Pattern- and Similarity-Based Realtime Risk Monitoring of SSH Brute Force Attacks with Bloom Filters

Günter Fahrnberger
University of Hagen
Hagen, North Rhine-Westphalia, Germany
guenter.fahrnberger@studium.fernuni-hagen.de

***Abstract*—This study explores the vulnerability of Internet-exposed services and the prevalence of Brute Force Attacks (BFAs) as an intrusion method. It highlights the significance of anti-hammering mechanisms in thwarting such attacks but acknowledges that some services lack these protective measures. Moreover, the potential consequences of successful breaches vary, ranging from data leakage to complete system compromise. This paper explores research opportunities presented by studying the quality of BFAs, with a specific focus on Secure Shell (SSH) attacks. It discusses previous research in this area and proposes methodological improvements, including enhanced pattern matching techniques and the evaluation of similarity metrics based on Bloom Filters (BFs). Its conclusion critically reflects on the findings and suggests directions for future research.**

***Index Terms*—Bloom Filter (BF), Brute Force Attack (BFA), Monitoring, Risk monitoring, Secure Shell (SSH), Similarity assessment, Supervision, Surveillance**

I. Introduction

Once a service becomes publicly exposed to the Internet, network scanners usually discover its ports briskly as attackable targets. At best for an offender, they can exploit a known Common Vulnerability Exposure (CVE) to deface such a service, attain sensitive data, or even access the underlying operating system. If a service requires secret login credentials and no CVE exists, BFAs may remain the sole possibility for intrusion apart from snatching illegal access by dint of social engineering. A Brute Force Attack (BFA) iteratively attempts all relevant elements of dictionaries or combinations of character sets until success, exhaustion, or cancellation. Anti-hammering mechanisms thwart the endeavors of BFAs with constant/increasing delays or temporary/permanent account lockout after unsuccessful tries. Regrettably, not all services protect themselves with such safeguards and, thus, lure BFAs. Nonetheless, despite the absence of anti-hammering methods, the situation has not deteriorated immediately, provided that only long and complex credentials grant access to a service. It goes without saying that deemed-as-secure credentials become weak if used at another service that has leaked them. Be that as it may, topical service log files, Security Information and Event Management (SIEM) systems, Intrusion Detection Systems (IDSs), and Intrusion Prevention Systems (IPSs) prove that BFAs have not fallen out of fashion yet. Attackers continue to conduct BFAs aggressively.

The impact of successful breaches vastly differs between miscellaneous services. Bad enough, the cracked administration portal of a website gives free reign to leakage of its data, defacement, and deletion. Much worse could be the conquest of a container, an operating system, or a hypervisor. All three types can be administered by the SSH, the contemporary protocol for remote consoles. All guides highly encourage making SSH only accessible to the Internet via a Virtual Private Network (VPN) rather than directly. However, stubborn administrators and borderline cases let an SSH Daemon (SSHD) listen on a public interface via Transmission Control Protocol (TCP) port 22, at worst, without any anti-hammering protection. As aforementioned, BFAs will not be long in coming.

Hard to believe, but carelessly opening SSH directly to the Internet also offers opportunities for scientific research. While SIEM systems and IDSs recognize the quantity of BFAs, and IPSs also prevent them, none of them assesses the quality of BFAs. Quality in this context means the similarity between attempted and correct credentials. This document operates on the hypothesis that smaller differences between attempted and genuine credentials lead to higher quality and greater risk of credential guessing. Based on this, the research question arises as to whether the quality and, consequently, the risk can be thoroughly monitored in realtime. Two existing scientific papers already try to answer this research question by explicating realtime risk monitoring of SSH BFAs [1], [2]. For this reason, the technique in [1] counts the matching patterns of attempted and true passwords in case of coincident usernames. The approach in [2] improves the accuracy compared with [1] by calculating three Bloom Filter (BF)-based similarity metrics. Both treatises apply different threshold (pair) computation techniques from [3]–[5] and recommend the candidate with the least False Positive (FP) alerts.

In spite of expedient results, room for refinement always emerges. Consequently, incorporating the following changes should better address the research question concerning the appraisal of the quality of BFAs.

1) Literature review dedicated to pattern matching based on Bloom Filters (BFs)
2) Flowcharts depicting BF creation instead of pseudocode

3) Verification of efficiency of novel metrics through an experiment conducted on a modern Condition Monitoring System (CMS)
4) Evaluation of closeness between (patterns of) attempted and legitimate passwords based on adapted BFs (as applied in [2]) and original BFs

This scholarly piece fulfills all four change requests. While [1] sheds light on related work about SSH BFAs, [2] surveys existing publications about similitude determination of character strings by dint of BFs. Section II follows up with an overview of academic approaches for pattern analysis by means of BFs. Since their utility for this treatise's objectives shapes up as limited, Sect. III describes how to effectively log abortive SSH logon attempts with the help of password patterns and BFs. Section IV specifies the creation of various metrics based upon the logged foundered SSH login tries. The deployment referenced in Sect. V demonstrates the feasibility and plausibility of these devised metrics and provides comparative analysis between them. Section VI self-critically reflects and emphasizes the merits of this paper while hinting at auspicious prospective work.

II. Related Work

The predecessor to this disquisition primarily cites related work on the reliable recognition of SSH BFAs, aiming to improve them by counting coherent password patterns [1]. Its sequel reviews related work focused on assessing the similarity of character strings supported by BFs [2]. Unsurprisingly, it refines the precision of the method in [1] by employing a modified BF. Owing to this groundwork of both predecessors, this section deliberately omits any references about BFAs and BF-based similarity calculations and, instead, focuses on present ideas of pattern matching using BFs. For the sake of completeness, it must be admitted here that writers evenhandedly use the term *pattern* for character strings and their structural description.

Chronologically, the first stems from the year 2010, authored by Tuan, Hieu, and Thinh [6]. Just as every scholarly proposal based upon BFs does, it rightly justifies their existence. During the compilation of a BF, a binary index table arises that contains all elements of an arbitrary set in a condensed form to render subsequent time- and space-efficient membership queries possible. Predefined hash functions handle this compression. For every member, each hash function ensures a one at a specific position in the BF. Unfortunately, BFs can produce False Positives (FPs), i.e., wrongly confirm memberships. For that reason, Tuan, Hieu, and Thinh additionally employ Bloomier filters to eliminate FPs. While BFs depend on bitwise index table entries, their Bloomier counterparts store more information per entry, which entails higher complexity. The authors combine both filter types to accelerate the performance of pattern matching in the malware scanner ClamAV by minimizing off-chip memory access times for exact pattern comparison.

Because the authors' initial draft merely supports patterns between ten and 20 characters, they (Hieu, Tuan, and Thinh) add pattern fragmentation in a successor publication that processes all static patterns of ClamAV irrespective of their length [7]. Furthermore, Hieu, Tuan, and Thinh generalize their extension to make it adaptable for other applications. They term it BBFex, which again poses an exact matching system without any FPs. Realtime risk monitoring of SSH BFAs can deal with sporadic FPs without difficulty if it imposes retention periods of several minutes to render them toothless. Therefore, Bloomier filters currently do not play a role in this paper but might be considered for future opportunities.

Liu, Kang, Chen, and Ni present another use case of pattern matching via BF [8]. It derives from the search for cellphone users whose communication patterns most resemble a given pattern across all base stations of a mobile network. The option of shipping all relevant data to a sole main data center for the sake of aggregation and calculation gets dismissed due to unreasonable communication costs, particularly in large networks like in China, where tons of data incur. Local processing in base stations turns out to be a more effective alternative, albeit causing a so-called distributed incomplete pattern matching problem. To make matters worse, the amount of FPs must be kept low, which renders ordinary BFs infeasible. Instead of resorting to Bloomier filters, the developers utilize Weighted BFs (WBFs). Such derivatives store weights in the form of integers in their arrays rather than bits. Although WBFs significantly reduce FPs, the images of their generating functions have much greater cardinalities compared to classic BFs. At the same time, WBFs disperse over larger codomains, collide less frequently with other WBFs, and become more distinct.

Pande and Bakal refine the suggestion of Liu, Kang, Chen, and Ni with unsupervised Machine Learning (ML) and term the resulting data structures unsupervised WBFs [9]. Hence, their system also handles untrained patterns, which enhances accuracy and communication efficacy. Nevertheless, the increased chance of uniqueness achieved by (unsupervised) WBFs facilitates guessing their plaintext input. This might be acceptable for noncritical use cases but certainly not for securely mapping passwords as required for realtime risk monitoring of SSH BFAs. An additional future treatise could scientifically compare the likelihood of breaking the privacy of WBFs and classical BFs.

Al-Tariq et al. address the undesirable correlation between the FP rate and the number of members comprised in a BF [10]. A BF offers a certain level of accuracy based upon its array length. This level remains as long as a BF does not contain too many members. As soon as the member count exceeds a tipping point, an excess of ones in the BF causes the FP rate to deteriorate. Before such deterioration occurs, Al-Tariq et al. increase the size of BFs and evenly redistribute all members among them. They term this concept *Extended BF*. Membership queries across multiple BFs undoubtedly require more computing resources but can be fortunately parallelized. Since passwords typically consist of a manageable number of characters for memorization, realtime risk monitoring of SSH BFAs does not necessitate this type of horizontal scaling.

El-Ghamrawy dedicates herself to a knowledge management framework for imbalanced data [11]. Such an imbalance occurs when the cardinality of one mined data class considerably surpasses those of other classes. This compromises accuracy optimization in knowledge discovery processes due to the neglect of relative distribution of classes. The proposed solution utilizes a BF in its frequent pattern mining algorithm for two reasons. Firstly, it ensures accurate mining of frequent patterns, storing, and counting k-itemsets. Secondly, it reduces the time required to discover knowledge by leveraging the ability of BFs to locate ingested data all at once. Realtime risk monitoring of SSH BFAs cannot leverage the findings of this paper on knowledge management in any way.

Wada, Matsumura, Nakano, and Ito suggest an efficient method for byte stream pattern tests by utilizing multiple bit arrays with manifold distinct rolling hash functions to create BFs [12]. For instance, signature-based anti-malware and anti-intrusion software detect malicious patterns in byte streams of network traffic with such tests. Field Programmable Gate Arrays (FPGAs) achieve the necessary throughput. The utilized Field Programmable Gate Array (FPGA) in the conducted experiment attains 49.5 Gbps for 48 byte streams and 100k patterns, i.e., 227 times more than an Intel Core i7 Central Processing Unit (CPU). Wada, Matsumura, Nakano, and Ito also formally elaborate on the FP probability of BFs. Additionally, they practically evaluate the FP likelihood of BFs based on randomly generated byte streams and Wikipedia articles. Since realtime risk monitoring of SSH BFAs only contrasts BFs of entered passwords respectively their patterns with those of real ones, applying the idea of Wada, Matsumura, Nakano, and Ito allures, but would be an overkill.

Stiawan et al. feel obliged to track employees who play online games during working hours [13]. The author group applies Deep Packet Inspection (DPI) on network traffic to discern sessions of the fantasy game Dragon Nest. A BF serves as a tool to check the existence of any Dragon Nest's pattern in captured dumps. An additional visualization of the observations resulting from the DPI process proves an FP ratio of approximately 0.4 percent. The article of Stiawan et al. suggests implementing a realtime visualization of detected online game traffic patterns in its concluding section. The graphical capabilities of the prototype for realtime risk monitoring of SSH BFAs could be readily reused for this suggested implementation.

Bhat, Thilak, and Vaibhav strive for fast pattern matching with the support of BFs [14]. To this end, they benchmark the BF assembly time with a variety of cryptographic and non-cryptographic hash functions on Central Processing Units (CPUs) and Graphical Processing Units (GPUs). Generally, non-cryptographic hash functions outperform their cryptographic counterparts in execution speed but exhibit less randomness, resulting in a higher FP ratio. In contrast, cryptographic hash functions excel in stability. Each CPU consists of one or a few cores that sequentially run programs. Conversely, GPUs feature massively parallel architecture with thousands of small cores that efficiently handle a multitude of simultaneous tasks. Originally designed for graphics processing, GPUs now enjoy popularity for general-purpose computing due to their additional processing power. An empirical study by Bhat, Thilak, and Vaibhav confirms the expected superiority of non-cryptographic hash functions over cryptographic ones in terms of speed. The input size determines the competition between CPUs and GPUs. CPUs outperform GPUs with small inputs due to memory transfer latency in GPUs. However, a larger amount of data shifts the advantage to GPUs with their parallel mode of operation. Given the comparatively small input of realtime risk monitoring of SSH BFAs, which involves pairwise comparisons of password-fed BFs, the use of GPUs appears redundant.

Even though extensive literature research has revealed some interesting contributions, none of them seems suitable for implementing the four change requests of Sect. I.

III. Data Collection

This section examines how to reveal and manage plaintext credentials when offenders perform BFAs on an SSHD in any Linux operating system. For the sake of secrecy, an SSHD records metadata of every login attempt independently of its success. The list below exemplifies the most prominent ones.

1) Timestamp
2) Target host name
3) SSHD process identifier
4) Authentication result
5) Entered username
6) Source Internet Protocol (IP) address
7) Source Transmission Control Protocol (TCP) port

Opposing the primary focus of this project, a log file of this nature falls short in providing details on entered passwords to maintain confidentiality. Fortunately, modern Linux distributions incorporate Pluggable Authentication Modules (PAM) as a robust framework for revealing plaintext passwords. Altering a PAM configuration necessitates superuser privileges, effectively preventing regular users from wiretapping foreign passwords.

In brief, this section elucidates the modification of eight files as outlined subsequently.

1) /etc/ssh/sshd_config
2) /etc/pam.d/sshd
3) ∼/.google_authenticator
4) /usr/bin/passwd
5) /root/shadow
6) /root/sshd
7) /root/sshd.log
8) /etc/logrotate.d/sshd

The location of three files in the root directory ensures their confidentiality and protection from unauthorized access by non-administrator accounts.

A. */etc/ssh/sshd_config*

To capture all login attempts, an SSHD must forward each attempt to the PAM without rejection. This requires not restricting any groups or users by either commenting out (as indicated by the hash characters at the beginning of the upper four lines in the following list) or removing corresponding entries in the SSHD configuration file */etc/ssh/sshd_config*. Additionally, SSH access must be explicitly allowed for the root account, as shown in the fifth line. The sixth and final adjustment delegates the authentication process to the PAM.

1) # AllowGroups
2) # AllowUsers
3) # DenyGroups
4) # DenyUsers
5) PermitRootLogin yes
6) UsePAM yes

B. /etc/pam.d/sshd

By default, the PAM configuration file for SSHD (located at */etc/pam.d/sshd*) authenticates each local user with their password as a single factor, the hash code of which resides in the shadow file */etc/shadow*. To achieve this, a typical */etc/pam.d/sshd* usually calls the file */etc/pam.d/common-auth*, which contains common authentication settings for all services (as demonstrated by the middle line in the subsequent listing). Before invoking the settings in */etc/pam.d/common-auth*, */etc/pam.d/sshd* must execute an executable file, */root/sshd*, responsible for recording the entered password and creating its corresponding BF. The top-listed line indicates this additional step. Following the invocation of */etc/pam.d/common-auth*, an appended third line adds extra security by enforcing Two-Factor-Authentication (2FA) with temporary numbers. The logon process compels users to provide their currently valid time-based code immediately after entering their correct password. For this purpose, each user must utilize a code generator (app) that generates a fresh valid temporary number with six digits every 30 seconds.

1) auth optional pam_exec.so expose_authtok /root/sshd
2) @include common-auth
3) auth required pam_google_authenticator.so

C. ∼/.google_authenticator

In the current scenario of enforced 2FA, PAM coerces each user to set it up by executing */usr/bin/google-authenticator*. This command displays the initialization string for the code generator (app) and saves it (along with a few emergency codes) to the hidden file *∼/.google_authenticator* located in the home directory. Alternatively, a proficient user can manually create *∼/.google_authenticator*. The following example illustrates the structure of *∼/.google_authenticator*.

1) OWICUAE07LEN5Z4I247CZ0QW9W
2) " TOTP_AUTH
3) 00334742
4) 09035716
5) 46867186
6) 57704378
7) 87270038

D. /usr/bin/passwd

The well-known Linux command */usr/bin/passwd* allows a local system user to modify their password. Each update rewrites the user's hash in */etc/shadow* using a cryptographically secure one-way function. The function's design intentionally makes it practically impossible to convert a hash back to a cleartext password, for privacy purposes. Therefore, */usr/bin/passwd* must also compute all necessary BFs for a new password during its runtime, i.e., it has to perform the illustrated BF creation algorithm in Fig. 1 in addition.

For the sake of contrasting juxtaposition, */usr/bin/passwd* combinatorially produces BFs for the three dimensions as itemized hereinafter, i.e., 3 x 2 x 2 = 12.

1) **BF length:** 16, 32, and 64
2) **BF originality:** modified BF, original BF
3) **Test item:** password, password pattern

Once a user $USR has entered a new password $PWD, the additional program component initializes all BFs with zeros. This includes four hexadecimal-based BFs, each initialized with 16 zeros; four base32-based BFs, each with 32 zeros; and four base64-based BFs, each with 64 zeros. Additionally, it duplicates $PWD to create a string $PWD_PAT, of the same size, which will represent the pattern of $PWD after further modifications later in this algorithm. The subsequent loop iterates over all characters of $PWD that nearly spans the remainder of the BF creation algorithm.

Firstly, the loop modifies $PWD_PAT by transforming each character of $PWD position-wise. Digits in $PWD become *0* in $PWD_PAT, uppercase characters in $PWD turn to *A* in $PWD_PAT, lowercase characters in $PWD morph into *a* in $PWD_PAT, and special characters in $PWD mutate into _ in $PWD_PAT. Next, it applies the SHA3_384 and SHA3_512 hash functions separately to each character of $PWD and $PWD_PAT. To avoid redundant hashes, the input for both functions equals the concatenation of the character's position number, the character itself, and $USR.

A nested loop follows, compiling all hexadecimal-based BFs during its first iteration, all base32-based BFs during its second iteration, and all base64-based BFs during its last iteration. Accordingly, hexadecimal, base32, and base64 versions of the aforementioned hashes kick in.

The modified BF for $PWD being constructed flips its bit representing the first character of SHA3_384($position||$PWD[$position]||$USR]) from zero to one or vice versa. The same applies to the representing bit of the starting character of SHA3_512($position||$PWD[$position]||$USR]).

Creating the original BF for $PWD turns out to be simpler. Its bits representing the initial characters of SHA3_384($position||$PWD[$position]||$USR]) and SHA3_512($position||$PWD[$position]||$USR]) unconditionally become one regardless of their prior states.

Similarly as for $PWD, the algorithm continues by generating the modified and original BFs for

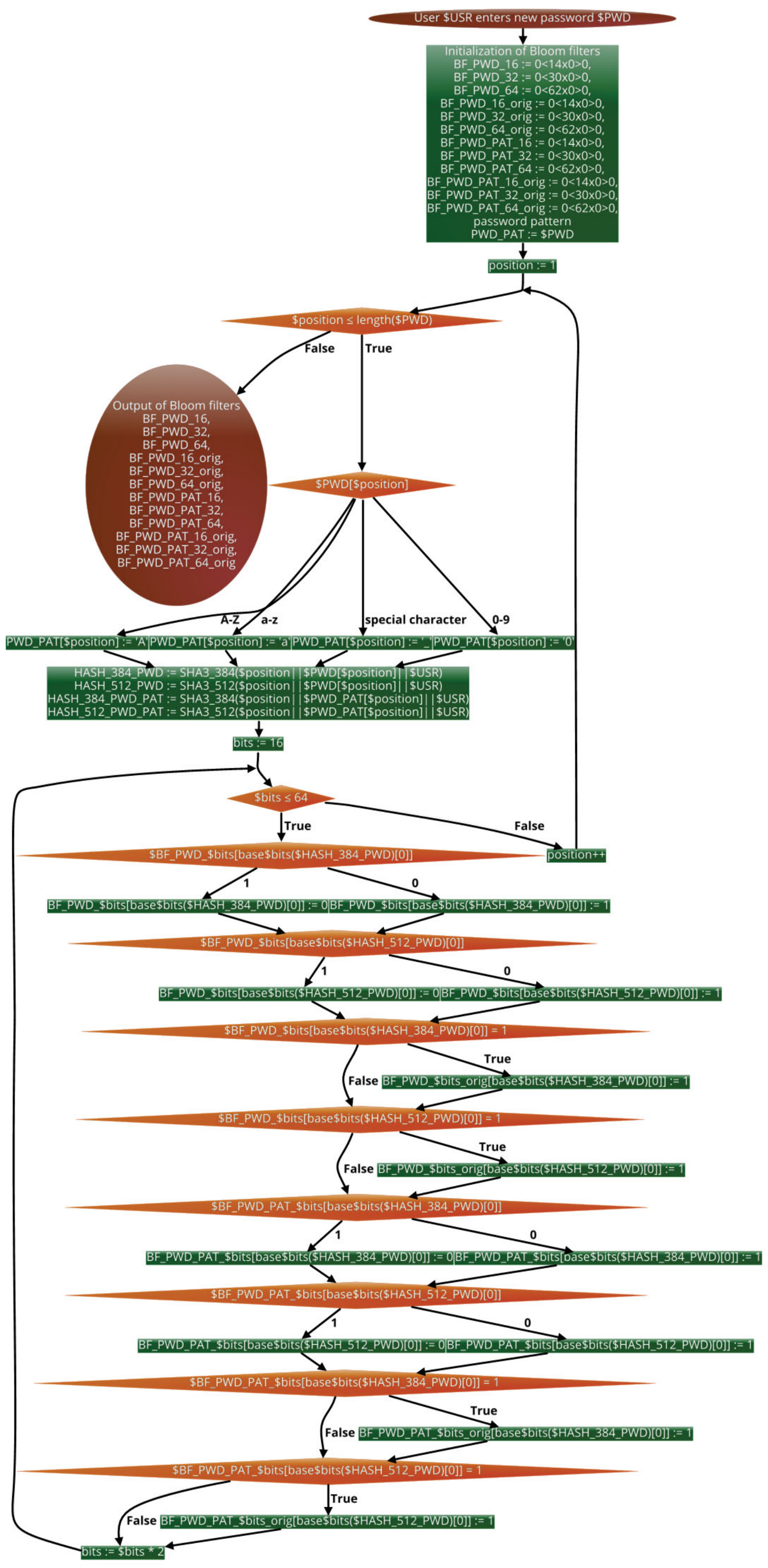
User $USR enters new password $PWD
Initialization of Bloom filters
BF_PWD_16 := 0<14x0>0,
BF_PWD_32 := 0<30x0>0,
BF_PWD_64 := 0<62x0>0,
BF_PWD_16_orig := 0<14x0>0,
BF_PWD_32_orig := 0<30x0>0,
BF_PWD_64_orig := 0<62x0>0,
BF_PWD_PAT_16 := 0<14x0>0,
BF_PWD_PAT_32 := 0<30x0>0,
BF_PWD_PAT_64 := 0<62x0>0,
BF_PWD_PAT_16_orig := 0<14x0>0,
BF_PWD_PAT_32_orig := 0<30x0>0,
BF_PWD_PAT_64_orig := 0<62x0>0,
password pattern
PWD_PAT := $PWD
position := 1
$position ≤ length($PWD)
False
True
Output of Bloom filters
BF_PWD_16,
BF_PWD_32,
BF_PWD_64,
BF_PWD_16_orig,
BF_PWD_32_orig,
BF_PWD_64_orig,
BF_PWD_PAT_16,
BF_PWD_PAT_32,
BF_PWD_PAT_64,
BF_PWD_PAT_16_orig,
BF_PWD_PAT_32_orig,
BF_PWD_PAT_64_orig
$PWD[$position]
A-Z
a-z
special character
0-9
PWD_PAT[$position] := 'A'
PWD_PAT[$position] := 'a'
PWD_PAT[$position] := '_'
PWD_PAT[$position] := '0'
HASH_384_PWD := SHA3_384($position||$PWD[$position]||$USR)
HASH_512_PWD := SHA3_512($position||$PWD[$position]||$USR)
HASH_384_PWD_PAT := SHA3_384($position||$PWD_PAT[$position]||$USR)
HASH_512_PWD_PAT := SHA3_512($position||$PWD_PAT[$position]||$USR)
bits := 16
$bits ≤ 64
True
False
$BF_PWD_$bits[base$bits($HASH_384_PWD)[0]]
position++
1
0
BF_PWD_$bits[base$bits($HASH_384_PWD)[0]] := 0
BF_PWD_$bits[base$bits($HASH_384_PWD)[0]] := 1
$BF_PWD_$bits[base$bits($HASH_512_PWD)[0]]
1
0
BF_PWD_$bits[base$bits($HASH_512_PWD)[0]] := 0
BF_PWD_$bits[base$bits($HASH_512_PWD)[0]] := 1
$BF_PWD_$bits[base$bits($HASH_384_PWD)[0]] = 1
True
False
BF_PWD_$bits_orig[base$bits($HASH_384_PWD)[0]] := 1
$BF_PWD_$bits[base$bits($HASH_512_PWD)[0]] = 1
True
False
BF_PWD_$bits_orig[base$bits($HASH_512_PWD)[0]] := 1
$BF_PWD_PAT_$bits[base$bits($HASH_384_PWD)[0]]
1
0
BF_PWD_PAT_$bits[base$bits($HASH_384_PWD)[0]] := 0
BF_PWD_PAT_$bits[base$bits($HASH_384_PWD)[0]] := 1
$BF_PWD_PAT_$bits[base$bits($HASH_512_PWD)[0]]
1
0
BF_PWD_PAT_$bits[base$bits($HASH_512_PWD)[0]] := 0
BF_PWD_PAT_$bits[base$bits($HASH_512_PWD)[0]] := 1
$BF_PWD_PAT_$bits[base$bits($HASH_384_PWD)[0]] = 1
True
False
BF_PWD_PAT_$bits_orig[base$bits($HASH_384_PWD)[0]] := 1
$BF_PWD_PAT_$bits[base$bits($HASH_512_PWD)[0]] = 1
True
False
BF_PWD_PAT_$bits_orig[base$bits($HASH_512_PWD)[0]] := 1
bits := $bits * 2

Fig. 1. BF creation

$PWD_PAT, derived from the initial characters of SHA3_384($position||$PWD_PAT[$position]||$USR) and SHA3_512($position||$PWD_PAT[$position]||$USR).
Finally, */usr/bin/passwd* replaces the twelve BFs in the line for $USR in the file */root/shadow* with the output of the BF creation algorithm.

E. /root/shadow

The file */root/shadow* stores the twelve aforementioned BFs for the configured password of each user account. The following stanza illustrates the order in which the algorithm in Fig. 1 stores all the BFs for each user account in a line within */root/shadow*.

$USR
$BF_PWD_PAT_16_orig $BF_PWD_PAT_32_orig
$BF_PWD_PAT_64_orig
$BF_PWD_PAT_16 $BF_PWD_PAT_32
$BF_PWD_PAT_64
$BF_PAT_16_orig $BF_PAT_32_orig
$BF_PAT_64_orig
$BF_PAT_16 $BF_PAT_32
$BF_PAT_64

The following paragraph visualizes the BFs for the user $root$ with the password $abcdef$ in */root/shadow*, which would be a one-liner there.

```
root
1100000101100111 01010000000000010010110000111001
0010001000000000000000000000000100001100011000000000101010000001
1100000000000011 01010000000000000000110000111001
0010001000000000000000000000000000001100011000000000101010000001
1000010100100111 01000000001100110000100000011001
0010000000000000000001110000001010000000001000000000001001000010
1000010000100111 01000000000100000000100000011001
0010000000000000000001110000000000000000001000000000001001000010
```

F. /root/sshd

This publication gains merit from the presence of the executable file */root/sshd*. To reiterate, the line *auth optional pam_exec.so expose_authtok /root/sshd* in */etc/pam.d/sshd* executes */root/sshd* during every login attempt, which expects input of $USR and $PWD for further processing.
In the event of a correctly entered $PWD, its hash matches that in */etc/shadow*. Since the significance of successful logins dwindles to zero during BFAs, */root/sshd* immediately terminates under such circumstances.
Conversely, the opposite case requires verifying the similarity between $PWD and the actual password by performing a binary comparison of their BFs. To that end, */root/sshd* computes all twelve BFs of $PWD (just as */usr/bin/passwd* does) without saving them to */root/shadow*. Instead, it contrasts them with their counterparts in */root/shadow* bit by bit. The quotient of matching and total bits yields the desired similarity index per BF, i.e., */root/sshd* calculates twelve similarity indexes. In the end, */root/sshd* logs the timestamp, $USR, and all twelve likeness indexes to the file */root/sshd.log*.

G. /root/sshd.log

The log file */root/sshd.log* retains the timestamps, usernames, and all twelve similitude indexes of all past failed login attempts that have occurred since its last rotation (as described in the next subsection). Furthermore, the corresponding source IP addresses and cleartext passwords enhance each log line for the purpose of manually verifying the similarity indexes. The stanza below reveals the order of the variables recorded per line.
$timestamp $s_PWD_PAT_16_orig $s_PWD_PAT_32_orig $s_PWD_PAT_64_orig $s_PWD_PAT_16 $s_PWD_PAT_32 $s_PWD_PAT_64 $s_PAT_16_orig $s_PAT_32_orig $s_PAT_64_orig $s_PAT_16 $s_PAT_32 $s_PAT_64 $ip $PWD
The following paragraph provides a fictitious example illustrating these variables with specific values.
202403020100 root 0.5 0.53125 0.671875 0.625 0.625 0.71875 0.5 0.46875 0.671875 0.5 0.625 0.75 1.2.3.4 336331jum

H. /etc/logrotate.d/sshd

The volume of */root/sshd.log* steadily grows with each failed login attempt. This would progressively hinder the computation of metrics in Sect. IV if it continuously scanned the abundance of historical login failures. Therefore, the configuration content below from the file */etc/logrotate.d/sshd* ensures daily rotation of */root/sshd.log* and preservation for 366 days, also archiving all log files of a leap year entirely.

1) /root/sshd.log
2) {
3) copytruncate
4) daily
5) missingok
6) notifempty
7) rotate 366
8) }

Restarting SSHD triggers the outlined changes in this section, necessary for generating metrics.

IV. Metrics Generation

As */root/sshd* calculates and preserves twelve distinct similarity indexes for every failed login attempt, it stands to reason to prepare an equal number of independent metrics. Each metric will indicate the riskiest login failure(s) from the previous time span, namely those with the highest similarity score. This necessitates adjustments in both of the listed files thereafter.

1) /root/metric
2) /etc/snmp/snmpd.conf

A. /root/metric

The executable file */root/metric* requires invocation by a CMS alongside one parameter determining which similarity index, as described in Sect. III, it will utilize for metric generation. As per the request, */root/metric* scans all corresponding similarity values logged by */root/sshd* in */root/sshd.log* during the last time interval. The program evaluates the peak value and exits by outputting it. Evaluating the highest similarity instead of summing all scanned scores makes more sense, as a single login attempt with a very similar password often poses a greater threat than multiple attempts with less similarity.

B. */etc/snmp/snmpd.conf*

Executing */root/metric* merely provides a snapshot of the requested metric rather than its complete history. A CMS lends itself to regularly fetching metrics, visualizing their histograms, and identifying and notifying anomalies. Among other options, the Simple Network Management Protocol (SNMP) [15] suits securely and efficiently transmitting metrics from SNMP-capable nodes to a CMS using Object Identifiers (OIDs). Specifically, SNMPv3 (the third version of SNMP) supports the confidential and unaltered transport of OIDs [16]. As an SNMP Daemon (SNMPD) authenticates incoming SNMPv3 requests based on usernames, having an operational SNMP account ensures accessibility for a CMS. The following excerpt from a sample SNMPD configuration file */etc/snmp/snmpd.conf* authorizes the username *checkmk* to fetch all twelve metrics.

1) rouser checkmk
2) exec PWD_PAT_16_orig /root/metric PWD_PAT_16_orig
3) exec PWD_PAT_32_orig /root/metric PWD_PAT_32_orig
4) exec PWD_PAT_64_orig /root/metric PWD_PAT_64_orig
5) exec PWD_PAT_16 /root/metric PWD_PAT_16
6) exec PWD_PAT_32 /root/metric PWD_PAT_32
7) exec PWD_PAT_64 /root/metric PWD_PAT_64
8) exec PWD_16_orig /root/metric PWD_16_orig
9) exec PWD_32_orig /root/metric PWD_32_orig
10) exec PWD_64_orig /root/metric PWD_64_orig
11) exec PWD_16 /root/metric PWD_16
12) exec PWD_32 /root/metric PWD_32
13) exec PWD_64 /root/metric PWD_64

Each line containing the keyword *exec* assigns a leaf within the Object Identifier (OID) subtree 1.3.6.1.4.1.2021.8.1.101 to the mentioned metric. This allocation occurs sequentially, meaning SNMPD begins by mapping the subordinate OID 1.3.6.1.4.1.2021.8.1.101.1 to PWD_PAT_16_orig and concludes with 1.3.6.1.4.1.2021.8.1.101.12 for PWD_64.
Upon SNMPD's operation with the updated */etc/snmp/snmpd.conf*, a CMS can then request the twelve metrics using the appropriate credentials.

V. Experiment

Before delving into the details of the conducted experiment, a few thoughts on the anticipated metrics deserve contemplation.
Firstly, given that each similarity score computed by */root/sshd* results in a positive fractional value equal to or less than one, it follows that */root/metric* can only provide metrics with maximum values equal to or smaller than one.
Secondly, an alikeness score of zero indicates that every bit of a BF differs from that of another, while a likeness score of one signifies their complete equality. Moreover, */root/sshd* never records similitude indexes of succeeded logons in */root/sshd.log*. Consequently, an alikeness score of one suggests the submission of a highly similar, albeit incorrect password rather than a correct one.
Thirdly, the original BF [17], utilized for six out of twelve similarity indexes, maintains set bits without resetting them to zero during its operation. In contrast, the adapted BF [2], employed for the remaining six alikeness scores, may reset ones to zeros during its compilation. Consequently, the occurrence of zeros and ones in modified BFs converges with increasing password length, resulting in similarity scores of 0.5 even for markedly dissimilar passwords. This characteristic solely renders alikeness indexes (derived from adapted BFs) between 0.5 and one meaningful.
Fourthly, the simultaneous utilization of multiple hash functions leads to the alteration of an equal number of bits in a modified BF per password character. Both */usr/bin/passwd* and */root/sshd* employ the SHA3-512 and SHA3-384 hash functions, resulting in an equal distribution of ones and zeros in these BFs. Consequently, these BFs only yield likeness indexes as positive fractions with even numerators, satisfying the condition: $\frac{\text{count of coherent bits}}{\text{BF length}} | \text{count of coherent bits} \equiv 0 \pmod{2} \wedge \text{BF length} \in \{16, 32, 64\}$. Hence, the use of two hash functions reduces the resolution of indexes derived from adapted BFs by half compared to using a single hash function.
Typically, adhering to best practices involves utilizing complex and lengthy passwords, which undeniably enhances security. However, this approach diminishes the experiment's presentability in terms of BF-based real-time risk monitoring of SSH BFAs. Why? Simply put, attackers would fortunately struggle to guess well-configured strong passwords or those that only marginally deviate from them. Consequently, the twelve generated similarity values from */root/sshd* and metrics from */root/metric* would tend to fluctuate around their baselines, with occasional spikes as FPs. To address this security versus presentation dilemma, configuring a deliberately weak password does the trick, providing no access under any circumstance. Choosing the superuser account root for this purpose seems appropriate because attackers commonly target root accounts for BFAs. Furthermore, root access should ideally be restricted to using the commands */usr/bin/su* or */usr/bin/sudo* through intermediary low-privileged accounts, rather than allowing direct login. Deliberately weakening security with a simple root password and then removing */root/.google¯authenticator* dooms all login attempts to root to fail, even those with correct passwords. To further weaken the root password, selecting a dictionary entry proves advisable. The infamous *rockyou.txt* file, stemming from the 2009 incident involving the US company RockYou, serves as a rich source for deliberately vulnerable passwords. This repository contains a staggering number of distinct cleartext passwords from compromised RockYou customers, which can also be configured for Linux system accounts. Easily accessible via Internet searches and available for download from various sources, the file has evolved over time, with the 2021 version, *rockyou2021.txt*, containing even more leaked passwords. Among the subsets within these files, words consisting of

precisely six lowercase characters stand out, with *abcdef* being a notable representative. As both predecessors of this disquisition explain, the latter becomes the experimental root password [1], [2]. Admittedly, the deliberate choice of easily guessable credentials enhances presentability but simultaneously limits the relevance of the results for real-world use cases.

An SSHD instance running on Ubuntu Linux 23.10, configured according to the specifications outlined in this document, including the use of simplistic credentials, became exposed to the Internet in March 2024. The network architecture depicted in Fig. 2 illustrates the SSHD's placement within a Demilitarized Zone (DMZ), along with the utilization of SNMP and SSH protocol, and their corresponding daemons on Ubuntu Linux 23.10. Upon execution of */root/metric*, the desired metric got evaluated by considering all recorded events from the preceding minute. Consequently, a CMS polled a fresh value for each metric every 60 seconds via an SNMPv3-request. While the predecessors' performed experiments [1]–[5] utilized the long-established Nagios as CMS, this writ breaks the mold by relying on an up-to-date version of Checkmk, a product developed from Nagios Core [18].

To ensure consistency with previous papers [1], [2], the CMS computed a fixed threshold (pair), three individual dynamic (critical) thresholds, and three techniques for selecting dynamic threshold pairs for each metric, as outlined below.

1) **Fixed approach:** The (critical) threshold corresponds to a similarity score that arises from comparing two BFs differing in exactly one character (two bits for original BFs or four bits for revised BFs).
2) **Three sigma rule without prior outlier removal:** The (critical) threshold resides three standard deviations above the arithmetic mean of an unfiltered metric history [1]–[5].
3) **Three sigma rule with prior outlier removal:** The (critical) threshold sits three standard deviations above the arithmetic mean of an metric history cleared of outliers [1], [2], [4], [5].
4) **Maximal value:** The (critical) threshold takes on the maximum value from an outlier-free metric history [1], [2], [4], [5].
5) **Tolerant approach:** The maximum of an unfiltered array with dynamical thresholds serves as the critical threshold, with the warning threshold set at the median [1], [2], [5].
6) **Balanced approach:** The maximum of an unfiltered array with dynamical thresholds represents the critical threshold, with the minimum serving as the warning threshold [1], [2], [5].
7) **Strict approach:** The critical threshold derives from the median of an unfiltered array with dynamical thresholds, with the minimum serving as the warning threshold [1], [2], [5].

The CMS calculated dynamic thresholds for each metric by considering up to 52 previous values collected over the past 365 days. Specifically, it utilized metric values from exactly one week ago, two weeks ago, and so forth up to 52 weeks ago. This comprehensive approach ensures coverage across all potential attack seasons throughout the year.

To mitigate desensitization resulting from overwhelming surges of notifications caused by short-term peaks, a notification will only be triggered after ten consecutive threshold exceedances, i.e., after a nine-minute retention period. This protocol accommodates scenarios where eligible users unintentionally input slightly incorrect passwords, leading to metric spikes beyond their thresholds.

All value pairs (each separated by a slash, indicating the quantity of threshold exceedances and resultant notifications) in the subsequent twelve tables serve as evidence of the effectiveness of the nine-minute retention time interval. This becomes apparent through the count of notifications on the right of slashes, consistently being lower than the threshold exceedances on the left. All twelve tables show the count of threshold exceedances categorized by the aforementioned threshold (pair) calculation methods, leading to warning and critical conditions followed by triggered notifications. The tables differ in their employed BF size and, thus, have an individual fixed (critical) threshold set, which increases with the BF size.

A pairwise contrasting juxtaposition of all twelve tables would be indisputably tedious and meaningless. The comparison graph in Fig. 3 provides a remedy by remarkably condensing the analysis to seven expressive comparisons, as follows hereinafter.

A. *Comparison between PWD_PAT_16_orig, PWD_PAT_32_orig, and PWD_PAT_64_orig*

The elaborated fixed (critical) thresholds in Tables I (PWD_PAT_16_orig), II (PWD_PAT_32_orig), and III (PWD_PAT_64_orig) ensure that all keyed-in passwords with barely one wrong character (equating to two deviating bits in original BFs) exceed them. It must be noted that even a BF of a password with more than one incorrect character can deviate purely in two bits from the true password's BF, leading to excessively high assessed similitude scores and threshold exceedances. This type of FPs occurs if correct and incorrect characters produce identical hashes and, thus, set identical bits in a BF, resulting from hash collisions. A glance at the three tables substantiates that smaller BFs tend to register more FPs of fixed (critical) threshold exceedances and resultant notifications, as they possess a smaller size. The decreasing rate of FPs argues in favor of employing larger BF sizes.

An examination of the dynamic threshold (pair) evaluation approaches consistently reveals zero or between 333 and 337 threshold exceedances in Tables I (PWD_PAT_16_orig) and II (PWD_PAT_32_orig), whereas Table III (PWD_PAT_64_orig) significantly exceeds this range, with figures ranging from 897 to 908. This further supports the preference for larger BF sizes, as they exhibit greater sensitivity to threshold exceedances without triggering redundant notifications.

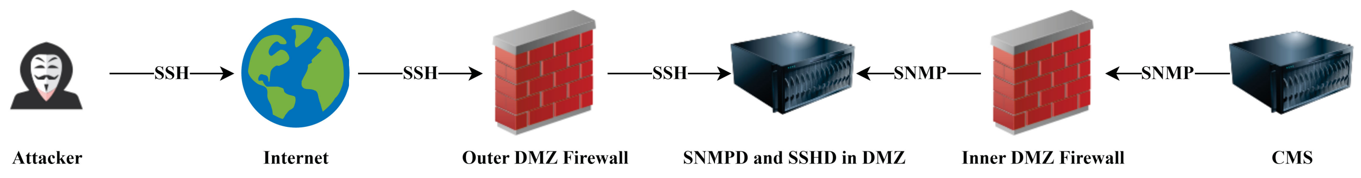

Fig. 2. Experimental rigging

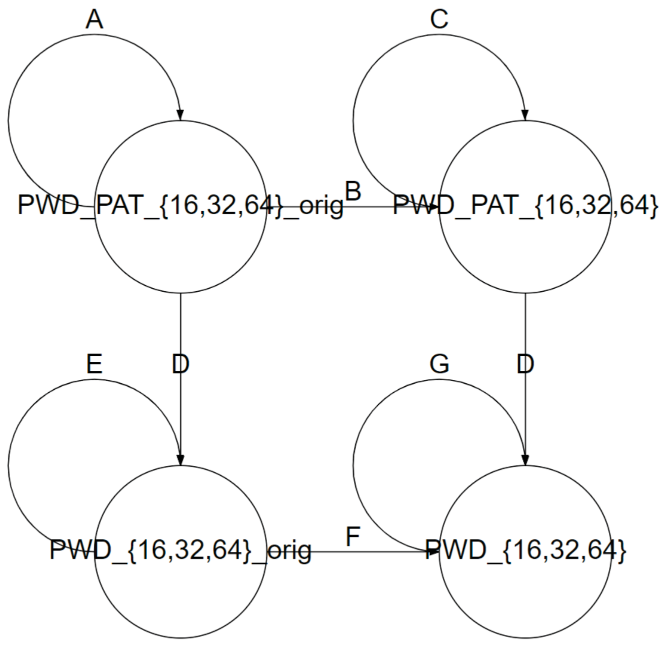

Fig. 3. Comparison graph

Ultimately, one can observe that the regularity of the BFAs contributes to the mastery of all dynamic threshold (pair) derivation methods, leading to zero notifications.

B. Comparison between PWD_PAT_{16, 32, 64}_orig and PWD_PAT_{16, 32, 64}

The lower fixed (critical) thresholds immediately catch a reader's eye when comparing the usage of original BFs with that of revised BFs. This lowering happens because exchanging one character of a string may flip up to four bits in an adapted BF, rather than maximally two in the case of an original BF. This reduction results in a fixed (critical) threshold of 0.75 in Table IV (PWD_PAT_16) rather than 0.875 in Table I (PWD_PAT_16_orig), 0.875 in Table V (PWD_PAT_32) instead of 0.9375 in Table II (PWD_PAT_32_orig), and 0.9375 in Table VI (PWD_PAT_64) in place of 0.96875 in Table III (PWD_PAT_64_orig). Just as original BFs, modified BFs of keyed-in passwords with more than one wrong character can have four or fewer incorrect bits due to hash collisions. On average, Tables IV (PWD_PAT_16), V (PWD_PAT_32), and VI (PWD_PAT_64) display more exceedances of their fixed (critical) thresholds than Tables I (PWD_PAT_16_orig), II (PWD_PAT_32_orig), and III (PWD_PAT_64_orig). This suggests more hash collisions.

TABLE I
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_16_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.875	N/A	17,396/235
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	333/0
Tolerant approach	0/0	0/0
Balanced approach	335/0	0/0
Strict approach	337/0	0/0

TABLE II
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_32_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.9375	N/A	15,800/159
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	333/0
Tolerant approach	0/0	0/0
Balanced approach	333/0	0/0
Strict approach	334/0	0/0

TABLE III
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_64_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.96875	N/A	15,578/156
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	897/0
Tolerant approach	0/0	0/0
Balanced approach	897/0	0/0
Strict approach	908/0	0/0

The threshold exceedances and notifications of dynamical thresholds based on revised BFs bear resemblance to those in Table III (PWD_PAT_64_orig).

C. Comparison between PWD_PAT_16, PWD_PAT_32, and PWD_PAT_64

The comparison in Subsect. V-A describes many more exceedances of dynamical thresholds in Table III (PWD_PAT_64_orig) than in Tables I (PWD_PAT_16_orig) and II (PWD_PAT_32_orig) at the same count of notifications,

TABLE IV
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_16)

	Warning	Critical
Fixed (critical) threshold of 0.75	N/A	18,362/268
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	877/0
Tolerant approach	0/0	0/0
Balanced approach	888/0	0/0
Strict approach	865/0	0/0

TABLE V
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_32)

	Warning	Critical
Fixed (critical) threshold of 0.875	N/A	17,431/230
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	877/0
Tolerant approach	0/0	0/0
Balanced approach	895/0	0/0
Strict approach	899/0	0/0

TABLE VI
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_PAT_64)

	Warning	Critical
Fixed (critical) threshold of 0.9375	N/A	17,255/220
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	867/0
Tolerant approach	0/0	0/0
Balanced approach	877/0	0/0
Strict approach	870/0	0/0

TABLE VII
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_16_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.875	N/A	152/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	2,411/0
Tolerant approach	0/0	0/0
Balanced approach	2,433/0	0/0
Strict approach	2,404/0	0/0

TABLE VIII
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_32_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.9375	N/A	119/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	2,343/0
Tolerant approach	0/0	0/0
Balanced approach	2,322/0	0/0
Strict approach	2,309/0	0/0

TABLE IX
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_64_ORIG)

	Warning	Critical
Fixed (critical) threshold of 0.96875	N/A	87/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	2,258/1
Tolerant approach	0/0	0/0
Balanced approach	2,291/1	0/0
Strict approach	2,280/1	0/0

which favors large-sized BFs. A glance at Tables IV (PWD_PAT_16), V (PWD_PAT_32), and VI (PWD_PAT_64) surprises with almost identical figures regarding exceedances of dynamical thresholds in the range from 865 to 899, as well as the absence of resultant notifications. This demonstrates that even small-sized revised BFs keep pace with large-sized original BFs.

D. Comparison between PWD_PAT_{16, 32, 64}[_orig] and PWD_{16, 32, 64}[_orig]

To recap, Tables I to VI depict threshold exceedances and resulting notifications based on similarity indexes of BFs compiled by password patterns. Tables VII to XII perform the same analysis directly with the received passwords. Their conversion to patterns does not apply in these tables. Given the reused fixed (critical) thresholds, the exceedances and resulting notifications notably fall lower than those in the first six tables. This can be explained by the increased difficulty for an attacker to guess a correct password compared to its pattern.

A contrasting juxtaposition of the dynamic threshold exceedances reveals a surprising trend. Ranging between 1,495 and 2,512, they conspicuously surpass their counterparts in Tables I to VI and the measured exceedances of the fixed (critical) thresholds in Tables VII to XII. An examination of the histograms in the CMS reveals that many passwords during BFAs deviate significantly from genuine ones. Consequently, the employed threshold (pair) reckoning algorithms determine lower thresholds than the fixed ones, resulting in more exceedances.

E. Comparison between PWD_16_orig, PWD_32_orig, and PWD_64_orig

Once more, a decrease in the exceedances of the fixed (critical) thresholds can be observed when examining Table VII (PWD_16_orig) before Tables VIII (PWD_32_orig) and IX (PWD_64_orig). This again confirms the negative correlation between BF size and FP rate. No threshold exceedance streak lasted longer than nine minutes and triggered any notification. Among all the tables considered in this subsection, Table IX grabs the reader's attention with three threshold (pair) elaboration techniques, each resulting in one notification. A glance at the notification history in the CMS reveals and validates the corresponding notifications on 3/18/24 at 11:34 A.M., each provoked by ten consecutive threshold exceedances beforehand. Only the BF with the largest size in this survey detected this anomaly. This once again suggests the importance of utilizing BFs with larger sizes whenever possible.

F. Comparison between PWD_{16, 32, 64}_orig and PWD_{16, 32, 64}

Tables XI (PWD_32) and XII (PWD_64) exhibit moderately higher exceedances of their fixed (critical) thresholds compared to Tables VIII (PWD_32_orig) and IX (PWD_64_orig). The jump from 152 exceedances in Table VII (PWD_16_orig) to unrealistic 1,173 in Table X (PWD_16) stands out conspicuously and indicates too many FPs, likely due to its relatively low fixed (critical) threshold of 0.75. Subsection V-A delves into this phenomenon.

A similar scenario arises with the dynamic thresholds. Of all the tables, Table X (PWD_16) once more draws attention with its markedly lower number of exceedances across all dynamic thresholds compared to those present in Table VII (PWD_16_orig). 16 bits clearly do not suffice for revised BFs to operate accurately, once more emphasizing the necessity for larger BFs. Minor but noteworthy for the sake of completeness, both the maximum value and the strict approach in Table XII (PWD_64) triggered a second notification in comparison to Table IX (PWD_64_orig).

G. Comparison between PWD_16, PWD_32, and PWD_64

Table X (PWD_16) presents another novelty among all twelve tables. A single minute during the observation period witnessed an instance where at least one BFA exceeded the critical threshold of the strict approach. The CMS reveals that the aforementioned minute occurred on 3/17/24 at 03:11 A.M. Further investigation reveals that ten consecutive threshold exceedances on 3/25/24 at 06:04 P.M. caused the second notification concerning the two approaches mentioned earlier in Table XII in Subsect. V-F.

What inference can be drawn from the seven recent contrasting juxtapositions?

As expected, the comparison in Subsect. V-D satisfactorily demonstrates universally lower threshold exceedances for password-based metrics compared to their pattern-based counterparts. This results from the higher difficulty of guessing passwords compared to their patterns.

The exertion of BFs with solely 16 bits particularly challenges this assertion. Just through a single exchanged character in a string, up to two bits in its original and four in its adapted BF can be altered. Considering 16 bits of a BF, this may impact 25% and 12.5% of them, respectively, opening the floodgates to FPs. This deficiency in sufficient BF bits leads to outlier figures in Tables I (PWD_PAT_16_orig), II (PWD_PAT_32_orig), and X (PWD_PAT_16), strongly advocating for the utilization of 64-bit long BFs.

This suggests that both original and adapted BFs can be prone to FPs when they comprise insufficient size. Even Table II (PWD_PAT_32_orig), based on 32-bit long BFs, embodies FPs. Opting for revised BFs seems to be the safer choice against FPs.

TABLE X
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_16)

	Warning	Critical
Fixed (critical) threshold of 0.75	N/A	1,173/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	1,520/0
Tolerant approach	0/0	0/0
Balanced approach	1,495/0	0/0
Strict approach	1,503/0	1/0

TABLE XI
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_32)

	Warning	Critical
Fixed (critical) threshold of 0.875	N/A	160/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	2,512/0
Tolerant approach	0/0	0/0
Balanced approach	2,480/0	0/0
Strict approach	2,500/0	0/0

TABLE XII
THRESHOLD EXCEEDANCES/NOTIFICATIONS OF BFAS AGAINST VALID USERNAMES (PWD_64)

	Warning	Critical
Fixed (critical) threshold of 0.9375	N/A	150/0
Three sigma rule without prior outlier removal	N/A	0/0
Three sigma rule with prior outlier removal	N/A	0/0
Maximum value	N/A	2,380/2
Tolerant approach	0/0	0/0
Balanced approach	2,368/1	0/0
Strict approach	2,368/2	0/0

VI. CONCLUSION

Section I introduces four change requests that this document must address to more effectively answer the research question related to assessing the quality of BFAs.

Section II provides a literature review dedicated to pattern matching based on BFs. Apart from the predecessors of this publication, it introduces nine related citations and explains why they do not satisfactorily address the four aforementioned

claims.
Figure 1 depicts an algorithmic flowchart for BF generation instead of pseudocode. The executables */usr/bin/passwd* and */root/sshd* strictly implement this algorithm.
In line with Fig. 2, a server based on Ubuntu Linux 23.10 in a DMZ entices BFAs against its SSHD. This host also generates log data as outlined in Sect. III and derives appropriate metrics as explained in Sect. IV. Subsequently, a state-of-the-art CMS like Checkmk securely collects the prepared metrics via SNMPv3. The outcome of the experiment at the end of Sect. V addresses the initial research question by confirming the improved efficiency of this workflow.
In detail, Tables I to VI evaluate the similarity between patterns of attempted and legitimate passwords based on original and adapted BFs. Additionally, Subsects. V-A to V-C quantitatively describe the discrepancies between them. Tables VII to XII, along with Subsects. V-E to V-G, provide comparable insights into the similarity between tried and actual passwords based on original and modified BFs. Additionally, Subsect. V-D contrasts the first six tables with the last six. All the contrasting juxtapositions together suggest that both original and adapted BFs may experience FPs when their size remains insufficient. Nevertheless, choosing revised BFs appears to present a safer option against FPs and, therefore, serves as a favored way to answer the research question.
Eventually, it should be noted that conventional ML algorithms can and ought to be prospectively employed to calculate thresholds from BF-based metrics.

ACKNOWLEDGMENT

Many thanks to Bettina Baumgartner from the University of Vienna for proofreading this paper!

REFERENCES

[1] G. Fahrnberger, “Realtime Risk Monitoring of SSH Brute Force Attacks,” in *Innovations for Community Services*, ser. Communications in Computer and Information Science, F. Phillipson, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds. Springer Cham, jun 2022, pp. 75–95. [Online]. Available: https://doi.org/10.1007/978-3-031-06668-9_8

[2] ——, “Bloom Filter-Based Realtime Risk Monitoring of SSH Brute Force Attacks,” in *Innovations for Community Services*, ser. Communications in Computer and Information Science, U. R. Krieger, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds. Springer Cham, sep 2023, pp. 48–67. [Online]. Available: https://doi.org/10.1007/978-3-031-40852-6_3

[3] ——, “Reliable Condition Monitoring of Telecommunication Services with Time-Varying Load Characteristic,” in *Distributed Computing and Internet Technology*, ser. Lecture Notes in Computer Science, A. Negi, R. Bhatnagar, and L. Parida, Eds. Springer International Publishing, jan 2018, pp. 173–188. [Online]. Available: https://doi.org/10.1007/978-3-319-72344-0_14

[4] ——, “Outlier Removal for the Reliable Condition Monitoring of Telecommunication Services,” in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2019 20th International Conference on*. IEEE, dec 2019, pp. 240–246. [Online]. Available: https://doi.org/10.1109/PDCAT46702.2019.00052

[5] ——, “Threshold Pair Selection for the Reliable Condition Monitoring of Telecommunication Services,” in *Innovations for Community Services*, ser. Communications in Computer and Information Science, U. R. Krieger, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds. Springer International Publishing, may 2021, pp. 9–21. [Online]. Available: https://doi.org/10.1007/978-3-030-75004-6_2

[6] N. D. A. Tuan, B. T. Hieu, and T. N. Thinh, “High Performance Pattern Matching Using Bloom-Bloomier Filter,” in *ECTI-CON2010: The 2010 ECTI International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. IEEE, may 2010, pp. 870–874.

[7] B. T. Hieu, N. D. A. Tuan, and T. N. Thinh, “BBFex: An Efficient FPGA-based Design for Long Patterns in Pattern Matching System,” in *2010 International Conference on Intelligent Network and Computing (ICINC 2010)*. IEEE, nov 2010, pp. 157–162.

[8] S. Liu, L. Kang, L. Chen, and L. Ni, “Distributed Incomplete Pattern Matching via a Novel Weighted Bloom Filter,” in *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, jun 2012, pp. 122–131. [Online]. Available: https://doi.org/10.1109/ICDCS.2012.24

[9] J. S. Pande and J. W. Bakal, “Distributed Incomplete Pattern Matching Using Unsupervised Weighted Bloom Filter,” *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, vol. 4, no. 4, pp. 250–253, apr 2015. [Online]. Available: https://www.ijaiem.org/Volume4Issue4/IJAIEM-2015-04-30-89.pdf

[10] A. Al-Tariq, A. R. M. Kamal, M. A. Hamid, M. Abdullah-Al-Wadud, M. M. Hassan, and S. M. M. Rahman, “A Scalable Framework for Protecting User Identity and Access Pattern in Untrusted Web Server Using Forward Secrecy, Public Key Encryption and Bloom Filter,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 23, pp. 1–25, jun 2016. [Online]. Available: https://doi.org/10.1002/cpe.3863

[11] S. M. El-Ghamrawy, “A Knowledge Management Framework for Imbalanced Data Using Frequent Pattern Mining Based on Bloom Filter,” in *2016 11th International Conference on Computer Engineering & Systems (ICCES)*. IEEE, dec 2016, pp. 226–231. [Online]. Available: https://doi.org/10.1109/ICCES.2016.7822004

[12] T. Wada, N. Matsumura, K. Nakano, and Y. Ito, “Efficient Byte Stream Pattern Test using Bloom Filter with Rolling Hash Functions on the FPGA,” in *2018 Sixth International Symposium on Computing and Networking (CANDAR)*. IEEE, nov 2018, pp. 66–75. [Online]. Available: https://doi.org/10.1109/CANDAR.2018.00016

[13] D. Stiawan, M. Y. Idris, D. Aryandi, A. Heryanto, T. W. Septian, F. Muchtar, and R. Budiarto, “Behavior Pattern Recognition of Game Dragon Nest Using Bloom Filter Method,” *International Journal of Communication Networks and Information Security (IJCNIS)*, vol. 11, no. 1, pp. 128–133, apr 2019. [Online]. Available: https://doi.org/10.17762/ijcnis.v11i1.3789

[14] R. Bhat, R. K. Thilak, and R. P. Vaibhav, “Hunting the Pertinency of Hash and Bloom Filter Combinations on GPU for Fast Pattern Matching,” *International Journal of Information Technology*, vol. 14, no. 5, pp. 2667–2679, may 2022. [Online]. Available: https://doi.org/10.1007/s41870-022-00964-3

[15] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin, “A Simple Network Management Protocol (SNMP),” RFC 1157 (Historic), may 1990. [Online]. Available: https://doi.org/10.17487/RFC1157

[16] U. Blumenthal and B. Wijnen, “User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3),” RFC 3414 (Internet Standard), dec 2002. [Online]. Available: https://doi.org/10.17487/RFC3414

[17] B. H. Bloom, “Space/Time Trade-Offs in Hash Coding with Allowable Errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, jul 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[18] I.-A. Ionel, “A Signal Theory Model for Security Monitoring using CheckMK,” in *Proceedings of the International Conference on Cybersecurity and Cybercrime (IC3)*. Cybercon, may 2023, vol. X, pp. 141–148. [Online]. Available: https://doi.org/10.19107/CYBERCON.2023.19