Points breakdown: Q1: 20 (10+10), Q2: 30 (7+8+7+8), Q3: 20 (10+10), Q4: 30.

Question 1.

1.1. Windows

Yes, you can provide this functionality. Store $h(P)$, the hash of the user's password, as usual. When someone enters a password P', hash it as well as a case-reversed version P''; if either hash matches $h(P)$, accept the user.

Alternative scheme: Store both $h(P)$ and $h(P^*)$, where P^* is a case-reversed version of P. When someone enters a password P', hash it and if the result matches either stored value, accept them.

Security analysis. The loss of security is at most a two-fold reduction in search space. If you can guess a user's password in n trials in this system, you can guess it with at most $2n$ trials in a conventional system (for every guess G in this system, try guessing both G and a case-reversed version of G against the conventional system).

Alternative analysis: Alternatively, we can calculate the reduction in search space as follows. Suppose the user chooses a uniformly random password of length ℓ, consisting of digits and letters. There are 62^ℓ possible passwords. If we consider two passwords equivalent if one is a case-reversed version of the other, then there are only $10^\ell + (62^\ell - 10^\ell)/2 \approx 62^\ell/2$ equivalence classes, so this system reduces the effective search space from 62^ℓ down to about $62^\ell/2$.

Of course, this analysis is a little bit bogus, because few people choose their password as a uniformly-random string of digits and letters, but I accepted it because it still gives a reasonable way to quantify the loss of security.

1.1. Mac

You can provide the functionality, but it comes at a non-trivial loss in security. One method is to store $h(P^*)$, where P^* is a version of P with all letters converted to upper case. When someone enters a password P', upper-case it, hash it, and if the result matches either stored value, accept them.

Alternative scheme: Store just $h(P)$. When someone presents password P' containing k letters, try all 2^k variations of P' (each letter either lower case or upper case), hash each variation, and if any of those hashes matches the stored hash, accept the user.

Security analysis. The loss of security is about a 2^k-fold reduction in search space, for passwords containing k letters. This is a more significant loss of security, and there seems to be no way to avoid it.

I accepted answers that said this loss of security might be acceptable, because the site can just encourage people to choose a longer password. I also accepted answers that said this loss of security is not acceptable, because passwords already tend to have low entropy and you can't stand to lose any more entropy.

Alternative analysis: Alternatively, we can calculate the reduction in search space as follows. Suppose the user chooses a uniformly random password of length ℓ, consisting of digits and letters. There are 62^ℓ possible passwords. However, taking into account the equivalence between passwords that differ only in case, there are only 36^ℓ non-equivalent passwords. This is a $(62/36)^\ell \approx 1.7^\ell$ reduction in the search space. For instance, for $\ell = 8$, this the difference between 47 bits of entropy vs 41 bits of entropy.

Question 2.

Preliminaries: Call a user *weak* if he/she uses a 10-bit password, or *strong* otherwise. There are about 100,000 weak users. Let's look at how many users there are, with each possible password:

- For each possible 10-bit password, we expect about 100 weak users with that password. More precisely, the number of such users is normally distributed with mean 97.7 and standard deviation 9.9; thus, with overwhelming probability (≈ 0.9999993), each 10-bit password will have at least 50 weak users with that password.
- For each 20-bit password, we expect about 9 strong users with that password (mean 9.4, standard deviation 3.1). With overwhelming probability, every 20-bit password will have fewer than 49 strong users with that password.

Let's assume that no one else is attacking the site and that users never make mistakes in entering their own passwords. Then, if we connect to the site and provide a username of a weak user, we will always be asked for a CAPTCHA; whereas if we provide a username of a strong user, we will never be asked for a CAPTCHA. This provides a simple way to tell whether a user is weak or strong: enter in the username into the webpage, and see whether it asks for a CAPTCHA or not. Also, it follows that every guess we want to make at a weak user's password will require solving a CAPTCHA, whereas we can guess strong users' passwords without solving any CAPTCHAs as long as we don't go over the 10-failed-login-per-user threshold.

2.1. Targeted attack, closed site

If we target a weak user, we'll have to solve on average 512 CAPTCHAs (worst case: 1024 CAPTCHAs) before we find their password. This will cost about $1.00 and take 512 requests (on average).

If we target a strong user, we'll have to try about 2^{19} guesses at the password on average (worst case: 2^{20}) before we find their password, which requires solving about 2^{19} CAPTCHAs. This will cost about $1050 and take 2^{19} requests (on average).

Averaging over these possibilities, the expected cost for targeting a random user is about $1040 and 520,000 requests (on average).

2.2. Untargeted attack, closed site

The cheapest attack is to guess passwords of strong users, making about 10 guesses at the password against many different strong users. More precisely, we repeat the following many times until we have successfully broken some user's account:

1. Pick a username at random.
2. Enter this username into the site. Did we get a request for a CAPTCHA? If yes, go back to step 1 (they are a weak user; no good). If no, continue on (they are a strong user).
3. Make a random guess at their 20-bit password. Try logging in with this password. Try this for 10 different password guesses. If any of them succeeds, we are done. Otherwise, go back to step 1.

Since attacking strong users does not trigger any CAPTCHA requests, this attack requires no CAPTCHAs and about $2^{19}/0.99$ requests (on average). It is free: the cost is \$0.

Comment. It's a bit counter-intuitive that the cheapest attack is to go after *strong* users, not weak users. This is a consequence of the fact that we're minimizing cost, not requests. Attack weak users minimizes the number of requests, but increases our costs because we have to solve a CAPTCHA for every guess we make (since weak users always trigger CAPTCHA requests).

2.3. Targeted attack, open site

If we target a weak user, we can't do any better than in Q2.1; we have to solve one CAPTCHA for each guess at their password. Average cost: \$1.00, plus 512 requests.

If we target a strong user u, we can use the open nature of the site to our benefit to help us find their password. We know that (initially) an attempt to log in as u does not trigger any CAPTCHA request. To test whether u's password is p, we create 50 accounts all with password p, then enter u into the site; if this triggers a CAPTCHA request, we know that u's password is p. We can repeat this once for each possible 20-bit password. This never requires solving any CAPTCHAs, so it is free: the cost is \$0. It takes about $2^{19} \times 51$ requests (on average).

If we target a random user, the average cost will be \$0.01 and around $2^{19} \times 50.5$ requests.

2.4. Untargeted attack, open site

We can use the same attack from Q2.2. Cost: \$0, and about 2^{19} requests (on average).

Alternative attack: Use the open nature of the site to help us, like this:

1. Scan about one million usernames and see which ones trigger a CAPTCHA.
2. Pick a 20-bit password p and create 50 new accounts with this password.
3. Finally, re-scan those one million users again to see which ones trigger a CAPTCHA this second time.

Assuming there is no other activity on the site, if a username does not trigger a CAPTCHA on the first scan but does trigger a CAPTCHA on the second scan, then their password must be p, so now we know their password. With good probability, we will find at least one such user. This attack is free; it does not require solving any CAPTCHAs, so its cost is $0. It uses about 2^{21} requests.

Alternative "attack": The attacker can create a new account, with a known username and password. Now the attacker knows an account on the site. Job finished. OK, this is a bit cheesy, but I accepted it. (I admit I didn't think of this when I wrote the question, but I guess it *does* satisfy all of the conditions I specified in the statement of the question, so I'll take it.)

Extensions

This question suggests some fun follow-on thought problems:

- If you want to crack the password of all of the site's strong users, without solving any CAPTCHAs, what's the most efficient way to do it? (Naively, you could repeat the attack from Q2.3 10 million times, but this would require about 250 trillion requests. You can do better!)
- More usefully, can we find a better way to protect web sites that does not have this sort of weakness? It would be nice to have a scheme that increases the cost of password search, while maximizing usability, i.e., minimizing how often legitimate users (or, at least, legitimate users who choose good passwords) have to solve a CAPTCHA.

I'll reserve solutions to these problems until the end of this solution set, to give you a chance to think through these problems on your own if you like.

Question 3.

3.1. Precise measurements

We will start by trying all the one-character passwords, `a`, `b`, `c`, ..., `9`, and measuring how long each takes. One of them will take about 10 nanoseconds longer; that one reveals the first character of the secret password. Measurement error is not an issue, since it is so much smaller than the difference between the time for correct vs incorrect guesses.

Once we know the first character of the secret password, say `q`, we next try all ways of extending this to a two-character password: `qa`, `qb`, `qc`, ..., `q9`. One will take longer, revealing the second character of the password. We continue in this way until we find the complete password.

This takes $62 \times 20 = 1240$ guesses.

(It's possible to reduce this to about 620 guesses on average, using our knowledge of how long an incorrect and a correct guess should take.)

3.2. Noisy measurements

Yes, it is still possible. The attack is similar, but now we'll attempt to dampen down the noise by averaging many observations. Instead of trying the one-character `a` (once), we'll collect many measurements of passwords that all start with `a`, say n of them, and we'll average these measurements to get our estimate of the time taken by the loop when the first character is `a`. We'll do the same with n passwords starting with `b`, to estimate the time taken by the loop when the first character is `b`, and so on for each other possibility at the first character of the secret password. If n is large enough, the correct guess at the first character of the secret password will have a time estimate that is noticeably larger than all others.

How large do we need n to be, for this attack to succeed? Well, I did some calculations, and $n = 550,000$ should be large enough. If we average 550,000 measurements, then the standard deviation decreases from 1000 nanoseconds to about $1000/\sqrt{550,000} = 1.35$ nanoseconds. If the error (after averaging) exceeds 5 nanoseconds, our attack goes awry. This corresponds to $5/1.35 = 3.7$ standard deviations. The probability that a normal distribution falls more than 3.7 standard deviations below the mean is about 0.0001, so we have only a 1 in 10,000 chance of going wrong in a guess at a single character of the password.

We have to repeat this many times, calculating 1260 estimates in all. Too much error in any one of these estimates may cause the whole attack to fail. The probability that none of those 1260 guesses fails is $0.9999^{1260} \approx 0.88$. So, this attack has at least a 88% chance of success. In total, the attack requires $1260 \times 550,000 = 693$ million guesses. If we can try 1000 guesses per second, this will take about 193 hours, or a bit over a week—feasible, but only just barely.

Comment: avoiding the heavy math. If working out the math/stats stuff was too much, there was a simpler approach available: you could just simulate it. It wouldn't be too hard to code up a simple simulation that estimates the probability of success for some particular value of n, and then use binary search to find the smallest value of n that is sufficient.

Comment: better bounds. The above analysis is actually a bit conservative, and you can get by with a somewhat smaller value of n: $n = 250000$ should suffice. Thus, $1260 \times 250,000 = 315$ million guesses are enough, which will take under 4 days if you can try 1000 guesses per second.

To justify that $n = 250000$ is sufficient requires a somewhat more sophisticated analysis. If you're curious about the math, it goes something like this. Let $T_{\mathtt{a}}, T_{\mathtt{b}}, \ldots, T_{\mathtt{9}}$ denote the 62 estimates obtained for the 62 possibilities at the first character of the password. If the correct first character is, say, `q`, then $T_{\mathtt{q}} \sim \mathcal{N}(10, 1000^2/n)$ and $T_i \sim \mathcal{N}(0, 1000^2/n)$ for all other estimates. Let E denote the event that $T_{\mathtt{q}} > \max(T_{\mathtt{a}}, T_{\mathtt{b}}, \ldots, T_{\mathtt{p}}, T_{\mathtt{r}}, \ldots, T_{\mathtt{9}})$ (i.e., the event that our procedure correctly infers the first character of the secret password). What is the probability of E? Some fancy statistics shows that this probability is ≥ 0.99 if $10/\sqrt{1000^2/n} \geq 4.92$. Solving for n yields $n \approx 250000$. With this value of n, the probability of success across all 20 characters of the password is $0.99^{20} \approx 0.82$.

Question 4.

4.1. A protocol

Store a 32-bit counter C in the fob, initialized to zero. When the user presses the key, send C, T where $T = \mathrm{MAC}_K(C)$ is a 32-bit value computed using a suitable message authentication code (e.g., AES-CMAC computed on C), then increment C.

The remote has its own counter. When the remote receives a value C, T, it checks that C matches its own counter, recomputes the MAC, and checks that T matches the MAC it computed. If all of these checks succeed, it opens/closes the garage door and increments its own counter.

Modification to handle unreliable links: Suppose the remote receives C, T, and its internal counter is C^*. If $C \geq C^*$ and the tag is valid, then the remote updates its counter to C and opens/closes the garage door. (Optionally, we could impose a stricter condition, such as $C^* \leq C \leq C^* + 50$, but this isn't strictly necessary for security.) We assume that if the user presses the button but the garage door does not open, the user will try again, pushing the button again until the door does open. This modified protocol recovers from lost/garbled transmissions (as well as random button-presses of the fob while the user is out of range).

4.2. Security analysis

The best that an attacker can do is to blindly guess the message authentication tag: i.e., send C, R where R is a random 32-bit string. This has a $1/2^{32}$ chance of success on each trial. If the attacker can send 100 trials per second, it will take about 250 days (on average) before the attacker succeeds. That seems more than adequate: it's not likely that an attacker is going to wait that long.

Eavesdropping on valid C, T pairs does not help the attacker learn K or predict future MAC tags.

There *are* two potential attacks, which sadly seem unavoidable in this model:

- If the user happens to press the button on their fob while they are out at the store, and if the attacker can eavesdrop on the 64-bit value transmitted by the fob, then the attacker can try to race back to the user's home and replay that 64-bit value to get into the user's garage. There doesn't seem to be any obvious way to stop this threat without two-way communication between the fob and lifter. Fortunately, this risk seems like something we might be willing to live with.
- If the attacker can jam a transmission from the fob while simultaneously capturing the contents of the transmission (e.g., by jamming the last bit of the MAC tag and then guessing the value of that bit), then more dangerous attacks become possible. Suppose the user is leaving home to go shopping and uses the fob to close their garage door on their way out. When the user presses the button, the attacker can jam and record the 64-bit value C, R. When the user sees that the garage door did not close, the user will presumably press the button a second time. The attacker can jam and record the second transmission C', R', too, and then replay the previously recorded value C, R. The user will see the garage door close and assume all is well. Later, while the user is out shopping, the attacker can replay C', R' and get into the user's house. This attack is more serious, and unfortunately, there's no obvious way to prevent it with only one-way communication between the fob and lifter.

Other answers

Using AES as the MAC. One approach is to instantiate the MAC, by letting T be the first 32 bits of $\mathrm{AES}_K(C)$. This works fine, precisely because this construction yields a secure MAC (when the message is fixed-width and less than 128 bits long).

Encrypting the counter. Some folks suggested encrypting the counter, e.g., sending $E_K(C)$ where E is the encryption algorithm. This does not work with the revised scheme that handles unreliable links: an attacker can send a random bit-string, and it will decrypt to *something*; with high probability, the decryption will be a counter value that is larger than the current counter, thus allowing the attacker to open the garage trivially.

Other suggested adding some redundancy to the message before encrypting, e.g., encrypting $E_K(C\|C)$, where $\|$ denotes concatenation. (In other words, we concanate C with itself to get a 64-bit value, and then encrypt that.) However, this has some problems, depending upon what encryption algorithm is used. If a stream cipher is used, this is thoroughly insecure, as the attacker can flip bits to cause controlled modifications in the message. For example, suppose an eavesdropper observes a ciphertext $Y = E_K(C\|C)$ that is formed by a valid encryption. Now the attacker can form $Y' = Y \oplus \texttt{0x0000000100000001}$ (i.e., flip the low bit and bit 32). With probability about $1/2$ (namely, if C is even), Y' will decrypt to $(C+1)\|(C+1)$. Therefore, the attacker can send Y' at some point later, and the garage door lifter will accept it and open the garage door.

Some folks were more specific: they suggested sending $Y = E_K(C\|C)$, where E is a 64-bit block cipher. This works.

A pseudorandom sequence, generated by iterated encryption. One person suggested using AES to generate a sequence of pseudorandom codes, which would be transmitted over the air. In particular, when the user presses the button, the key fob generates a new code $C' = F_K(C)$ (the first 32 bits of the AES encryption of the current 32-bit value C), transmits C', and then stores C' in its 32-bit non-volatile memory. In effect, the sequence of codes emitted by the key fob will be $C_0, C_1, C_2, \ldots$ where $C_{i+1} = F_K(C)$.

This is a solid approach, but it turns out it has a non-obvious security problem. Because of a birthday-paradox effect, this sequence repeats, with a typical cycle length of around 2^{16} or so. (The birthday paradox says: given 22 people, it's likely that some pair has the same birthday; and on a planet with d days in the year, once you have about $\sqrt{d}$ people, it's likely that some pair of people will have the same birthday.) Of course, once the sequence repeats, anyone who has captured the entire sequence until then will be able to predict all future elements of the sequence, defeating the security of the scheme.

Here's why the cycle length is so short. Consider the sequence $C_0, C_1, C_2, \ldots$, as above. This looks like a sequence of random 32-bit values. By the birthday paradox, once we have $\sqrt{2^{32}} = 2^{16}$ such values, it is likely that there is some pair which are equal, say $C_i = C_j$. However, given the way that the sequence is generated, if $C_i = C_j$, it follows that $C_{i+1} = C_{j+1}$, $C_{i+2} = C_{j+2}$, etc., so the sequence has entered a cycle of length $j - i$. This means that it is likely this sequence will start to repeat within around 2^{16} iterations or so.

On the surface, this weakness isn't so terrible. If a person uses their key fob 4 times a day (twice on the way out in the morning, twice on the way back), it would take 45 years before the device has been used enough for 32-bit code values to repeat. The device will probably break, or the battery will run out, far before then, so this doesn't seem so bad.

If we posit that an attacker might plant special gear to eavesdrop on all transmission over a period of eight months (enough to observe about 2^{10} codes), then there is about a $1/2^{13}$ chance that the sequence will begin to repeat within this period. Thus, an attacker who is patient enough to wait for 8 months will have about a $1/2^{13}$ chance of breaking the scheme.

All in all, this doesn't seem like a serious problem in practice. The risk seems quite acceptable in most real-world situations. However, given that it is easy enough to design a scheme that doesn't have this shortcoming, it might be preferable to choose one that provides full 32-bit security.

Appendix: Cracking all user passwords, for the site in Q2

Suppose we want to find the password for all 9.9 million strong users of the site, without spending a single cent to crack a single CAPTCHA. How can we do it? There's a clever way!

Suppose we have a set S of candidate passwords. Here is a procedure to let us identify all strong users who are using a password from the set S:

1. Scan about all usernames and see which ones trigger a CAPTCHA.
2. For each password $p \in S$, create 50 new accounts with this password[1].
3. Re-scan all users again to see which ones trigger a CAPTCHA this second time.

This process takes $2 \times 10^7 + 50 \times |S|$ requests. Any user who did not trigger a CAPTCHA in the first scan but did on the second scan must have a password in the set S. This reveals one bit of information about the password for each user: namely, the answer to the question "is this user's password in the set S"?

We can now play a game of 20 questions to identify every user's password. The attack involves 20 phases (fittingly). Each 20-bit password can be identified with a 20-bit string. In the ith phase, we define the set S_i to be the set of all 20-bit passwords whose corresponding bit-string has a 1 in its ith bit. After 20 phases, we have enough information to deduce what every strong user's password is. The total number of requests required is $4 \times 10^8 + 50 \times 2^{19} = 425$ million requests.

You might be able to do a bit better still by optimizing the size of the set (instead of size 2^{19}, a slightly smaller size might be more efficient). However, this is already a 600,000-fold improvement, so it's good enough for me.

[1] Or, if we have previously executed this attack, change their passwords in a corresponding way.

Appendix: A better scheme

Tweaks to avoid information leakage

The scheme in the homework is well-intentioned—it seeks to increase the cost of password search, while minimizing the usability impact on most users (at least those who choose strong passwords)—but as we saw in the homework, it has some significant problems. One of the biggest problems with the scheme in the homework is that the presence or absence of a CAPTCHA leaks information about the user's password. Here is a revised scheme that reduces this problem:

- Step 1. The user enters their username and password into the page. This is immediately sent to the website.
- Step 2. The website looks up the information about this user in the database. If the username does not exist, or provided password is incorrect, or this user exists and has at least 10 consecutive failed login attempts, or at least 50 or more other users share the same password as this user, the user is shown a web page that contains a CAPTCHA and a textbox where the user is prompted to enter in the text shown in the CAPTCHA. Otherwise (if the username does exist, the provided password is correct, there are at most 9 consecutive failed logins, and at most 49 other users shared the same password as this user), the user is immediately logged in without further ado.
- Step 3. If the user was prompted to solve a CAPTCHA, the user enters their solution and it is sent to the website. The website checks whether the CAPTCHA was solved correctly. If the previous username and password were correct and the CAPTCHA was solved correctly, the user is logged in, otherwise he site displays an error message and does not log the user in.

Some additional care must be taken to ensure that the number of consecutive failed login attempts is incremented even in a situation where the user is shown a CAPTCHA but the user closes the window and doesn't try to solve it. Also, additional care must be taken to ensure that concurrent access to the password database does not allow vulnerability (the number of consecutive failed logins should be incremented in Step 2, as soon as the username is looked up, even though we don't actually know whether this attempt will be a failure or not; it can be subsequently cleared if the login is ultimately successful). Also, creation of a new account should involve solving a CAPTCHA.

This variant is somewhat better than the strawman scheme highlighted in the homework. With these improvements, a targeted (or untargeted) attack on a weak user will cost about $1, and a targeted attack on a strong user will cost about $1000. Untargeted attacks on strong users can be done for free. (The latter shortcoming can be addressed by keeping track of the ratio of total failed logins to total attempted logins, and requiring a CAPTCHA for all logins if this ratio gets too high.)

Why isn't something like this used in practice? Probably because it adds complexity, it doesn't add all that much security—and many users find CAPTCHAs annoying (understandably).

If you enjoyed this, here is some related research:

- Benny Pinkas, Tomas Sander, "Securing Passwords Against Dictionary Attacks." ACM CCS 2002.
- Paul C. van Oorschot, Stuart Stubblebine, "Countering Online Dictionary Attacks with Login Histories and Humans-in-the-Loop." ACM TISSEC, vol. 9 issue 3 (Aug. 2006), pp.235–258.

Prohibiting weak passwords

A different approach is to check the password at account creation time, and forbid use of especially weak passwords. One intriguing proposal in this vein is to forbid choosing any password that is already in use by at least (say) 50 other users. Thus, at account creation time, when a new user tries to create an account, we would check whether the user's proposed password has already been used by 50 other users; if so, we would require the user to choose a different one. The same restriction could be checked any time a user tries to change their password.

This is appealing, because it has the potential to increase the entropy of user passwords and make it harder for attackers to guess them. However, it raises a special challenge of its own: how do we store passwords in a way that allows us to determine whether a password is permissible, without compromising security? Of course, if we stored passwords in the clear, the restriction would be trivial to enforce, but storing unhashed passwords is usually considered poor practice: if the database is breached, then attackers learn all of the users' passwords (which users often re-use on other sites as well). The restriction would also be easy to enforce if we stored the passwords using an unsalted hash, but that too is considered poor practice. So how can we make this restriction compatible with good practice regarding password storage (i.e., store a salted hash of the password)?

It turns out there has been some research on this subject, and the following paper proposes a data structure to enable this:

- Stuart Schechter, Cormac Herley, and Michael Mitzenmacher, "Popularity is Everything: A New Approach to Protecting Passwords from Statistical-Guessing Attacks." HotSec 2010.