uc3m | Universidad Carlos III de Madrid

Master Degree in Cibersecurity
2021-2022

Master Thesis

Avoiding compromised passwords in Django using AMQ-Filters

Miguel González Saiz

Tutor: Pedro Reviriego Vasallo

Avoiding compromised passwords in Django using AMQ-Filters

Miguel González Saiz

Abstract—Using passwords to access personal accounts, whether for social networks, online banking or other services, has become an indispensable habit in today's world. This fact makes passwords an attractive target for cybercriminals, who look for any possible vulnerability in order to obtain them. The National Institute of Standards and Technology, in one of its latest publications, refers to the increase in data breaches in which hundreds of millions of passwords are exposed and warns the danger of having an account with compromised credentials, urging to verify at each account creation that the passwords chosen by users do not belong to previous breach corpuses. This may sound simple, but the large number of known hacked passwords makes a sequential or any other exhaustive check completely unfeasible. Fortunately, there are data structures called Approximate Membership Query (AMQ) that can serve as filters and be a good solution to act as verifiers since they are characterized by not returning false negatives while being fast and using little memory. In this work a search among the AMQ filter family aiming for the most memory efficient one will be performed, followed by an optimization of its existing implementation with the use of Intel's AVX2 intrinsics. Subsequently, this implementation will be integrated into a module of a well-known web application framework such as Django, which despite its multiple advantages and widespread use, lacks a proper compromised password validator to deal with this worrying problem. In this way, Django is provided with the necessary resources to follow the NIST recommendations.

Index Terms—Django module, Django validator, Django applications, pwned passwords, AMQ filters.

1 Introduction

1.1 Motivation

Nowadays, it is well known that passwords [1] have become one of the most valuable resources in our lives. This is simply because they are used to gain access either to specific data, an account, a computer system or a protected space [2]. One of the most common uses for passwords is online accounts, such as those related to social networks, banks, as well as to different subscribable services such as OTT [3] (over-the-top) services.

Due to their high value, they become a very juicy target for cybercriminals, who put all their efforts into trying to steal them either directly from users, especially through different phishing techniques [4], or even by performing specific attacks to different server databases [5] where these passwords are stored. In both cases obtaining passwords in clear text is not difficult at all, in the first case it is straightforward and in the second, despite the fact that passwords in server databases are usually stored in hashed format, this is not necessarily a problem for these criminals, given that with enough time and resources, brute force tools such as *John the Ripper* [6] or *Hashcat* [7] can crack certain hashes in a relatively short time.

This interest in stealing passwords or other information from online server databases is not new, but the number of data breaches has grown considerably in recent years, probably due to the rise of the Internet and data storage on cloud servers. This reality affects all types of companies, from the smallest to the most powerful. In fact, some of the latest leaks [8] have affected such well-known and prestigious companies as Facebook, Twitter, Yahoo, LinkedIn, Adobe, Zoom, Twitch, or eBay.

To address this issue, the National Institute of Standards and Technology (NIST), in its Special Publication 800-63b [9], refers to recommended requirements for *memorized passwords*, which are defined as

> secret values intended to be chosen and stored by the user

, and establishes guidelines for password verifiers. The only requirement in terms of format for these user-memorized passwords (which are the most common for online accounts) is that they should consist of at least 8 characters, as demanding greater complexity has been shown to lead to major security issues. On the other hand, it is stressed that verifiers must check whether passwords chosen by users are compromised, i.e. that they do not belong to previous breach corpuses [10].

Following the aforementioned NIST guidelines, any application should verify that its users' passwords were not obtained from any of the existing leaked passwords lists. As regards this, one of the complications could be to gather all those compromised passwords in a common list, since there occur many different leaks [8]. However, there is already an entity called *haveibeenpwned* [11] that provides in an easy and non-profit way a zip file gathering all these passwords. Since this file contains hundreds of millions of compromised passwords, the final complication lies in the verification, because it is completely unfeasible and unscalable to exhaustively check each user password against this endless list.

There are still platforms that do not have this type of verification recommended by NIST. Without going any further, Django [12] is one of the most well-known and widely used Python open-source web application frameworks and, despite having the infrastructure to validate passwords, it does not have any type of validator that verifies, prior to approval, whether a password is compromised. While it is true that the Django's community implements numerous modules in which they make available to users different novelties, there is only one similar in this regard. The web page *haveibeenpwned* has a rest API to which online queries can be made in order to check whether they are compromised or not. The existing module leverages this functionality and uses it to approve passwords on account creation, which allows for an easy and simple way to do this check, but at the same time can be an impediment for security reasons in many organizations or enterprise environments.

1.2 Objectives

The work of this article will consist in making a module for the Django's community to serve as an offline validator of compromised passwords. The target validator to be implemented will require a filter consisting of a data structure that can hold hundreds of millions of keys (the collected in a file by *haveibeenpwned*) and is as memory efficient and fast as possible. This type of structure will be sought and the one that best optimizes memory expenditure will be chosen, to later optimize it with the use of SIMD [13] type instructions and integrate it into a Python module for Django.

1.3 Methodology

In more detail, for solving this practical problem the following methodology has been applied:

1) Different types of filters that may be interesting to solve the problem due to their characteristics are presented and compared theoretically.
2) The most suitable one is selected according to the proposed requirements and its existing implementation is optimized, if possible.
3) Once optimized, the filter would be integrated into a module for Django and packaged for subsequent upload to Pypi and GitHub, in order to make it available to the community.

1.4 Outline of research

The rest of the paper is organized as follows, in Section 2 a brief overview of Django's framework and of the approximate membership filters is discussed. Next, in Section 3 the proposed design containing the specific chosen data structure and its implementation is exposed followed by the evaluation of this one in Section 4. The integration of the implementation in a module for Django is explained in Section 5. The paper finishes, in Section 6, with the conclusions of the work done including some possible proposals to be carried out in the future.

2 Preliminaries

2.1 Django Framework

Django is an open source web development framework, which is written in Python and complies to some extent with the Model-View-Controller (MVC) paradigm. It was developed to manage several news-oriented pages of the Lawrence Journal-World in Kansas, and was released to the public under a BSD [14] license on July 1, 2005. Python language is used in all parts of the Framework, including configurations, files, and data models.

The fundamental goal of Django is to facilitate the creation of complex websites by emphasizing reusability, component connectivity and extensibility, rapid development, and the *Don't Repeat Yourself* (DRY) principle [15]. Moreover Django provides a CRUD (create, read, update and delete) [16] administrative interface option that is dynamically generated through introspection and configuration via admin models. Some sites have been using Django for a long time, such as the Spotify website [17], which is a site from which you can listen to music over the Internet without having to buy or download it. In addition, its application allows to manage the user's music libraries from any digital device.

Django's origins as a news page manager are evident in its design, as it provides a number of features that facilitate the rapid development of content-oriented pages. For example, instead of requiring developers to write controllers and views for the administration areas of a page, Django incorporates an application to content management. This application can be included as part of any page made with Django and can manage multiple pages made with Django from a single installation. This application allows the creation, update and deletion of content objects, keeping track of all actions performed on each one.

Furthermore Django provides an interface for managing users and user groups (including a detailed assignment of permissions). In addition to the files discussed above Django also includes a URL redirection system, a commenting system, an independent and lightweight web server for development and testing, a caching environment that can be used for any of the caching methods, a serialization system that can produce and read XML (eXtensible Markup Language) and JSON (JavaScript Object Notation) representations of Django model instances, and much more.

The core of Django consists of a mapping between object and relationship that exists between:

- Data models (defined as Python classes) and a relational database (*model.py*).

- A system to process the requests with a web template system (*Views.py*) and a regular expression obtained by URL (*Controller*).

Django appears to implement the MVC pattern, but still there are some differences due to the fact that Django's design was not intended to be tied to anything in particular, but rather to develop a tool that would work as well as possible. The truth is that for Django the view describes what data will be presented and not how it will be seen. The formatting of the data is where the templates come into play. That is the reason why the MVC "controller" in Django is called "view" and the MVC "view" is "template".

Considering this architecture, the following is an outline of how a request is processed in Django. When Django receives a request, the first thing it does is to create an *HttpRequest* object that represents it and that will serve as an abstraction to work on different servers. Then the URL resolution is performed. This consists in selecting the function of the view (of the URL specified in the request) that will participate in the creation of the response. Once the function that will resolve the specified URL is resolved, the view function is invoked with the request as the first parameter. This is where the heavy lifting is done, such as database queries, template loading and HTML generation, resulting in an HttpResponse object or exception.

In addition, Django provides three different points where it allows running Middleware classes, previously defined in the configuration file. The same class can be executed in more than one point, these are the options:

- **Request middleware:** runs after creating the HTTP request object, but before resolving the URL, allowing to modify the request object or return a response of its own before the rest of the application is executed.
- **View middleware:** executes after URL resolution, but before executing the corresponding view. It allows operations to be executed before and after the execution of the view. The view may not be executed at all.
- **Response middleware:** it is executed at the end, after creating the response object and before delivering it to the client. It is used to make the final modifications.

Figure 1 shows a schematic of the described operation for a better understanding.

Django has become very popular in recent times, to the point that more than 80,000 different websites around the world use this framework as a base [18], probably because of the facilities it offers. Among the many advantages that make it a winning choice, the following stand out:

1) Its particularly attractive ORM (object relational mapping) [19], which allows a very practical interaction with a database.
2) Its administration interface, which makes it possible to manage application data with very little additional work.
3) The wide selection of modules and libraries, both official and provided by the community, which allows incorporating all kinds of functionalities, such as providing an API (application programming interface).

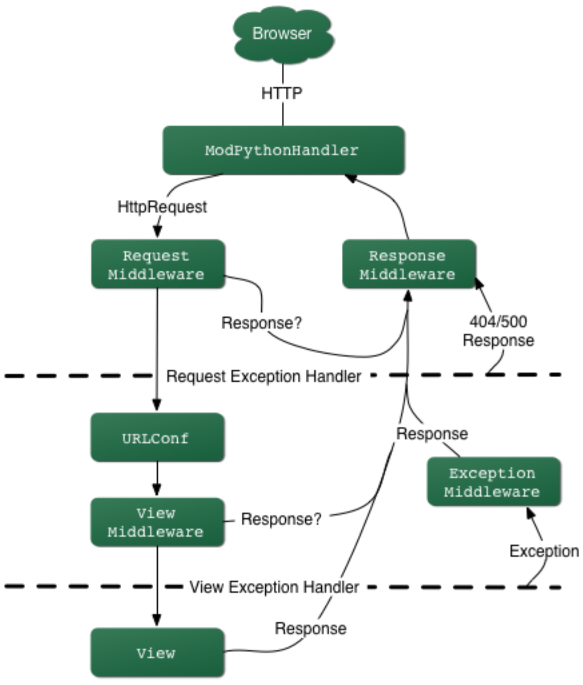

Fig. 1. Diagram summarizing the main inner workings of Django.

2.2 Approximate Membership Query (AMQ) Filters

AMQ filters arise from the following need, also known as Approximate Membership Query Problem:

> Given a large set of words W, we want to be able to quickly determine whether a word in query q belongs to the set, resulting impossible to get a $False$ value if it does belong. On the contrary, it would be valid to obtain a $True$ value with a word that does not belong to the set W [20].

To solve this, a new data structure is presented that allows to answer these queries very quickly, even for huge sets, if the words are not too long and the query is close enough. One useful application is to limit password guessing by verifying, before a password is approved, that it is not too close to a dictionary word [20].

These data structures called Approximate Membership Query (AMQ) also give name to the filters that use them. The interface they provide for representing sets, in which elements can be inserted, queried and, depending on the use case, also deleted, is simple. A query returns a true value if the element has been previously inserted. Querying an element can return a true result even if it has not been inserted, which would be called a false positive. The probability that a filter, given a non-inserted element, will result in a false positive is called the false positive rate. This rate varies depending on the capacity of the filter, so it can usually be chosen when constructing the filter. The main advantages of these AMQ data structures over other set representations are that they are fast and space efficient. One disadvantage is that, like hash tables, most AMQ data structures have to be initialized with their capacity, which can become a problem for their space efficiency when the final size is unknown a priori.

The typical applications of AMQ-Filters are distributed systems and database systems, where these filters are used as an in-memory data structure to avoid costly disk accesses. The operation would be as follows (Figure 2): The AMQ-Filter functions as a proxy for the key set of a database or remote memory. When performing a query, the AMQ-Filter is used to approximate whether the key is in the set before the query is made to the database or remote memory. In this way, the slow query would only be performed if the AMQ-Filter returns a true result, so only a false positive result leads to an unnecessary I/O or remote access operation.

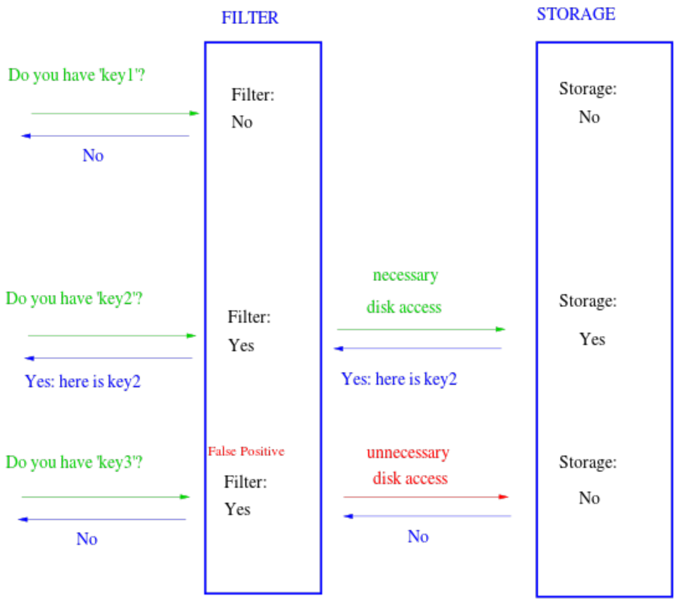

Fig. 2. AMQ Filters typical operation, in which only false positive results will lead to unnecessary I/O or remote accesses [21].

All AMQ-Filters support the lookup operation, which as mentioned before, once an element is inserted the lookup for this element must return true, but not all of them solve the problem in the same way. Each type of filter within this family has its configuration parameters that differentiate it from the rest, obtaining different performances usually either in search of a higher speed of construction and/or query or, on the contrary, a higher efficiency of size in memory. However, for all cases, there is a trade-off between storage size and false positive rate (FPR). Increasing the storage space reduces the false positive rate, although the theoretical lower bound is $log_2(1/fpr)$ bits for each element [22]. In any case, AMQ filters have different ranges of false positive rates and space requirements, a fact that makes the choice of the best filter application-dependent.

Some of the most popular AMQ filters are listed below along with a brief description of the filters:

- **Bloom filter**: A Bloom filter is a bit array of m bits with k hash functions. Each hash function maps an element to one of the m positions in the array. In the beginning, all bits of the array are set to zero. In order to insert an element, all hash functions are calculated and all corresponding bits in the array are set to one, whereas to lookup an element, all k hash functions are calculated and if all corresponding bits are set, true is returned. False positive rate an be reduced by increasing the number of hash functions and m [23].

- **Quotient filter**: The idea of Quotient filters is to hash an element and to split its fingerprint into the r least significant bits called the remainder d_R and the most significant bits called the quotient d_Q. The place in the hash table where the remainder is stored is determined by the quotient. In order to resolve soft collisions (same quotient but different remainders) additional three bits for every slot in the hash table are used. The space used by Quotient filters is comparable to Bloom filters, but Quotient filters can be merged without affecting their false positive rate [24].

- **Cuckoo filter**: Cuckoo filters are based on Cuckoo hashing [25], but only fingerprints of the elements are stored in the hash table. Each element has two possible locations being the second calculated based on the first location and the fingerprint of the element. This will allow moving elements already inserted, which would be necessary if the two possible slots for an element are full. The insertion speed of Cuckoo filters degrades after reaching a load threshold, as it is possible that an insertion fails, and the table must be rehashed [26].

- **Xor filter**: Xor filters [27] are based on a Bloomier filter [28] and use the idea of perfect hash [1] tables. As similar to Cuckoo filters, they save fingerprints

1. In computer science, a perfect hash function f for a set S is a hash function that maps distinct elements in S to a set of m integers, with no collisions.

of the elements in a hash table. The idea is that a query for an element x is true if the xor [2] of three given hash functions h_0,h_1,h_2 is the fingerprint of x. While building the hash table, each element is assigned one of its three slots in a way that no other elements are assigned to this slot. After all elements are assigned, it is set for each element the value of its slot to the xor of the two other (not assigned) slots of the element and the fingerprint of the element. Despite this hash table can be constructed using only 1.23 times the theoretical minimum FPR bits per element, the construction algorithm can fail and no dynamic insertions are possible without rebuilding the hash table. A disadvantage of this filter is that the data structure may have to be rebuilt if additional elements are added, making it viable only for applications where elements do not have to be added afterwards and space is of importance.

Despite the previous filters are the most well-known, probably because they were the first to appear, there exist also more modern versions of these ones that include some modifications and improvements:

- **Split Block Bloom filter**: The Split Block Bloom filter is a design of Jim Apple that achieves certain improvements in speed and space requirements over the original Bloom filter. It is based on the next ideas:
  1) Use block Bloom filters to reduce the number of cache lines to access down to one [29].
  2) Within each block, use a "split" Bloom filter that sets one bit in each of several sections, rather than several bits in one section [30].
  3) Use eight hash functions in order to fit cleanly into SIMD[3] lanes.
- **BinaryFuse filter**: The binary fuse filter is a variation on the xor filter. It is made of a fingerprint function, an array of fingerprints and a hash function from keys to locations in the array of fingerprints [31].
- **Ribbon filter**: Peter C. Dillinger together with Stefan Walzer introduce a faster, simplified and more adaptive Gaussian elimination algorithm (Ribbon) for the static function data structure from [32]. Based on Ribbon, they have developed a family of practical and highly space-efficient XOR filters [33].

2.3 Notation

When comparing these types of AMQ filters, as each has its own nomenclature and structure, it is difficult to use terms that mean the same thing for all filters down to the smallest detail. Therefore, even if there are small conceptual flaws, a common nomenclature will be used to avoid this document becoming too technical and precise. The idea is to have an overview of these filters and proceed to the choice of the best one for the intended application. Before proceeding to describe the proposed design, the notation used in the paper is summarized in Table 1.

2. Referred to the bitwise XOR: a binary operation that takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits.

3. Single instruction, multiple data (SIMD) is a type of parallel processing in Flynn's taxonomy. More information in [30].

3 Proposed Design

This section describes the proposed design, in the following order: The objectives of the work will be recalled and the filter with the ideal characteristics will be chosen. Subsequently, the optimizations achieved on the existing implementation of the filter will be explained.

3.1 Main objective

Recall that the main objective of this work is to provide a practical tool for Django, with which to verify whether a password is compromised or not before registering it in an account. In this way we would obtain a compromised password validator with which to follow the NIST recommendations. This tool will consist of an AMQ filter and the priority in its selection will be memory efficiency, i.e., we want to minimize the memory cost.

3.2 AMQ Filter choice

Given the requirements of the previous subsection 3.1, an AMQ filter should be chosen from those presented in Section 2.2 to address the problem and achieve the objectives. In one of the referenced articles, specifically the one by Dillinger and Walzer [33], a comparison of the fastest filters is made (Figure 3) where not only the speeds in construction and query are represented, but also the resulting overhead space of each filter, making it a splendid source for the choice of filter. This Figure 3 consists of an upper and two lower graphs. The first one shows the different filters and their corresponding capacities taking into account their false positive rate and their resulting overhead space in memory. The lower graphs show the construction and query speeds given the point clouds where the filters are located in the upper graph. A priori, the filter that uses the least memory is the Balanced Ribbon. However, the problem with this filter is that it is quite complicated to implement and the balance between occupied memory and speed is not very good, since, although it uses little memory, it has poor performance regarding the construction and query times, something that could become a problem in the target application. Nonetheless, Homogeneous Ribbon, the next best memory-optimizing filter, offers much more balanced construction and query times, so it has been considered a more viable option for integration into Django.

TABLE 1
Notation and terms related to AMQ filters used throughout this paper.

Symbol and/or term	Meaning and/or formula
fpr or FPR	Filter's false positive rate. It can be expressed per unit or as a percentage.
r	Number of bits of the filter's fingerprint. Typically $r = 8$.
$nkeys$	Number of keys with which the filter has been constructed.
ϵ	Excess bit space per key with respect to r. An ideal filter has $\epsilon = 0$
$Space$	The total bits per key occupied by the filter in memory. $Space = r * (\epsilon + 1)$
γ	Given a false positive rate, the minimum number of bits that the filter's fingerprint can consist of. $\gamma = log_2(1/fpr)$
Efficiency or η	Number of bits per key required according to the theoretical minimum divided by the number of bits per key actually used. $\eta = \gamma/Space$ For an ideal filter $\rightarrow \eta = 1$
Space overhead	Another way of expressing efficiency used by Dillinger in [33]. Space overhead $= Space/\gamma - 1 = 1/\eta - 1$ For an ideal filter $\rightarrow$ Space overhead $= 0$

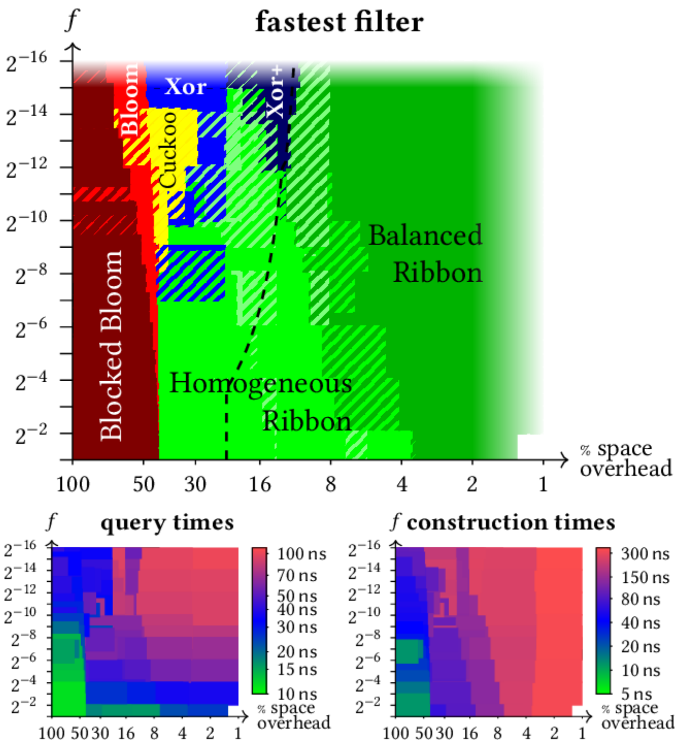

Fig. 3. Figure from [33]. Top graph: Fastest filter with the given combination of space overhead and false positive rate, considering a mix of construction and query times. Bottom graphs: Construction and query times for fastest approaches in top graph.

3.3 Homogeneous Ribbon Filter

Once the choice of the filter has been made, it only remains to examine the existing implementations in order to proceed with the optimization, but first it is important to know the exact operation of this type of filter.

All filters are constructed from a number n of keys, but each one has its own structure. The Homogeneous Ribbon filter is based on the creation of a linear and homogeneous system of equations whose solution will be the filter itself. This system of equations is solved in the Galois Field GF(2) [34], where the elements are binary and the used operations XOR and AND. Starting from a number n of keys, a matrix of dimensions n x m is formed, which contains the equations and whose coefficients are taken from the keys. In order for the equations to be linearly independent, the matrix has to be upper triangular, so there are a number of coefficients that are forced to 0 for this to be true. Specifically, from each key w bits are taken to make up the ribbon, and w' bits to obtain a ribbon's index indicating the exact row to place the ribbon, i.e., the position of the corresponding equation in the matrix. Figure 4 shows a schematic that may clarify the appearance of the matrix.

Once the matrix is filled in, the next step is to find not one but r solutions to the system. Obviously, the larger r is, the more solutions there will be and, therefore, the lower the false positive rate of the filter, although all at the cost of a higher memory occupancy.

Another feature of the coefficient matrix formation is that there is a possibility of collision between keys. This means that the bits chosen for the index of one key coincide with those of another key, so the position of the respective ribbons within the matrix would be the same. This problem is solved as follows:

1) Firstly, when choosing bits w and w' of the key, for the ribbon and the ribbon's index respectively, the first bit of the ribbon is forced to 1 before inserting it into the upper triangular matrix. This

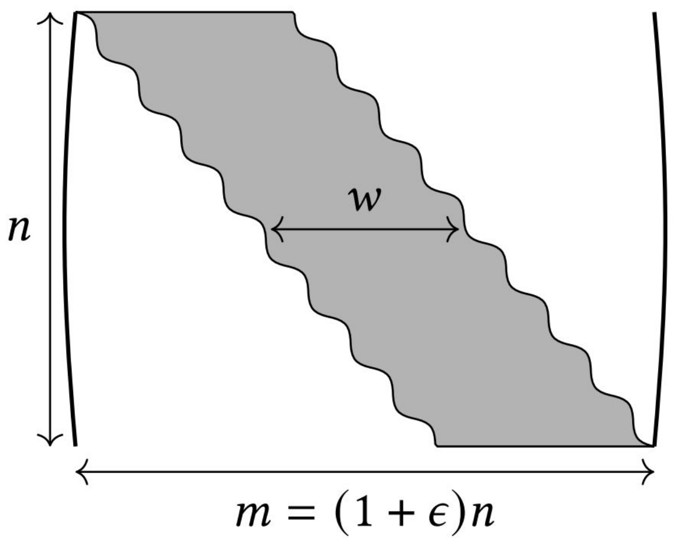

Fig. 4. Figure from [33]. A matrix of coefficients containing the ribbons, with w bits, extracted from the keys. This forms a linearly independent homogeneous system of equations whose solution will yield the Homogeneous Ribbon filter.

is a necessary condition for independent linearity between the equations.

2) Assuming that when inserting a ribbon $r1$ in the matrix, the position it should fill is already occupied by another ribbon $r2$, a linearly dependent ribbon $r2'$ is calculated between the two ribbon $r1$ and $r2$. This operation is performed with the binary operand XOR. The result of this operation shifts the position of the new ribbon $r2'$ as many positions downwards as the number of 0's in front of it. That is, $r2$ is no longer inserted into the matrix, but $r2'$, which is a linear combination of $r1$ and $r2$. The row in which this $r2'$ is inserted is x positions lower than $r1$, being x the number of bits to 0 that $r2'$ contains before the first bit to 1 (read from left to right obviously).
3) This process is repeated iteratively until an empty row for the corresponding $r2'$ is found. In the case of finally obtaining an $r2'$ without 1's, i.e. all 0's, this would mean that this equation is already contemplated in the matrix, since it is already a linear combination of the rest.

Once the system is solved and the filter is constructed, the way to verify if a key is contained in the filter is by means of a query. The queries are performed as follows:

1) From the key to be queried, the bits w and w' are chosen, to form the ribbon and the ribbon's index respectively, just as it was done in the construction process to insert into the matrix.
2) With the ribbon index, the r solutions of the equation formed by the ribbon coefficients are obtained. Subsequently, by means of binary AND and XOR operations between these solutions and the coefficients that form the consulted ribbon, 0 is obtained if the key is in the filter. If the key is not in the filter, 0 could also be obtained but with a probability of fpr.
3) It is important to note that queries have a false positive rate directly dependent on r, since the higher r, the higher the number of solutions and, therefore, the lower the probability of false positives. Recall that there is always a theoretical minimum and that for the case of $r = 8$, the minimum achievable false positive rate is $fpr = 1/2^r = 1/256 \approx 0.39\%$.

Finally, in Figure 4 it can be seen that $m = (1+\epsilon) * n$. This means that by varying the parameter ϵ we enlarge the matrix, obtaining as well a higher degree of freedom of the system of equations, and consequently increase the size occupied by the resulting filter. This parameter, together with r, will be of utmost importance in the configuration of the filter since the efficiency and false positive rate of the filter will depend mainly on them.

3.4 Implementation and optimization work

As already mentioned, once the choice of the filter with the most suitable characteristics has been made and the operation of the chosen filter, Homogeneous Ribbon, is understood, the implementation and optimization part of the filter remains. There is a GitHub repository [35] that contains the original code sources of this filter, that is, those of the inventor himself. These sources use C++ as language and are quite complex to understand due to the excessive use of templates. Even so, what can be perceived is that there is no use of Intel intrinsics. This makes real the opportunity to achieve a more optimized implementation.

Since the goal is to have absolute control over the code flow, and the only existing implementation of Homogeneous Ribbon is too complex to modify, it was decided to start from scratch and make an implementation in pure C to be optimized with Intel's AVX2 intrinsics. This has been possible thanks to the detailed explanation of the filter's algorithm in the paper [33]. Regarding the choice of the filter within the Homogeneous Ribbon, the implementation $w = 128$ bits has been chosen. The reason is that the filter is going to be constructed with the file provided from the [11] web page previously mentioned in the introduction, which contains the keys in SHA1 hash format. This format maps the keys into 160 bits, of which 128 bits (w) will be chosen for ribbon and 32 bits (w') for the ribbon's index.

At the time of programming the way of proceeding to structure the code has been the following one:

1) In this same repository [35] are also the sources of other filters such as BinaryFuse and Xor. These have been the model to follow as far as their structure is concerned. Most of the AMQ filters have mainly two methods, one that constructs the filter and fills the structure so that it remains in memory during execution, and another to perform queries to it. Despite the implementation that has been done contains several other methods, the first to be coded were these, the most important ones.
2) When the implementation of the most basic methods is finished, the construction method and the one that performs the queries, we check that it works as expected in order to proceed to add more functionalities and optimizations. So far all the code is done in C, but without without the integrations of the Intel intrinsics.
3) Finally, once the correct operation and therefore the correct understanding of the algorithm has been verified, we try to add functionalities to save unnecessary memory costs and modify the code to use Intel's AVX2 intrinsics.

Starting with added functionality to save memory costs, some of these enhancements include:

- Most AMQ filter implementations found in [35] construct the filter in two steps: they first load the entire compromised keys file into memory and then iterate these memory-hosted keys for the filter's construction, which also incurs a memory cost. In the proposed implementation, the construction is not performed on the raw key file of [11], but on an already preprocessed one containing the keys in binary format, thus saving all the memory involved in having the key file preloaded.
- Moreover, in the construction, at least in the Homogeneous Ribbon filter, there are usually two structures: one for the coefficient matrix and one for the filter containing the fingerprints. Since the latter needs the former to be filled, it seems strictly necessary to contain both in memory, but this is not quite the case. It is possible to allocate a little more space than necessary for the coefficient matrix and when it is full and it is time to solve the system, as the system is solved from bottom to top, overwrite the matrix while the filter with the fingerprints is formed from the bottom.
- Other minor functionalities of the code include filter saving. It consists in saving into a file all the information (filter type, important parameters such as r or ϵ, and of course the fingerprints) of the filter, already constructed, that is in memory. This allows the possibility of loading the filter at another time without having to go through the construction process again and the time it takes when the filter consists of hundreds of millions of keys.

Among the features or improvements mentioned, one that has affected performance the most has been the use of a pre-processed password file, as this avoids having to use large amounts of memory space to store so many passwords. Furthermore, this could be considered necessary since the compromised passwords that come in [11] are followed by a count of how many times they have been seen in source data breaches, which is not at all necessary for filter construction. The idea, therefore, is to preprocess all keys beforehand by obtaining a key file in binary format, where all keys have a fixed and unique size of 160 bits (of which 128 bits are for the ribbon and 32 bits for the ribbon's index in the coefficient matrix). Thus, at construction time, it is known in advance how many keys will be used for the construction (because each key has a fixed size and the size of the preprocessed key file is also known) and they can be inserted one at a time into the matrix. The key to make this process as fast as possible is to use asynchronous I/O, so that at the same time that a key is being read from the preprocessed key file, the corresponding construction operations are being performed with the previous one already read. This is done using the POSIX Asynchronous I/O (AIO) interface, the documentation for which can be found in [36].

Finally, focusing on optimizations, what has been done is to use special SIMD [13] type instructions, specifically from the AVX2 set, where the registers contain 256 bits, to speed up the construction and query process. The explanation of why this can be done is as follows: In the filter that is implemented, Homogeneous Ribbon, multiple XOR operations must be performed to solve the system and find the corresponding fingerprints. These operations can be parallelized using the available AVX2 registers, instead of the normal 64-bit ones, achieving, with this change, a 4-fold increase in the speed of the overall process. In the code, all this translates into the use of intrinsic functions of the C compiler, GCC [37], which are mapped to the instructions of the Intel's processor in question.

Once the optimization of the chosen filter was done, it was decided to make a comparison with some of the filters that can also be found in the same mentioned repository [35]. The comparison will be made with the Xor, a classic of the AMQ family, the BinaryFuse, also known as Xor+ and the Split Block Bloom, which despite not being available in this repository, the source code of the same can be found in the article of its author [38]. The results will be shown in the following evaluation section. As for the work that has been described in this section, the source code can be found in the following repository [39].

4 Evaluation

This section will present results showing both the new performance obtained with the optimization of the Homogeneous Ribbon filter, as well as the comparison of this one with some other filters of the AMQ family, such as Xor, BinaryFuse and Split Block Bloom. All comparisons in this section have been made on an equal footing for all filters, i.e., the added features discussed in the previous section, such as having a preprocessed key file and the use of asynchronous I/O, have been applied to all filters equally.

Performance tests have been carried out using a laptop with limited capabilities in a Linux environment. The relevant technical specifications of the machine[4] are listed below:

- CPU: 2.7GHz Intel Core i7-5700HQ (quad-core, 6MB cache)
- RAM: DDR3L,up to 1600 MHz, slot *2, max 16GB
- Storage: 1TB HDD 7200rpm

4.1 Optimization results

The previous Section 3 contains a detailed explanation of the optimization work performed on the existing implementation of the Homogeneous Ribbon filter. However, in order to be called an optimization, it must be demonstrated that there is a change for the better in performance. Therefore, it has been compared the existing implementation with the new one, seeking to show the significant changes in speed that a code using Intel's AVX2 intrinsics is able to provide.

Since the two filters being compared here are Homogeneous Ribbon, it is meaningless to show differences in false positive rate or memory efficiency because they are exactly the same, except in the implementation, thus the only variation will lead in the filter's construction and query times. Figure 5 and Figure 6 show the comparison of construction and query times respectively between the existing implementation of the Homogeneous Ribbon filter and the one optimized in this work. Both graphs plot the time already normalized for each key as a function of the total number of keys used for construction in each case, taking into account that the construction conditions and parameters[5] have been the same.

It is important to note that as more keys are used in the construction of any of the AMQ filters, both the construction and query times increase (a fact that can be clearly observed in both curves compared in Figure 5 and Figure 6), mainly due to the loss of efficiency involved in handling such a large amount of memory. In fact, it can be seen how the swap [40] comes into operation in Figure 5 when the number of keys is very high and, therefore, the performance worsens drastically.

4. More details of this specific model are available at https://es.msi.com/Laptop/GE72-2QD-Apache-Pro/Specification

5. Recall that two of the most significant parameters in the construction of the Homogeneous Ribbon filter are r, w and ϵ

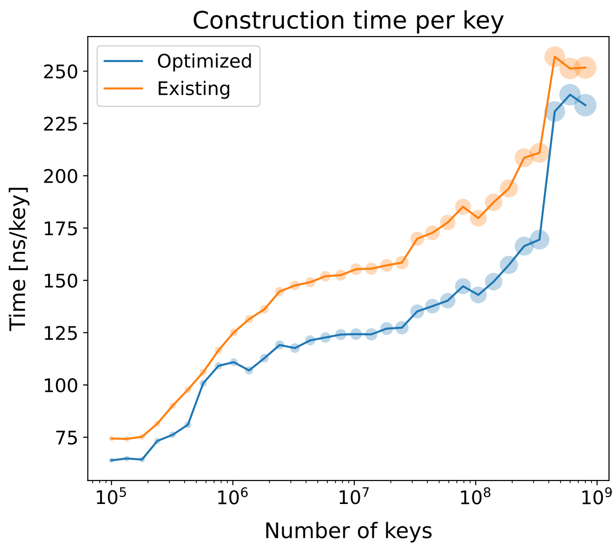

Fig. 5. Homogeneous Ribbon's filter optimized implementation vs. existing implementation construction time per key based on the total number of keys. Results have been obtained for $w = 128$ and $r = 8$ in both curves.

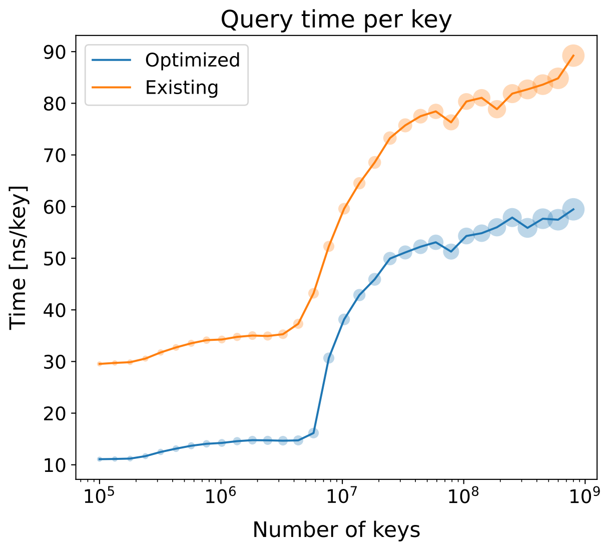

Fig. 6. Homogeneous Ribbon's filter optimized implementation vs. existing implementation query time per key based on the total number of keys. Results have been obtained for $w = 128$ and $r = 8$ in both curves.

Nonetheless, the improvement achieved in the optimization over the existing implementation is perfectly visible, both in Figure 5 and Figure 6, and highlights the importance of code optimization for speed and performance purposes. Such is the case that an optimization in filter construction speed of up to 20% has been achieved. The query speed, however, is where the change has been most noticeable, since the speed has been improved up to 30% for the section formed by more than 10 million keys and up to 60% for filters with less than 10 million keys.

In any case, the percentages given are approximate, since the aim of this work is not to compare the new Homogeneous Ribbon implemented by the authors with the existing implementation, but to integrate the most efficient filter possible into Django so that it can be used by the community in those applications that require it. Therefore, from now on in this paper, when referring to Ribbon or comparing it with other filters of the AMQ family, the implementation referred to is the one optimized in this work. Note that when referring from now on to *Ribbon*, reference is actually made to Homogeneous Ribbon with $w = 128$, since this is the one implemented and no other type of Ribbon filter is contemplated in the rest of the paper.

4.2 Parameterization

The parameterization of AMQ filters is important because it can modulate the memory efficiency at the expense of the false positive rate. This means that, depending on the case, it may be more important to prioritize the false positive rate over the amount of memory occupied by the filter or vice versa. The goal of this work is to find the filter that best optimizes memory, so the parameterization will follow that direction.

Let us recall the efficiency formula described in Table 1: $\eta = (log_2(1/fpr))/(r * (\epsilon + 1))$ where fpr refers to the false positive rate, r to the number of bits of a fingerprint, and ϵ to the excess bit space per key with respect to r.

Having clarified the definition of efficiency η and ϵ, it only remains to find the point of maximum efficiency for each of the filters to be compared and see to which ϵ corresponds in each case. The Xor and BinaryFuse filters will be exempt from parameterization in this case, since the tests performed with them have been black box, which means that their implementation has not been modified. On the other hand, in the case of Ribbon, being the optimized implementation in this work, maximum efficiency will be sought in order to leave the most suitable configuration. Similarly, the same will be done with the Split Block Bloom filter, whose implementation in this case has been more customized, although maintaining the author's own functions.

Two graphs are shown, one for the Ribbon filter (Figure 7) and one for the Split Block Bloom filter (Figure 8). In both there are two curves, each of which corresponds to a Y-axis, and it is possible to differentiate which axis is referred to by the color. One of the curves represents the efficiency η and the other the false positive rate, both as a function of ϵ. Thus if we find the maximum point of the first curve we will obtain the ϵ for the optimal memory efficiency, which can then be seen to which false positive rate corresponds in the second curve.

As a result, it can be seen that the maximum achievable efficiency of the Split Block Bloom filter is 63%, for which the false positive rate corresponds to approximately 1%. As already mentioned, the false positive rate could be reduced but, in such a case and as can be shown

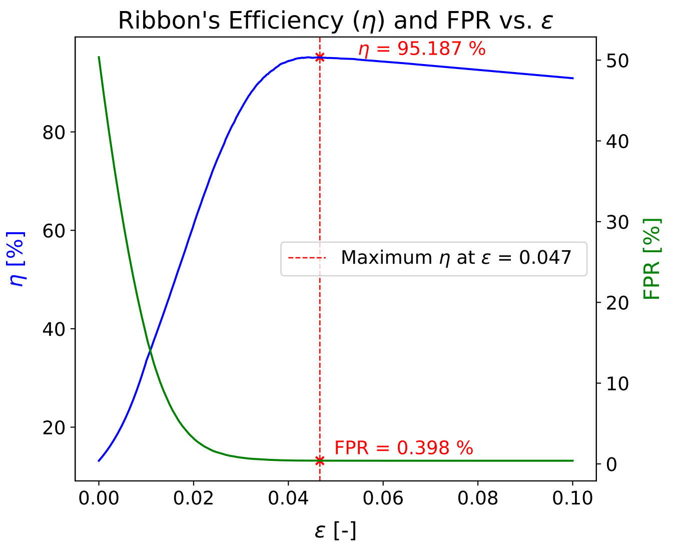

Fig. 7. Ribbon's efficiency and false positive rate as a function of ϵ. The maximum efficiency is pointed out in order to find the most efficient ϵ and to which value of FPR corresponds to.

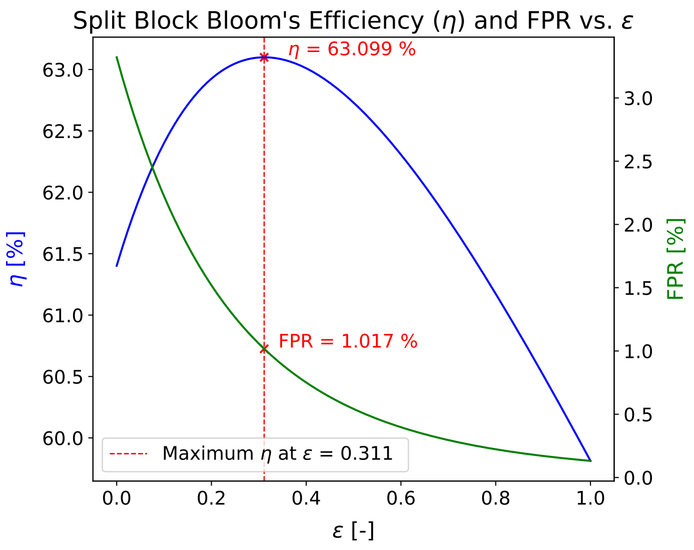

Fig. 8. Split Block Bloom's efficiency and false positive rate as a function of ϵ. The maximum efficiency is pointed out in order to find the most efficient ϵ and to which value of FPR corresponds to.

from the curve in Figure 8, the memory efficiency would decrease a lot, so it would not make much sense, at least according to the aim of this work. In contrast, the Ribbon filter achieves a maximum efficiency of 95% and a consequent almost unbeatable[6] false positive rate of 0.4%.

In the next subsection 4.3 a full comparison of the discussed filters will be made and the Ribbon and Split

6. Recall that for $r = 8$ the theoretical minimum false positive rate is $1/256 \approx 0.39\%$

Block Bloom filters will be considered to be configured with the ϵ parameter already optimized for memory.

4.3 Ribbon vs. other AMQ Filters

The last step for this evaluation section is none other than the comparison of the four mentioned AMQ filters[7] to conclude which of all is the most optimal to integrate in Django taking into account the special interest for a low memory occupancy above all, but without neglecting the importance of speed in filter's construction and queries.

To start this comparison, the construction and query time of each of the filters was measured (having $r = 8$ in all cases), depending on the number of keys. Thirty measurements have been made for each group of keys and the minimum value has been taken since the time (especially in the construction, which is a longer process) is very variable and is often interfered by the processes of the operating system. The construction algorithm of the Ribbon and Split Block Bloom filters is practically fixed time, so taking the minimum of the measures is not unreasonable, while in the case of Xor and BinaryFuse a favor is being done to them, since they are time-varying algorithms and it is going to be taken the result of the most favorable case, which is not necessarily the most frequent. Regarding the queries, they are all fixed time, so taking the minimum is the most appropriate.

Whereas Figure 9 shows the construction times, Figure 10 shows the query times. In both graphs, the minimum of the 30 measurements for each key group is represented as a circle, which becomes larger as the number of keys increases. While it is true that the Ribbon filter has the worst query times, its performance is not the worst, as the speed in construction outperforms the Xor and BinaryFuse filters for higher numbers of keys. Unquestionably, the number one position for both processes is reserved for the Split Block Bloom filter, as could be expected, which given the times it sets and the difference with the rest of the AMQ filters compared seems to play in another league.

Speed is an important aspect of AMQ filters, but as it has been repeated throughout the paper, it is not the priority in this case. Now it is the turn of the comparison from the point of view of memory occupancy. For this study, the same dynamics as in the previous graphs of this subsection have been followed, i.e., the same number of measurements and groups of keys have been taken, but this time showing different data. Since the feature wanted to be observed is the amount of memory needed for each filter once constructed with each key group, the Y-axis is representing the number of Bytes occupied by the filter in memory divided by the number

7. The four mentioned at the beginning of the section were Xor, BinaryFuse, Split Block Bloom and the implemented Homogeneous Ribbon (or also referred to as just *Ribbon*)

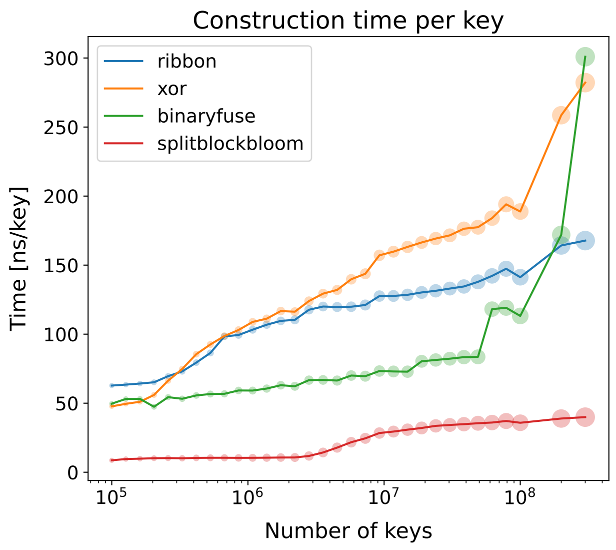

Fig. 9. Comparison of the studied AMQ filters' construction times per key as a function of the total number of keys.

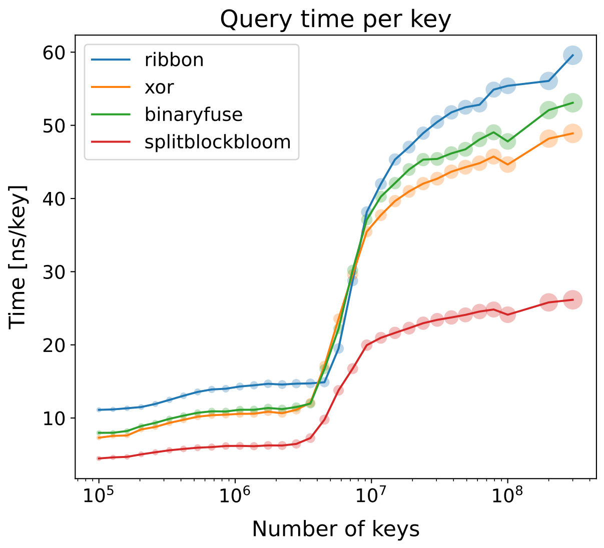

Fig. 10. Comparison of the studied AMQ filters' query times per key as a function of the total number of keys.

of keys that form it. With this information two graphs are shown, one whose X-axis is the query time (Figure 11), to be able to observe the two important things once the construction process is finished: the memory cost and the time used for each query; and the other whose X-axis shows the false positive rate (Figure 12). This last one (Figure 12) is closely related to the graphs that have been shown in the parameterization Section 4.2, since, by looking closely, it can be appreciated that the FPR of the Split Block Bloom filter is the same as configured (≈1%), as well as its occupation in memory because, the optimal epsilon was 0.31, which means that each byte of information implies 0.31 real bytes more and the filter size per each key is of about 1.31. The same fact can be checked analogously with Ribbon, where the FPR

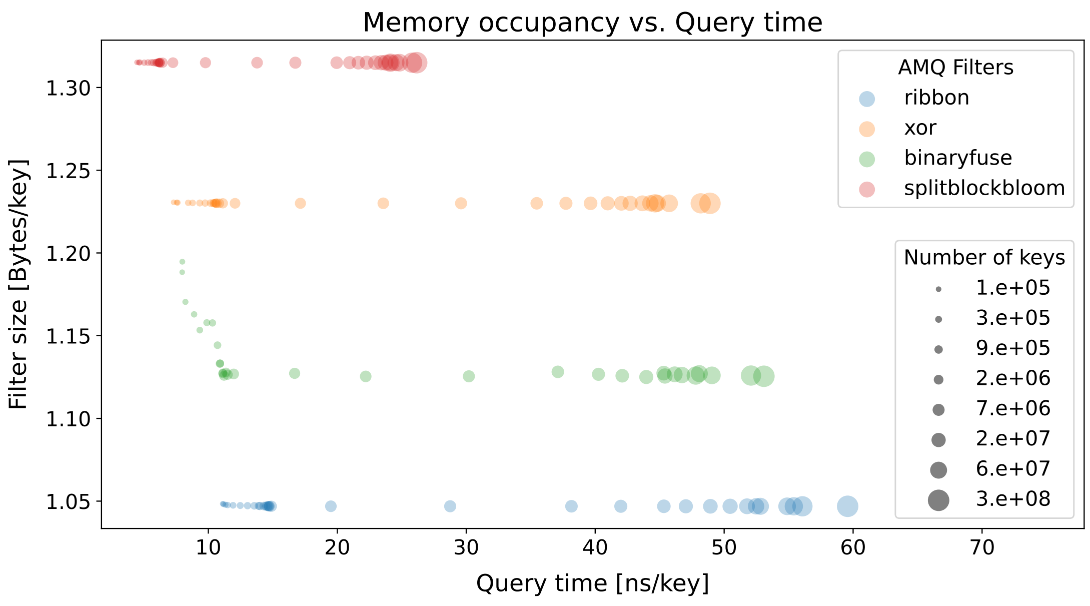

Fig. 11. Comparison of the studied AMQ filters' memory occupancy as a function of their respective query times based on the total number of keys. Ribbon's filter as well as Split Block Bloom's filter have been configured with the most efficient ϵ previously calculated in Section 4.2.

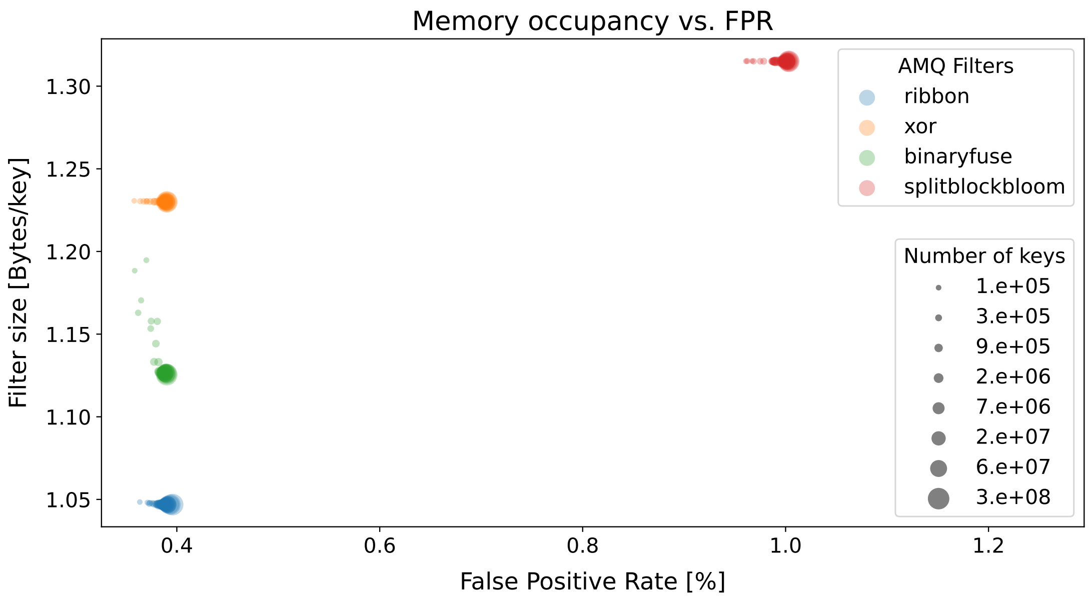

Fig. 12. Comparison of the studied AMQ filters' memory occupancy as a function of their respective false positive rates based on the total number of keys. Ribbon's filter as well as Split Block Bloom's filter have been configured with the most efficient ϵ previously calculated in Section 4.2.

obviously corresponds to the one previously configured and the size of the filter also coincides with 1 + the optimal ϵ found.

The results of Figure 11 are quite conclusive, it can be seen how for all filters the query time increases the greater the number of keys that form the filter, being the Ribbon filter the slowest, although with little difference with respect to the Xor or BinaryFuse filters. This fact was already observable in the Figure 10, but in this figure it can also be seen how the occupation of each key in memory exceeds from the one of an ideal filter. In the case of an ideal filter, each byte of information would correspond to a real byte, so, in that sense, if this filter existed and was represented in the graph, the points of the Y-axis would always be located at 1, since each key would occupy only 1 byte and no more.

For a better understanding, as an example, based on the results obtained, if we were to form a filter with 100 keys, this is what it would occupy in memory in each case (always taken into account $r = 8$):

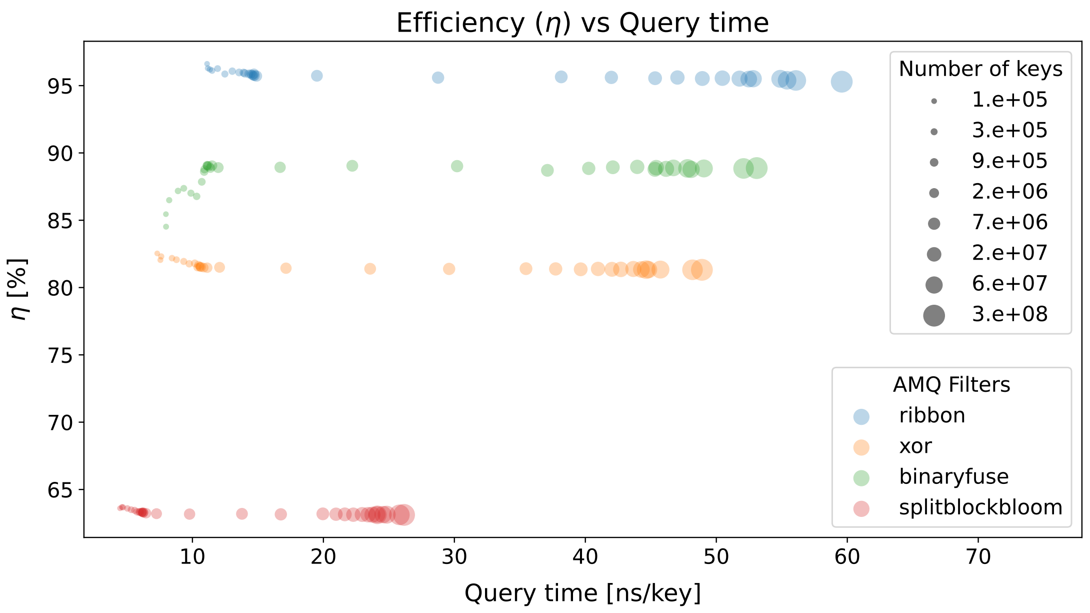

Fig. 13. Comparison of the studied AMQ filters' efficiency as a function of their respective query times based on the total number of keys. Ribbon's filter as well as Split Block Bloom's filter have been configured with the most efficient ϵ previously calculated in Section 4.2.

- Ideal filter: 100 bytes
- Ribbon filter: $\leq$105 bytes
- BinaryFuse filter: $\leq$113 bytes
- Xor filter: $\approx$123 bytes
- Split Block Bloom filter: $\approx$131 bytes

This added to the results obtained from Figure 12, which as already mentioned, in the case of Ribbon and Split Block Bloom are just a reminder of the consequences of the parameterization performed, leads to the following conclusions:

1) The Ribbon filter exceeds only 5% in memory with respect to the ideal and obtains a false positive rate practically equal or very close to the ideal.
2) The Xor and BinaryFuse filters occupy 13% and 23% more memory than the ideal respectively and both obtain for those sizes a false positive rate also very close to the ideal.
3) The Split Block Bloom filter is the farthest in size from the ideal, occupying 31% more, and obtaining a less competitive false positive rate than the rest, 1%.

Finally, to conclude the evaluation section, and in order to corroborate the results obtained so far, Figure 13 shows the maximum efficiency of each filter. The efficiency represented takes into account not only the actual bytes per key that each filter occupies in memory, but also the ideal bytes it should occupy given the false positive rate they provide in each case. Once again, the Ribbon filter is the one that best optimizes the memory used, while the Split Block Bloom filter leaves much to be desired in this aspect. It is important to note, although mentioned above, that this efficiency shown is the maximum efficiency achieved for $r = 8$ in all cases. The Ribbon, Xor and BinaryFuse filters provide an almost perfect false positive rate, as it is very close to the ideal minimum, whereas the Split Block Bloom filter does not go below 1%. It is true that for the latter case it can be reduced, as explained in the parameterization Section 4.2, but it would be at the cost of a drastic reduction in efficiency.

5 Integration in Django

In this section the steps followed to make possible the integration of sources written in C in a framework whose main language is Python, such as Django, will be discussed. In addition, it will be shared how an installable application has been built in this framework to be later packaged and uploaded to Pypi, from where it will be available for anyone who wants to use it. All this will be divided into three subsections, each of which summarizes the necessary steps to achieve the goal.

In the previous section it has been shown that the filter that best optimizes memory and therefore best meets the main requirement of this work is the Ribbon filter. However, since the intention is to make this Django application public and not everyone will have the same requirements, it has been decided to integrate the four filters studied and thus demonstrate that it is possible to integrate any desired filter into this application module.

5.1 Using Python/C API

The first step of this integration in Django is quite intuitive. As Django is a framework written in Python and the filter sources are written in C, it is necessary to write some Python wrappers (for Cross-language/runtime interoperability). For this purpose Python/C API [8], a manual that documents the API used by C and C++ programmers who want to write extension modules or embed Python [41] was used. The process is not complex, for each of the filters a wrapper is written in C containing all the functions to be invoked from Python. This wrapper, making use of the API methods, is in charge of converting the C types into the corresponding Python ones, capturing the necessary arguments of each function and mapping them with the desired keywords, choosing different names for the Python function calls, etc. Once the wrappers of each filter are finished, it will be possible to invoke the methods of these filters in Python, so everything would be ready to start writing the code for Django.

5.2 Writing installable Apps

To use Django, the most usual way is to create a project and write applications inside it. Django already has some pre-installed modules and one of them manages the creation of accounts for the whole project. This module provides the user with a few default validators, which are just functions to validate passwords, but you can add as many as you want. Among the default validators there is one that checks the password length and sets a minimum number of characters, another one that checks the similarity with the username, and so on. Unfortunately, there is no default validator that verifies that the chosen password is not compromised. Therefore, the intended application is a new validator that achieves this purpose and can be installed and added to any Django's project.

The idea is, given a file of compromised passwords, to create a type of filter from the four studied and keep it in memory for the duration of Django's execution. When an account is created, the chosen password will be checked against the filter already constructed and in memory, i.e. a query will be made and depending on the result the password will be validated or not. In this way, NIST recommendation[9] would be fulfilled and there will be no possibility of having any account with passwords belonging to breach corpuses in Django's database.

This described application is a kind of self-service client, in the sense that this Django instance has to construct the filter and keep it in memory. The problem with this is that it can become inefficient in schemes such as an organization or company where there are different instances of Django, since having an in-memory filter for each instance is useless. For this reason, a server application has also been developed to host the in-memory filter and receive HTTP requests. This server is an application that receives as a query the passwords via HTTP in JSON format and responds if they are compromised or not. When the server receives an external query, it performs a query to the filter it has in memory, so the server application must have a client application running which is the one that constructs the filter when Django starts. In this way there are two applications, one that works as a client and the other as a server, which have been named *filterclient* and *filterserver* respectively.

The way to manage the configuration of a project in Django is through the "settings.py" file and it is used for all installed applications. Therefore, the same dynamic has been followed for the developed applications, that is, to change the default configuration of the applications it has to be done through this same file. For example, in the case of the *filterclient* application there are two modes: one in which the filter is constructed and used locally and another in which no filter is constructed nor used. In the first one the queries are performed against the filter in memory locally and in the second one through an HTTP request to a remote server. This server, by the way, must have both the *filterserver* application and the *filterclient* application in the first of the described modes installed. In a certain Django instance with both applications installed, the *filterserver* application would only handle requests received via HTTP and redirect the queries to the *filterclient* application, which would perform the queries against the filter in memory. In order to manage which mode of the *filterclient* application is the one running (either the local or the remote one), the user must specify it in the *settings.py* file before starting Django. There are many other parameters that can be specified or changed to vary the operation of these applications at the whim of each user. For instance, the mode in the *filterclient* application is selected with the variable *FILTER_MODE* whose possible values are *LOCAL* or *REMOTE*. Table 2 shows a short incomplete list of the configurable parameters of both applications. For more information it is recommended to consult the GitHub repository [39].

The two developed applications have been fully tested and have their own tests so that when they are installed they can be verified to work correctly before they go live, as specified in the Django best practices manual. The client application (*filterclient*) has tests for both *LOCAL* and *REMOTE* modes, in the first case the tests simulate and verify the construction of all the integrated filters (the four studied) combining all possible configurations and situations, and in the second case a remote server is simulated to perform password query requests and verify the correct operation. In addition, for the server

8. API's Official documentation at https://docs.python.org/3/c-api/index.html

9. Recall that NIST stated that *when processing requests to establish and change memorized secrets, verifiers SHALL compare potential secrets against a list containing passwords obtained from previous breach corpuses.*

TABLE 2
Some configurable settings of the developed applications, indicating on each case the operation mode they are referred to, the variable's setting name, their meaning, their default value and their possible values. Mode is only relevant to *filterclient* application and is also managed with a variable named *FILTER_MODE* whose possible values are *LOCAL* (indicating the local mode in which the queries are performed locally to a filter in memory) and *REMOTE* (indicating the remote mode which is only feasible with an active *filterserver* application on a remote Django instance).

Application	Mode	Variable's name	Meaning	Default Value	Possible Values
Filterclient	*LOCAL*	*FILTER*	Type of filter to be used	*ribbon128*	*ribbon128*
					xor
					binaryfuse8
					splitblockbloom
					dummy
		NKEYS	Number of keys to be included in the filter, 0 means all	*0*	(Whatever number)
		RBYTES	Number of bytes of r (only applicable to $ribbon128$ and xor filters)	*1*	*1*
					2
		PWDFILE	File containing all compromised passwords extracted from ref	-	Place the absolute path to that file
	REMOTE	*REMOTE_SERVER*	URL of the server's application including the page where it is expecting the requests	-	Example: ”http://127.0.0.1:8000/reqfilter”
		SERVER_CREDENTIALS	Credentials of user needed at Django's server instance if SERVER_OPEN = False	{*'username':'admin', 'password':'admin'*}	(Must follow the indicated JSON format)
Filterserver	No mode needed, but Filterclient app must be installed in order to work	*SERVER_URL*	Relative web page where server is waiting for HTTP requests	-	Example: ”filterapi/”
		OPEN_SERVER	Whether if the server is open to requests no matter who they come from.	*True*	*True*
					False

application (*filterserver*) there are also tests that simulate HTTP requests and verify correct operation in all possible scenarios.

5.3 Packaging and Uploading

Once the implementation of the respective commented applications is finished, it is necessary to find a way to package everything into a single installable package or module. One of the best ways to do this is through *setuptools* [42], which is a commonly used extension library for distributing Python libraries and extensions. It extends *distutils*, a basic module installation system included with Python, to also support several more complex builds that make it easier to distribute larger applications. It is mainly characterized by:

1) Providing support for dependencies: A library or application can declare a list of other libraries it depends on, which will be automatically installed for you. This feature is essential since the development of these applications has required the use of external Python libraries.
2) Providing package registration: *setuptools* registers your package with your Python installation. This allows you to query the information provided by one package from another package. The most well-known feature of this system is the entry point support that allows a package to declare an *entry point* that another package can hook into to extend the other package.
3) Serving as an installation manager: allowing PIP [43] to install other libraries for you.
4) Compiling C or C++ sources using the Python/C API: This allows you to include raw C or C++ sources and compile them according to the given compilation information and corresponding wrappers.

These features make this library the preferred one for creating an installable module for Django. Following the general steps of this tutorial [44], and particularizing them for this case, we have built a package that contains everything necessary, both the C sources of the different filters and the Python sources of both applications, to be installed with PIP. This package has been uploaded to the Pypi platform and is already available for use in the alpha 0.0.1 version at [45].

The way to download it is simple, everything is achieved through the *pip* command [10]. The application would be installed as the rest of the applications in Django, adding it to the *INSTALLED_APPS* section of the configuration file *settings.py* and to make use of the validator it would be necessary to also add the filter's validator in *AUTH_PASSWORD_VALIDATORS* obtain-

10. Visit GitHub repository at https://github.com/migonsa/django-pwnedpass-validator for more detailed information.

ing a verification of compromised password at each account creation.

6 Conclusions and Future Works

This paper describes the work done to solve a practical problem in Django. An installable module has been developed for the Django framework that enables the possibility of using a validator to avoid compromised passwords. NIST, in one of its latest special publications, recommends to check each password chosen by a user in an account creation process against a list containing compromised passwords from different data breaches. That recommendation has been implemented through the development of this module, which uses AMQ filter technology to ensure that no compromised passwords can be registered to any type of account.

AMQ filters can be constructed from a gigantic amount of compromised passwords extracted from different breach corpuses and are known to be able to check quickly and reliably whether a password belongs to that dataset or not. Their reliability lies mainly in the fact that there is no possibility of obtaining false negatives, which makes them perfect for this purpose. The only negative aspect is perhaps that this type of filters must remain in memory to work properly. For this reason, we have chosen the filter, within the AMQ family, that best optimizes the memory space and we have made substantial improvements in the performance of the existing implementation of the filter in terms of construction and query speeds.

In addition, the module incorporates not only this optimized filter but also many others of the family with which it has been compared to in this paper in terms of efficiency and speed, so that they can be used indistinctly in different situations depending on the requirements of each user. Last but not least, the module includes two applications that allow its use in local or remote mode. The local mode is intended for single Django instances that need the validator locally, having the filter constructed in memory, while the remote mode is intended for those organizations that have multiple Django instances that communicate with each other. The latter mode enables the possibility of having a single instance of Django acting as a server with the filter in memory and the rest of the instances acting as clients performing the corresponding queries against the server remotely instead of consuming local memory.

The results obtained, both for the optimization work and for the subsequent integration into the Django framework have been a complete success. However, there is always work to be done in this regard and that is why it is convenient to point out some aspects to be enhanced in the future. In the academic environment it would be interesting to deepen the study of the AMQ family filters not only in the comparisons but also in the effects that the use of Intel's AVX2 intrinsics in the implementation have on the performance. It might also be worthwhile to adapt the different implementations to the new instructions (such as those related to AVX512) incorporated in the latest Intel processors, as well as to investigate the relationship between the size of the L2 and L3 caches and the speed achieved for construction and query processes. On the other hand, on a more practical note, it would be beneficial to incorporate to this module some more filters to adapt more flexibly to the different interests of each user. Furthermore, it would be good as future work to include this module in other existing web application frameworks not necessarily based on Python, in order to spread its use and make it more globally available.

7 Acknowledgments

The authors of this article would like to thank researchers and professionals in the field for their support. To Jim Apple, for his patience and help in the parameterization of his filter Split Block Bloom, and to Peter Dillinger and Stefan Walzer for recommending the use of their his high memory-efficient Homogeneous Ribbon filter. They would also like to give special thanks to Pedro Reviriego for his guidance throughout the work.

References

[1] G. Notoatmodjo and C. Thomborson, "Passwords and perceptions," in *Proceedings of the Seventh Australasian Conference on Information Security-Volume 98*. Citeseer, 2009, pp. 71–78.

[2] K. Chanda, "Password security: an analysis of password strengths and vulnerabilities," *International Journal of Computer Network and Information Security*, vol. 8, no. 7, p. 23, 2016.

[3] C. Roberts and V. Muscarella, "Defining over-the-top (ott) digital distribution," *The Entertainment Merchants Association*, pp. 3–4, 2015.

[4] A. Aleroud and L. Zhou, "Phishing environments, techniques, and countermeasures: A survey," *Computers & Security*, vol. 68, pp. 160–196, 2017.

[5] S. Kulkarni and S. Urolagin, "Review of attacks on databases and database security techniques," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 11, pp. 253–263, 2012.

[6] K. Marchetti and P. Bodily, "John the ripper: An examination and analysis of the popular hash cracking algorithm," in *2022 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, 2022, pp. 1–6.

[7] R. Hranický, L. Zobal, O. Ryšavý, D. Kolář, and D. Mikuš, "Distributed pcfg password cracking," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 701–719.

[8] A. T. Tunggal, "The 66 biggest data breaches (updated august 2022): Upguard," Aug 2022. [Online]. Available: https://www.upguard.com/blog/biggest-data-breaches

[9] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y. Choong *et al.*, "Draft nist special publication 800-63b digital identity guidelines," *National Institute of Standards and Technology (NIST)*, vol. 27, 2016.

[10] D. Jaeger, C. Pelchen, H. Graupner, F. Cheng, and C. Meinel, "Analysis of publicly leaked credentials and the long story of password (re-) use," *Hasso Plattner Institute, Universidad de Potsdam. Disponible en https://bit. ly/2E7ZT01*, 2016.

[11] "Pwned passwords." [Online]. Available: https://haveibeenpwned.com/Passwords

[12] C. Burch, "Django, a web framework using python: Tutorial presentation," *Journal of Computing Sciences in Colleges*, vol. 25, no. 5, pp. 154–155, 2010.

[13] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of simd extensions," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, 2005.

[14] A. Sinclair, "License profile: Bsd," *IFOSS L. Rev.*, vol. 2, p. 1, 2010.

[15] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley *et al.*, "Best practices for scientific computing," *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[16] A. Leffert, "The crud methodology," *A Record of The Proceedings of SIGBOVIK 2009*, p. 97, 2009.

[17] J. Haupt, "Spotify," 2012.

[18] "Django market share and web usage statistics." [Online]. Available: https://www.similartech.com/technologies/django

[19] M. Keith and M. Schnicariol, "Object-relational mapping," in *Pro JPA 2*. Springer, 2009, pp. 69–106.

[20] U. Manber and S. Wu, "An algorithm for approximate membership checking with application to password security," *Information Processing Letters*, vol. 50, no. 4, pp. 191–197, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019094000328

[21] Tok.wiki. [Online]. Available: https://hmong.es/wiki/Quotient_filter

[22] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman, "Exact and approximate membership testers," in *Proceedings of the tenth annual ACM symposium on Theory of computing*, 1978, pp. 59–65.

[23] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[24] A. Geil, M. Farach-Colton, and J. D. Owens, "Quotient filters: Approximate membership queries on the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 451–462.

[25] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[27] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *Journal of Experimental Algorithmics (JEA)*, vol. 25, pp. 1–16, 2020.

[28] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Citeseer, 2004, pp. 30–39.

[29] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *Journal of Experimental Algorithmics (JEA)*, vol. 14, pp. 4–4, 2010.

[30] J. Lu, Y. Wan, Y. Li, C. Zhang, H. Dai, Y. Wang, G. Zhang, and B. Liu, "Ultra-fast bloom filters using simd techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 953–964, 2018.

[31] T. M. Graf and D. Lemire, "Binary fuse filters: Fast and smaller than xor filters," *Journal of Experimental Algorithmics (JEA)*, vol. 27, no. 1, pp. 1–15, 2022.

[32] M. Dietzfelbinger and S. Walzer, "Efficient Gauss Elimination for Near-Quadratic Matrices with One Short Random Block per Row, with Applications," in *27th Annual European Symposium on Algorithms (ESA 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 39:1–39:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/11160

[33] P. C. Dillinger and S. Walzer, "Ribbon filter: practically smaller than bloom and xor," *arXiv preprint arXiv:2103.02515*, 2021.

[34] R. Lidl and H. Niederreiter, *Finite fields*. Cambridge university press, 1997, no. 20.

[35] T. M. Graf and D. Lemire, "fastfilter_cpp," https://github.com/FastFilter/fastfilter_cpp, 2019.

[36] [Online]. Available: https://man7.org/linux/man-pages/man7/aio.7.html

[37] B. J. Gough and R. Stallman, *An Introduction to GCC*. Citeseer, 2004.

[38] J. Apple, "Split block bloom filters," *arXiv preprint arXiv:2101.01719*, 2021.

[39] M. G. Saiz, "django-pwnedpass-validator," https://github.com/migonsa/django-pwnedpass-validator, 2022.

[40] T. Kay, "Linux swap space," *Linux Journal*, vol. 2011, no. 201, p. 5, 2011.

[41] M. Hu and Y. Zhang, "The python/c api: evolution, usage statistics, and bug patterns," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 532–536.

[42] P. von Weitershausen, "Packaging and deployment," *Web Component Development with Zope 3*, pp. 441–457, 2008.

[43] A. Lancaster and G. Webster, "Getting started with python," in *Python for the Life Sciences*. Springer, 2019, pp. 1–12.

[44] joelbarmettlerUZH, "How to upload your python package to pypi," Jul 2020. [Online]. Available: https://medium.com/@joel.barmettler/how-to-upload-your-python-package-to-pypi-65edc5fe9c56

[45] M. G. Saiz, "django-pwnedpass-validator," https://pypi.org/project/django-pwnedpass-validator/, 2022.