A Critical Security Analysis of the Password-based Authentication Honeywords System Under Code-Corruption Attack

Ziya Alper Genç, Gabriele Lenzini, Peter Y.A. Ryan and Itzel Vazquez Sandoval

Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg, Luxembourg
{ziya.genc, gabriele.lenzini, peter.ryan, itzel.vazquezsandoval}@uni.lu

Abstract. Password-based authentication is a widespread method to access into systems, thus password files are a valuable resource often target of attacks. To detect when a password file has been stolen, Juels and Rivest introduced the Honeywords System in 2013. The core idea is to store the password with a list of decoy words that are "indistinguishable" from the password, called honeywords. An adversary that obtains the password file and, by dictionary attack, retrieves the honeywords can only guess the password when attempting to log in: but any incorrect guess will set off an alarm, warning that file has been compromised. In a recent conference paper, we studied the security of the Honeywords System in a scenario where the intruder also manages to corrupt the server's code (with certain limiting assumptions); we proposed an authentication protocol and proved it secure despite the corruption. In this extended journal version, we detail the analysis and we extend it, under the same attacker model, to the other two protocols of the original Honeywords System, the *setup* and *change of password*. We formally verify the security of both of them; further, we discuss that our design suggests a completely new approach that diverges from the original idea of the Honeywords System but indicates an alternative way to authenticate users which is robust to server's code-corruption.

Keywords: Honeywords · Password-based Authentication · Secure Protocols Design · Formal Analysis · ProVerif

1 Introduction

Password-based authentication is a simple and widespread way to validate user identity [6]: it requires users to have a public login and a secret password. It is not the most secure though. For that, passwords must remain secret, users must chose them hard-to-guess, not to share them, and transmit them only over encrypted channels. Servers, in turn, should not store passwords in cleartext but keep them hashed (usually with some "salt") in a file called the *password file*.

Such valuable files are naturally the target of hackers, who try to steal them from servers for then retrieve passwords by off-line dictionary attacks. A taste of the extension of the problem can be read from the news. In 2016, Yahoo! was reported to have had, in 2014, 500 million user accounts hacked, a number that was corrected, later, to be 1 billion accounts [8], and further to be 3 billion [10]. MySpace, Tumblr, and LinkedIn were also be reported to have had millions of login credentials stolen (64 million Tumblr accounts and more than 360 million MySpace accounts [1]). The theft would have passed unnoticed if it was not for someone who tried to sell the credentials in the black market.

These examples, in addition to the high number of passwords lost, surprise because of the time that has passed between the attacks and their detection. Such a delay is a problem as serious as the reason that led to the leak because puts off the application of countermeasures that could limit the damage.

To improve the awareness of passwords theft, computer security research has proposed solutions. For instance, Google monitors suspicious activities and invites users to review from what device and from which location they have accessed their account. But of course, it is more critical and valuable to ensure that a service becomes aware of the theft of a password file because, in such situation, a great deal of passwords is exposed at once. This problem is the starting point of some recent research.

2 Juels and Rivest's Honeywords System

Aiming to make password-cracking detectable, in 2013, Juels and Rivest proposed to modify the classical password-based authentication scheme with one called *Honeywords System* [9].

A Honeywords System hides and stores a user (hashed) password in a list of decoy words, called *honeywords*. Honeywords are chosen to be indistinguishable from the password, for instance "redsun3" is a good honeyword for "whitemoon5", a property which is called *flatness* [9, 5]. Honeywords should also be chosen in such a way that is unlikely that a user types a honeyword purely by mistake. From those properties from any attempt to log in with a honeyword instead of the password one can soundly concludes that the password file must have been leaked. So does the Honeywords System, which flags the event and initiates some contingency procedure (e.g., system administrators are alerted, monitors are activated, user's execution rights are reduced, user's actions are run in a sandbox, and so on).

The Honeywords System's architecture is logically organized in two modules: (1) a "computer system" which, according to Juels and Rivest, is "any system that allows a user to 'log in' after she has provided a username and a password" (*ibid*) and which we call the *Login Server (LS)*; (2) an auxiliary *hardened secure* server that assists with the use of honeywords, which Juels and Rivest call the *Honeychecker (HC)*. For each registered user u, the LS keeps (in the password file) the ordered list of u's *sweetwords* (so are called collectively honeywords and password), denoted here by $[h(w_x)]_u$, with $x \in [1,k]$ where k is the fixed number of sweetwords. The HC stores c_u, the index of u's password in such list.

The system's behavior comprises three phases: (1) *setup*, (2) *authentication* (i.e., the login), and (3) *change of password*. Authentication is the most critical phase, so we describe it first. We leave the setup and modification for later sections. At authentication, the LS receives username and password (u,w) from the user; then, it searches the hashed version of w in the list $[h(w_1),\ldots,h(w_k)]_u$ of (hashed) sweetwords of u. If no match is found, login is denied. Otherwise the LS sends to the HC the message (u, j), where j is the found position. This communication occurs over dedicated and/or encrypted and authenticated channels. The HC checks whether $j = c_u$. In the case that the test succeeds, access is granted. In case the test fails, it is up to the HC to decide what to do. Juels and Rivest say: "Depending on the policy chosen, the honeychecker may or may not reply to the computer system when a login is attempted. When it detects that something is amiss with the login attempt, it could signal to the computer system that login should be denied. On the other hand it may merely signal a 'silent alarm' to an administrator, and let the login on the computer system proceed. In the later case, we could perhaps call the honeychecker a 'login monitor' rather than a 'honeychecker'."(*ibid*). Figure 1 illustrates the authentication protocol considering a responsive honeychecker.

The Honeywords System's goal is not to impede the stealing of a password file: an intruder who has retrieved by an off-line dictionary attack the sweetwords can still succeed in guessing the correct password but, assuming that the adversary has no other clue than the sweetwords, he has probability $(k-1)/k$ to fail and reveal the leak.

Juels and Rivest have left open several problems. One of them reads as follows "How can a Honeywords System best be designed to withstand active attacks e.g., malicious messages issued by a compromised computer system or code modification of the computer system?' (*ibid*).

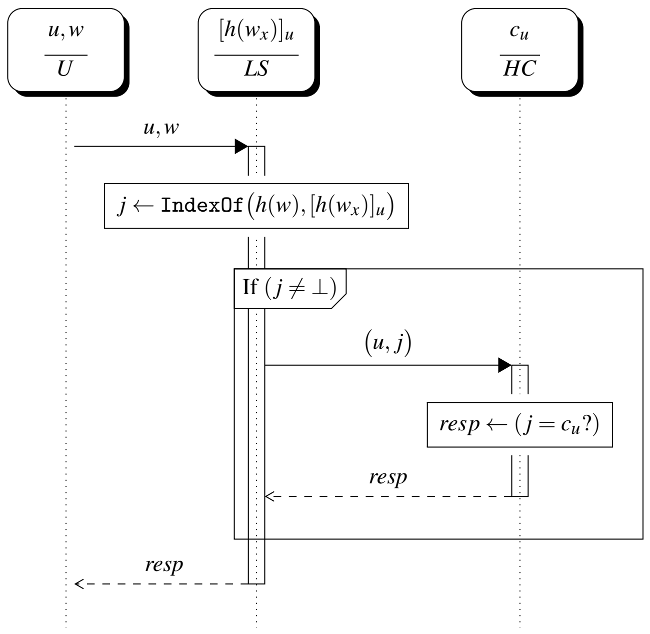

u, w
U
$[h(w_x)]_u$
LS
c_u
HC
u, w
$j \leftarrow \texttt{IndexOf}\big(h(w), [h(w_x)]_u\big)$
If $(j \neq \perp)$
(u, j)
$resp \leftarrow (j = c_u?)$
$resp$
$resp$

Fig. 1. Honeywords System Authentication Protocol. (Taken from [7])

In a conference version of this work [7], we took on the task to discuss the part of the problem regarding "*to withstand [..] code modification of the computer system*".

Generally speaking, awaiting to define in detail what "code modification" means (see next section), the corruption of the Honeywords System raises interesting questions. Juels and Rivest discuss in [9] that: "compromising only the honeychecker at worst reduces security to the level it was before the introduction of honeywords and the honeychecker". The situation worsens if the honeychecker *and* the module that we called the LS were both corrupted: we are inclined to believe that if the whole Honeywords System is compromised there is little to do to avoid that the intruder can get access unnoticed (although only a rigorous analysis, out of scope here but suitable for future work, can provide evidence to this claim). The last interesting case is to analyse the security of the Honeywords System against an adversary that has succeeded in "modifying the code of LS". In [7] we propose a solution, a new protocol that here we describe and analyze in full detail. We also extend the analysis to the two remaining protocols, the setup and the change of a password, which we also redesign and prove it be secure against a code modified LS.

3 Paper Outline

We first discuss the notion of "code modification". It was left informal in Juels and Rivest's work but so stated it does not help understand the real nature of the threat. We need to define it rigorously. So, in §4 we give our definition of "code modification" or, as we will call from now on, "*code corruption*": we prefer this term since it stresses the maliciousness of the act. We also state a few foundational assumptions before thoroughly giving the analysis of security of the original Honeywords System under the threat.

In §5 we prove the Honeywords System insecure, illustrating an attack that works when the LS's code has been corrupted according to our model and under our assumptions. The attack reveals that when confronted against the threat, the original Honeywords System has a core weakness. From studying the root cause of the attack we elicit a security requirement and by fulfilling it we are able to provide a solution to the problem.

In §7 we describe a new cryptographic protocol for authentication which we argue that removes the weakness and hence restores security. We sustain this statement formally in §7.2 by modeling in ProVerif the protocol together with the code-corrupting adversary and running an automatic analysis. The results of the verification confirm that the previous attack is no more possible. Actually, we prove that there are no more attacks against the new protocol, in the given model.

In §9 we discuss the Honeywords System's setup phase and extend our new protocol to cover this stage. The corresponding formal analysis addressing issues detected in a code corrupted Honeywords System follows in §9.1. Likewise, §10 and §10.1 are devoted to the change of password phase.

Our solution is meant to be primarily of theoretical interest, but because its cryptographic primitives rely on a generous use of exponentiation, we thought useful to implement the authentication protocol and benchmark its performance with respect to the original Honeywords System's authentication. The results are reported in §11: they show that although slower than the original Honeywords System, the loss in performance is linear in k, the number of sweetwords. Roughly speaking, our scheme can handle a few hundred authentication requests per second on a laptop with the service running on a virtual machine. It is reasonable to expect better results on more efficiently performing servers.

At the end, in §12, we discuss our solution in a wider perspective. We look at it from distance to conclude that, although it solves the open problem and works against the code-corruption threat that we have defined, it actually suggests a completely innovative design for password-based authentication that is far stronger to be used only as a fix for Honeywords System.

4 Code Corruption and Threat Model

What is a reasonable goal for an adversary that intends to code corrupt LS? What is *code corruption*? What levels of corruption are interesting to study? We have to answer all these questions to understand the threat.

We premise that if the meaning of code corruption should be taken literally, it suggested an ability to change the code at will. Stated in this way, it seems to be a very disruptive ability and not all its instances are interesting, in the sense that they do not bring to insights that help up understand the fundamental weaknesses of the system design. What understanding do we gain from a code corruption that, for instance, causes a shutdown of the entire system or that let anyone log in? It is necessary to establish specific *assumptions* to limit the extension of the threat.

We start with an obvious assumption, one that follows from the original Juels and Rivest's paper:

Assumption 1 *The adversary, before corrupting the LS, knows the sweetwords but not the passwords.*

Assumption 1 says that the adversary has stolen the password file and has retrieved all the k sweetwords of, say, user u: yet, s(he) does not know which one among the u's sweetwords is the password.

Let us call the fact of logging in without the HC's raising an alarm a “successful log-in”. Assumption 1 states that the probability of a successful log-in for the intruder occurs when s(he) naïvely picks at random a sweetword. We exclude that the adversary have access to other sources to increase such probability, for instance, possessing social information about a specific user (e.g., relatives' birthdays, pets' names, etc.) which could be reflected in the password choice. Oppositely, for simplicity, we also exclude that the adversary failed to retrieve some sweetwords from the hashed values: all the sweetwords have been recovered.

“What is a reasonable goal for an adversary that intends to code corrupt the system?”. We believe that a reasonable goal is to increase the intruder's probability of a successful log-in to a value higher than the one that (s)he would have by guessing the password and with an honest LS.

Definition 1. *The goal of a code corruption attack is to increase the adversary's probability to successfully log in with respect to the probability of guessing the password among the sweetwords retrieved from the passwords file.*

We answer the second question "What is code corruption?" together with "What levels of corruption are reasonable to consider"?

Definition 2. *Let* ls.exe *be the code of the LS's protocols. Code corruption of LS means changing* ls.exe.

With its code corrupted, LS can change completely its behavior. An intruder can reprogram it to do whatever, e.g., to play chess.[1] However, we are not interested in attacks that change the functionality of the LS, for the reason that they do not help the adversary to increase its probability of successfully logging in. For a similar reason, we are not interested in attacks that shut-down the systems or cause Denial-of-Service. These are important attacks from which to seek defense, but out-of-scope in this study.

We also exclude attacks such as those consisting in changing ls.exe to always grant access (but of course if would make sense to change ls.exe to grant access selectively e.g., to the adversary Mallory, if that were possible, see later).

But in excluding an access-for-all corruption we have also a technical reason. The original paper does not give full detail of the architecture of the "computer system", our LS, but it seems reasonable to assume that Honeywords System implements a *separation of duties* [3]. And if so the duty of LS is only to search the proffered password in the password file, to inform the HC, and possibly to report the decision to the user, but not to grant or deny accesses. So there is no simple to way to open access to anyone, unless LS can foul the HC and this is actually the core of the threat (see next section).

Assumption 2 *A code corruption against LS does not change the LS's observable behaviour.*

The rationale of this assumption is that, if the adversary changes the observable behaviour of LS, this would result in an anomaly that can be detected, triggering an alert in response to which a safe version of the `ls.exe` can be restored. Since the adversary may have a once-in-lifetime opportunity to corrupt LS's code, he may not want to see his efforts vanishing in this way. Of course not all attackers will be so concerned about being undetected. They can be satisfied by managing to log in and say ex-filtrate sensitive data might be fine, even if this leaves a trail. But we decided to scope our analysis only within the context of Assumption 2.

However, even under Assumption 2 there are subtleties that need to be addressed. Interpreted strictly it does not allow the creation of any back door between the adversary and the LS that this last can use at anytime to leak information. This is because, interpreting strictly the term "undetectability", an exchange of messages from the LS towards the adversary and outside the protocol's message flow can be eventually detected (e.g., by monitoring the net traffic), leading to have a safe version of the `ls.exe` re-installed.

Thus according to this interpretation, Assumption 2 says that if the intruder wants to communicate with the corrupted LS, it must use the same channels from which legitimate users log in, and must respect the message flow of the honest protocol. This does not exclude that, when re-coding `ls.exe`, the adversary can use the knowledge he has gained from having hacked the password file and hard-code in the corrupted `ls.exe` a user's IDs, the sweetwords, or other information s(he) may have.

Still, if we take Assumption 2 less strictly, it admits that some information can flow back to the adversary, for example, in message *resp*. And, as we will discuss in detail in §5, letting LS to communicate back to the adversary leads to a powerful attack that breaks the original Honeywords System. In short, the attack works because LS can learn *u*'s password (or the hash of it). This is a feature more than a vulnerability but a feature

[1] This is what R. Gonggrijp did when, in 2006, proved insecure a Dutch electronic voting machine.

that a collusive adversary able to invert the hash can exploit to know the password. So, an incentive for code corrupting the LS is exactly to create this retroactive communication and we cannot exclude this possibility in our analysis.

We propose thus the following methodology: by default we interpret Assumption 2 strictly but, always, we discuss what happens if we relax this constraint and let LS leak information to the intruder.

Notably, the new protocols that we describe in §7-§10, although designed to secure the Honeywords System under an Assumption 2 interpreted strictly turn out to be efficient also when we relax it. The new protocols will not impede the leak nor stop the adversary from learning u's password, but will make that information useless for the adversary. Somehow the ideas behind our protocols reduces considerably the role of the password as the only authentication token.

5 Attacks against the Authentication Protocol

As future reference, we write down how ls.exe looks like for the authentication protocol. Algorithm 1 shows it in pseudo-code, using a notation whose commands are self-explanatory. Here, passwd is the password file, passwd_u is the row of user u, and H is a hash function (e.g., SHA-3 [11]). We also assume that u is a legitimate user's name. The algorithms presented here were introduced first in [7].

Algorithm 1 Login Server Authentication

```
1: procedure ls.exe(passwd)
2:     while true do;
3:         ReceiveFrom(U; $(u,w)$);
4:         j ← IndexOf($H(w)$, $\text{passwd}_u$);
5:         SendTo(HC; $(u,j)$);
6:         ReceiveFrom(HC; $resp$);
7:         SendTo(U; $resp$);
```

Algorithm 2 Code Corrupted LS

```
1: procedure ls'.exe(passwd)
2:     $(u',w') \leftarrow (\perp,\perp)$                         ▷ init good $(u,w)$
3:     while true do;
4:         ReceiveFrom(U; $(u,w)$);
5:         if $(u' \neq \perp) \wedge (u = \texttt{Mallory})$ then
6:             $(u,w) \leftarrow (u',w')$
7:         j ← IndexOf($H(w)$, $\text{passwd}_u$);
8:         SendTo(HC; $(u,j)$);
9:         ReceiveFrom(HC; $resp$);
10:        if $(resp = \texttt{granted})$ then
11:            $(u',w') \leftarrow (u,w)$                         ▷ good $(u,w)$
12:        SendTo(U; $resp$);
```

If the adversary can corrupt ls.exe, even under our Assumption 2 taken strictly, there is an obvious attack. The corrupted ls′.exe is reported in Algorithm 2. When LS notices a good user's password, it stores the valid pair of credentials (user, password) and then reuses that knowledge to let the adversary gain access, when (s)he reveals her/himself at the log-in with a specific user name (e.g., "Mallory").

Algorithm 2 represents an ideal attack. Actually, LS could just remember the valid index j (in step 11) and, in a next round, skip searching the passwd_u (step 7) and send that j to the HC (step 8). But the corrupted ls.exe outlined mimics the behaviour of LS more faithfully and shows also that *LS gets knowledge of a user's valid password*. This, we will see, is the root of a serious vulnerability.

Note that not always, in instruction 10, the LS learns u's password with certainty. This may happen, for instance, when the HC follows a contingency policy that dictate to respond by granting access even when it receives a sweetword, as suggested in the original work (see also our quote about it in §2). However, the following strategy gives the LS at least a good chance to guess the password, especially when the strategy is coordinated with the adversary: since the adversary can submit honeywords on purpose, it refrains itself from trying to access for a certain time. During this interval, the only requests that arrive to the LS pretending to be from user u are actually from the legitimate u; all the w that come with the requests then must be the u's legitimate password. Surely, the user can sometimes misspell the password, but that will never collide with a honeyword (because honeywords are flat, see §2). It is therefore possible for the LS, purely by statistical analysis and by cross comparison between what u submits, to infer the u's real password and at that moment the LS can so help the adversary as we illustrated in our ideal version of the attack. The adversary has raised its probability to gain a successful access.

This attack is already serious but under a relaxed Assumption 2, LS can further send the password back to the adversary, who now can use the u's credentials at any time.

Discussion. The root cause of the attack seems therefore to lie in the fact that LS gets to know u's password. Only hashing the password will not help, since the LS can search for the position of such hash value in u's row in the password file or, under a relaxed Assumption 2, send the hash back to the adversary who can reverse it. The main problems seem then rooted into three concomitant facts: (a) LS knows username and password in clear (even if it receives them over a secure channel); (b) LS can query HC as an oracle to know whether the submitted password is the user u's valid password (in this way it also gets to know the hash of the password); (c) LS can retrieve the index of the password in passwd_u (and with that he can foul the HC to grant access).

So, if a solution exists that makes the system secure despite a corrupted ls.exe then it would be such that it impedes LS to perform all these three actions (a)-(c) together. We state this finding as a requirement:

Requirement 1 *An authentication protocol resilient to code corruption should not (1) let the LS receive sweetwords in clear; (2) let it know when a sweetword is a valid password; (3) allow it to reuse that knowledge to retrieve a valid index at any moment that is not when the legitimate user logs in.*

6 Towards a Solution

In searching for a solution we are not interested in pragmatic fixes such as checking regularly the integrity of `ls.exe` and reinstalling a safe copy. Our lack of interest is not because solutions like that are not fully effective (e.g., the intruder can still execute its attack before any integrity check is performed) but because such pragmatic fixes do not give any insights about intrinsic weakness. The same argument holds in relation to best practices like forcing users to change their password frequently.

If a solution exists then it must be searched in a strategy that satisfies our requirement's items (1)-(3).

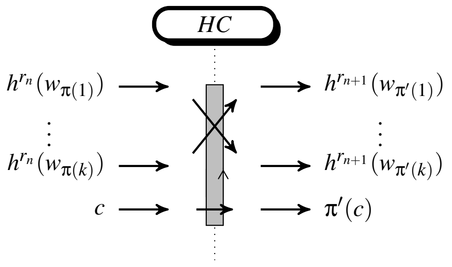


Fig. 2. Shuffling/re-hashing w's and updating c (Taken from [7]).

One way to comply with them is by implementing the following countermeasures: (i) passwd_u is *shuffled each time* LS queries HC: this avoids that LS can reuse an index j that it has learned to be the index of u's password; (ii) passwd_u is *re-hashed each time* LS queries HC: this avoids that LS can search again for the index of a typed password that it got to know being a valid u's password; (iii) let the *LS know what to search in* passwd_u *only when user u is logging in*: this precaution is to avoid that LS can perform off-line searches on passwd_u.

The countermeasures (i)-(ii), and so requirements (1)-(2), can be implemented *leaving HC in charge of shuffling and re-hashing the password file each time that a user logs in and that the LS questions the HC about index j.*

The shuffling does not require particular explanation. It must be randomized but is a standard step: given a row $[w_1,\ldots,w_k]$, and a permutation π, it returns $[w_{\pi(1)},\ldots,w_{\pi(k)}]$.

The re-hashing, instead, needs to be explained. It is implemented by *cryptographic exponentiation*. For each user, HC possesses g, a generator of a multiplicative subgroup $\mathbb{G}$ of order q (so, actually, g should be written g_u, but to lighten the notation we omit the index u). When first the list of sweetwords is generated, the file is initially hashed using g^{r_0}, where $r_0 \in \{1,\cdots,q-1\}$ is a random number. The u's row of the file is therefore $[g^{r_0\cdot w_1},\ldots,g^{r_0\cdot w_k}]$, which we write $[h^{r_0(w_1)},\ldots,h^{r_0(w_k)}]$ to stress that this is a hashing. More synthetically we also write it as $h^{r_0}(\overline{w})$.

To rehash the row and obtain $h^{r_1}(\overline{w})$, HC choses a new random number $r_1 \in \{1,\cdots,q-1\}$ and, for each element w_i of the row, it calculates (introduced in [7])

$$h^{r_0}(w_i)^{\frac{r_1}{r_0}} = (g^{r_0\cdot w_i})^{\frac{r_1}{r_0}} = g^{r_0\cdot\frac{r_1}{r_0}\cdot w_i} = g^{r_1\cdot w_i}$$

The process can be iterated: to re-hash token $h^{r_n}(\overline{w})$, HC selects another number $r_{n+1} \in \{1,\cdots,q-1\}$ and computes $(h^{r_n}(\overline{w}))^{r_{n+1}/r_n}$ which is the re-hashed token $h^{r_{n+1}}(\overline{w})$.

In fact, HC reshuffles and re-hashes passwd_u in one single step as shown in Figure 2.

So far, we are envisioning a message flow as follows: when HC receives from LS a check query, it also receives $\mathsf{passwd}_u^{r_n}$, which it shuffles using a new ordering π', and re-hashes using a freshly generated r_{n+1}. The re-hashed, re-shuffled row of u, passwd_u, is therefore $[h^{r_n}(w_{u,\pi'(i)})]_{i\in\{1,\ldots,k\}}$, which we write compactly as $\mathsf{passwd}_u^{r_{n+1}}$. HC *performs these three steps indivisibly*: the $\mathsf{passwd}_u^{r_n}$ should not be accessed by concurrent versions of the HC before it has been shuffled and re-hashed.

What explained so far implements countermeasures (i) and (ii). However, each n^{th} time that a user u logs in and submits the password w, LS needs to calculate $h^{r_n}(w) = g^{r_n\cdot w}$ before being able to search for w's index in $\mathsf{passwd}_u^{r_n}$.

Letting LS to do this while avoiding that it gets to know u's password (i.e., by taking advantage of knowing the re-hashed password file $\mathsf{passwd}_u^{r_n}$ and the re-hashed $h^{r_n}(w)$, so anticipating the search and using

HC as an oracle) is not obvious. We need to implement countermeasure (iii) and *prevent LS from searching the file at any time that is not when a legitimate user u logs in*.

Our final solution is explained in §7 and its workflow is illustrated in Figure 3. The core idea is to inform the HC when a user is logging in, but without passing through the LS which may otherwise interfere with the communication. Because of the risk of man-in-the-middle attacks, this communication should not be over the Internet either. Instead, it *must happen on a secure second channel between the user and the HC*, which we suggest to be the *ether* and implement by letting them use a One-Time-Password (OTP) device. We are aware that, introduced without an adequate explanation, the need of a second channel and our suggestion to use an OTP may appear arbitrary and unjustified. They are not. In the rest of this section we briefly explain our reasons, but the reader interested only in the new protocol can skip it, and restart the reading from §7.

Why a Second Channel? Before concluding that we need a *second channel* between the user and the HC we tried to comply with countermeasure (iii) by other ways. One attempt was to add a module, called Keys Register (KR), to keep r_n. Abstractly, this suggests to outsource the calculation of the hash of the submitted password out from the LS. In particular, we let KR receive (u,w) and calculate the $h^{r_n}(w)$. The token is thus forwarded to the LS, who also receives the username u. Notably, KR's role cannot be played by LS itself. This would lead it to know the hash of the password and so its valid index, consequently enabling an attack as we have described previously. KR's role apart, the authentication process is not different from what we described before, with the HC that also shuffles, re-hashes, and returns the password file to the LS, but at the end the HC sends the new r_{n+1} only to the KR, which is ready for a new session.

This solution is secure, but only if KR cannot be code-corrupted. This is not an assumption that we intend to take easily. According to Juels and Rivest, the only component that is hardened secure is the HC. Thus, KR should be considered corruptible. And if it is so, the intermediate solution has a flaw. An adversary can compromise both kr.exe and ls.exe and, even under a strict Assumption2 with no back doors, manage to successfully log in. The attack is implemented by the following corrupted code, presented first in [7], where we assume h' and passwd'_u to be updates of h and passwd_u. The corrupted instructions are in red:

Algorithm 3 Code Corrupted KR

```
1: procedure kr'.exe($r_n$)
2:     while true do;
3:         ReceiveFrom(U; $(u,w)$);
4:         SendTo(LS; $h^{r_n}(w)$);
5:         ReceiveFrom(HC; $r_{n+1}$);
6:         SendTo(LS; $h^{r_{n+1}}(w)$);
```

KR resends the last w, re-hashed using the new r_{n+1} received from HC. KR does not know whether w is a valid password, but a corrupted LS does. The attack works because LS gets pieces of information beforehand, using which, he can anticipate querying the new password file and get a valid j that can be used to let the adversary in.

Alternative ways to implement (iii), such as using timestamps from the user's side as a proof of freshness do not work either since LS stands in the middle and can compromise those messages. For all this follows our conclusion that if there must be a "synchronization" between users and the HC, it must be happening over a channel that is not under the control on any module of the Honeywords System nor of the adversary. We of course welcome, and we leave it as an open challenge, to find a secure solution that does not use a second channel between the user and the HC.

7 The New Authentication Protocol

One way to realize requirement's items (i)-(iii) in agreement with the Honeywords System solution, is to empower the user (i.e., the user's browser) with the ability to hash his password w with g_n^r using the same r_n that is generated by the HC. It is (almost) equivalent to let the user play the role of KR.

However, letting HC send r directly to the user over the Internet leaves the channel exposed to man-in-the-middle attacks and introduces other issues such as that of ensuring authentication of the user. The channel through which the HC "communicates" with the user must be a second channel and not in the Internet. We already justified this choice in the previous section.

The solution that we are about to discuss now and prove secure in the next section requires that the HC and the user share an OTP device. This is employed to generate a new seed r each time that the OTP is used, a seed which is also the same for the user and the HC. The protocol message sequence diagram is detailed in Figure 3.

The OTP serves as pseudo-random generator but also as proof of freshness, since what it generates is synchronized with what the OTP generates by the HC. Here we talk of an OTP that generates a new seed each time that it is pressed.

In Figure 3, we have indicated with $\texttt{OTP}(n)$ the action of using the OTP for the n^{th} time (step **1**). The user sends to the LS, the username u and the hashed version of its password, $h^{r_n}(w)$, where the hashing takes the n^{th} OTP-generated number r_n as parameter (step **2**).

Then, the protocol follows as expected: the LS searches for an index in the password file (step **3**); the file has been reshuffled and re-hashed in a previous session by the HC, which has used in anticipation the same OTP number that the user has now used to hash the password (we will discuss in 7.1 how to handle when a user "burns" a generated number by pressing the OTP accidentally outside the login). The found index j is submitted together with the username and the row of the password file that LS has just used in the search (step **4**).

The HC checks first j against c_u (i.e., the index of the user's password) to determine whether to grant access or not (step **5**), then shuffles and re-hashes the password file's row. It also updates the c_u according to the index's re-ordering (steps in **6**). The shuffled and re-hashed file is returned to the LS (step **7**) and LS notifies the user (step **8**).

7.1 Informal Security Analysis.

We argue that there is no corruption of the LS that under our assumptions can lead to a successful attack. In particular, even if the LS learns that a particular $h^{r_n}(w)$ is a valid password, LS cannot make any use of it to anticipate the index that w will have in the new reshuffled and re-hashed password file. LS could retain an old file, but the index retrieved from it would not correspond to the new c_u' that the HC holds. It could send to the HC the username and sweetwords file's row of another user and so have this later reshuffled and re-hashed. The only gain is that LS will likely have the request rejected without never getting to know whether that hashed honeyword (and consequently the j calculated) were good for access. Note that even if two users log-in with the same password, it is very unlikely that the hashes are the same if we assume that each user has its own OTP. LS can send the username u and password file's row of another user to know the answer about the correctness of j without having the file's row of u reshuffled and re-hashed. But then, HC changes c_u and so the LS will not be able to take advantage of what he has learned; besides, the effect seems to be disastrous in terms of compromising the integrity of a future check, when u logs in again. This counts as a Denial-of-Service (DoS) but not as an attack according to Definition 1 since it does not increase the probability of the adversary to gain access, which remains $1/k$.

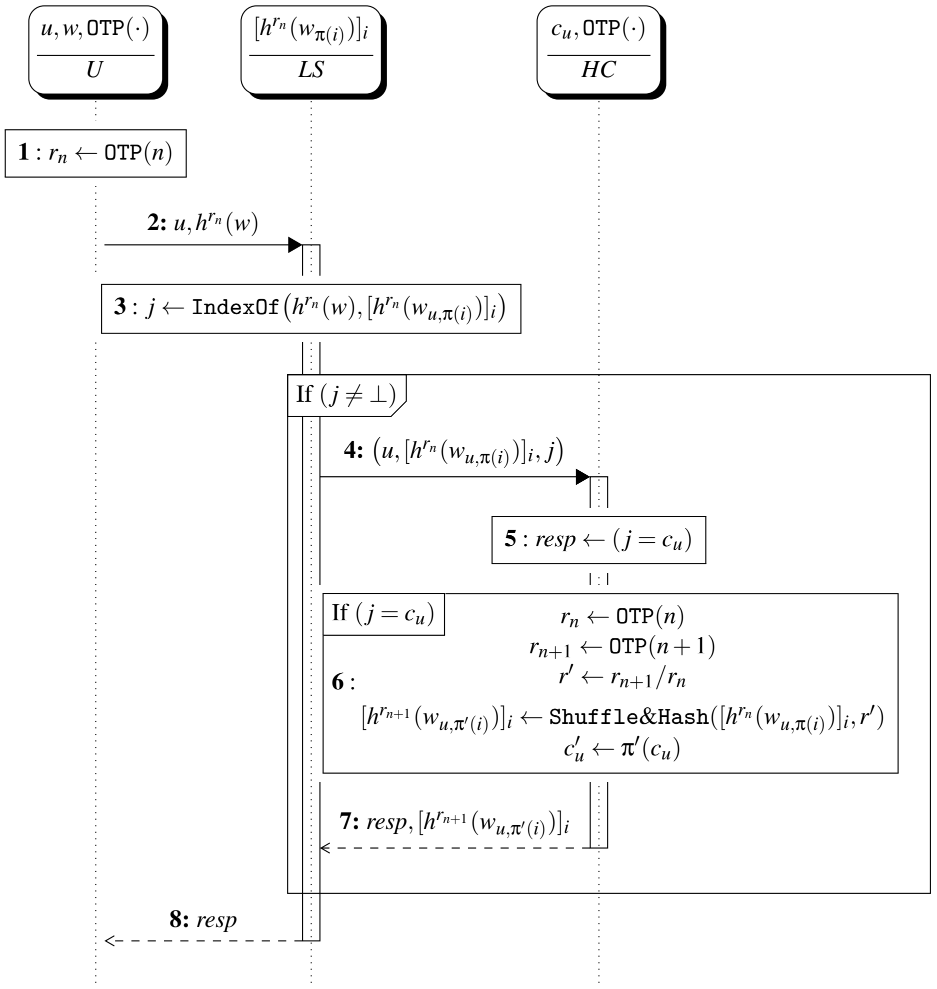

$u, w, \texttt{OTP}(\cdot)$
U
$[h^{r_n}(w_{\pi(i)})]_i$
LS
$c_u, \texttt{OTP}(\cdot)$
HC
$\mathbf{1}: r_n \leftarrow \texttt{OTP}(n)$
$\mathbf{2:}\ u, h^{r_n}(w)$
$\mathbf{3}: j \leftarrow \texttt{IndexOf}\big(h^{r_n}(w), [h^{r_n}(w_{u,\pi(i)})]_i\big)$
If $(j \neq \perp)$
$\mathbf{4:}\ \big(u, [h^{r_n}(w_{u,\pi(i)})]_i, j\big)$
$\mathbf{5}: resp \leftarrow (j = c_u)$
If $(j = c_u)$
$\mathbf{6}:$
$r_n \leftarrow \texttt{OTP}(n)$
$r_{n+1} \leftarrow \texttt{OTP}(n+1)$
$r' \leftarrow r_{n+1}/r_n$
$[h^{r_{n+1}}(w_{u,\pi'(i)})]_i \leftarrow \texttt{Shuffle\&Hash}([h^{r_n}(w_{u,\pi(i)})]_i, r')$
$c'_u \leftarrow \pi'(c_u)$
$\mathbf{7:}\ resp, [h^{r_{n+1}}(w_{u,\pi'(i)})]_i$
$\mathbf{8:}\ resp$

Fig. 3. New Authentication Protocol (Presented in [7])

Finally, our protocol is secure even under a relaxed Assumption 2. Even if the LS, learned that a particular $h^{r_n}(w)$ is a valid password, sends it back to the adversary which in turn retrieves the w, the adversary cannot use either w or the token $h^{r_n}(w)$ to gain access. He needs the token $h^{r_{n+1}}(w)$ which he cannot generate without holding also the OTP.

Before concluding, we comment on what to do if the user "burns" some of the valid OTPs. A classic solution is that the HC anticipates new versions of the password file using a certain number, say m, of the next OTPs. The file's row for user u becomes a matrix where each row is ordered with the same π':

$$\begin{bmatrix} h^{r_n}(w_{u,\pi'(1)}), & \cdots, & h^{r_n}(w_{u,\pi'(k)}) \\ & \vdots & \\ h^{r_{n+m}}(w_{u,\pi'(1)}), & \cdots, & h^{r_{n+m}}(w_{u,\pi'(k)}) \end{bmatrix}$$

The HC stores one c_u as before, but when shuffling and re-hashing the matrix for the new run, it discards all the rows that correspond to the OTP numbers that the user has accidentally burned, including the one used in the current submission (which HC receives from LS).

7.2 Formal Security Analysis

We modeled the original protocol and our proposal (Fig. 3) in the *applied-π calculus* and used ProVerif [2] to formally verify their compliance with authentication security goals. ProVerif is an automatic verifier for cryptographic protocols under the Dolev-Yao model.

Honeywords System Authentication We start by analyzing the original Honeywords System. We know already that there is an attack, but our aim is to test the proper way to model a LS that has been code corrupted according to Assumption 2. Moreover, we need to correctly interpret the results, discarding attacks originated from stronger attackers than the one defined in our threat model.

We built our formal design upon the following considerations. There are three parties: the User (U), the LS and the HC. The LS is an active attacker since it is able to read and send messages from and to the HC; the channel between LS and HC is thus public. In contrast, the channel between U and LS is private, otherwise the attacker can learn a correct pair of user and password from the beginning, contradicting Assumption 1.

Note that this decision together with the fact that the password is never transmitted in the public channel, prevents the attacker to know the submitted password at any time. It also rules out the simplest guessing (password) attack, which is the first one that ProVerif finds in the analysis, allowing the verifier to find attacks more related to the protocol's flow. We know already that a guessing attack is always possible, since Honeywords System is not designed to avoid it.

The attack described in §5 violates the following security property, introduced in [7]:

$$\mathit{correctIndex}(u,j) \implies \mathit{injct}(\mathit{indexFound}(u,p,j)) \quad \&\& \quad \mathit{injct}(\mathit{usrLogged}(u,p))$$

It expresses that, whenever the HC sends a positive answer to the LS for a submitted pair of user and index (u,j), all of these three actions occurred: (1) a user logged in with a pair of credentials (u,p) (2) the index j found by the LS corresponds to (u,p) and (3) the value stored in HC for u is equal to j. Injectivity in the expression (*injct*) captures the fact of HC processing only once each request that LS submits after events (1) and (2), to prevent interaction between LS and HC in the absence of a user.

Result. As expected, the verification indicates that the property does not hold. The attack found shows how once the attacker (in this case the LS) gets a positive answer from the HC, it is able to send a new check request to HC with the correct user and index, gaining access to the system and thus contradicting injectivity, because there was not a new $usrLogged(u,p)$ event for that second request. These observations support our model design for code-corruption and provide formal evidence that a Honeywords System resilient to the flaw must satisfy Requirement 1.

New Authentication Protocol We are now ready to apply the analysis to the new protocol. In this ProVerif model, all channels are public since the LS can send requests at any time and can learn the inputs from U and HC. We choose this design to discover any attack using any information available. Conversely, the LS's function that retrieves the index of a sweetword is private, because LS can get information from the password's file but cannot modify it.

Unlike in the original, in this protocol each instance of U is synchronized with a HC instance by a *seed*, representing that both parts generate the same OTP at the beginning of a round; the HC knows as well the index of the password. Then, to give LS the opportunity to attempt an attack using the knowledge gained during the run of the protocol, we model the fact that HC keeps running with the updated index after reshuffling. The LS is almost as in the original protocol, except that this time it receives a hashed password parametrized by the OTP, instead of a plain password. An index is a term determined by the hashed word searched and the row of sweetwords where it is searched. We introduced in [7] its representation in ProVerif as

$$indexOfHw(hashWord(w,getOTP(n)),\ shuffleNhash(u,n))$$

where *hashWord* is the hash of the plain submitted word w calculated with the seed n; *shuffleNhash* is the sweetwords' row for user u hashed with seed n.

Our equational theory relies on the *checkEqual* function in the HC, which returns *true* only when all the parameters of the indexes under comparison are equal. After a successful match, the index hold by the HC is affected by the next seed value, becoming $indexOfHw(...getOTP(\mathbf{next(n)}),...(u,\mathbf{next(n)}))$. Therefore, after this point the evaluation of *checkEqual* will be *false* for any submitted index not obtained with the new seed.

We verified in [7] that our protocol fulfills a property equivalent to the one which the Honeywords System does not:

$$\begin{aligned} correctIndex(u,j) \implies & injct(usrLogged(u,p)) \quad \&\& \\ & injct(indexFound(j,\ hashWord(p,x),\ shuffleNhash(u,y))) \end{aligned}$$

It states that every time an index j is equal to the one in the HC's database for u, then (a) the owner u of j logged in with password p and (b) j corresponds to the index of the hashed value of p in the sweetwords row for u. The conjunction ensures the execution of every step in the protocol; the injectivity ensures that each is executed only once.

In addition, we introduce the property $event(unreachable)$ to verify that LS cannot retrieve a sweetword's index of a word not submitted by a user; the event *unreachable* is triggered if the HC's check function returns *true* after shuffling and rehashing, when applied to a previously submitted hashed password.

The model also assumes, as we stated in §6, that HC must process LS's requests *atomically*, finishing a request before starting the next. Failing to implement HC this way, leads to an attack as we are going to explain in the next section, which prove that atomicity is in fact necessary.

Result. All properties were verified to be true almost immediately. It follows that even knowing that a certain $hashWord(p,getOTP(n))$ is a valid password, LS cannot use it to anticipate the new good index, since it depends on the seed value possessed only by U and HC.

The analysis also proves that event *unreachable* is indeed so; this implies that LS cannot get any advantage even if using HC as an oracle if using messages obtained from previous runs with U and HC.

We also verified the necessity for the HC to execute all its tasks (as an atomic block) concerning a request from user u before processing another request from the same u. Removing this constraint reveals an attack: let HC_1 and HC_2 be parallel runs of the HC, then (1) After a LS request, HC_1 verifies that the submitted index is correct and sends the answer to LS (2) LS submits again the correct index, HC_2 processes it, finishes the protocol and grants access (3) HC_1 continues its execution and grants access as well.

8 Discussion

The new authentication protocol just described ensures that an adversary cannot improve its chances to get access even if (s)he manages to corrupt the code of the LS, and we have demonstrated formally this claim. Nevertheless, at the eye of someone, the new design may seem overkilling and the use of the OTP unconventional. But the new protocol is simply and rightly fulfilling the requirements that we have determined at first: to avoid that the LS could retrieve a good index by processing, on- and off- line, the information it handles. Re-shuffling, for us, is the operation that makes the LS's previous knowledge of the index become obsolete; and the use of the OTP provides freshness to each re-hashing, nullifying any attempt of the LS to use the password file to calculate, off-line, a good user index. In our protocols, password authentication becomes one-time.

Our use of the OTP differs from what is common in authentication procedure: OTPs are proofs of possession of a device and, sometimes, of freshness of a session. We elevate them to become proof of possession of the password and of freshness of the session; at the same time, we use them as random seeds, achieving that no one, not even the server, can learn the password from the messages exchanged. The new reassignment of the OTP's role is a price we think is worth to be paid if one wants to be sure that no unauthorized log in ca happen despite a code-corruption of the server (under our assumptions).

Another consequence of our new design is that the Honeywords System changes considerably. It does up to a point that the use of honeywords becomes unnecessary. What the LS stores and what the HC handles are random bitstrings with no linguistic meaning. It will be self-evident in the new *setup* and *change of password* protocols (see next sections): in them, the orginal procedures suggested in Juels and Rivest to generate flat sweetwords do not make sense any more: our new protocols generate decoy words which are arbitrary, distinct, strings and are not honeywords anymore. Still, we preserve that is the user who begins the process by choosing a password; and we believe it is important that he chooses it in a way for him meaningful from a usability viewpoint: but our OTP is used to add randomness, and this seems compensating any poor choice of passwords, thus protecting from phishing and from shoulder-surfing attacks until the user remains in possess of the OTP device. More research is however required to give evidence to this last claim.

9 Setup Protocols

So far we have discussed only the authentication phase. In this section we extend our study to cover the setup phase. We introduce first the Honeywords System's setup protocol to later present how that phase operates for our protocol described in §7.

At registration of user u the Honeywords System creates an identifier *uid* for u and then generates a randomly ordered list $[w_x]_u$ of k sweetwords ($x \in [1,k]$), containing $k-1$ honeywords based on the given password *pwd* and the password itself. Next, it retrieves the index c_u of *pwd* in such list and creates a new one with the hash value of each w_i. LS stores this list, $[h(w_x)]_u$, and securely notifies c_u to HC, who in turn stores the entry (uid, c_u). The protocol is depicted in Fig. 4.

A code corrupted LS completely invalidates the Honeywords System's registration process given that it obtains all the knowledge (password, user and even index) of every new user, hence, LS can evidently store any valid pair of credentials and directly use it during the login phase as described in Algorithm 2 in §5. On the contrary, the LS possessing this information does not represent a critical threaten in our solution since the OTP generated value is needed too; that the LS gets the password in clear opens nonetheless doors to attacks. We will discuss this after presenting our protocol's setup phase.

Our protocol requires the user to collect in person an OTP device. On its side, HC shares the OTP's generating algorithm which outputs the same r as the user's device.

The registration protocol (Fig. 5) then proceeds as follows: user u obtains the first number from the OTP device (**1**), which is used to send a hashed version of his/her password $h^{r_1}(pwd)$ (**2**). On reception, LS creates an identifier *uid* for u (**3**) and sends it to HC together with the password received (**4**). HC performs then all the registration tasks: retrieves r_1, the same OTP number obtained by u; creates a list of $k-1$ random words and hashes them using r_1 as a parameter for the hash function (**5**); inserts $h^{r_1}(pwd)$ in the list and stores its index. At this point, HC has already the password hidden with decoy words, the rest of the actions prepare the system for the authentication phase: HC obtains the next OTP number and uses it to rehash and reshuffle the list of hashes (**6**), this is needed in order to rule out the chance of LS using the known hash $h^{r_1}(pwd)$; it also updates the index of the hash corresponding to the password, according to the reshuffling output (**7**). The referred index is stored as valid for *uid* in HC (**8**), who finally sends the reshuffled row back to LS (**9**).

An important feature of this protocol is that even if an adversary gets in possession of the password, (s)he still needs to get the OTP's number to generate the hash value that would eventually grant access. This leads us to remark that our new protocol achieve much more than to detect whenever a password's file has been stolen; it actually help us to detect whenever the LS has been corrupted. We verify these security claims in the following part.

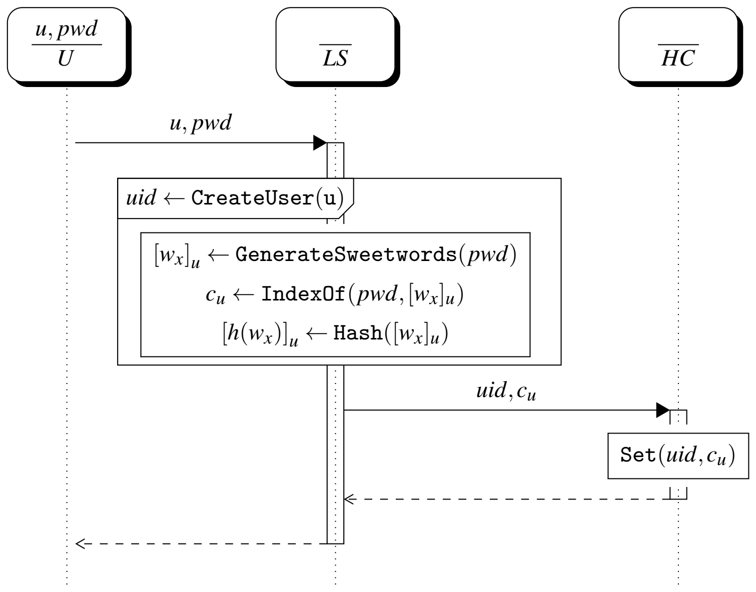
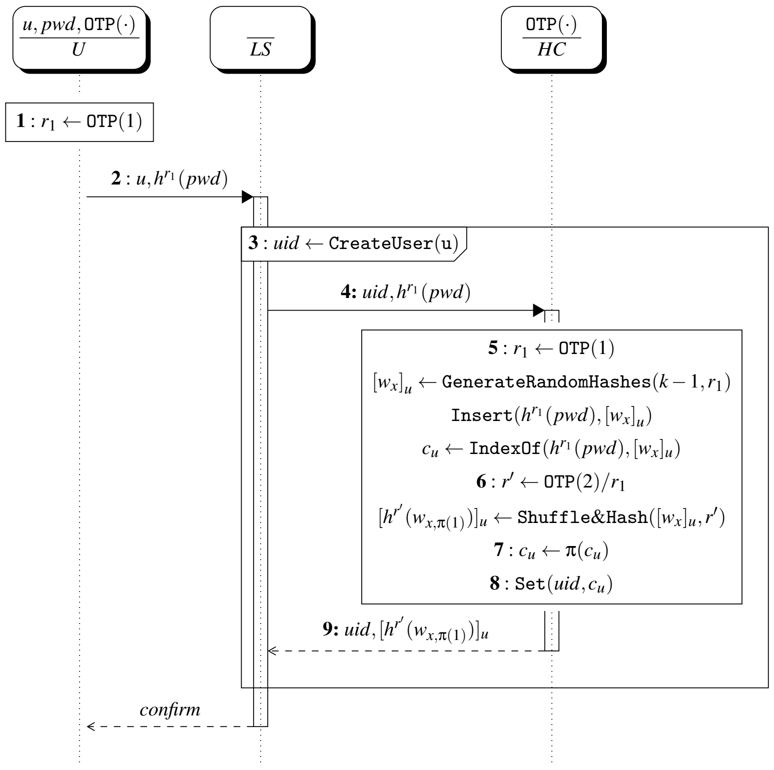

$\frac{u, pwd}{U}$
LS
HC
u, pwd
$uid \leftarrow$ CreateUser(u)
$[w_x]_u \leftarrow$ GenerateSweetwords(pwd)
$c_u \leftarrow$ IndexOf($pwd, [w_x]_u$)
$[h(w_x)]_u \leftarrow$ Hash($[w_x]_u$)
uid, c_u
Set(uid, c_u)

Fig. 4. Setup protocol in the Honeywords System.



$u, pwd, \texttt{OTP}(\cdot)$ / U
LS
$\texttt{OTP}(\cdot)$ / HC
$\mathbf{1}: r_1 \leftarrow \texttt{OTP}(1)$
$\mathbf{2}: u, h^{r_1}(pwd)$
$\mathbf{3}: uid \leftarrow \texttt{CreateUser(u)}$
$\mathbf{4}: uid, h^{r_1}(pwd)$
$\mathbf{5}: r_1 \leftarrow \texttt{OTP}(1)$
$[w_x]_u \leftarrow \texttt{GenerateRandomHashes}(k-1, r_1)$
$\texttt{Insert}(h^{r_1}(pwd), [w_x]_u)$
$c_u \leftarrow \texttt{IndexOf}(h^{r_1}(pwd), [w_x]_u)$
$\mathbf{6}: r' \leftarrow \texttt{OTP}(2)/r_1$
$[h^{r'}(w_{x,\pi(1)})]_u \leftarrow \texttt{Shuffle\&Hash}([w_x]_u, r')$
$\mathbf{7}: c_u \leftarrow \pi(c_u)$
$\mathbf{8}: \texttt{Set}(uid, c_u)$
$\mathbf{9}: uid, [h^{r'}(w_{x,\pi(1)})]_u$
confirm

Fig. 5. New Setup Protocol.

Note that although the original idea of the honeywords seems to be invalidated in our solutions, it is not the case though; the use of decoy words to disguise the password, even if they are not related to the password itself anymore, still lowers the probability of an adversary that manages to reverse hashes.

9.1 Formal Security Analysis

Honeywords System Setup Our model of a code corrupted LS follows the considerations detailed for the authentication protocol in §7.2. In this case though, the property to verify is:

$$storedHC(uId(u), j) \implies injct(submittedCred(u, p)) \quad \&\& \quad injct(indexOfPwd(uId(u), j))$$

It expresses that a pair of user identifier and index $(uId(u), j)$ is stored in the HC, only when both occur: (1) the user u submitted a registration request with password p and (2) the index of p in the row of u's sweetwords generated by LS is exactly that j.

The property is violated if the LS simply sends the HC a password different than the one submitted by the user. Another attack consists on the LS selecting and recording any index, then, sending it to the HC; since the only task for the HC is to store the values received, there is no risk for the attacker to be discovered of giving misleading information. Both attacks reflect as DoS to u. Here we are not interested in such kind of attacks, however, the attack trace found in the analysis confirms that even when the registration process is successful, a corrupted LS gets to know the password in clear, hence, it is able to use it in the login phase as previously discussed.

New Setup Protocol We represent a corrupted LS in our setup protocol by making public the channel through which it sends messages to the HC; this reflects that the attacker is authorized to observe anything from and to send anything to the HC. The security property captures the same idea expressed for the Honeywords System, except that this time we need to consider as well the random seed used for hashing.

$$\begin{aligned}&\mathit{saveDecoys}(u, \mathit{shuffleNhash}(\mathit{row}, \mathit{nextOTP}(s)))\ ^{(\mathbf{C})} \implies \mathit{injct}(\mathit{setupRequest}(u, \mathit{hashWord}(p, \mathit{getOTP}(s))))\ ^{(\mathbf{E1})}\\ &\qquad\&\&\quad \mathit{injct}(\mathit{storedHC}(u, \mathit{indexOf}(\mathit{hashWord}(p, \mathit{getOTP}(s)), \mathit{row})))\ ^{(\mathbf{E2})}\end{aligned}$$

In words, we verify that whenever the LS stores a row for user u reshuffled and rehashed with the seed $s+1$ (**C**), two events must have occurred: (**E1**) u has sent a request for registration with password p and using the OTP number s, and (**E2**) HC stored initially as a valid pair (u, j) where j is the index of $h^s(p)$ in the list of decoy words created for u. Note that such index is updated after each reshuffling.

Analogous to the Honeywords System's case, the attack that we found for this property consists in the LS sending to HC a different password than the one submitted by u.

Additionally, we verified that the password remains secret at every moment. Remark that unlike in the original Honeywords System, learning the plain text password in this case does not allow a corrupted LS to directly obtain access during the login phase; to successfully login it requires the hashed value of such p, which is calculated with a value that the user obtains from the OTP device. However, under a relaxed Assumption 2, learning the password introduces the possibility of the adversary carrying out off-line dictionary attacks. In opposition, the knowledge of previous hashed values does not give any clue to the attacker about the corresponding new hashes.

10 Change of Password Protocols

The change of password phase involves a combination of the setup and authentication protocols. After a password update request, the user u is required to provide his credentials in order to validate that the modification attempt is authentic (as a side implementation note, carrying out the authentication in the first place assists to keep separated the management of OTPs between stages). On successful authentication, the protocol proceeds similarly to the setup phase, excluding the creation of a new user.

In the Honeywords System the LS creates and hashes sweetwords based on the new password newW, and then retrieves u's id and the index of newW in the sweetwords' list. It sends the retrieved pair to the HC, which in turn updates the index value stored for u. Fig. 6 represents this protocol, where $authentication(u, w)$ is the response from executing the authentication protocol in Fig 1 and w is the old password.

As for the new protocol, LS performs the authentication in Fig. 3 with the old password w (**1**); for this step it uses the OTP number n, i.e., the pair submitted for authentication is $(u, h^n(w))$. On a successful response, the user u is allowed to submit the new password newP which is hashed with a freshly generated OTP number $n+1$ (**2**); LS gets u's identifier and proceeds sending the new credentials to HC (**3**). The tasks performed by HC are the same as for the setup, taking the current $n+1$ instead of the first generated OTP. Also, in this case u' index is updated instead of inserted in HC. The protocol is displayed in Fig. 7.

10.1 Formal Security Analysis

Honeywords System Password Update Given that the core of the protocols for password update and setup is essentially the same, both protocols are subject to identical attacks. A simple attack to the update in Honeywords System consists on the LS (remembering and) sending to the HC the password's index f of a selected user. Like in the setup phase, LS can use the recorded information to gain access during

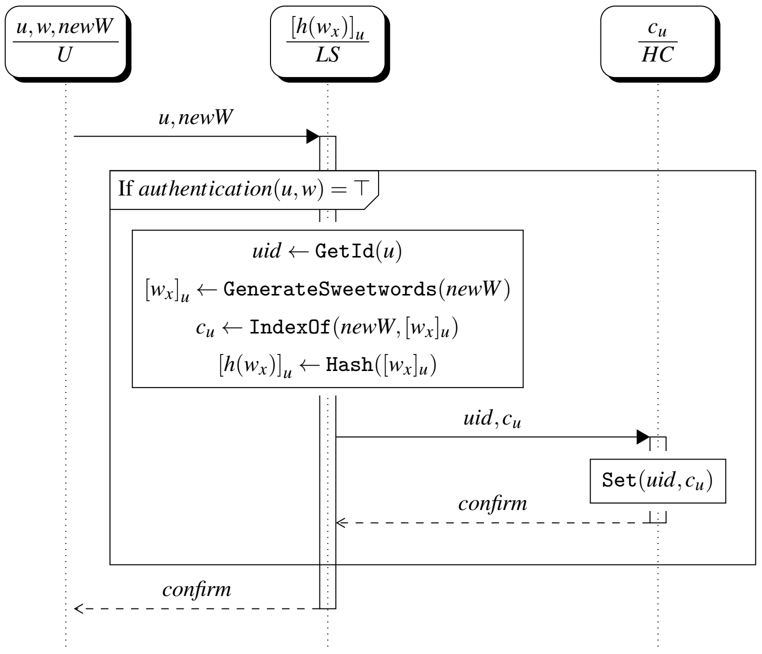

$u, w, newW$
U
$[h(w_x)]_u$
LS
c_u
HC
$u, newW$
If $authentication(u,w) = \top$
$uid \leftarrow \texttt{GetId}(u)$
$[w_x]_u \leftarrow \texttt{GenerateSweetwords}(newW)$
$c_u \leftarrow \texttt{IndexOf}(newW, [w_x]_u)$
$[h(w_x)]_u \leftarrow \texttt{Hash}([w_x]_u)$
uid, c_u
$\texttt{Set}(uid, c_u)$
confirm
confirm

Fig. 6. Password's Update Protocol in the Honeywords System.

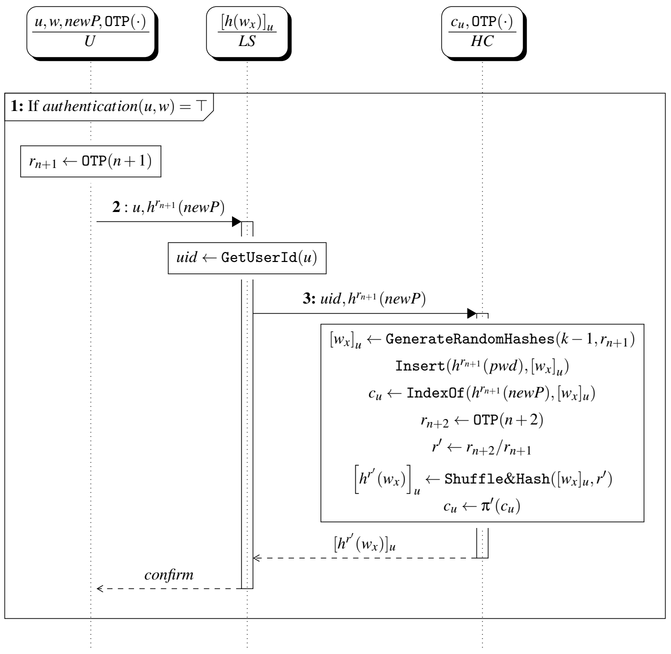

$u, w, newP, \texttt{OTP}(\cdot)$
U
$[h(w_x)]_u$
LS
$c_u, \texttt{OTP}(\cdot)$
HC
1: If $authentication(u,w) = \top$
$r_{n+1} \leftarrow \texttt{OTP}(n+1)$
2 : $u, h^{r_{n+1}}(newP)$
$uid \leftarrow \texttt{GetUserId}(u)$
3: $uid, h^{r_{n+1}}(newP)$
$[w_x]_u \leftarrow \texttt{GenerateRandomHashes}(k-1, r_{n+1})$
$\texttt{Insert}(h^{r_{n+1}}(pwd), [w_x]_u)$
$c_u \leftarrow \texttt{IndexOf}(h^{r_{n+1}}(newP), [w_x]_u)$
$r_{n+2} \leftarrow \texttt{OTP}(n+2)$
$r' \leftarrow r_{n+2}/r_{n+1}$
$\left[h^{r'}(w_x)\right]_u \leftarrow \texttt{Shuffle\&Hash}([w_x]_u, r')$
$c_u \leftarrow \pi'(c_u)$
$[h^{r'}(w_x)]_u$
confirm

Fig. 7. New Password Update Protocol.

authentication. Notice too that once more the new password is directly observed by the LS, thus, the protocol is powerless in front of a code corrupted LS.

For this phase we want to ensure that whenever HC sets j as index of user u, it must have been a password update request coming from u, who was correctly authenticated in the system with the old password p. Also the LS updated the sweetwords row of u with words based in the new password *newP* and the index of *newP* in the sweetwords row is j. This is expressed by the property:

$$\begin{aligned} \mathit{indexUpdated}(\mathit{uId}(u),j) \implies & \mathit{injct}(\mathit{updateRequest}(u,p,\mathit{newP})) \quad \&\& \quad \mathit{injct}(\mathit{authenticated}(u,p)) \\ & \&\& \quad \mathit{injct}(\mathit{passwordUpdated}(u,\mathit{newP})) \quad \&\& \quad \mathit{injct}(\mathit{indexOfPwd}(\mathit{uId}(u),j)) \end{aligned}$$

Our analysis found the attack trace corresponding to sending a self-selected index.

New Password Update Protocol Similarly to the Honeywords System's case, the new setup protocol could be considered as an instance of the new update protocol executed with the first OTP number, with the only difference that, instead of updating, the LS stores the hashed row obtained from the HC. Besides, we abstract the OTP as a function parametrized by a counter. Therefore, our formal model is not significantly affected and changes consist merely in renaming functions.

The attacks stand hence according to the findings for the setup protocol and the analysis is to verify that whenever the LS stores a row for user u reshuffled and rehashed with the seed $s+1$, then (i) u has sent a change of password request with p and s as input parameters for password and OTP respectively and (ii) the valid index stored in HC is the index of $h^{s+1}(p)$ in the list of decoy words created for u. This is what the following formula captures:

$$\begin{aligned} \mathit{updateDecoys}(u,\mathit{SandH}(\mathit{row},s+1)) \quad \implies \quad & \mathit{injct}\,[\mathit{updateRequest}(u,\ \mathit{hash}(p,\mathit{OTP}(s)))] \\ \&\& \quad \mathit{injct}\,[\mathit{indexInHC}(u,\ \mathit{indexOf}(\mathit{hash}(p,\mathit{OTP}(s+1)),\mathit{SandH}(\mathit{row},s+1)))] & \end{aligned}$$

The code of all the formal models of the protocols presented here, as well as the results of the analysis, are available at `https://github.com/codeCorruption/HoneywordsM`.

11 Complexity and Performance

The contribution of this research is mainly theoretical but we judged useful to test the performance of what we propose. We sketch a complexity analysis and benchmark an implementation of both the original and our authentication protocols. We test them with respect to different implementations of the elliptic curve (EC) multiplication, which we used to execute the main operation of our protocol: exponentiation.

Although we implemented as well the setup phase, the analysis and benchmarks focus uniquely in the authentication phase.

Complexity analysis. The analysis assumes that an elliptic curve multiplication takes constant time $t_{\texttt{CURVE}}$ (which depends on the employed `CURVE`): this protects implementations against remote timing attacks [4].

Let us now consider the operations that affect the performance. Once received the password, LS calls `IndexOf` to search the index of the submitted password among the sweetwords. Given that the sweetwords are not ordered and also are constantly reshuffled, this is a linear search. In the worst case it can be done in $O(k)$ time, where k is the number of sweetwords per user. In case of a match, the HC checks the validity of the index in $O(1)$ time. Next, the HC calls `Shuffle&Hash`; this function shuffles the sweetwords in $O(k)$ time and performs k times an EC multiplication in $k \cdot t_{\texttt{CURVE}}$ time. The last equation is linear in k for a fixed

CURVE. Since each of the previous operations takes at most $O(k)$ time, the time complexity of the new protocol is $O(k)$. As well, for a fixed k, the execution time increases linearly as $t_{\texttt{CURVE}}$ grows. Moreover, EC multiplication is CPU intensive and dominates the total execution time. This is also confirmed by our empirical results (see Fig. 8(a)).

Communication Cost. In the original Honeywords system, the communication cost per login comes from messages (u, j) and *resp*. We denote the number of bytes required to encode (u, j) and *resp* by $|(u, j)|$ and $|resp|$ accordingly, and obtain the data transfer rate per login as $C = |(u, j)| + |resp|$. While the data flow remains the same, our protocol brings the following communication overhead to the original Honeywords system: LS sends the sweetword hashes $[h^{r_{n+1}}(w_{u,\pi'(i)})]_{i=1}^{k}$ and receives the updated ones. The number of bytes required to encode a password hash depends on the employed curve and is denoted by $\mathtt{H_{CURVE}}$. Thus, LS sends $|(u, j)| + k\mathtt{H_{CURVE}}$ bytes and receives $|resp| + k\mathtt{H_{CURVE}}$ bytes per login. As a result, the total data transfer rate per login between LS and HC is computed as $C + 2k\mathtt{H_{CURVE}}$ bytes.

Since k, the number of sweetwords, is a constant defined by the system, and $\mathtt{H_{CURVE}}$ is constant too, the overload in communication is bounded. We have not simulated nor evaluated how much this may affect a server's ability to process a great number of log-in attempts per unit of time, but we are inclined to believe that this loss in performance is not so dramatic. Of course one may will to discuss whether the solution that emerges from our analysis by fitting our requirements is not actually an overkill in itself. This is a legitimate question which we discuss in § 12.

Implementation. We implemented our solution in C# at the Microsoft .NET framework.[2] Elliptic curve operations are performed using Bouncy Castle Cryptographic Library, although a faster version may be obtained by native language implementations or libraries.

In our implementation, u, j and *resp* are implemented as integers, hence C equals 12 bytes and $\mathtt{H_{CURVE}}$ takes 57, 65, 97, and 133 bytes for P-224, P-256, P-384 and P-521 accordingly. Fig. 8(c) compares data transfer rates with different settings.

Performance Analysis. This section provides experimental results about the efficiency of our proposed authentication protocol with two questions in mind: *How does number of verifications per second correlates with the number of honeywords? What is the impact of the selected curve on verification speed?* The results presented have been performed on notebooks with Intel Core i7 CPU and 8GB of RAM over an idle network. We measured the total execution time on server side computations and communication over the network separately. Roughly speaking, our prototype reaches a decision for each login request below 9 ms. Table 1 summarizes the overall performance with different settings.

Another performance consideration is the cost of avoiding login failures due to out-of-synchronization of OTPs. System policies may follow the strategy discussed in Section 7. The computational overhead of both, Login Server and Honeychecker, increases linearly on the number of copies in the password file.

It is reasonable to expect that the time required for re-encryption directly depend on the number of honeywords for a user. Fig. 8 illustrates the time measurements. It can be seen that the time required for verifying a single user increases linearly with the number of honeywords per user. The Honeychecker performs one EC multiplication for each honeyword, which is the most expensive part of its function, and the result is aligned with our theoretical expectations. Our solution preserves the computational characteristics of the original honeywords protocol: performance is linearly dependent on the number of honeywords. On the other hand, we can see from Fig. 8 (and from Table 1) that the time to run the employed curves increases with the number of honeywords.

[2] Source code is available under GPLv3 at `https://github.com/codeCorruption/HoneywordsM`.

Table 1. Performance results of our implementation. Login Server and Honeychecker columns display the time in milliseconds for a single authentication on LS and HC, respectively. Throughput column shows the maximum number of verifications per second. Round-Trip Time (RTT) is the network delay during the experiments (Taken from [7]).

k	Curve	Login Server (ms)	Honeychecker (ms)	Throughput (login/s)	RTT (ms)
5	P-224	0.011	1.709	581	24.446
5	P-256	0.009	1.796	554	28.917
5	P-384	0.009	2.242	444	31.502
5	P-521	0.010	2.541	392	30.812
10	P-224	0.009	2.680	372	24.534
10	P-256	0.009	3.317	301	29.885
10	P-384	0.010	4.365	229	34.918
10	P-521	0.010	4.793	208	29.414
15	P-224	0.009	3.856	259	27.063
15	P-256	0.010	4.868	205	30.896
15	P-384	0.009	6.240	160	36.253
15	P-521	0.010	6.842	146	31.445
20	P-224	0.009	5.016	199	26.867
20	P-256	0.010	6.301	158	29.355
20	P-384	0.010	8.220	122	32.724
20	P-521	0.011	8.965	111	31.944

Figure 8(b) compares our protocol with the reference implementation. The client side latency of both, original and improved protocols stays almost constant. Considering the delays caused by the network, the computational overhead of our protocol is relatively small. It might not be even noticed by the clients.

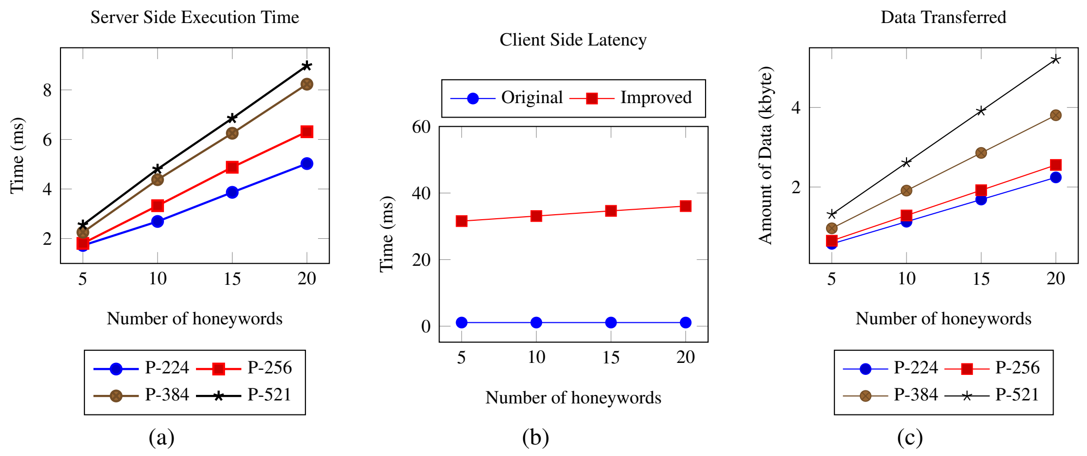


Fig. 8. (a) CPU time required to verify a user depending on the number of honeywords and employed curve. (b) Client side latency comparison between original protocol and our proposal with NIST Curve P-256. (c) The amount of the data (in kilobytes) transferred between the Login Server and the Honeychecker (Taken from [7]).

12 Conclusion

In this work we proposed a password-based authentication system motivated by a challenge left open by Juels and Rivest in [9]: how to protect a Honeywords System against a code corrupted login server (LS). A Honeywords System's architecture is intended to detect when a password file has been stolen.

We provided protocols for the three functionalities defined for a Honeywords System: the *setup* or registration, the *log-in* or authentication and the *change of password* or update. The study of the setup and the change of password extends the initial scope presented in the conference paper [7], where only the authentication protocol is addressed.

We introduced a precise definition of code-corruption, according to which, the adversary model is less powerful than the Doloev-Yao model but powerful enough to correctly guess a users's password, hidden among a list of decoy words known by the adversary, with higher probability of success than with a honest LS. The flaw resides indeed in the LS knowing eventually the user's valid (hashed) password.

The solution that we propose prevents the LS to make, off-session, any good use of what he knows, but based on the requirements derived from studying attacks on the original Honeywords System, the new protocols consist in shuffling and rehashing the password (plus decoy words) after any user's attempt to log in. This solution impedes as well to the LS to interact with the HC in the absence of a legitimate user's message. In order to control such event, the user and the HC need some synchronization through a channel not controlled by the LS or by a man-in-the-middle. We propose OTPs for this purpose. Our protocols' security is supported by a formal analysis in Proverif.

Aiming to assess its feasibility, we implemented the setup and authentication phases in `C#`. A benchmark analysis on the authentication protocol shows that it performs reasonably well.

The new protocols invalidate until a certain point the need to detect when a password file is stolen because the adversary cannot gain access to the system without the OTP that creates the authentication token (i.e., the hash of the password). Of course leaking a password is still a serious weakness because users may reuse the same password across different sites. Yet, the strategy that we proposed suggests a completely new direction for password authentication, a procedure that is resilient even if a password is lost. In this sense, the proposed system's goal is not anymore to detect passwords' leakage but to detect whenever the code of a LS has been corrupted.

This approach puts in the table a password-based authentication process where users still type their passwords but where the token that the LS checks in the password file is one-time-valid. An implementation of this concept still differs from current OTP-based solutions used e.g., in home-banking, due to the assumption that it must work even when the LS has been code-corrupted. We consider this an open problem in password-based authentication and an interesting line for future work.

References

1. Beck, K.: Hackers are selling account credentials for 400 million Tumblr and MySpace users. Machable (May 2016), `http://mashable.com/2016/05/31/myspace-tumblr-hack`, `http://mashable.com/2016/05/31/myspace-tumblr-hack` (last access: 4th September 2017)
2. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE Computer Security Foundations Workshop. pp. 82–96. IEEE (2001)
3. Botha, R.A., Eloff, J.H.P.: Separation of duties for access control enforcement in workflow environments. IBM Systems Journal **40**(3), 666–682 (2001)
4. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: European Symposium on Research in Computer Security. pp. 355–371. Springer (2011)

5. Erguler, I.: Achieving flatness: Selecting the honeywords from existing user passwords. IEEE Transact. on Dependable and Secure Computing **13**(2), 284–295 (2016)
6. Furnell, S.M., Dowland, P., Illingworth, H., Reynolds, P.L.: Authentication and supervision: A survey of user attitudes. Computers & Security **19**(6), 529–539 (2000)
7. Genc, Z.A., Lenzini, G., Ryan, P.Y.A., Vazquez-Sandoval, I.: A security analysis, and a fix, of a code-corrupted honeywords system. In: Proceedings of the 4th International Conference on Information Systems Security and Privacy (2018)
8. Goel, V., Perlroth, N.: Yahoo Says 1 Billion User Accounts Were Hacked. NT Times Online (Dec 2016), `\url{https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html}`, `https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html` (last access: 4th September 2017)
9. Juels, A., Rivest, R.L.: Honeywords: Making password-cracking detectable. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 145–160. ACM (2013)
10. Newman, L.H.: Yahoo's 2013 email hack actually compromised three billion accounts. Wired (Oct 2017), `https://www.wired.com/story/yahoo-breach-three-billion-accounts/`
11. NIST: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (2015)