HoneyTree: Making Honeywords Sweeter

Kuntal Das, Jafar Haadi Jafarian, Ersin Dincelli, Ellen Gethner, Thomas Bekman
University of Colorado Denver
{kuntal.das, haadi.jafarian, ersin.dincelli, ellen.gethner, thomas.bekman}@ucdenver.edu

Abstract

Cyber deception is an area of cybersecurity based on building detection systems and verification models using decoys or controlled misinformation to confuse or misdirect the adversaries into revealing their presence and/or intentions. In the era of online services where our data is usually protected on the cloud relying on a secret key, even the most secure cyber systems can get compromised, losing highly confidential data to the attackers, including hashed passwords that can be cracked offline. Prior work has been done in carefully placing traps in the systems to detect intrusion activities. The Honeywords *project by Juels and Rivest is the most straightforward and successful technique in detecting and deterring offline-password brute-force by placing multiple plausible decoy passwords together along with the real password. In this paper, we enhance this approach and combine it with the concept of the Merkle tree to build a new model called HoneyTree. Our model achieves twice the level of security as the Honeywords project at the same storage cost. We perform a detailed comparison of our approach to the original Honeywords project and analyze its pros and cons.*

1. Introduction

Nowadays, most online account security is based on textual passwords, which need to be stored on third-party servers in some form to verify the actual user during authentication. However, since these servers can get compromised and the credentials can get leaked, passwords are not stored in their plaintext form. Instead, the system computes a hash of the password using a cryptographic hash function, such as MD5, SHA1, and SHA256, and stores the hash instead. Hash functions being lossy cannot be inverted, and so even if the database gets leaked, the plaintext passwords cannot be obtained back from the hashes. However, in reality, the attackers use an ingenious technique known the hash inversion to obtain the plaintexts from the hashes.

Precisely, this form of attack is not about inverting the hash. In this case, attackers use various techniques like high-performance cluster computing (HPCC), a smaller scale brute-force using multiple cores and threads running in parallel, or online lookup tables, such as rainbow tables, to find their plaintext counterparts or pre-images [1]. Since these kinds of attacks usually happen away from the source or offline, the site owners have no mechanism to protect their data against these attacks. Once a pre-image is discovered, the attackers try it online to get access to the victim's account.

In some cases, the result of the hash inversion does not match with the actual password, although they correspond to the same hash. This property of hash functions in which multiple plaintexts generate the same hash is called hash-collision [2]. As a result of the collision, the actual password sometimes does not get revealed. However, when it comes to compromising accounts, it makes no difference as the attacker can use any pre-image of the hash and gain access to the account as the authentication system verifies the hash of the password, and all the pre-images generate the same hash. The system cannot differentiate the real password from the other pre-images as the server does not store the actual password or any of its lossless forms. To counter this, cyber deception mechanisms are introduced into the authentication systems to add detectability [3].

In this paper, we discuss the use of decoy passwords in detecting offline brute-force. The idea is to introduce a bunch of decoy passwords in the database along with the real password and have a different server as the verification agent. If the attacker uses the decoys instead of the real password, they will be detected. The idea was originally used in the Honeywords project [4]. Based on this concept, we designed our approach, called HoneyTree, which serves as a structural improvement to the original work. Our contribution lies in applying the Honeywords concept in Merkle tree [5] and designing a new way of achieving about twice the security standard using the same number of honeywords as compared to the original Honeywords project.

URI: https://hdl.handle.net/10125/80259
978-0-9981331-5-7

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 offers an overview of our approach, defines the terminologies and the notations that are used throughout the paper. Section 4 discusses the methodology of HoneyTree, the associated components like the server models, their communication policies, their file systems, and the underlying assumptions in finer detail. In section 5, we define the algorithms we discuss in section 4 from adding to verifying users. Finally, in section 6, we analyze the various aspects of the scheme with regards to the known attacks and the execution overheads. We also perform a detailed comparison with the Honeywords project and discuss the pros and cons of our approach.

2. Related Work

Passwords serve as the front line of defense to prevent unauthorized access to users' sensitive and confidential information [6]. The brute-force attack is one of the most common methods for cracking passwords. To counter brute-force attacks, several mechanisms have been proposed that use decoys in adversary detection like the Honey Encryption [7] and D3CyT [8]. However, these methods are most suitable for encryption purposes that sometimes generate believable fake values upon decryption using the wrong key. Another method called the collisionful hash function [9] generates multiple collisions upon inversion of even the simplest of hashes, out of which only one is correct. However, it fails to deterministically generate meaningful pre-images, which is why the actual password can be distinguished. It is more suitable for systems based on nonsensical passwords managed by password managers. Since here we are dealing with meaningful user passwords, collisionful hash functions cannot be used.

The Honeywords project [4] by Juels and Rivest is one of the most-studied approaches that popularized the concept of the use of decoy passwords. In this system, the server stores multiple plausible passwords (honeywords) together with the real password (sugarword) of the user in the form of a list called the sweetwords. The index of the sugarword is stored in a separate, highly secure server called the Honeychecker. During authentication, the Honeychecker fetches the index of the requested user and passes the index to the main server, which then retrieves the sugarword from the list and compares it against the one entered by the user.

Suppose an attacker found a copy of the database and, using hash inversion, discovered all the plaintext forms of the sweetwords for a particular user. To access the account, they try to authenticate by guessing the sugarword. If the entered password is correct, the attacker gets authenticated. However, if they enter a honeyword instead, the system flags it as a successful detection. Since it is completely based on probabilities, the authors recommended using about 19 honeywords per user so that the chances of getting caught is $19/20$ or 95%, which is significantly higher than guessing the right one successfully, which is $1/20$ or just 5%.

However, in reality, the probabilities of all the sweetwords are not necessarily equally flat. This brings us to the problem of flatness which measures how equally likely each of the sweetwords appears [4, 10]. By using passive information or techniques like pattern recognition, some of the sweetwords can be eliminated, and the list can be narrowed down.

A lot of the works built on the foundation of the Honeywords project focus on qualitative advancement, which involves improving the mechanism of generating plausible passwords or achieving flatness. This is mostly done using either surveys [10] or using Deep Adversarial Neural Network-based models like PassGAN [11]. The number of honeywords can also be increased to achieve better security. However, this requires a lot of space, and it becomes inefficient while up-scaling. None of the relevant prior works focused on the quantitative side or worked on optimizing the scheme.

In the following section, we discuss a potential approach called the HoneyTree to cut the storage overheads of honeywords by half making it more efficient. Or conversely, one can achieve about double the effective security by spending the same storage cost. Since this is purely quantitative advancement, any honeyword generation technique based on improving the quality of the honeywords or achieving flatness [4, 10] can be coupled with this approach.

3. Methodology Overview

In our approach, we define an online service where n users have registered with their respective user-ids and passwords. The user-ids are defined by u_0, u_1, u_2, ...u_n, where u_i denotes the user-id of the i_{th} user. We represent p_i as the password of u_i. Upon sign-up of a user u_i with input password p_i, we first select a tree depth d, which is a global constant. The value of d is stored in a system file called F_{sys}.

Next, we generate n_i honeywords for the user based on the chosen d and hash them along with the real password. Suppose $h_{i,0}$ denotes the hash of real password, and $h_{i,1}, \ldots, h_{i,n_i}$ denote the hash of honeywords.

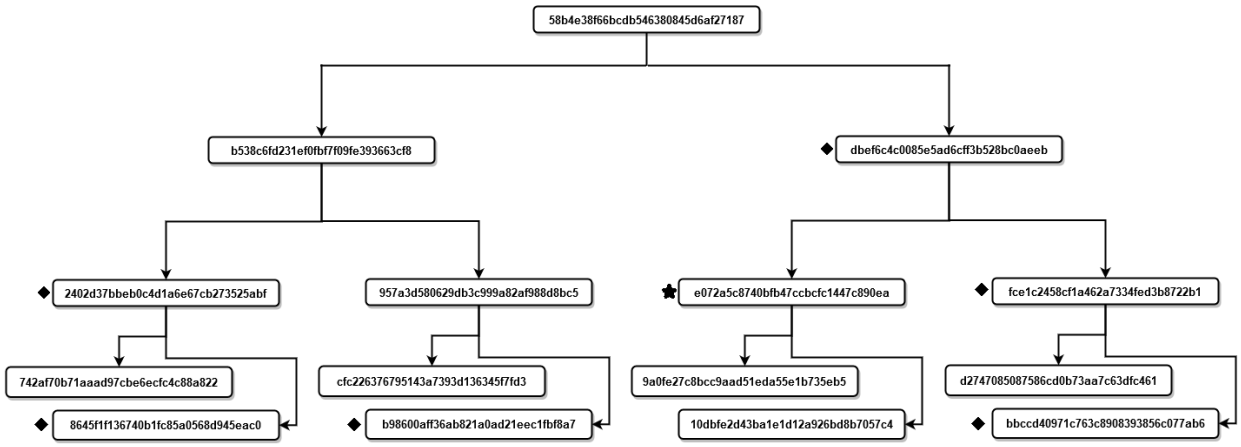

Figure 1. A complete binary tree formed by hierarchical SR-XOR

Given d, n_i, and $h_{i,j}$s, we generate a complete binary tree of depth d where $h_{i,j}$s are randomly distributed on nodes such that for every parent node and its two siblings, at most two of them are assigned. Then, the value for the remaining unassigned node is computed by applying a modified version of bitwise xor , called SR-XOR, between the two other assigned nodes.

The result of this modified xor is an arbitrary binary string with the same size as real hashes; however, given that the output size of a hash function has a fixed size, any arbitrary string should be the hash of an infinite number of inputs. We refer to these computed binary strings as *honeyhashes*. Unlike hashes of honeywords, the pre-images of honeyhashes are not known. Figure 1 shows a complete binary tree of depth 4. The honeywords are marked using a diamond symbol, while the real hash is marked with a star symbol. The rest of the nodes represent honeyhashes.

For a depth d, the tree has $k = 2^{(d-1)}$ leaf nodes which are either assigned with $h_{i,j}$s or a computed honeyhash. Suppose

$$L_i = \{l_{i,1}, l_{i,2}, ..., l_{i,k}\}$$

denote the values assigned to leaves of this tree. For each user, we only store values in L_i in the database.

For authentication of user u_i, we first retrieve L_i. Then starting from the leaves, we calculate the value of every parent by performing the SR-XOR of its two children in a bottom-up fashion until the root is computed. Then, the hash of the given password is compared against node values, and the index of the matching node is sent to the Honeychecker. If the index is verified, the user is successfully authenticated. If the hash matches that of a node in the tree but is not the sugarword, then an alarm is raised.

Given d, for user u_i, the maximum number of honeywords that could be assigned to the tree is

$$n_i = 2^{d-1} - 1$$

To be able to reconstruct the tree, we need to store $2^{d-1} = n_i + 1$ leaf values. This is exactly equal to the space required for the Honeywords approach.

Since the tree is full and complete, it includes $N_i = 2^d - 1$ nodes. Therefore, if an adversary discovers the database and correctly recovers L_i, they have to try to reverse $N_i \simeq 2 \cdot n_i$ hashes. Out of these N_i values, only one is the hash of the real password, almost half of them are honeyhashes, and the other half are hashes of honeywords. In theory, the adversary has a probability of $1/N_i$ to succeed. Therefore, HoneyTree doubles both the effort required for reversing the hashes, and the likelihood of detection while requiring the same extra space as compared to the Honeywords project.

However, in practice, HoneyTree complicates the reversing process because the attacker can not distinguish between hashes of real passwords and honeyhashes. Therefore, if they try to brute-force a hash, this confusion does not let them know when to stop, since they do not know if they are trying to break a honeyhash, which could be very hard to brute-force, also known as toughnuts [4], or the hash of a real but strong password.

Especially by selecting a strong password for the user (longer than ten characters) while choosing weak but believable passwords as honeywords (7-8 characters using dictionary words), the adversary could be given a higher likelihood of successfully - and potentially only - breaking and trying the honeywords.

4. Detailed Methodology

In this section we discuss all the components of the HoneyTree approach in details.

4.1. The Components

In our HoneyTree approach, we have the following components. There are two servers: The authentication server S_{auth} and the index server S_{ind}. The authentication server, S_{auth} receives the initial request for authentication, user registration, user deletion, password change, and all others types of requests related to accounting information management and modification. This server contains two different files F_{sys} and F_{usr} that can be accessed by all the algorithms running on S_{auth}, but not S_{ind}. The file F_{sys} contains the global constants, which are assumed to be public information. The constants include the depth of the tree d and a value v, known as the pivot. The F_{usr} file, as we already mentioned, contains a table of the users and their corresponding leaf nodes required to build the tree.

The index server S_{ind}, on the other hand, is a highly secure server that verifies whether the tree generated and sent by the S_{auth} corresponding to a particular user is legitimate or not during authentication. During user registration, it generates the tree based on the password sent by S_{auth} and sends the leaf nodes back to it. It contains two different files F_{pass} and F_{ind}. F_{pass} serves as the honeyword repository of HoneyTree. It contains hashes of real passwords obtained from secure online sources and is assumed to be frequently updated. The second file F_{ind} contains a table of the users, the userids, and their corresponding real index r. For a user u_i, r_i acts as the position of the hash of their real password in their tree.

4.2. Registration

During user registration, S_{auth} sends the new user's login credentials which are basically the userid u_i and the hashed password p_i, and the global constants, depth d, and pivot v, to S_{ind}, through a secure channel, for building the tree. Upon receiving the information, S_{ind} uses algorithm 1 and d to calculate the total number of nodes of the tree. Next, it uses algorithm 4 to calculate the nodes of the tree one by one in a top-down fashion. In the beginning, the algorithm selects $2^{d-1} - 1$ honeywords randomly from F_{pass}, inserts the real password in the list, and shuffles them together randomly to form a list of sweetwords s.

Next, at the root, it generates a random number between $[0, 1]$ to determine if the root node will be an entry from s or a similar length random number that will act as a honeyhash. The root can be any sweetword (the real hash or a honeyword) or a honeyhash. If the root happens to be a honeyhash, we assign a random number of the same length as that of the p_i to the root node and remove one honeyword from s. During all other node operations, we first pick a sweetword as the left-child, compute the honeyhash as the right-child using algorithm 2, and choose a random number between $[0, 1]$. Based on this number, we either keep the ordering or swap the children. We discuss the swapping process in section 4.5 in detail. Once the entire tree is built, the real hash p_i is located. We denote its index by r_i. Finally, the leaf nodes L_i are returned back to S_{ind}, and r_i is stored in F_{ind} along with the userid u_i. As this server is secure, all the intermediate processes and values stay confidential from the adversaries. Once the leaf nodes are received, S_{ind} stores them in file F_{usr} corresponding to the new user u_i, and the registration gets completed.

4.3. Authentication

During user authentication, S_{auth} sends the requested userid u_i, the leaf nodes L_i, the hash of the entered password p_i, and the constants d and v, to S_{ind} through an encrypted channel, for authentication. Upon receiving the information, the algorithm 5 first looks up in F_{usr} for checking the validity of the requested user. If the user does not exist, a "$FAILURE$" flag is returned; otherwise, algorithm 3 is invoked to generate the tree. This algorithm first fetches the real index r_i of the user from F_{ind}. Then it uses SR-XOR on the leaf node values in a bottom-up fashion using a queue to build the entire tree in a level-order format. Next, it checks if the received hash p_i is present at r_i index of the tree from the top. If it does, the algorithm returns $True$ otherwise $False$. Finally, S_{ind} returns either "$SUCCESS$" or "$FAILURE$" based on the output of authentication. S_{auth} on getting a "$SUCCESS$" response redirects the user to the profile page. On the contrary, upon receiving a "$FAILURE$" response, it flashes a failure message stating that either the userid or the password is incorrect. In algorithm 5 two timer functions can be noted. The first function $start\_timer(200)$ initiates a timer of 200ms in a separate thread and assigns the reference to a variable t_0. The timer runs in the background, and t_0 gets updated after every millisecond. The second function $timer\_wait(t_0, 0)$ pauses the current thread till t_0 becomes 0. That means no matter how much time has elapsed on the main thread; it will only be able to return an output after the 200 milliseconds mark from the start is reached. These functions essentially add noise to flatten the time requirements for all the

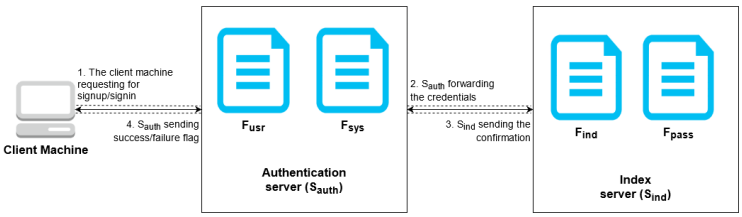

Figure 2. A schematic of the HoneyTree Topology for user authentication

possible outcomes to make them look equally likely. In other words, this noise prevents the attackers from launching a side-channel timing attack for passive information gathering. We discuss the details of these two functions and the attack in section 6.3. Figure 2 presents a schematic of the authentication process.

4.4. Index Server / Honeychecker (S_{ind})

The index server is considered to be a highly secure and hardened machine with strict security standards. All the communications with this server are through encrypted channels. Considering all these policies, we assume that this machine and its contents will never get compromised. The motivation has been drawn from the concept of Honeychecker from the Honeywords project [4]. It consists of a file F_{ind} that contains indices of the real password of all the users as shown in Table 1. For any given user u_i, these indices are values ranging from 0 to $2^d - 2$.

userid	index
john123xy	5
kent_94	2
cool_maurice2	12
beth@ny	17
max_007	25
.	.
.	.
.	.
katie_cute	9

Table 1. Structure of F_{ind}

4.5. SR-XOR

As mentioned in Section 3, we build a tree of hex strings where each nodal operation is performed using function 2. We call this operation SR-XOR (denoted by $\boxtimes$), which stands for Split-Reverse xor. It is used instead of regular xor to make the attack on our system exponentially harder or time-consuming as it makes the operation non-commutative. Here, we perform a simple transformation on the first operand before xor-ing it with the second one. The transformation has three parts; namely, split, reverse and merge. First, we split the string into two parts from a pivot index v. The element at the pivot index is included in the first half. Next, we reverse the two strings, and finally, we merge them back as described in Algorithm 5.2. Figure 3 shows the flow of the entire operation, performed on two example hex strings, with a pivot value of 6.

Properties of SR-XOR

Non-Commutative: The main reason we transform the xor operation is to achieve non-commutativity. In other words, unlike normal xor, for a given v, $a \boxtimes b$ is not same as $b \boxtimes a$ except for very rare cases. We choose SR-XOR to make brute-force attacks exponentially harder compared to ordinary xor. In Section 6.6, we discuss why we choose SR-XOR instead of ordinary xor. Also, for a given pair of strings, if we change v, the output changes as well. However, we consider the pivot to be public knowledge, and hence it does not actively contribute anything towards the deception of the system. So we arbitrarily choose a pivot value for the system at the beginning and keep it unchanged. We also like to mention that if both of the substrings of the first operand about the pivot are palindromes, the operation works the same as ordinary xor. However, since we are dealing with password hashes and some random numbers, the chances of having two palindromic substrings merged at the given pivot value are rare. We tested the same by generating a mixture of md5 hashes of $500,000$ different texts and 128 bit random hexadecimal numbers with different pivot values. None of them contain two palindromic substrings around any of the pivots.

Child Swapping: For SR-XOR, we follow the in-order traversal convention ($leftchild$, $parent$, $rightchild$). This means we perform $leftchild \boxtimes parent$ to calculate the $rightchild$. By this convention, the $leftchild$ will always be a honeyword, and the $rightchild$ will be honeyhash, defeating the purpose of the approach. To counter that, we swap the $leftchild$ and $rightchild$

with a 50% probability, such that:

$$rightchild \boxtimes parent = leftchild$$

First, we assign the honeyword to the $rightchild$. Next, we split-reverse the parent string (XOR with 0 causes no change) and finally apply SR-XOR on the $rightchild$ and the split-reversed $parent$, with the same pivot v, to calculate the left child as:

$$leftchild = rightchild \boxtimes (parent \boxtimes 0)$$

Generating Parent: To generate the parent node from the children, we calculate SR-XOR between the left and the right child as follows:

$$parent = leftchild \boxtimes rightchild$$

5. Algorithms

In this section we define all the algorithms used in this paper.

5.1. Add User

Algorithm 1 shows the pseudo-code of the user addition process.

Algorithm 1 add_user(u_i, p_i)

$user\_dict := file\_read\_to\_dict(F_{usr})$ #read the file as a dictionary
if u_i in $user\_dict.keys()$ **then**
  return "$FAILURE$"
end if
$d, v = file\_read(F_{sys})$ #get depth and pivot
$leaf\_list = gen\_tree(p_i, d, v)$
$user\_dict[u_i] = leaf\_list$
$Write\_File(F_{usr}, user\_dict)$ #function to write to file
return "$SUCCESS$"

5.2. SR-XOR

Algorithm 2 shows the pseudo-code of the SR-XOR algorithm. Initially, we split the first operand about a pivot element v, reverse each of the segments and then merge it back. Once the split-reverse transformation is complete, we xor it with the second operand to get the result.

Algorithm 2 sr_xor(x, y, v)

$end := |y|$
$y_1 := y[0 : v]$ #v is the pivot
$y_2 := y[v + 1 : end - 1]$
$y_3 := string\_reverse(y_1)||string\_reverse(y_2)$
$z := x \oplus y$
return z

5.3. Password Verification

Algorithm 3 shows the pseudo-code of the password verification process.

Algorithm 3 verify_password(u_i,p_i,L_i,d,v)

$tree = L_i.copy()$
$tree.reverse()$
$temp = L_i.copy()$
$lvl\_list = []$
$l = d - 1$ #level
$count = 0$
while $temp.length() > 1$ **do**
  $val = sr\_xor(temp[0]), temp[1], v)$
  $temp.append(val)$
  $temp.pop(0)$ #pop the left child
  $temp.pop(0)$ #pop the right child
  $count = count + 2$
  $lvl\_list.append(val)$
  if $count == 2^l$ **then**
    $lvl\_list.reverse()$
    $tree = tree + lvl\_list$ #merge list
    $lvl\_list.clear()$ #clear level list
    $l = l - 1$ #level decrease
    $count = 0$ #reset level counter
  end if
end while
$tree.reverse()$ #ascending order
$u_i, r_i = file\_get\_record(u_i)$
$r = tree.index(p_i)$
if $r == r_i$ **then**
  #index matched
  return $TRUE$
else
  return $FALSE$
end if

5.4. Tree Generation

Algorithm 4 shows the pseudo-code for tree generation.

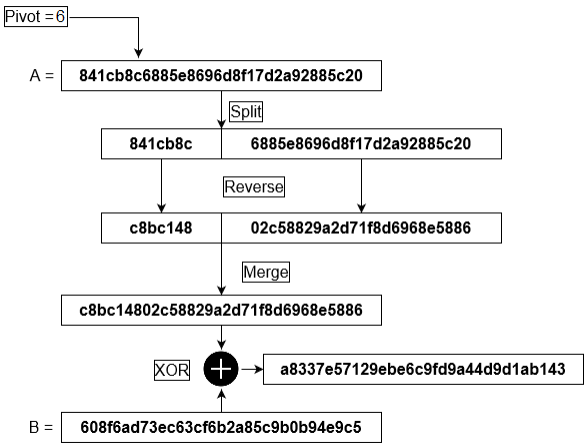

Figure 3. A schematic depiction of SR-XOR operation with two 128-bit hex strings

Algorithm 4 gen_tree(p_i, d, v)

```
tree_list = [] # create an empty list
honey_list = file_randomFetch(F_pass, 2^d)
r_0 = random(0, 1)
if r_0 == 1 then
    #Random Root
    root = randomHex(p_i.length)
    honey_list.pop(0)
    r_1 = random(0, honey_list.length)
    honey_list[r_1] = p_i
else
    #Sweetword Root
    r_1 = random(0, honey_list.length)
    honey_list[r1] = p_i
    root = honey_list[0]
    honey_list.pop(0)
end if
tree_list.append(root)
j = 0
while honey_list.length 0 do
    parent = tree_list[j]
    left = honey_list.pop(0)
    right = sr_xor(left, parent, v)
    r_2 = random(0, 1)
    if r_2 == 1 then
        right = left
        left = sr_xor(right, sr_xor(parent, 0, v), v)
    end if
    tree_list.append(left)
    tree_list.append(right
    j = j + 1
end while
leaf_nodes = tree_list[2^(d-1) - 1 : 2^d - 2]
return leaf_nodes
```

5.5. Authentication

Algorithm 5 shows the pseudo-code of the user-authentication process.

Algorithm 5 authenticate(u_i, p_i)

```
t_0 = start_timer(200)
#read the file as a dictionary
user_dict := file_read_to_dict(F_usr)
if u_i not in user_dict.keys() then
    timer_wait(t_0, 0)
    return "FAILURE"
end if
L_i = user_dict[u_i] #List of Leaf nodes
d, v = file_read(F_sys) #get depth and pivot
correct = verify_password(u_i, p_i, L_i, v, d)
if correct == True then
    timer_wait(t_0, 0)
    return "SUCCESS"
else
    timer_wait(t_0, 0)
    return "FAILURE"
end if
```

6. Analysis

In this section, we perform several analyses on our proposed HoneyTree approach.

6.1. Resilience against Hash Inversion attacks

In hash inversion brute-force attacks, we assume that the attacker has already performed an exfiltration on S_{auth} and dumped a copy of the F_{usr} and F_{sys} on their disk for offline analysis. Being offline, these

attacks are a lot more effective than the previously mentioned blind guessing. We assume that the attacker already knows the HoneyTree algorithm as well as the base hashing algorithm, which is MD5 here, as it is required to perform the hash inversion. Upon downloading the files, the attacker runs a script that generates the trees using the leaf node values from F_{usr}, corresponding to each user. Next, they perform a hash inversion on each of the $2^{d_{max}} - 1$ nodes of the tree using pre-computed hashtables. In our system, we used $d_{max} = 5$. Therefore, out of all the 31 nodes, 15 of them are sweetwords including the correct password, and the remaining 16 are honeyhashes, which are there only to waste their computational power and time and occasionally mislead them.

6.2. Attack on the Index Server

The index-server contains the information which, if obtained by any means, will be sufficient to compromise the HoneyTree system. Although this server is assumed to be secure, we need to have a fail-safe mechanism as well to make sure the system remains safe in case it fails or crashes due to encountering an unprecedented situation. In this case, if the index server is offline due to maintenance or a DDoS attack, the system falls back to a fail-safe mode, in which every password will be rejected. This helps in preventing the attackers from testing the outcomes of their hash inversion attack to discover the sugarword. In this way, our fail-safe mechanism ensures that even if the server fails, no user data or passive information about their account gets leaked.

6.3. Side-channel Timing Attack

Side-channelling refers to a form of passive information gathering using which the attacker narrows down their search space. The side-channel timing attack is based on calculating the average time for different login scenarios [12]. This type of attack is very effective against systems involving multiple servers like ours. Let us consider the first authentication scenario to be using a correct user id and a wrong password. The second scenario is based on logging in using an userid not known to the server. If the user is not present, the system returns a "$FAILURE$" flag. On the other hand, if it exists, then the password check is computed, and if it does not match, then a "$FAILURE$" flag is returned. Although the failure message corresponding to the failure flag says "incorrect userid or password" figuring the precise reason is not difficult. The attacker enters some gibberish string as an userid to get the timing for non-existent users and then calculates their average after some trials. Next, they check against some of the most common userids as per the online repositories of leaked credentials. Now, if there is an userid that is present on the server's records, the timing shoots up as the authentication server processes further if and only if the user id exists. The latencies include intra-server communication with the index server, reading from the disk, and some extra computation. This helps in guessing which userids are present on the server's records. We ran several simulations using Python 3. User authentication, on average, took about 135ms, when the userid exists in F_{usr}. The maximum delay reached up to 186ms. However, when the user did not exist in F_{usr}, the timing was 42ms on an average with the maximum touching about 73ms.

To counter this, we add some noise [13] using two functions, $start\_timer(initial\_mark)$ and $timer\_wait(timer\_object, fin\_mark)$. At the beginning of the authentication routine, the $start\_timer()$ function is invoked in a separate thread T_{child} and an timer object t_0 is returned. The function works like a timer, initialized with an initial value. We used 200ms in our implementation. We chose it to be slightly greater than 186ms, which is the maximum delay we encountered during our simulation. To determine the best value for the timer, one needs to run several tests in their setup and pick a number slightly greater than the maximum delay they found. Once invoked, the function keeps on decreasing the counter every millisecond by 1. While this is running in the background, the main operations run on the main thread T_{main}. The second timer function, $timer\_wait()$ running on T_{main}, waits for t_0 to reach the desired $final\_mark$ which is 0 in our case. In this way, the $timer\_wait()$ function pauses the authentication function and prevents it from returning any output flag, until t_0 reaches the $final\_mark$. This whole process ensures that the duration of authentication at the server's end will always be the difference of the $initial\_mark$ and the $final\_mark$ irrespective of the outcome.

6.4. Time and Space Complexity

In this section, we evaluate the time and space requirements of different components and operations of HoneyTree. We assume our hash function has an output hash of x bits. We also assume that the maximum length of password allowed is comparable to the length of the hash and that computing a hash requires a constant number of passes on the string; therefore its complexity is $\mathcal{O}(x)$. For MD5, x is 128.

SR-XOR: The number of characters in hexadecimal

will be $x/4$. The time required to split and reverse the first string takes $\theta(x/4)$. Next, the xor operation of the two strings takes $\theta(x)$. Therefore, the total time required to compute the SR-XOR of two hex strings is of the order $\mathcal{O}(x)$. In terms of space, computing SR-XOR takes up at most the length of the hash, which is x bits. Therefore its space complexity is in $\mathcal{O}(x)$.

Now, further assume we have n leaf nodes per user, which means $n = 2^{d-1}$, where d is the depth of the tree. Also, for a tree of depth d, we have at most $n + 1$ sweetwords. We evaluate the time and space requirements for both Tree Generation and Password Verification.

Tree Generation: For each sweetword other than the root node, we have one SR-XOR operation. Therefore for $n + 1$ sweetwords we have a total of $\mathcal{O}(x) * (n + 1)$ operations. Hence the time complexity is of the order $\mathcal{O}(nx)$.

In terms of space, the maximum list size it uses is equal to the size of the tree, which has $2n - 1$ nodes, each of which is of x bits. This makes the total space requirements of the order $\mathcal{O}(x) * (2n - 1)$ or simply $\mathcal{O}(nx)$.

Password Verification: For each node pair, we have one SR-XOR operation. Therefore we have at most n such operations, as the last node or the root node is a single hash. As mentioned before, each SR-XOR takes up $\mathcal{O}(x)$ operations. So we have a total of $\mathcal{O}(x) * (n)$ operations. Hence the time complexity is of the order $\mathcal{O}(nx)$.

In terms of space, like tree generation, the maximum list size required is equal to the size of the tree, which has $2n-1$ nodes, each of which is of x bits. This makes the total space requirements of the order $\mathcal{O}(nx)$.

6.5. Comparison with Honeywords

Since HoneyTree is based on the Honeywords project, we also compare the pros and cons in this section. In the Honeywords project, during password verification, the actual index is sent from the Honeychecker to the database server for fetching the real password before comparison. This design choice has a flaw. When the attackers get into the system and dump the database, it is likely to assume that they have some control over the database server. Therefore the sugarword, once fetched, will be present in the memory, which might be accessible to them as well. In HoneyTree, we send the password to the index server along with the stored leaf nodes, and it only returns "$SUCCESS$" or "$FAILURE$" flags based on the results of the comparison. This prevents the sugarword from getting revealed at the database server even if it gets compromised.

The actual Honeywords project placed honeywords in the user database along with the real password. It uses the userid to fetch the secret index and retrieve the password, which was then sent for comparison. Comparing two passwords of x bits require $\mathcal{O}(x)$ comparisons which are significantly better than HoneyTree, which takes n times more computation where n is the number of leaf nodes which is about half of the number of honeywords stored in the Honeywords project.

This brings us to the space comparison. HoneyTree takes up about half of the space required by the Honeywords project as it generates the other half during authentication. The Honeywords project also introduced the concept of toughnuts or hard-to-crack hashes, which they insert into the sweetword set randomly. In our approach, we use the honeyhashes for the same purpose, but it is more deterministic, which could be worse if the sweetwords are easy to crack; otherwise, it helps by slowing down the brute-force significantly more than the toughnuts in the Honeywords project. Finally, the introduction of the time variables helps to prevent side-channel timing attacks, which were not considered in the Honeywords Project.

6.6. Advantages of SR-XOR over XOR

The strength of the HoneyTree approach comes from the Merkle tree structure of combining hashes. For a depth d, we have 2^{d-1} leaf nodes and $2^d - 1$ nodes in the entire tree. So we have a total of 2^{2^d-1} possible trees. Out of all these possibilities, only one tree is correct and only one way of combining the leaf nodes to generate that. The nature of the actual xor ($\oplus$) operation is commutative. Therefore, for any two numbers (A, B), $A \oplus B = B \oplus A$. Now suppose we use xor instead of SR-XOR. Then, we can combine every pair of siblings in two different ways instead of a particular order. The total number of sibling pairs in a tree is given by $2^{d-1} - 1$. Therefore, the total number of possible ways of solving a tree is given by $2^{(2^{d-1})-1}$. Since the security of our system relies on combining the correct values in the correct order, the commutative property of xor reduces the complexity of the problem of combining the correct pairs. For $d = 5$, this value becomes 2^{15}. So instead of one correct way of combining the sibling pairs, we have 32768 ways. To avoid this, we designed SR-XOR, which is non-commutative by nature (except in very rare cases as discussed in Section 4.5). So at every

node calculation, one needs to know the exact order to build the correct tree. In other words, SR-XOR makes brute-force exponentially harder than ordinary xor.

6.7. Limitations of HoneyTree

Although our HoneyTree approach opens up a wide range of possibilities surrounding the actual concept of Honeywords, it has its own limitations. As we have seen in Section 6.5, it reduces space requirements by adding extra computations, which negatively impact the time needed for authentication. Moreover, it uses a lot more and a predictable number of honeyhashes which could be easier to detect if the sweetwords are not strong. Furthermore, like the Honeywords project, the system relies on the unbreakable security of the index server, which is unrealistic to assume in real life. Also, there is a design-based limitation in the Honeywords project when it comes to applying cyber deception, and HoneyTree is no exception. The problem comes from the fact that multiple words are packed together, out of which only one is correct. This immediately raises suspicion due to the bodyguard of lies surrounding the real password. The attacker may decide not to step on the trap. While this causes deterrence and the attack can be avoided, it also reduces the chance of studying the attackers, their exploits, and the flaws in the server's security which they used in the first place to download the password file.

7. Conclusion and Future Work

In this study, we reviewed a few cyber deception schemes and how they can be used to detect the presence of passive adversaries. We designed HoneyTree, a very efficient technique of storing honeywords and verifying user credentials that can be deployed in any honeyword-based authentication system irrespective of their underlying honeyword generation schemes. Next, we conducted a neck-to-neck comparison with the original Honeywords project with respect to various security and performance metrics and studied its pros and cons. We concluded that with the same space requirements, one could achieve twice the level of security offered by the original Honeywords project, although it comes with an additional computation overhead. As a part of future work, we plan to extend our current model by adding more features like variable tree depth and designing a new variant of xor that uses the pivot for another level of deception. We also plan to partially combine multiple HoneyTrees from different users to build a HoneyForest making it several times harder to crack with the same space overhead.

References

[1] K. Kim, "Distributed password cracking on gpu nodes," in *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pp. 647–650, IEEE, 2012.

[2] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for hash functions md4, md5, haval-128 and ripemd.," *IACR Cryptol. ePrint Arch.*, vol. 2004, p. 199, 2004.

[3] P. Aggarwal, C. Gonzalez, and V. Dutt, "Cyber-security: role of deception in cyber-attack detection," in *Advances in human factors in cybersecurity*, pp. 85–96, Springer, 2016.

[4] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 145–160, 2013.

[5] E. Mykletun, M. Narasimha, and G. Tsudik, "Providing authentication and integrity in outsourced databases using merkle hash trees," *UCI-SCONCE Technical Report*, 2003.

[6] E. Dincelli and I. Chengalur-Smith, "Choose your own training adventure: Designing a gamified seta artefact for improving information security and privacy through interactive storytelling," *European Journal of Information Systems*, vol. 29, no. 6, pp. 669–687, 2020.

[7] A. Juels and T. Ristenpart, "Honey encryption: Security beyond the brute-force bound," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 293–310, Springer, 2014.

[8] K. Das, E. Gethner, E. Dincelli, and J. H. Jafarian, "D3cyt: Deceptive camouflaging for cyber threat detection and deterrence," in *Advances in Information and Communication*, pp. 756–771, Springer International Publishing, 2021.

[9] L. Gong, "Collisionful keyed hash functions with selectable collisions," *Information Processing Letters*, vol. 55, no. 3, pp. 167–170, 1995.

[10] N. Chakraborty, S. Singh, and S. Mondal, "On designing a questionnaire based honeyword generation approach for achieving flatness," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 444–455, IEEE, 2018.

[11] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "Passgan: A deep learning approach for password guessing," in *International Conference on Applied Cryptography and Network Security*, pp. 217–237, Springer, 2019.

[12] M. Čagalj, T. Perković, and M. Bugarić, "Timing attacks on cognitive authentication schemes," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 3, pp. 584–596, 2014.

[13] N. Chakraborty, G. S. Randhawa, K. Das, and S. Mondal, "Mobsecure: A shoulder surfing safe login approach implemented on mobile device," *Procedia Computer Science*, vol. 93, pp. 854–861, 2016.