} sequences for the key_2 from round 1 to n, denoted abusively $\{key_2^{(1..n)}\}$ in the following for clarity reason.

Using the value $key_2^{(1)}$, we can complete the 2 missing bits of $key_2^{(2)}$ and similarly for other key_2 val-

[2]To be correct, we should consider the password in bytes instead of characters. Therefore, most ZIP encryption softwares only accept ASCII passwords in which each character is encoded by a single byte.

[3]E.g. for covering length 7 of the ASCII readable range (95 chars) it would already need more than three weeks on a normal CPU.

[4]As discussed in https://hashcat.net/forum/thread-5709.html and https://github.com/magnumripper/JohnTheRipper/issues/718

ues up to round n. Then, from each value $key_2^{(i)}$, we can determine $MSB(key_1^{(i+1)})$ by still using the equation. The list $\mathcal{C}$ is now containing 2^{22} sequences $\{MSB(key_1^{(3..n)}), key_2^{(2..n)}\}$.

Given $MSB(key_1^{(n)})$ and $MSB(key_1^{(n-1)})$ from each candidate in $\mathcal{C}$ and given Equation 2, we can calculate 2^{16} values of $key_1^{(n)}$ increasing the size of $\mathcal{C}$ up to 2^{38} sequences of candidates.

$$(key_1^{(n)} - 1) = \frac{key_1^{(n-1)} + LSB(key_0^{(n)})}{0x08088405} \pmod{2^{32}} \quad (2)$$

Exploiting Equation 2, we know the 24 most significant bits of $key_1^{(n-1)}$ for each candidate in $\mathcal{C}$. The last byte can be retrieved thanks to the knowledge of $MSB(key_1^{(n-2)})$, determining at the same time the value $LSB(key_0^{(n)})$. We consequently know at this point a list of 2^{38} sequences $\{LSB(key_0^{(5..n)}), key_1^{(4..n)}, key_2^{(2..n)}\}$.

A value key_0 can be reconstructed using the least significant bytes of four consecutive values thanks to the linearity of the CRC32 function. Such value can also be determined by its predecessor or successor when the corresponding plaintext is known. As a consequence, $key_0^{(n-3)}$ can be computed for each sequence in $\mathcal{C}$. The previous values of key_0 can then be derived from this value and compared with the corresponding LSB contained in the sequence. Considering five of those bytes, it is certain that a fraction of 2^{-40} sequences will match. As a consequence, as long as we have a list containing less than 2^{40}, a single list should be output by this process. In practice, the list of candidates can be reduced by at least a factor of 2^8 because of redundant values, therefore requiring only 4 bytes of key_0 for the matching process.

Once in possession of a single candidate, and therefore one triplet (key_0, key_1, key_2), the cipher can be rewind into its initial state. Such state is sufficient to decrypt completely the archive and any archive that has been encrypted using the same password.

3.2 Retrieving the Password

Biham and Kocher describe in the original article (Biham and Kocher, 1994) how to retrieve the password from the initial internal state of the stream cipher with a complexity of $2^{8(l-6)}$ where l is the length of the password allowing the instant recovery of passwords of length lower or equal to 6. For passwords of length $l > 6$, the last $l-6$ characters need to be retrieved with traditional password guessing techniques (i.e. exhaustive search, dictionary with mangling rules...), whilst the 6 first characters are determined automatically for each evaluated candidates using construction properties of the stream cipher.

The step-by-step approach of the attack is summarized in Figure 1 where $(key_0^{(1)}, key_1^{(1)}, key_2^{(1)})$ denotes the initial internal state retrieved by the known plaintext attack. To simplify the notations, we denote abusively $(key_0^{(0)}, key_1^{(0)}, key_2^{(0)})$ (the subscript should be $1-l+6$) the internal state that has been obtained by *rewinding* the internal state using the $l-6$ last characters of the evaluated candidate. This initial step, also called step 0 later, is for example depicted in Figure 1 where the $(key_0^{(0)}, key_1^{(0)}, key_2^{(0)})$ is calculated from the internal state $(key_0^{(1)}, key_1^{(1)}, key_2^{(1)})$ using the last three letters from the candidate. The following steps are aiming at retrieving the missing values of the intermediate internal states to a point where the first six characters can be determined for the evaluated candidate. The last step of this approach is therefore to check if the obtained l characters match the known initial internal state $(key_0^{(-6)}, key_1^{(-6)}, key_2^{(-6)})$.

Step 1: $key_1^{(-1)}$ can be computed using the regular update function of $key_0^{(0)}$ and $key_1^{(0)}$. We cannot go further at this stage as we do not know $key_0^{(-1)}$. $key_2^{(-1)}$ can be computed using $key_1^{(0)}$ and $key_2^{(0)}$. As we already know $key_1^{(-1)}$ we can also compute $key_2^{(-2)}$.

Step 2: We do not know the most significant byte of $key_1^{(-2)}$, yet we can determine the three most significant bytes of $key_2^{(-3)}$ and continuing to determine the two most significant bytes of $key_2^{(-4)}$ and the most significant byte of $key_2^{(-5)}$.

Step 3: We can use the partial knowledge of the previous key_2 to determine the most significant byte of $key_1^{(-5)}$ to $key_1^{(-2)}$. Those bytes are sufficient to compute values $key_2^{(-5)}$ to $key_2^{(-3)}$.

Step 4: This step will reconstruct the missing values key_1 and the lowest significant bytes of the key_0. For this reconstruction, we are using the following equation derived from the key update algorithm of the stream cipher, where $const = 0x08088405$.

$$MSB\left(\frac{\frac{key_1^{(-1)}-1}{const} - 1}{const}\right) \quad (3)$$

$$= MSB(key_1^{(-3)}) + MSB\left(\frac{LSB(key_0^{(-1)})}{const}\right) \quad (4)$$

As we know the most significant byte of $key_1^{(-3)}$ and we can compute the left part of the equation as well, we can obtain the second value of the right part of the equation. We then use a lookup table to check

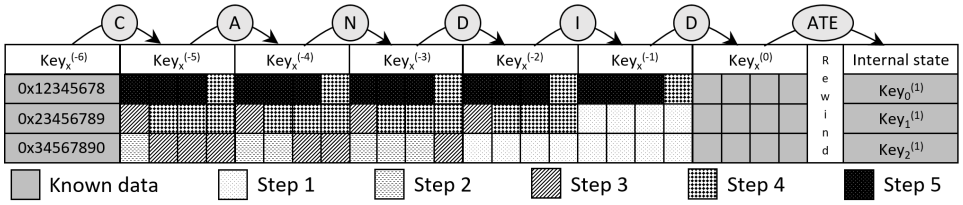

Figure 1: Summary of the Process Recovering the Password.

if there exist pair of values $(key_1^{(-2)}, LSB(key_0^{(-1)}))$ that would generate such value as done in libzc (Ferland, 2018). If not, we can abort the search for the candidate and start again in step 1 for the next candidate. If there are one or two of such pair (there cannot be more than two), we reconstruct the next pair of values in a similar fashion until either all the values key_1 and the lowest significant bytes of the key_0 are either retrieved or all the branches lead to a failure.

Step 5: Thanks to the linearity of the CRC function we can extract the six first bytes of the password candidates and reconstruct totally the key_0 values. If the reconstructed value $key_0^{(-6)}$ matches the initialization value, then we have found a valid password candidate.

This password guessing process allows to reduce the keyspace in a noticeable manner. For example, all the possible password candidates of length 10 (a final space of 95^{10} candidates) are tested with a maximum of 95^4 evaluations. As it will be exhibited in Section 3.4, a space of 14 characters can be explored in a reasonable amount of time with a single modern GPU.

3.3 OpenCL Implementation

There is no known GPU implementation available for retrieving the password from a given internal state. We implemented a high-performance OpenCL kernel to run the attack on GPU. We have used Hashcat (Steube, 2018) to run our OpenCL kernel, as it is open source and offers a well designed structure for GPU implementations of hash algorithms.

To allow the derivation of the 6 first bytes, we had to do the calculation backwards starting from the internal state (*key0*, *key1*, *key2*) and trying to reach the initial state (*0x12345678*, *0x23456789*, *0x34567890*) which would show that we have the right password.

As table lookups heavily affect GPU performances, we tried to keep them to a minimum. We needed both the CRC32 and INVCRC32 lookup tables and additionally another one to be used in the step 4 (see Subsection 3.2). By optimizing the way in which the lookups are stored and used, we were able to shrink the lookup table to one third of its original size which fits into the fast memory of the GPU cores. In contrary to the more logical approach, we encapsulated all iterations happening in step 4 into nested loops (depth-first) to avoid having to save intermediary data and being able to abort earlier.

3.4 Results

We compared our new GPU implementation to existing applications, namely pkcrack and libzc. There are other CPU applications available being able to run the known plaintext attack, but none of them is able to just take the internal state to retrieve the password. In the case of pkcrack we also modified it to run on multiple cores as it was only implemented for single core to have one of the single-core applications compared to our kernel running on CPU.

An Intel Xeon CPU E5-2623 3.00GHz was used to benchmark the CPU implementations. A Nvidia GTX 1080 as well as a 1080 Ti were used to benchmark the GPU variants. Figure 2 shows the speeds, in terms of number of candidates evaluated per second, during the password guessing process. Note that this speed gives the number of candidates tested, excluding the recovery of the 6 additional bytes. Even running on the CPU with OpenCL our implementation is faster than the available applications. Performance

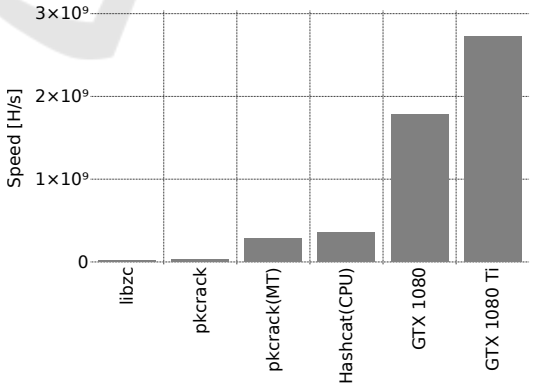

Figure 2: Speed comparison of existing implementations.

on GPUs is noticeably higher than on CPUs. The full readable ASCII range (95 characters) can be exhaustively explored for passwords up to length 14 within a reasonable amount of time, less than 30 days to be precise, using a single GTX 1080 Ti. Having 30 of

them would complete the same process in less than a day. Considering one hundred cards, performing the exhaustive search of the full readable ASCII range for passwords up to length 15 is achieved in approximately 27 days.

4 HIGH PERFORMANT PKZIP ATTACK IMPLEMENTATION

When the stream cipher internal state of a PKZIP archive cannot be retrieved, there is the possibility to retrieve the password using a general PKZIP attack.

4.1 PKZIP Hash Format

In order to crack a PKZIP password with John the Ripper, the relevant information first needs to be extracted from the archive and saved in a hash format which is readable by John afterwards. This is done with the utility zip2john giving a hash in the following format (JohntheRipper, 2017):

```
$pkzip2$C*B*[DT*MT{CL*UL*CR*OF*OX}
        *CT*DL*CS*TC*DA]*$/pkzip2$
```

Note that the part of the hash encapsulated here with the curly brackets is not always present, but only when the data type of the file is 2 or 3. The size of the PKZIP hash can vary depending on the amount and size of the files inside the zip archive. The following elements are the most important in our case:

C Hash Count: Specifies how many files are contained between the square brackets.

B Bytes: Number of valid bytes (1 or 2) in the checksum. The Linux command line zip saves 2, all other programs only 1.

CR The CRC32 checksum of the plaintext file allows to check the consistency of the extracted file.

CT The encrypted file data can either be compressed (value 8) or just stored in plaintext (value 0), depending if the plaintext was considered to be compressible or not (or the user creating the archive manually enforced no compression).

CS and TC Depending on the tool with which the archive was created there are either 2 or just 1 checksum bytes present. These contain the last 2 respective 1 bytes of the IV for the file.

DA If the file inside the archive is not too big, the full file data is included in the hash, otherwise the data is only referenced by the zip's filename. These bytes include the 12 IV bytes at the beginning, which can be discarded after the decryption. If the file was also compressed, the data has to be inflated to retrieve the original data. If the DT is 1 or 3 only the first 36 bytes are present.

4.2 Retrieving the Password

A straightforward approach for retrieving the password used to encrypt a PKZIP archive would be: create the internal stream cipher out of the password candidate, decrypt the full file data with the stream cipher, inflate the data (if it is compressed), compute the corresponding CRC32 checksum and finally check if it matches the one provided in the hash. However, inflating and checking the CRC32 are costly operations with a cost increasing with the size of the file(s). Several checks can be performed to detect if the process can be aborted at an earlier stage, therefore speeding up the overall process. John the Ripper has already implemented such checks in the CPU implementation. We have reused and adapted some of them in our OpenCL implementation.

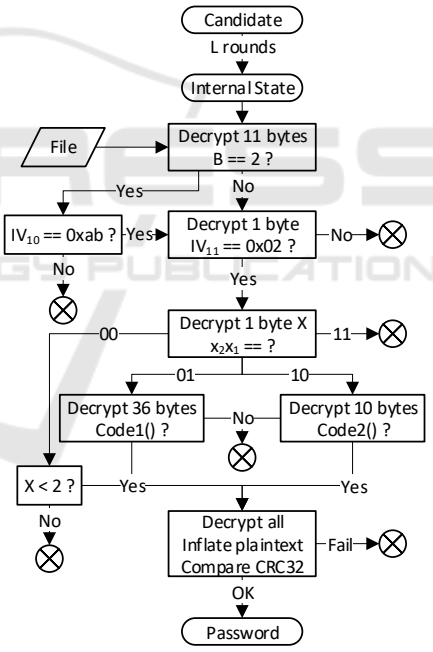

Figure 3: Password candidates evaluation process.

Figure 3 shows the order of the checks which are executed when attacking PKZIP. The circled crosses shows when the process can be aborted and started again with another candidate. The CODEx and Inflate checks are only possible if the data is compressed with deflate, if we have a non-compressed file, after the first IV checks, we have only the possibility to fully decrypt the file and check the CRC32 checksum. We use the term *encoding method* in the following de-

scription to refer to the 2nd and 3rd bit of the first byte of the compressed data, which is defined in the deflate compression specifications.

IV Check 1. If possible (when B is 2), after decrypting 11 bytes of the IV, we can check if it matches one of the second bytes of the provided checksums (CS and TC). If not, we abort.

IV Check 2. After decrypting 12 bytes of the IV, we can check if it matches one of the first bytes of the checksums (CS and TC). If not, we abort.

CODE0. If the encoding method is 0 (raw/stored block), the only bit which is allowed to be set, is the first one. We abort if any other bit is set.

CODE1. If the encoding method is 1 (static Huffman), we can check if there is a proper encoding present in the next 36 bytes. If not, we abort.

CODE2. If the encoding method is 2 (dynamic Huffman), we check if the next 10 bytes contain valid encoded data, otherwise we abort.

CODE3. If the encoding method is 3, we abort as such value is reserved and should not be used.

Inflate. If the inflation algorithm reports any problem in reading the data, we abort.

CRC32. We calculate the CRC32 of the inflated full file data and check if it matches the checksum provided. If yes, we have found the password.

Empirical results of our implementation have shown the proportion of candidates for which we can abort at an earlier stage because of invalid checks. We have to distinguish between the two cases where we either have one or two checksum bytes. Table 1 shows the rejected percentages when having two checksum bytes, Table 2 shows the rejected percentages when only having one checksum byte. When the hash provides two checksum bytes, it allows to reject more candidates at an earlier stage of the process, which avoids all the more costly checks.

Table 1: Number of rejected candidates per check for 2-byte hashes (Total 543'257'459 candidates).

Check	Candidates	Rejected
IV Check 1	543257459	541136146 (99.6%)
IV Check 2	2121313	2113156 (99.6%)
CODEx	8157	8069 (89.9%)
CODE0	2024	1948 (96.2%)
CODE1	2054	2050 (99.8%)
CODE2	2026	2018 (99.6%)
CODE3	2053	2053 (100%)
Inflate	88	83 (94.9%)
CRC32	5	4 (80.0%)

Table 2: Number of rejected candidates per check for 1-byte hashes (Total 543'257'459 candidates).

Check	Candidates	Rejected
IV Check 1	543257459	– (–%)
IV Check 2	543257459	541136764 (99.6%)
CODEx	2120695	2099858 (99.0%)
CODE0	530845	514353 (96.8%)
CODE1	530247	527827 (99.5%)
CODE2	529821	527896 (99.6%)
CODE3	529782	529782 (100%)
Inflate	20837	18984 (91.1%)
CRC32	1853	1852 (94.8%)

4.3 OpenCL Implementations

The first challenge to have the PKZIP cracking process in OpenCL was the ability to inflate data. In CPU implementations the libzip library bindings[5] can be used to achieve this, but in OpenCL this is not possible. The libzip implementation is quite complex and heavily dependent on all the components in the library. Miniz (Miniz, 2018) is an alternative implementation that is very compact and licensed with MIT. Only the functions which were required to do an inflation were extracted and included in our OpenCL kernel. We had to do some minor modifications to the code to get it working in OpenCL to overcome some specific casting of pointers which crashed the kernel.

Due to the variety of inputs (from the hash format) output by zip2john, the implementation looks different in OpenCL, as some checks/parts are only needed for specific cases. To always benefit from the most optimal implementation for the type and get the best possible speed, we decided to split all the hash variants into three families.

Compressed. If there is a single compressed file in the archive the normal attack described in Figure 3 and Section 4.2 is performed.

Uncompressed. If there is a single uncompressed file in the archive, we cannot benefit from the CODEx checks but we also do not have to include the inflate code in the kernel. Therefore, the kernel is running slower (depending on the file size) but is also cleaner and does not require the CODEx lookup tables.

Multifile. The process can use the 1 (or 2) byte checksums of the files without knowing more than the first few bytes of each file. John The Ripper already proposed a similar approach for exactly 3 files with 2 byte checksums available. The size of the output space, namely $(2^8 \times 2^8)^3 = 2^{48}$, is relatively small. Therefore, this approach would quickly lead to collisions with our GPU implementation. For example, a col-

[5]The libzip library is widely used to handle/modify/create zip archives and is provided as a C implementation which can be used by many other languages and applications.

lision would be found in less than an hour using 30 GTX 1080 Ti. Our kernel consequently accept up to 8 files as input with 1 or 2 checksum byte reducing the chance of having a collision.

4.4 Results

We compared the high-performance GPU implementation to the existing PKZIP password retrieval applications. There are no other known implementations of this attack on GPU. We used the same hardware for the benchmark, namely an Intel Xeon CPU E5-2623 3.00GHz, a GTX 1080, and a GTX 1080 Ti. In order to have a fair comparison between the available CPU implementations and our OpenCL implementation we also ran Hashcat with only using the CPU OpenCL device (-D 1) and all runs were executed with the same hash. As Figure 4 shows, even on the CPU, the OpenCL implementation is faster than the other applications. Using the hashcat kernel on GPUs we can further increase the speed compared to todays available PKZIP password retrieval solutions by a factor of more than 16, comparing the fastest CPU implementation to a GTX 1080 Ti.

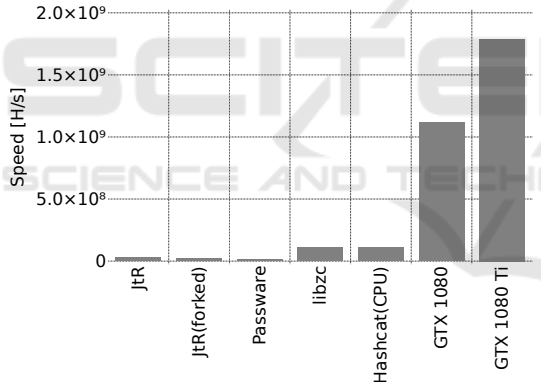

Figure 4: Speed of PKZIP cracking implementations

5 CONCLUSION

In this paper we have presented the first GPU implementations for the recovery of PKZIP stream cipher passwords. Our results show that our GPU implementations are able to outperform even the fastest existing implementations by at least one order of magnitude. These results invalidate the existing hypothesis that GPU implementations would not outperform CPU implementations significantly.

When enough known plaintext is available, our implementation can recover passwords up to 15 characters long in practical time. For example, a 15-characters long password can be recovered in less than 27 days using one hundred Nvidia 1080 Ti GPU. To make the parallel with the fastest CPU implementation, it would require one thousand Xeon E5 CPU to recover such password within the same amount of time. This offers a higher chance for forensic analysts to recover the password of an encrypted archive.

In addition to the five kernels presented in this article, further implementations could be done in the future to increase the performance of the attack by targeting specific scenarios. As an example, separate kernels could be developed to handle different maximum sizes of encrypted files. This further specialization of the kernels could allow to use the available computational resources more efficiently potentially leading to additional performance gains.

REFERENCES

Biham, E. and Kocher, P. C. (1994). A known plaintext attack on the pkzip stream cipher. In *International Workshop on Fast Software Encryption*, pages 144–153. Springer.

Elcomsoft (2018). https://www.elcomsoft.de/archpr.html. [Accessed Sept-2018].

Ferland, M. (2018). https://github.com/mferland/libzc. [Accessed Septr-2018].

Jeong, K. C., Lee, D. H., and Han, D. (2011). An improved known plaintext attack on pkzip encryption algorithm. In *International Conference on Information Security and Cryptology*, pages 235–247. Springer.

JohntheRipper (2017). zip2john utility. https://github.com/magnumripper/JohnTheRipper/blob/bleeding-jumbo/src/zip2john.c. [Accessed Sept-2018].

JohntheRipper (2018). John the ripper is a fast password cracker. https://github.com/magnumripper/JohnTheRipper/. [Accessed Sept-2018].

Miniz (2018). miniz: Single c source file zlib-replacement library. https://github.com/richgel999/miniz. [Accessed Sept-2018].

Passware (2017). Passware kit forensic. https://www.passware.com/kit-forensic/. [Accessed Sept-2018].

Stay, M. (2001). Zip attacks with reduced known plaintext. In *International Workshop on Fast Software Encryption*, pages 125–134. Springer.

Steube, J. (2018). World's fastest and most advanced password recovery utility. https://github.com/hashcat/hashcat/. [Accessed Sept-2018].