Just in Time Hashing

Benjamin Harsha
Purdue University
West Lafayette, Indiana
Email: bharsha@purdue.edu

Jeremiah Blocki
Purdue University
West Lafayette, Indiana
Email: jblocki@purdue.edu

***Abstract*—In the past few years billions of user passwords have been exposed to the threat of offline cracking attempts. Such brute-force cracking attempts are increasingly dangerous as password cracking hardware continues to improve and as users continue to select low entropy passwords. Key-stretching techniques such as hash iteration and memory hard functions can help to mitigate the risk, but increased key-stretching effort necessarily increases authentication delay so this defense is fundamentally constrained by usability concerns. We introduce Just in Time Hashing (JIT), a client side key-stretching algorithm to protect user passwords against offline brute-force cracking attempts *without* increasing delay for the user. The basic idea is to exploit idle time while the user is typing in their password to perform extra key-stretching. As soon as the user types in the first character(s) of their password our algorithm immediately begins filling memory with hash values derived from the character(s) that the user has typed thus far. We conduct a user study to guide the development of JIT e.g. by determining how much extra key-stretching could be performed during idle cycles or how many consecutive deletions JIT may need to handle. Our security analysis demonstrates that JIT can substantially increase guessing costs over traditional key-stretching algorithms with equivalent (or less) authentication delay. Specifically an empirical evaluation using existing password datasets demonstrates that JIT increases guessing costs by nearly an order of magnitude in comparison to standard key-stretching techniques with comparable delay. We provide a proof-of-concept implementation of a Just in Time Hashing algorithm by modifying Argon2.**

1. Introduction

In the past few years billions of user passwords have been exposed to the threat of offline cracking attempts. Recent high profile examples include Yahoo!, Dropbox, Lastpass, AshleyMadison, LinkedIn, AdultFriendFinder and eBay. Once a such a breach occurs the attacker can check as many password guesses as s/he wants offline. The attacker is only limited by the resources s/he invests to crack user passwords and by the underlying cost of computing the hash function.

Offline brute-force cracking attacks are increasingly dangerous as password cracking hardware continues to improve and as many users continue to select low-entropy passwords, finding it too difficult to memorize multiple strong passwords for each of their accounts. Key stretching serves as a last line of defense for users after a password breach. The basic idea is to increase guessing costs for the attacker by performing hash iteration (e.g., BCRYPT[75] or PBKDF2 [59]) or by intentionally using a password hash function that is memory hard (e.g., SCRYPT [74, 74], Argon2 [12]).

Unfortunately, there is an inherent security/usability trade-off when adopting traditional key-stretching algorithms such as PBKDF2, SCRYPT or Argon2. If the key-stretching algorithm cannot be computed quickly then we increase authentication delay for legitimate users. Since usability is a first order constraint most organizations select their key-stretching parameters to ensure that authentication delay is tolerable. Thus, usability considerations inherently upper bound the maximum number of calls that can be made to an underlying hash function (e.g., SHA256, Blake2b) as well as the amount of RAM that can be filled. For example, LastPass had been using PBKDF2-SHA256 with just 10^5 SHA256 iterations when they were breached, which means that it could potentially cost an attacker as little as $\$1$ (USD) to validate 10^{10} password guesses[1]. Blocki et al. [18] recently presented an economic argument that iterative hash functions such as BCRYPT and PBKDF2 *cannot* provide sufficient protection for user passwords without introducing an intolerable authentication delay [18]. Even with more advanced key-stretching mechanisms such as memory hard functions it is not clear whether or not it is possible to perform sufficient key-stretching to protect most (lower entropy) user passwords without *substantially increasing* authentication delay.

Contributions We introduce a novel client-side key stretching technique that we call Just In Time hashing (JIT hashing) which can substantially increase key-stretching *without* increasing authentication delay for the user. The basic idea is to exploit idle time while the user is typing in their password to perform extra key-stretching. As soon

1. While LastPass [25] claimed that "Cracking our algorithms is extremely difficult, even for the strongest computers", it has been estimated that the cost to evaluate the SHA256 hash function on customized hardware [44] is as low as $\$10^{-15}$ (USD) [23], which translates to $1 (USD) per 10^{10} computations PBKDF2-SHA256 with 10^5 SHA256 iterations.

as the user types in the first character(s) of their password our algorithm immediately begins filling memory with hash values derived from the character(s) that the user has typed thus far. The key challenge in designing such a function is that the final output must be a deterministic function of the input password, while users do not always enter their password at exactly the same speed.

We conduct a user study to investigate password typing habits and inform the design of JIT. In particular, we aimed to answer the following questions: How fast do users type when entering passwords? Is this password typing speed equivalent to regular typing speed? How often do users press backspace when entering a password? We find that for over 95% of PC users (resp. mobile users) the delay between consecutive key-strokes during password entry is over 180 ms (resp. 319 ms). While users do occasionally press backspace during password entry we find that the pattern is highly predictable (e.g., a user either erases ≤ 3 characters or erases the entire password). Both of these observations are encouraging trends for JIT since it means that there is lots of time to perform key-stretching between consecutive key-presses, and at any time we only need to be able to restore JIT state for the last three characters of the password that the user typed. In the unlikely case that there are additional deletions the algorithm can restart from the first state and reprocess, resulting in a delay for the user.

Several of our findings from the user study may be of independent interest. For example, we find that password typing speed is only weakly correlated with regular typing speed, which may have implications for the design and evaluation of implicit authentication mechanisms based on keystroke dynamics during password entry. We conjecture that the differences in typing times are due to muscle memory as well as the use of less common characters and/or character sequences in passwords.

We analyze the security of a JIT password hashing algorithm using graph pebbling arguments. On the negative side, our analysis demonstrates that JIT password hashing with hash iteration as the underlying key-stretching mechanism provides minimal benefits over traditional key-stretching algorithms based on hash iteration (e.g., BCRYPT, PBKDF2). On the positive side, we find that JIT hashing can be combined with memory hard functions to *dramatically* increase guessing costs for an offline attacker. For example, we find that if users select passwords similarly to those found in the Rockyou data set [36] then an offline attacker will need to perform more than 6 times as much work to crack a password protected with JIT hashing. Similarly, it will be more than 9 times as much work for an attacker to check every password in a standard password cracking dictionary [37]. We remark that these advantages are based on a pessimistic assumption that the adversary has an unbounded amount of parallelism available. If the adversary is sequential then he must perform 13.3 (resp. 12.5) times as much work to crack a RockYou (resp. cracking dictionary) password protected with JIT.

Finally, we provide a proof-of-concept implementation of JIT highlighting key design decisions made along the way. Our implementation is based on a modification of Argon2 [12], winner of the password hashing competition [3]. The execution of JIT can remain hidden from the user to provide the benefit of increased key-stretching without affecting the user's authentication experience.

2. Related Work

Password Cracking The issue of offline password cracking is far from new, and has been studied for many years [70]. Password cracking tools have been created and improved through the exploration of new strategies and techniques such as probabalistic password models [72], probabalistic context free grammars [91, 60, 90], Markov chain models [31, 32, 65, 89], and neural networks [68]. For sentence-based passwords attackers may turn to public or online resources as a source of password possibilities, or they may use training data from previous large breaches like Rockyou [93, 92]. Public and open-source password cracking tools like John the Ripper are easily available online, and can be modified or provided with specific strategies to attempt to crack lists of passwords offline [37].

Improving Password Strength It has proven difficult to convince or force users to choose stronger passwords [28, 62, 82, 83, 55, 80], and methods that do work have usability issues [1]. Strategies to convince users to select stronger passwords have included providing continuous feedback (e.g. password strength meters [63, 88, 30]) and providing instructions or enforcing composition policies [93, 21, 28, 62, 82, 83, 80, 81]. However it has been shown these these methods also suffer from usability issues [55, 84, 48, 1] and in some cases can even lead to users selecting weaker passwords [20, 62]. Password strength meters have also been shown to provide inconsistent feedback to users, often failing to persuade them to select a stronger password [88, 30].

Key Stretching Key stretching, the process of artificially increasing the difficulty of computation of a hash function, is designed to protect low entropy passwords and secrets from offline cracking attempts. By making each guess more expensive, it becomes more difficult for an adversary to crack each password as each attempt costs them more. The method was proposed by Morris in 1979 [70] who used it in the context of password security. Key stretching was originally performed by repeated calculations of the hashing function, i.e. rather than storing the hash of the password and salt the result is first run through the hash function many more times. This method is still used by the functions BCRYPT [75] and PBKDF2 [59]. However these functions require small amounts of memory, and the hash functions that these are based on can now be computed very quickly for a reasonable cost using hardware such as the Antminer [44], which can computer trillions of base functions per second e.g., Bonneau and Schechter estimated that, even if we ran PBKDF2-SHA256 with $\tau = 10^7$ hash iterations (1 second), a human would need to memorize a 56-bit secret to provide adequate security against an offline attacker [23]. Additional key stretching methods have been proposed to

introduce asymmetric costs by keeping part of the salt secret and requiring that it be guessed iteratively [17, 67].

Memory Hard Functions Memory Hard Functions (MHFs) were introduced in 2009 by Percival [73]. The key insight behind MHFs is that, while computation power is asymmetric between users and adversaries, the cost of using memory is more equitable. Ideally a MHF should have τ^2 area-time complexity, where τ is a parameter setting the amount of time and memory the function should use. Functions like BCRYPT or PBKDF2 would instead have an area-time complexity of τ as they use a constant amount of space. Data-dependent MHFs are MHFs which have a data access pattern that depends on the input. Examples include Argon2d [12] and SCRYPT [73]. Because data-dependent MHFs have a data access pattern that depends on the input they are potentially vulnerable to side-channel attacks that determine memory access patterns [11, 49]. Data-independent MHFs are a particular class of MHF that are designed to help prevent side-channel attacks, and have a data access pattern independent from the input. It has recently been shown that SCRYPT is optimally memory-hard [9]. Blocki et al. [18] recently presented an economic argument that iterative hash functions such as BCRYPT and PBKDF2 *cannot* provide sufficient protection for user passwords and argued that NIST standards should be updated to *require* the use of memory hard functions [18].

Other defenses against offline attacks It is possible to distribute the storage and computation of password hashes across multiple servers [24, 27, 45]. Juels and Rivest [58] proposed storing the hashes of fake passwords (honeywords) and using a second auxiliary server to detect authentication attempts that come from cracking the fake passwords. These methods require the purchase of additional equipment, which may prevent those with more limited financial resources from employing them. A second area of research has investigated the use of hard artificial intelligence problems that require a human to solve [29, 16, 19]. This would require an offline attacker to employ human oversight throughout the process by having them solve a puzzle (e.g. a CAPTCHA [29, 19]). In comparison, data independent MHFs have a set memory access pattern that does not involve the input. These data independent MHFs are typically the recommended type to use for password hashing [3, 12]. Several of the most prominent iMHFs from the literature are: (1) Argon2i [13, 14], the winner of the password hashing competition [3], (2) Catena [49], a PHC contestant which received special recognition from the PHC judges and (3) Balloon Hashing [22]. Several attacks have been found for Catena [13, 7, 4] and for Argon2i and Balloon Hashing [4, 5]. Constructions for iMHFs with cumulative complexity $\Omega(n^2/\log n)$ have been shown [6] using a concept called depth-robust graphs [43]. This is asymptotically the best possible result given the attack shown by Alwen and Blocki [4] showing that any "natural" iMHF has cumulative complexity $O(n^2 \log\log n/\log n)$, but the construction remains theoretical at this time.

Password Managers The number of passwords people must keep track of has increased greatly over the years, and will continue to increase in the future. Password managers such as PwdHash, Password Multiplier and LastPass [77, 50] offer to relieve much of this cognitive burden [52] by allowing a user to derive multiple password from a single master password. However, password managers have been criticized since the compromise of a master password (e.g., via key-logging, phishing or brute-force guessing) can be fatal [64]. JIT password hashing can be used to mitigate the risk of brute-force guessing in the event of a breach such as the one at LastPass.

Disk Encryption Users have plenty of options when it comes to encrypting the storage on their machines [15, 38, 54], and work has been done to design cryptographic systems completely dedicated to this task [47]. Each of these options in some way encrypts the user's disk(s) (or only a portion, such as a home folder) so that the data can only be accessed with a provided password. As with password managers, it is possible to run an offline attack against disk encryption, such as in the case of a stolen mobile phone.

3. Just in Time Hashing

Our proposed solution to the problem of increasing key stretching without inconveniencing users is Just in Time Hashing, a method that allows for extended key stretching without a user being aware that it is being done. Where a user may have noticed that authentication was taking several seconds before, using JIT hashing they could perform the same amount of key stretching and barely notice any delay once they have finished entering their password.

Formally we define a k-limited Just In Time hashing function as a streaming algorithm $\mathcal{A}$, with a random oracle H, and an initial state q_0 that makes at most k sequential calls to the random oracle for each state update. As each character c_i of the input enters the algorithms the state is updated, up until a special terminating character \$. The input must be of the form $C \in (\Sigma \setminus \$)^* |\$$, where once the terminating character is read the output τ based on the final state is returned. On each update a just in time hashing algorithm returns one of two types of outputs. If the character was not the terminating character \$ then the function returns a new state q_i from the set of possible states Q. If it is \$ then it returns an output t from the set of possible outputs T. The just in time algorithm transitions between states according to the following function:

$$\mathcal{A}^H(q_{i=1}, c_i) = \begin{cases} q_i \in Q & c_i \neq \$ \\ \tau \in T & c_i = \$ \end{cases}$$

We use $\mathcal{A}^H(C)$ to denote the final output given a sequence of the form $C \in (\Sigma \setminus \$)^* |\$$.

The Backspace Challenge: We allow the character set Σ to include a special character $\blacktriangleleft$ (backspace). We require that a $\mathcal{A}^H$ is consistent meaning that we should get the same output when the user types $1, 2, f, g, \blacktriangleleft, \blacktriangleleft, 3$ that we would if the user had typed the sequence $1, 2, 3, \$$ the output τ. Formally, for all input sequence $C \in (\Sigma \setminus \$)^* |\$$ we require that $\mathcal{A}^H(C) = \mathcal{A}^H(\mathbf{Prune}(C))$, where $\mathbf{Prune}(C) \in$

$(\Sigma \setminus \{\$, \blacktriangleleft\})^* |\$$ is the character sequence we obtain after applying each backspace operation $\blacktriangleleft$.

A naive way to handle backspaces would be to revert to state q_0 and repeat the entire computation, but this approach would result in noticeably large authentication delays for the user. A second way to handle backspaces would be to store all previous states so that we can quickly revert to a prior state. The key challenge is that states can quickly become very large (e.g., 1GB) because our instantiation of $\mathcal{A}^H$ is memory hard.

We can relax the requirement that $\mathcal{A}^H$ *always* updates after at most k sequential calls to the random oracle to say that $\mathcal{A}^H$ *always* updates after at most k sequential calls for β-good input sequences. Intuitively, a sequence is β good if it does not contain too many backspaces $\blacktriangleleft$ within a short interval so that once we are in state $q_{i+\beta}$ we will never be asked to revert to a state q_j for $j < i$. We allow for one exception: if the user wipes out the entire password then the sequence is *not* β-bad because it is easy to revert to state q_0.

Definition 1. *We say that a sequence $C = c_1, \ldots, c_t, \$ \in (\Sigma \setminus \$)^* |\$$ is β-bad if we can find indices $i \leq j \leq t$ such that*

$$\beta < \sum_{i=1}^{j} \left(\mathbb{1}_{c_i = \blacktriangleleft} - \mathbb{1}_{c_j \neq \blacktriangleleft} \right) ,$$

and $\mathbf{Prune}(c_1, \ldots, c_j) \neq \emptyset$. If no such indices exist then we say that the sequence is β-good. We say that a sequence is β-bad if $\mathcal{A}^H$ is a β-tolerant k-limited Just In Time hashing function if for all β-good sequences $C \in (\Sigma \setminus \$)^* |\$$ the algorithm $\mathcal{A}^H(C)$ never requires more than k-sequential calls to the random oracle between updates.

Examples: Say that someone is using a β-tolerant JIT hashing algorithm with $\beta = 3$, and that this user unwisely decided to use the password "password". If the user typed "$pasS \blacktriangleleft\blacktriangleleft ssw0 \blacktriangleleft ord$" then the sequence of keystrokes is β-good, but if the user typed "$passwo \blacktriangleleft\blacktriangleleft\blacktriangleleft s \blacktriangleleft\blacktriangleleft ssword$" then the sequence of keystrokes is β-bad. We stress that a β-tolerant JIT hashing algorithm will allow the user to authenticate in *either* case, though the user may experience some delay in the second instance. In particular, a β-tolerant JIT hashing algorithm is allowed to introduce authentication delay whenever the sequence of keystrokes is β-bad since the algorithm may need to revert to state q_0 even though the user has only erased part of the password. Similarly, if a user types his password faster than expected then the characters will be stored in a queue to be processed in the future. The user does not lose protection from fast typing, but instead notices a delay depending on how fast they typed since the JIT hashing algorithm may not have enough time to catch up before the user finishes typing the password. In Section 4 we present the results from our user study showing that 95% of users take more than 180ms per character typed. We remark that the JIT parameters could be adjusted when we know the user is a fast typist.

Applications: In this paper we focus on the context of password hashing and key stretching, specifically using the time users spend typing in their passwords. However, in the broadest sense JIT is a method to hide computation within idle cycles by streaming input instead of working in batches, and thus potential applications are not necessarily limited to password hash computation. For example, the JIT technique could be used to generate proofs of work for email. As the user types an e-mail the JIT algorithm could continually update the proof of work for the current email message. Using this approach could help deter spammers by making it prohibitively expensive to generate the proof of work for each message. One intriguing challenge would be to develop a JIT proof of work with a more efficient verification algorithm in case the receiver does not have time to regenerate the entire JIT proof. Another possible application domain for authentication would be to take advantage of the longer delays induced by two-factor authentication.

Salting: In the context of password hashing it is vital to have a method of introducing salt. We remark that a JIT hashing algorithm can incorporate salt by including it in the calculation of the initial state q_0 and then storing the salt value. In many contexts (e.g., disk encryption, password vaults) the salt value could simply be stored on the client machine. Similarly, a stateful password manager could store salt values on the client or on the cloud. A stateless password manager (e.g., PwdHash) could derive a salt value from public parameters (e.g., username, domain).

4. Usability Analysis

In the last section we introduced the notion of a β-tolerant k-limited JIT scheme which updates the state at most k times given any β-good input sequence. Before instantiating any JIT scheme it is crucial to understand how people type passwords in practice. In particular, to avoid delays during authentication we need to tune k so that the time to update the state is less than the expected delay between consecutive keystrokes. Thus, the parameter k will depend on the user's password typing speed. Furthermore, we also need to ensure that JIT is β-tolerant for a sufficiently large value of β to ensure that the input sequence we receive when a user types their password is β-good.

In this section we aim to answer the following questions. How quickly do users type their passwords? To what extent is password typing speed correlated with regular typing speed? What fraction of login attempts are β-good for $\beta = 1, 2, 3$? And to what extend does password typing speed Change over time?

To answer these questions we first analyze two publicly available datasets [34, 61]. While we can extract useful insights from both datasets, there are significant methodological limitations when we attempt to use these datasets to answer each of our questions e.g., users in the passwords typo dataset [34] were not actually typing their own passwords. To address these limitations we also conduct our own user study in which we asked users to type in their real passwords so that we could measure password typing speed.

4.1. Password Typos dataset

Chatterjee et al [34] ran a user study designed to "...identify common typos and trends". To do this they used Amazon Mechanical Turk (MTurk), an online platform that allows users to post or complete small tasks, called Human Intelligence Tasks (HIT), for compensation. In their study they asked users to type in several passwords sampled from the Rockyou password leak, a 2009 leak that exposed users passwords in plaintext [36]. For each password that users typed a JSON entry was created that contained a list of the user's key presses as well as the corresponding timestamps.

Analysis of Deletion Patterns: We analyzed deletion patterns in the password typos dataset and found that if a user starts deleting a password, there is a 74.9% chance that they will clear the entire field. For implementation of JIT hashing this is convenient, because it means that with most deletions the user starts from scratch and we don't have to worry about reverting. In addition, of those that delete some, but not all, of the password, 89.5% will delete fewer than 3 characters and 94.7% will delete fewer than 5. This suggests that it would be sufficient to ensure that the JIT algorithm is β-tolerant with $\beta \in \{3, 5\}$.

4.2. Keystroke Dynamics dataset

Killoughy and Maxion also performed a user study that recorded user's password typing habits with the goal of creating a dataset that could be used to test authentication methods based on user's individual typing trends [61]. In their study they recruited 51 subjects and had them type the same password 400 times. For each recording the researchers recorded key press timing data. All participants used the same preset password (.tie5Roanl).

Impact of Muscle Memory: We used the keystroke dynamics dataset to estimate the impact of muscle memory on password typing speeds. The dataset is particularly well suited for this estimation because each user typed the password 400 times. We found that the time spend typing passwords on their last entry was, on average, 0.4877% of the time spend typing it the first time ($\sigma = 0.1611$). This was strong evidence in favor of our hypothesis that over time people will tend to speed up their password typing speed over time as they gain more practice.

4.3. Limitations

Both of these studies provided valuable data for the analysis of just in time hashing, but the data was not without limitations for our purposes. In the typos data set users were given multiple passwords sampled from the Rockyou password list. While these passwords are sampled from a large database of user-chosen passwords, they are not necessarily the style of password that particular user may pick. In addition they typed these passwords only one time - while throughout their daily lives they may be typing their own chosen passwords multiple times, possibly improving their speeds along the way. Thus, it is possible that the typing speeds and deletion patterns observed during the study are not representative of real world password typing speeds/deletion patterns.

Similarly the users in the keystroke dynamics study all typed the same randomly-generated password. They were also typing the passwords in large batches during several sessions, which may not match everyday password typing habits.

These limitations restrict the ecological validity of any conclusions that we draw from these datasets, thus we perform our own user study to further investigate how JIT hashing could best be implemented.

4.4. Study Design

To address the previous limitations we designed a user study to investigate user's typing speed and correction habits on their real passwords. Briefly, in the study users were asked to type their password, type a paragraph and then type their password again. The instructions emphasized that we wanted users to type in their actual password and reassured users that we only collected statistics on typing speeds and would never receive their actual password. Previous work has found that conducting password studies poses many challenges, and that care must be taken when analyzing the results [46]. Thus we strived to ensure that we were learning valuable information while taking care to design the study properly.

To give an idea of how much key stretching could be performed with JIT hashing the specific data that is needed is how quickly people type their passwords in practice. While previous work did have people type in passwords, they were either typing a pre-defined password list [61] or randomly generated passwords [34]. To give an idea of how much time we have for key stretching in practice we need to know how long users spend typing per character on their own passwords. To obtain this information we performed a user study in which we collected the time it took for people to type in their real passwords.

The IRB-approved user study was conducted using MTurk and a website we hosted locally. We asked participants to take part in a quick 5 minute survey that was investigating password typing habits, and were recruited with the following advertisement on MTurk:

> **Participate in a study investigating password typing speeds and muscle memory**: For this HIT you will be asked to help investigate how long users take to type in their passwords. This involves several typing tasks, including filling in password boxes and typing a full paragraph. At no point in this study are the passwords you type transmitted or stored by the researchers.

We used Mechanical Turk due to its ability to recruit a larger amount of subjects than we would normally be able

to recruit locally. It is known that Mechanical Turk tends to recruit users who are younger, more educated, and more technically proficient than the general population [26]. In addition Mechanical Turk users tend to be more diverse than populations that would be recruited on a standard campus [56]. Mechanical Turk is not without its flaws, and has been criticized by Adar. Though he criticizes it, he states that he has no issues with its use if the study involves understanding humans and human interactions[2]. We believe that this study falls within this category, and that it has been used in an appropriate and useful manner. In addition, to ensure that we had reliable results we restricted our survey to MTurk users with an approval rating of 90% or higher. We recruited 400 participants, each of whom were paid $0.50 for an estimated 5 minute survey.

To participate, users were linked to a website where they were given instructions and consent information about this study. Participants were told that the study was investigating password typing habits, and that they would be asked to answer a few quick questions and type a short paragraph. They were also told that the study was not expected to take any more than 5 minutes to complete. To begin, we asked users what platform they were using, either desktop/laptop, a mobile device (phone or tablet), or something else. We only accepted those who selected the first two options.

Once users had consented to the study and entered in what platform they were using they were taken to the first page which contained a password input field and the following **bolded** instructions:

> In the form below please enter a commonly used password. **It is important to type one of your own commonly used passwords, as we are studying how quickly people normally type their passwords!**...

Subjects were also reassured that at no point do we transmit their actual password, and that we only collect the timing data from their input. Particularly concerned users had the ability to verify this by checking the code on the site. Once they had finished typing in the password, they were taken to a second timing page and asked to type in a short paragraph to gauge their typing speed. As before, the instructions told the participants to type the paragraph and hit enter once they were done. At the end they were taken to a page similar to the first password entry page, where they were asked to type in the same commonly used password as before in the same manner.

Previous research has found that when users in studies are asked to use either their real password or very similar ones, some participants will behave strangely or even in an antagonistic way [46]. Thus, it is possible that some users typed in a fake password instead of using one of their real ones as we repeatedly requested. To address these concerns we asked users to self-report whether or not they had typed in their real passwords. In particular, once the final timing data was collected the participants were taken to a page with the following statement:

> It is important for us to know if you used a commonly written password. If what you typed was not a commonly used password, please select the appropriate response. We appreciate that you may have done this in the interest of your own security, but we need to ensure the collected data is usable. Even if you select "I did NOT type a commonly used password on the previous page" you will be paid for completing the study.
> ☐ I typed a commonly used password on the previous page
> ☐ I did NOT type a commonly used password on the previous page

We felt it was important to clearly state that the users are paid whether or not they self-reported typing one of their real passwords to eliminate incentive to lie about the last question.

As a final step users were asked to optionally provide some demographic information, including age, ethnicity, education, and gender. After this page the users were directed to a page containing a code that they could enter on Mechanical Turk to claim their payment.

The main data that were collected were per character password and standard typing speeds. To record this, an action was triggered when the first character was entered into the provided field that recorded the starting time. If, at any point, a user cleared out the field, the timer was reset. Once the user hit enter or clicked the continue button the timer stopped, calculated the total time over the number of characters, and transmitted the per character speed over the encrypted connection. In addition to timing data we also collected data on how many consecutive backspaces occurred in the worst case, as well as how many times users cleared the entire field. The results were stored in a database at our institution for analysis.

Ethical Considerations As this study involved the use of human subjects and sensitive information great care was taken to ensure this study was designed and run in a way that would offer the most benefit with the least risk to users. A large number of security precautions to prevent password theft were put in place, described in the following paragraph. Another potential concern is that an attacker might conduct a copycat "study" to phish for user passwords. To minimize the risk of such copycat studies we provided full contact information for the PI and for the IRB board at Purdue. The study site was also hosted on an https server using a domain name affiliated with Purdue University. Finally, we note that in user studies in which users are asked to create a new password that many users simply type in one of their own passwords [62]. Thus the risk of phishing 'studies' is present whether or not the user is explicitly asked to type in their own password. The study was submitted to and approved by the IRB board at Purdue before the study was conducted.

4.5. Security precautions

We took several precautions to ensure that at no point would a user's password be revealed, either to us or even to someone monitoring the user's network traffic. The first

step for ensuring security is to make sure that all data involving the user's password was computed locally on the user's machine. To accomplish this we wrote Javascript code to monitor the time between key presses and watched for the enter key to be pressed when the user was done typing. Once they finished our code transmitted only the time typed per character, the number of field clears, and the maximum number of consecutive backspaces to the server. At no point was the password or its length transmitted, only the time it takes to type each character, which is the relevant information for tuning JIT parameters.

As a second layer of protection we required that all connections to our server be encrypted. Thus, even in the event that secure data was sent it would not be retrievable by observing network traffic.

As a final precaution all of the code for the survey was subjected to independent third-party analysis. The third party used the automated tool Checkmarx to test for security vulnerabilities. The analysis found no vulnerabilities that would expose any sensitive user data.

4.6. Results

Of the 400 MTurk participants recruited 335 self-reported that they had completed the study and used one of their own passwords. In our analysis we dropped data from the 65 users who self-reported not using their own password in the study. Additionally, we discarded data from the 7 PC users who left the password field blank (all mobile users filled in the password field). Of the remaining 335 users, 313 reported using a desktop or a laptop while 22 reported using some mobile device (phone, tablet, etc...).

Several users had exceptionally long typing times (2000ms+ per character typed). In each of these cases either the password per-character speed or the typing per-character speed were unusually large, never both. These values are excluded from the charts and tables in this section as they make it difficult to visualize the more common results. Statistical analysis was performed using the statistical package R [86], where one of the first things we looked at is whether or not the time taken to type each user's individual password had anything to do with their individual typing speed.

We split the analysis into mobile and non-mobile users. One of the first things to notice, especially in Figure 1, is that there isn't a very strong correlation ($R^2_{adj} = 0.1289, p < 0.001$) between observed typing time and password typing time. The non-mobile data showed the same weak correlation, meaning that typing time is not a particularly good measure of how quickly someone might type their passwords.

We noted that we found similar deletion habits to those from Chatterjee et al's data [34]. In particular, we observe that people rarely have more than 3 consecutive deletions without deleting the entire password, with only about 1% of participants doing so[2]. In total we saw that, of the 612 entries from 306 users (two entries per user) who self-reported using their real password and were on a non-mobile device, only 4 showed more than 3 consecutive deletions. Thus, we maintain that a large majority of users will not run into more than a small number of deletions. In particular, it should be sufficient to set $\beta = 3$ when implementing a β-tolerant JIT scheme.

Of particular interest to JIT hashing are some of the typing time percentiles, marked on Figure 1. For just in time hashing it is valuable to know how long we can safely run the key stretching per character so that users will not notice any odd delays or slowdowns from the system. From the provided data we can see that it should suffice to stop after 183 or 213 milliseconds of computation for a non-mobile user so that 95% and 90%, respectively, of users will notice no delay from the key stretching. With mobile data it does seem like we may have a bit more time to run key stretching due to overall slower password typing speeds, however due to the small mobile sample size this will likely require further study to come up with statistically significant claims. If it does turn out that we have more time on mobile devices this may be a benefit, as we can make up for some of the slower processing speeds with additional computation time.

Using Regular Typing Speeds to Select Cutoffs We further investigated the possibility of predicting typing speeds by categorizing users into broader categories. We began with the non-mobile users and then split this group into those with speeds under 250ms/ch, those between 250 and 500, and those taking more than 500ms per character typed. Each of these groups was further split into a training set and a testing set. Each training set contained 70% of the time group's results, with the remainder reserved for testing. Using the training data split by typing speed, we determined each group's 5th and 10th password typing speed percentiles. We then looked at the cutoff line for the percentiles and determined what proportion of the training data fell below the cutoff line, giving an idea of how accurate the predictions came out. The results are shown in Table 1, which shows the percentile cutoffs from the training data and the percentage of the testing data that fell below each cutoff. We observe that we obtain reasonable predictions of the testing percentiles, with the exception of the final timing category. This category turned out to be more difficult to predict due to the outliers contained in the set.

The practical benefit of being able to make some predictions based on larger standard typing time categories is the potential optimization of JIT hashing times per character by giving a user a typing speed test. If their typing speed is known, and if they turn out to be in one of the slower groups, our data suggests that it is possible to run just in time hashing for more time per character for that individual. While possible, usability may be an issue with this optimization. Users may become impatient with a required typing test before registering, and those with faster typing times can argue that they are being cheated out of additional key-stretching due to their typing speed. That is, they may prefer the extra security they would have gained by increasing the

2. The four users that did have larger numbers of consecutive deletions without wiping out the entire password had very large numbers (26,19,22 and 27) for the maximum number of consecutive deletions.

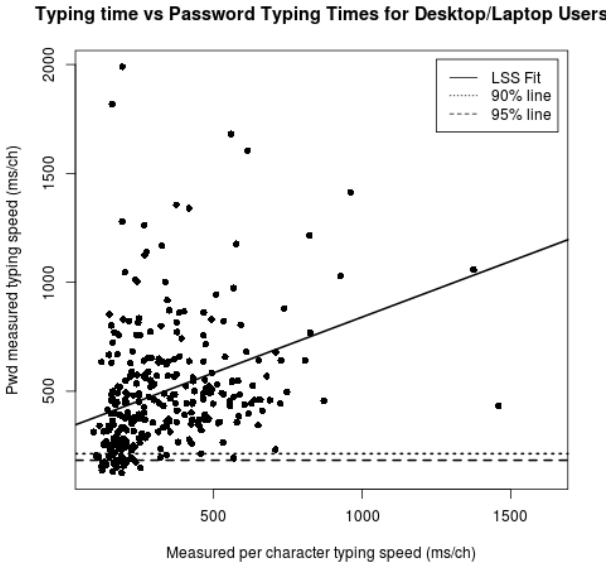

Figure 1. Results for non-mobile users

Typing speed range	Train 5'th (ms)	% ≤ Pred	Train 10th(ms)	% ≤ Pred
$0 \leq x < 250$	170.01	0.078	180.025	0.100
$250 \leq x < 500$	264.11	0.059	299.18	0.118
$500 \leq x$	310.12	0.083	356.400	0.167

TABLE 1. SUMMARIZED RESULTS FROM SUBSAMPLING TESTS

per-character running time. The benefit of this method over a universal set time would be that those who would have experienced some annoying delay when typing would no longer see this delay.

5. Security analysis

In this section we investigate the performance of JIT hashing with and without memory hardness. On the negative side, our analysis demonstrates that the benefits of JIT hashing without memory hardness are marginal. In particular, if an iterated hash function is run in JIT mode we will show that the adversary has a fairly efficient method to guess passwords i.e., the cost of checking every guess in a dictionary with JIT is only marginally higher than the costs the attacker would incur if a comparable iterated hash function (inducing the same authentication delay) were used. Given this attack we warn that JIT hashing does not offer its full benefits without memory hardness. In the second section we will examine JIT hashing when implemented with memory hardness. In this case we use a pebbling argument to demonstrate that JIT hashing substantially increases guessing costs for an attacker.

5.1. Adversarial model

For analysis we assume the adversary is:

1) **Offline:** The adversary has obtained a hash and salt of a password, and can verify password guesses offline.
2) **Informed:** The adversary is familiar with the specific implementation of JIT hashing being used, and knows exactly how the hash value they have was obtained. The adversary is also assumed to posses a reasonably large password dictionary containing all of the most likely user password choices. The adversary is interested in cracking the password using the minimum possible number of guesses, and will use their knowledge to optimize their strategy to crack the password with the minimal possible amount of work.
3) **Rational:** An attacker is willing to continue cracking as long as marginal guessing benefits (i.e., value of a cracked password times the probability that the next guess is correct) exceed marginal guessing costs. If expected guessing costs exceed expected reward then the attacker will quit his attack. In particular, it is possible to discourage the attacker by increasing the cost to validate each password guess.
   a) **Infinitely Parallel, Memory Unbounded:** The adversary has no time limit to their computation, although there is an opportunity cost to allocating additional resources (memory/processing cores) to password cracking. Since the adversary is rational the attacker may stop attacking if the opportunity costs exceeds the expected reward. This model may be overly pessimistic since a real world attacker does not have infinite memory.
   b) **Sequential, Memory Unbounded:** The attacker has limited memory and each memory chip is associated with a single processor. While this model may be overly optimistic we note that in practice it is difficult to route messages from a single shared memory chip to many different cores.

5.2. Password Model

We consider three types of password distributions:

1) **Empirical:** The user selects a password from the RockYou dictionary. Probabilities are weighted by their empirical frequency (e.g., in the RockYou dictionary contains passwords from $N = 32.6$ million user accounts and 291×10^3 users in the dataset selected '123456' so the probability our user selects the password '123456' is $\Pr[\text{`}123456'] \approx 0.009$).
2) **XKCD (Random Words):** The user selects several words uniformly at random from a dictionary of English words. In particular, we use Google's list of the 10,000 most common English words in our analysis.
3) **Cracking Dictionary:** Passwords are taken from a cracking dictionary created by Openwall and intended for use with John the Ripper [37]. This is designed to mimic how a criminal may perform an online attack against a standard password.

An additional analysis of uniform passwords is available in the full version of this paper. Briefly, our analysis shows that when we are protecting uniformly random passwords JIT offers no advantage against a parallel memory bounded

attacker. However, JIT can increase costs for a sequential attacker by an order of magnitude. We defer the analysis to the full version of the paper because real users tend not to pick uniformly random passwords.

5.3. JIT without memory hardness

In this section we analyze the performance of JIT without memory hardness. We will present an executive summary of our results and refer and interested reader to the full version of this paper for more details. To begin, assume that we have an unbounded adversary attempting to run through their list of possible passwords as quickly as they can, and that JIT hashing is being run using a hash function H and key stretching is performed through hash iteration. Note that under the JIT model the adversary can think of the list of possible passwords as forming a trie of possibilities. To explore all passwords the adversary simply needs to calculate the entire trie, ensuring that they visit every node at least once.

If we define the cost to traverse each edge to be $W = \mathbb{C}(\mathbb{H}) = 1$ and define our alphabet as Σ and assume that no password is of length $1 \leq$ length $\leq \ell$ then the adversaries total work to check all password guesses is given by the number of nodes in the trie. By comparison if we had not used JIT and instead simply hashed the final password with $\mathbb{H}$ then the total work is given by the total number of passwords in the dataset that the attacker wants to check (e.g., the number of *leaf* nodes in the trie). The advantage of JIT is given by the ratio: $\#nodes/\#leaves$.

Empirical Distribution: For each value of T we computed a trie from the T most popular passwords in the RockYou list. Figure 2 plots the ratio $\#nodes/\#leaves$ for each point T. A typical value of the ratio is about 1.5. Thus, JIT slightly increases the work that an attacker must to to check the T most popular passwords.

XKCD (Random Words): We computed the ratio $\#nodes/\#leaves$ for the trie for the dictionary containing all i-tuples of the $10,000$ English words for each $i \leq 5$. The typical value for the ratio (i.e. at 4 words) is 2.417 meaning that JIT yields a modest increase in the work that an attacker must to to crack an XKCD style password.

Cracking Dictionary: The cracking dictionary was analyzed in the same manner as Rockyou, with the results shown in Figure 2. We note that the ratio is slightly higher, closer to 2.4, for this dictionary.

5.4. Just in Time Memory Hard Hashing

In the previous section we saw that an adversary using a trie attack can obtain near optimal running times when attacking a JIT hashed function that key stretching via iterated hashing. The main vulnerability that allows this to happen is the fact that standard hashing results are easily stored in memory and referenced, producing a trie that can easily be stored entirely in memory on any standard home

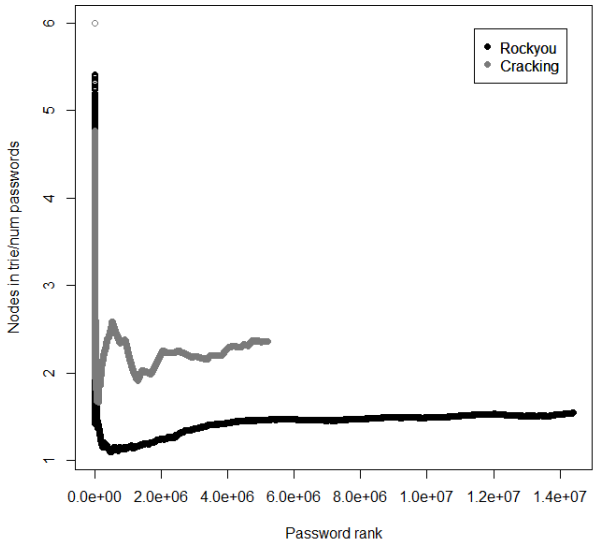

Figure 2. Average number of nodes added to pwd trie over time

computer. However the results are much different when using a memory hard hashing function rather than using a standard iterated hashing function.

Memory hard functions are functions that require the user to dedicate a set amount of memory to computing a function in addition to performing the computation cost, or at the very least to suffer an extreme runtime penalty if they do not want to store the function in memory. Several of these memory hard functions, such as SCRYPT, have a tunable memory use parameter that allows the user to specify how much memory they would like the hashing function to take up. We note that if a user were to simply select a value that a reasonable computer would have, such as using 1GB of memory, we already require more memory to calculate a single hash than it took to store the entire trie under the iterated hashing scheme. The question arises as to just how much would it cost to run an offline attack against a JIT password hash that used a memory hard function. To accomplish this we first introduce the notion of graph representations of MHFs, graph pebbling and cumulative complexity.

The Black Pebbling Game. One of the main techniques for analyzing iMHF is to use pebbling games played on graphs. First introduced by Hewitt and Paterson [53] and Cook [35] the (sequential) black pebbling game (and its relatives) have been used to great effect in theoretical computer science. Some early applications include space/time tradeoffs for various computational tasks such as matrix multiplication [87], the FFT [78, 87], integer multiplication [85] and solving linear recursions [33, 79]. More recently, pebbling games have been used for various cryptographic applications including proofs of space [42, 76], proofs of work [39, 66], leakage-resilient cryptography [40], garbled circuits [51], one-time computable functions [41], adaptive

security proofs [51, 57] and memory-hard functions [49, 7, 4, 8].

The black pebbling game is played on a fixed DAG in rounds. At each round, denoted $P_i \subseteq V$, certain vertices are considered to be pebbled if they are contained in P_i. The goal of the game is to pebble all sink nodes of G (not necessarily simultaneously). In the first round we set $P_0 = \emptyset$. P_i is derived from the previous configuration P_{i-1} according to two simple rules. (1) A node v may be pebbled (added to P_i) if, in the previous configuration all of its parents were pebbled, i.e., parents$(v) \subseteq P_{i-1}$. In this parallel pebbling game we allow for any number of pebbles to be placed in a single round, while in the sequential version only one pebble may be placed at each round. (2) A pebble can always be removed from P_i. A sequence of configurations $P = (P_0, P_1, \ldots)$ is a pebbling of G if it adheres to these rules and each sink node of G is contained in at least one configuration.

Cumulative Complexity When running a pebbling game on a DAG we have the concept of cumulative complexity. In this model we are thinking of each pebble as some unit of memory, and each round as a unit of time. Cumulative complexity of a DAG G, $CC(G)$, and a pebbling sequence $P = P_0, P_1, \ldots, P_k$ is a measure of the space time complexity of the pebbling. To model memory usage in a JIT MHF we slightly modify the standard cost definition to match the realities of MHF implementations. In a MHF placing a pebble involves filling an array of size m over m steps. During the placement we use $1+2+\ldots = \frac{m^2}{2}$ units of memory over the entire placement process. When keeping a pebble on the graph, we keep m units of memory filled for the m steps it will take to fill up a new pebble, meaning each pebble costs m^2 to keep around for an additional round. Note that in the original definition of cumulative complexity the graph being used represents nodes as blocks of memory in a memory hard function and the edges as the dependencies required to fill that block. For JIT hashing the graph represents a different point of view. Each node does not represent a single memory block, but rather represents the state that the JIT hashing function is in once a sequence of characters has been entered. For each character entry the JIT function represents the filling of m blocks of memory rather than a single block in the original definition. It is this distinction that leads to these modifications in the definition of cumulative complexity. Essentially, rather than a single node representing a single operation, it represents the sequence of operations required to update the JIT state from the previous state to a new state.

With these definitions in place we can now define the cumulative complexity of a sequence of pebbling moves P:

$$CC(P) = \sum_{i=0}^{k} \left(\left(\frac{m^2}{2} |P_i \setminus P_{i-1}| \right) + \left(m^2 \left| P_{i-1} \cap P_i \right| \right) \right)$$

To distinguish between parallel and sequential pebbling games we will denote the cumulative complexity $CC^{||}(P)$ for parallel pebbling and $CC(P)$ for sequential pebbling. Next, we define the cumulative complexity of an entire graph. Denote the set of all possible pebbling sequences P_G (resp. $P_G^{||}$ for parallel pebbling).

$$CC(G) = \min_{P \in P_G} CC(P)\,, \quad \text{and} \quad CC^{||}(G) = \min_{P \in P_G^{||}} CC(P)\,.$$

With parallel pebbling cumulative complexity ($CC^{||}(G)$) being defined in the same way when parallel pebbling has been used.

5.5. Cumulative Cost for JIT Hashing

In JIT hashing we have a window of size w that only allows us to select dependencies from the previous $w-1$ memory blocks. Recall that there is a trie representing the list of passwords that an adversary wants to try, with each node representing the addition of a new character. To create our JIT pebbling graph we start with this base trie. For each node, any node at distance at most $w-1$ may depend on it. To represent this for each node in the graph we add an edge to each of its descendants up to distance $w-1$. We set our list of sink nodes to be each node that corresponds to a password that is being guessed (e.g. in the path 1-2-3-4-5-6 we may set the nodes for 5 and 6 to be sink nodes, as they correspond to common passwords). We denote $T_{D,w}$ as the directed acyclic graph created in this manner using a dictionary D to form the base trie and windows size w. From this graph we derive two bounds on the cumulative complexity of running a brute force attack on a JIT hashed password. The upper bound on time is derived using a parallel pebbling argument while the similar lower bound is derived using a sequential pebbling game.

Notation: Given a node $v \in T_{D,w}$ we use height(v) (resp. depth(v)) to denote the height (resp. depth) of a node in the tree $T_{W,v}$ e.g., a leaf node is defined to have height 1 and the root node is defined to have depth 0.

Theorem 1. *For a parallel attacker with unbounded memory we have*

$$CC^{||}(T_{D,w}) = |T_{D,w}| \frac{m^2}{2} + \sum_{h>1} \sum_{v:\ \textit{height}(v)=h} \left(m^2 \min\{h-2, w-2\} \right)$$

For a sequential (memory bounded) adversary we have

$$CC(T_{D,w}) \geq \left(\sum_{d} \sum_{v:\textit{depth}(v)=d} m^2 \min\{d-1, w-1\} \right) - |T_{D,w}| \frac{m^2}{2}$$

Proof. (sketch) Consider the graph $T_{D,w}$ under the parallel black pebbling game. Let t be the height of the root node. We first observe that the there is a simple legal parallel

pebbling strategy $P \in P_{T_{D,w}}$ with

$$CC^{||}(P) \leq |T_{D,w}| \frac{m^2}{2} + \sum_{h>1} \sum_{v:\ \text{height}(v)=h} \left(m^2 \min\{h-2, w-2\}\right) .$$

In particular, we set $P_0 = \emptyset$ and we set $P_i = \{v :\ t+w-i \geq height(v) \geq t+1-i\}$. To see that the pebbling is legal we observe that P_t contains all leaf nodes in $T_{D,w}$ and that $P_{i+1} \setminus P_i = \{v\ :\ \text{height}(v) = t-i\}$ and that therefore $\text{parents}(P_{i+1} \setminus P_i) \subseteq \{v : height(v) = t-i\} \subseteq \text{parents}(P_i)$. Furthermore, since $P_{i+1} \cap P_i = \{v :\ t+w-i-1 \geq \text{height}(v) \geq t+1-i\}$, we have $CC^{||}(T_{D,w}) \leq CC(P) =$

$$\sum_{i=1}^{t} \left(\left(\frac{m^2}{2} |P_i \setminus P_{i-1}| \right) + \left(m^2 \left(P_{i-1} \cap P_i\right)\right) \right)$$

$$= \sum_{h} \sum_{v:\ \text{height}(v)=h} \left(\frac{m^2}{2} + m^2 \min\{\max\{h-2, 0\}, w-2\} \right)$$

$$= |T_{D,w}| \frac{m^2}{2} + \sum_{h>1} \sum_{v:\ \text{height}(v)=h} \left(\frac{m^2}{2} + m^2 \min\{h-2, w-2\} \right)$$

Note that if a node v is at height 1 (e.g., a leaf) or 2 then we keep a pebble on that node for exactly one round (total cost $m^2/2$). If a node v is at height > 2 then we keep a pebble on that node for exactly $\min\{h-2, w-2\}$ additional rounds *after* we initially place the pebble (total cost $m^2/2 + m^2 \min\{h-2, w-2\}$).

To see that $CC^{||}(T_{D,w}) \geq CC(P)$ we note that in *any* legal pebbling of $T_{D,w}$ we must place a pebble on each node in $T_{D,w}$ at some point and that the total cost of placing these pebbles on each node for the first time is at least $|T_{D,w}| m^2/2$. After we first place a pebble on node v we must keep a pebble on node v for an additional $\min\{w-2, \max\{h-2, 0\}\}$ steps to pebble the $\min\{w-1, h-1\}$ children of node v. The total additional cost is $m^2 \min\{w-2, \max\{h-2, 0\}\}$ for each node v. Therefore, $CC^{||}(T_{D,w}) \geq$

$$\sum_{h} \sum_{v:\ \text{height}(v)=h} \left(\frac{m^2}{2} + m^2 \min\{\max\{h-2, 0\}, w-2\} \right) .$$

Now consider an arbitrary sequential pebbling strategy and in particular consider the unique round i_v during which we first place a pebble on node v. We note that during round $i_v - 1$ we must have pebbles on all of v's parents, thus $|P_{i_v-1}| \geq |\text{parents}(v)| \geq \min\{w-1, \text{depth}(v)-1\}$. It follows that

$$\begin{aligned} CC(P) &\geq \left(\sum_{v} m^2 \min\{w-1, \text{depth}(v)-1\} \right) - |T_{D,w}| \frac{m^2}{2} \\ &= \left(\sum_{d} \sum_{v:\text{depth}(v)=d} m^2 \min\{d-1, w-1\} \right) - |T_{D,w}|\, m^2/2 . \end{aligned}$$

□

Under the sequential black pebbling game a similar approach works. In this case rather than the height of each node consider the depth of each each node, which is the distance to the root of the trie. For each node realize that you must pay the initial pebbling cost and also pay $min\{w-1, d-1\}$ to keep its parents in the trie. Thus we gain a similar bound for the sequential pebbling game:

To give some perspective we calculate the pebbling complexities for several password lists, including rockyou, xkcd-style passwords, and a cracking password list from Openwall, designed for use with John the Ripper[37]. We look at the advantage in terms of the CC of the JIT pebbling graph and the work required to calculate all passwords using memory hard hashing but not a JIT model i.e. each password requires only $m^2/2$ work to compute. We specifically define our advantages as $adv^{||}(D, w)$ and $adv(D, w)$ under parallel and sequential models for a dictionary of passwords D with a JIT hashing algorithm using a window of size w as

$$adv^{||}(D, w) = \frac{CC^{||}(T_{D,w})}{m^2 |T_{D,w}|/2}$$
$$adv(D, w) = \frac{CC(T_{D,w})}{m^2 |T_{D,w}|/2} .$$

This can be thought of intuitively as the amount of work necessary for the attacker to check all passwords in the dictionary when passwords are protected with JIT divided by the work the attacker must perform when passwords are protected by a standard memory hard function with equivalent authentication delay.

XKCD (Random Words) For XKCD passwords we use a semi-theoretical approach to calculate the cumulative complexity. First we divide the trie for a single word into a list of nodes by height/depth. For height, we can get the next largest XKCD trie's height list by multiplying the current list of heights by 10,000 (as there are 10,000 copies of this trie in the next largest trie). We add on the heights for the newly added nodes for the new first word, with all of the heights increased by the max height of the single word trie. This process can be continued to produce the height list for larger XKCD tries, which is sufficient to allow calculation of parallel cumulative complexity. Finding the depth lists follows a similar strategy, but is more complicated. Depth must take into account the depth of the

List	$adv^{\|\|}(D,w)$	$adv(D,w)$
Rockyou	6.028	13.260
Cracking(1k)	10.100	15.542
Cracking (10k)	7.057	10.654
Cracking (100k)	3.048	6.068
Cracking ($\sim$ 5 mill)	9.154	12.468
XKCD (4 word)	11.674	25.399

TABLE 2. ADVANTAGES OF JIT HASHING WITH SELECTED DICTIONARIES

location where each recursive tree begins. For all advantage calculations we assume that certain parameters have been set according to reasonable assumptions about available hardware and the results from the user study in Section 4.6. For the assumptions we assume that the user has set up their function to use 2GB of memory, and use 200MB per character. We assume that they allow 3 deletions, which sets our window to be $2GB/200MB - 3 = 7 = w$. We also assume that the same amount of memory would have been used without JIT hashing, giving identical m^2 values for each. The calculations based on these parameters can be seen in Table 2.

Empirical Distribution We again analyze the complexity using the Rockyou password list. In this case we are able to calculate the entire trie and feed the results into our formula. In this case we find that JIT offers an advantage of over 6 in the parallel pebbling model and over 13 under sequential. Exact number are in Table 2. The Rockyou leak consisted of primarily low-value accounts, yet it remains the largest plaintext password leak as of this time, and provides valuable insight into cracking attempts against these types of accounts.

Cracking Dictionary To model higher value accounts we use a general cracking dictionary designed by Openwall for use with John the Ripper [37]. We computed advantages using cracking dictionaries of size 1k, 10k, 100k, and roughly 5 million (whole dictionary). Our analysis shows that JIT can increase work by roughly an order of magnitude.

6. Implementation

A basic implementation of Just In Time hashing was created as a version of Argon2 - The winner of a Password Hashing Competition held from 2013-2015. [3][12] Argon2 is an open source memory-hard hashing function that runs in either a data-dependent (Argon2d), data-independent (Argon2i), or hybrid (Argon2id) mode. This implementation specifically contains a method to introduce characters into Argon2 on the fly. To modify Argon2i to run in JIT mode we needed to find a place to add in a character after a specified amount of time. To do this we modify the stage where two blocks are XORed together and passed into Blake2b. Whenever a character must be added, we take the character (currently as a byte, although this method is easily modified to work with larger character sizes) and XOR it with the first byte of the previous input block. The result of this is XORed with the second block from the iMHF dependencies and run through the same XOR, Blake2b, XOR process as the current version of Argon2. Once a character has been entered, Argon2 can be computed as normal for a set amount of blocks until it is time to wait for the next character.

For single-thread Argon2 an array of length m is declared and divided into a specified number of blocks. The first block is filled using the parameters (and optional salt) passed into the Argon2 function and the block is hashed using the Blake2b hash function [10]. At each successive stage in the first pass the next block is filled by XORing the previous block with another block chosen according to the iMHF dependencies defined. Once the XOR is completed, the block is hashed with Blake2b and XORed once more with the input to Blake2b. In further passes the dependencies are permitted to select blocks further ahead in the array, as long as they were filled during the last pass and not the current pass. The implementation introduces a few new parameters to Argon2 that the JIT mode requires. In JIT mode there are two significant parameters to set. The first of these is the number of blocks that each character corresponds to while running the function. This is tunable to any number, and is the way that running time per character is set. The second parameter is the number of permitted backspaces the algorithm should permit. This parameter, combined with the per-character block number, is how the window is determined by the algorithm. Once a number of permitted backspaces is set the window for permitted dependencies is set to $w = m - \texttt{blocks\_per\_char} * \texttt{max\_num\_del}$. As this is a proof of concept the prototype implementation currently supports a very basic $\beta = 0$-tolerant k-limited JIT hashing function i.e. it requires correct input. The code for the implementation is publically available at (https://github.com/JustInTimePwdHash2017/Just-In-Time-Hashing). For a more detailed description of the implementation please refer to the full version of this paper.

7. Discussion

7.1. Usability advantages

A great benefit of JIT hashing is that, from a user's perspective, there is nothing new to learn. So long as the system has been implemented correctly most users should expect to be able to authenticate with no detectable delay. In a case when a user does notice a delay this would be because they are either typing much faster than expected, such as faster than 95% of users, or because they have deleted a significant amount, but not all, of the characters that they entered. In these cases while there is a delay it is not a very significant delay, and should only last for a few seconds while the algorithm restarts computation from the beginning to catch up with the user.

From a developer's perspective JIT hashing will require some modification to their existing authentication systems. Current password hashing functions are set up to take the

entire password at once, while JIT is a streaming algorithm. Developers would need to modify their existing systems to accept passwords one character at a time, which may vary from simple to complex depending on the current systems they are working with. Beyond modification to be a streaming algorithm, the replacement of the function itself would be quick, only requiring the developer to import the function and set a few additional parameters.

7.2. Client vs Server-Side

In earlier sections we described JIT Hashing as a client-side hashing algorithm. The reason for this is that a naive implementation for a server-side version could include several serious security or usability risks. For example, a naive implementation may send (encrypted) characters to the server one at a time. An adversary could eavesdrop in this scenario to learn the exact length of a user's password. One way to address this issue would be to have the client send an encrypted packet every few miliseconds whether or not a character was typed. A second consideration is that of sever resources. Since JIT hashing involves extra work any server side implementation must also consider the potentially increased risk of denial-of-service attacks.

Acknowledgments

We would like to thank Intel for their support via a Research Assistantship grant. We would also like to thank Carlton Ashley for his review and code checking assistance, and James McKee for his help designing the user study website. This research was partially funded by NSF Grant 1704587.

Appendix

Corollary of Theorem 1

Theorem 1 showed the general form of computing parallel and sequential pebbling complexities of tries based on password dictionaries. The following corrolary is for the specific case that all passwords are uniformly random and of length t, i.e. $pwd \in \Sigma^t$.

Corrolary. *1.1*

With all passwords of the same length we can say that the number of nodes with height h is $|N_h| = |\Sigma|^{t-h}$ and that the number of nodes of depth d is $|N_d| = |\Sigma|^d$.

Begin with the definition of parallel cumulative complexity:

$$CC^{||}(T_{D,w}) = |T_{D,w}|\, m^2/2 + \sum_{h>1} \sum_{v:\ height(v)=h} \left(m^2 \min\{h-2, w-2\}\right)$$

As we have a complete trie of height t we can split this sum into the two components generated by the min function.

$$CC^{||}(T_{D,w}) = |T_{D,w}|\frac{m^2}{2} + \sum_{h=2}^{w} \sum_{v \in height(h)} m^2(h-2) + \sum_{h=w+1}^{t} \sum_{v \in height(h)} m^2(w-2)$$

As the inner sum is now a sum of constants we rewrite it as a product

$$CC^{||}(T_{D,w}) = |T_{D,w}|\frac{m^2}{2} + \sum_{h=2}^{w} |\Sigma|^{t-h} m^2(h-2) + \sum_{h=w+1}^{t} |\Sigma|^{t-h} m^2(w-2)$$

Factoring out terms and applying bounds from complete tries of length t to limit the sums gives

$$CC^{||}(T_{D,w}) = |T_{D,w}|\frac{m^2}{2} + |\Sigma|^t m^2 \left(\sum_{h=2}^{w} |\Sigma|^{-h}(h-2) + \sum_{h=w+1}^{t} |\Sigma|^{-h}(w-2) \right)$$

Which can now be evaluated to a closed form expression, giving

$$CC^{||}(T_{D,w}) = |T_{D,w}|\frac{m^2}{2} + m^2|\Sigma|^t \cdot \frac{|\Sigma|^{-w-1}\left(|\Sigma|^w - |\Sigma|^2(w-1) + |\Sigma|(w-2)\right)}{(|\Sigma|-1)^2 s} + m^2|\Sigma|^t \frac{(w-2)|\Sigma|^{-t-w}(|\Sigma|^t - |\Sigma|^w)}{|\Sigma|-1}$$

It is possible to perform a similar set of steps with sequential pebbling. The main difference is that rather than the number of nodes at a height being $|\Sigma|^{t-h}$ we instead have $|\Sigma|^d$. Substituting this change into the formula we can follow the same beginning steps until we reach the split sums, which work out to

$$CC(T_{D,w}) \geq \left(\sum_{d=1}^{w} |\Sigma|^d m^2(d-1) + \sum_{d=w+1}^{t} |\Sigma|^d m^2(w-1) \right) - |T_{D,w}|m^2/2$$

Which evaluates to

$$CC(T_{D,w}) = m^2 \left(\frac{|\Sigma| \left((w-1)|\Sigma|^{w+1} - w|\Sigma|^w + |\Sigma| \right)}{(|\Sigma|-1)^2} \right) + m^2 \left(\frac{|\Sigma|(w-1)(|\Sigma|^t - |\Sigma|^w)}{|\Sigma|-1} \right) - |T_{D,w}| \frac{m^2}{2}$$

For an example we take alphanumeric passwords of length 6 with window size 7 i.e. $|\Sigma| = 62, w = 7, t = 6$ and calculate the advantages. In this case we find a negligible advantage $(1 + \epsilon)$ for parallel pebbling and an advantage of 9.13 with sequential pebbling. While interesting, we note that uniform passwords are not a realistic model for actual user-chosen passwords, and that the results found in the main body of this paper are more useful to evaluate JIT in practice.

Extended implementation details

This appendix assumes some familiarity with the implementation and specifications of Argon2, which are available online at https://github.com/P-H-C/phc-winner-argon2. Several changes in addition to those mentioned in Section 6 were added to the Argon2 code. First, we want to note that this implementation is meant as a proof of concept, and that additional work would be required to make the implementation work under all Argon2 modes of operation. For now the only mode of operation that supports JIT hashing is Argon2i, with other modes being disabled if the JIT mode is activated. We also disable multi-lane versions of Argon2i. While originally done to simplify the proof-of-concept prototype we also note that recent work has shown that the parallel implementation of Argon2 has issues that need to be resolved [5]. Character addition is accomplished within the compression function Z, which takes two blocks and combines them to produce an output. During most stages of computation the compression function works exactly as specified in the Argon2 documentation. During character addition a slight modification is made to the inputs to this function. The previous block (A) has the new character c_i XORed with the first byte. While we are working with standard ASCII characters for now, we note that this can easily be extended as far as anyone would like, and supports as many types of characters as can fit into the specified block size. A diagram of character addition is seen in Figure 3, which is modified from Figure 3 in the original Argon2 paper [14].

Error handling. To safeguard against improper use of JIT hashing with the current implementation we extend the input checking and error reporting sections of code. Although we do not consider this a significant change to the code, we mention it so that those interested in extending the implementation are aware that these sections may require modification to reactivate certain features of Argon2.

Dependency window. Any implementation of JIT hashing that wishes to maximize usability must have the ability to revert to previous states. The current master branch of the provided repository represents a 0-tolerant k-limited JIT function, however we plan to add support for $\beta \geq 1$ tolerant JIT functions in the near future, which will appear at a branch of the main repository as work continues. The dependency window is being implemented by modifying the edge distribution functions in `core.c` and `ref.c`, where we are able to limit the chosen additional reference block. This works in a manner similar to the window limit imposed in the parallel version of Argon2, where edges are forbidden to be taken from the same slice in a different lane. In this version this is done to ensure deterministic results, however the same method serves to limit the edge distribution to a window in just in time hashing.

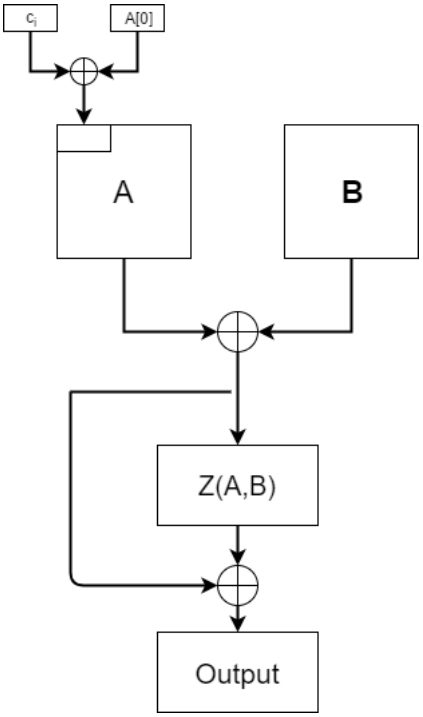

Figure 3. Character addition in JIT Argon2i

Client vs Server Side Implementation

One limitation of JIT hashing is that is only particularly useful on a user-side machine. We specifically make use of the time that the user is spending typing their password to hide the time it takes to do the additional key stretching. So, while the user may not be aware of the fact that quite a bit of extra key stretching is taking place, it is still happening in the end. A server would not have the benefit of typing time to hide this extra key stretching; they must bear the full brunt of the extra computation. For a larger server, this can rapidly become a problem for a server, as they will have to provide enough computing resources to handle this extra stretching. If a server was not already willing to do the amount of key stretching that JIT hashing would require with standard password hashing then it seems unlikely that they would

be willing to do it under this scheme. An additional issue with trying to implement JIT hashing on a server is the leakage of password length. To take advantage of the JIT method a server would need to receive characters one by one and run the computation on their own side while the user is still typing. Not only does this introduce uncertainty in the timing through network latency, but it also exposes the user to a man in the middle or eavesdropping attacks, where an adversary would be able to count the number of messages sent from the user to the server to determine the length of their password. This additional information can be used by the adversary to optimize their cracking attempts, both offline and online.

So, when using JIT hashing it is likely that it would somehow be implemented on the user side. For example, with a password manager such as LastPass, KeePass, or a browser password manager, a user may want to select to store their passwords in a JIT manner. When they want to enter a password for some website, they can either enter a master password or some password they have decided to use for that site. While typing the JIT hashing system is run, and at the end they have a hash that they can use as their password in the website. From there, it can be sent over the usual secure methods to the website where it is hashed yet again by the server and compared to their password records.

Optimizing Guess Order

In most of our analysis we focused on an attacker who wants to check every password in a dictionary. Placing a bound on the expected work for an adversary who would like to strategically order password guesses (e.g., it makes sense to check the most popular passwords such as '123456' and 'password' before checking less popular, but still common, passwords like 'monkey1') is more difficult. In the previous analysis an assumption was made that the adversary is working with a uniform distribution of passwords. In reality, the distribution of passwords is not uniform, and may be quite different even from website to website. To begin, we find an upper bound on the work an adversary would need to do under this scheme. As before, assume that the cost for each hashing step, including key stretching, is $W = \mathbb{C}(\mathbb{H}) = 1$. In addition, assume that the length of the i's most common password, π_i, is given by the function $L(\pi_i)$. An adversary may follow a strategy of trying the top n passwords. In the worst case for the adversary each password corresponds to are completely disjoint. In such a case the upper bound for the adversary's work to check the n most common passwords (W_n) is:

$$\mathbb{E}(W_n) = \sum_{i=1}^{n} L(\pi_i) \in O(n * max_i(L(\pi_i))$$

Their lower bound is a case where the top n passwords are all consecutive subsequences of each other. In this case the first password has a cost of $L(\pi_i)$, and each other password has a cost of 1 to extend the password. This gives the lower bound:

$$\mathbb{E}(W_n) \in \Omega(n)$$

Just In Time Hashing without Memory Hardness

In theory the adversary need only run one hashing operation per password that they want to try. However this is under the assumption that the adversary is attempting every single possible password that can be generated. The question remains how this scheme performs in practice, and what sort of bounds can be seen for known password data sets. To investigate the performance of JIT hashing in practice we construct a trie from the Rockyou password set. The chart below details how much work an adversary would need to do depending on how many passwords they are going to try. We calculate the work that an adversary is expected to do by taking the Rockyou password list and inserting each new password into a trie. After inserting each password, we record the number of nodes that had to be added to the trie, which corresponds to the number of hashing steps that would need to be done to calculate the value that belongs in each new node.

This figure reveals some interesting information about JIT hashing. First, note that early on the adversary's performance approaches the optimal amount of work required. The adversary gains the largest advantage (least work per password) when they are checking 1.14 million passwords, with an average of 1.139 hashing steps required per password at the curve's minimum point. As the adversary continues attempting more passwords their expected average work per password increases and stabilizes at a value of roughly 1.50 hashing steps required per password. If we assume that a fairly standard hashing function like SHA-256 or BCRYPT is being used then the expected amount of memory needed to store this trie structure is lower bounded by about 700MB of memory.

One fact we would like to note about the graph is that the number of hashing steps shown in this graph correspond to JIT hashing steps, not the number of standard hashing steps that would be required for a normal password. JIT hashing steps differ in that they have been run through key stretching for an amount of time based on user password typing habits. As we have previously seen from the user study this can mean that a few hundred miliseconds of key stretching can be performed. Even the time spent on a single keystroke can outpace the amount of key stretching being deployed in practice. Take as an example JIT hashing implemented with hash iteration with the time per character set to 183ms (non mobile 5th percentile time). Even if an adversary only needed to calculate a single stage for each password this still outpaces the key stretching used in leaks such as Dropbox [69], where BCRYPT with 2^8 iterations were used. Quick tests using a laptop with a 1.70 GHz processor showed that this can be calculated in roughly 10-15ms.

We see different results when applying different rules for passwords. For example, Randal Munroe, author of the

webcomic XKCD, suggests that users should pick passwords that are secure yet easy to remember [71]. Specifically he estimates that simply be selecting four random common words users can select easily remembered secure passwords. For our analysis we assume that these passwords are truly chosen at random from a list of the 10,000 most common words in English. This gives us a possible search space of $10^{4^4} = 10^{16}$ possible keys. When taking the trie-based approach each stage of 10,000 words can be compressed into 24163 nodes. Carrying this out with all possible 4 word combinations gives a total trie size of 2.15×10^{16} nodes. This method also is a reasonable description of other suggested password creation schemes, such as the Person Action Object (PAO) scheme suggested by Blocki et al [21] as a way to create secure passwords that are much easier for users to remember. In this scheme users are presented with two images, a location and a person, and are asked to envision that person at that location performing some action on an object. For example, a user may be shown a picture of Albert Einstein and a Beach, and be asked to remember the words "Kissing" and "Piranha". When they return to a site to log in, they will be shown the picture of Einstein and the beach and are required to remember the password kissingpiranha. As these passwords are also made up of concatenated common words, we use the same method that is used to estimate XKCD-style passwords to estimate the strength of JIT hashing for PAO style passwords. To generalize, note that we have a trie of size 24163 per word that is in the trie, and 10,000 terminal nodes within that trie that represent a word. Thus, if we would like to see the strength of passwords generated using the n most common words concatenated together m times, we note that the trie would have to be of size $\sum_{i=0}^{m-1} cn^i$, where c is the size of the trie that is generated by the n most common words. A sample table for the first few values of this function for $n = 10000$ are shown below.

Final key stretching

In the hash iteration mode being described it has currently been assumed that a version of JIT hashing was being used that performed a total of n rounds of key stretching total. When applying this model and a strategic approach to cracking can be obtained (1.14 hashing stages per password). As already discussed before the first additional advantage we have is that each of these hashing stages can be set to be as long as the average time delay between users' keystrokes, which may be more than the amount of key stretching than would otherwise be done. The second additional advantage than can be gained is to introduce the method of final key stretching. As stated earlier, with this method we first calculate the hash of the password Π, $H(\Pi)$, and then run this result through additional rounds of key stretching to obtain $H^r(H(\Pi))$. A key fact of the earlier hashing stage analysis is that the expected average work per password is to obtain $H(\Pi)$. Once this has been obtained, we have what can be considered a new random input for the hashing function H. At this point we can treat the result of the JIT hashing stage as any other password, and apply known key stretching methods to it in a way that further increases the difficulty of computing the function. In this case, however, the adversary gains no additional advantage by using a trie based attack. There is no relation between each successive iteration of hashing at this stage in the same way that there was during the JIT stage, which is what allowed the adversary to gain their advantage through the trie attack. When running the simulated trie attacker with final stretching added in the curve shifts upwards by a number of hashing steps equivalent to the amount of additional key stretching introduced.

References

[1] Anne Adams and Martina Angela Sasse. "Users are not the enemy". In: *Communications of the ACM* 42.12 (1999), pp. 40–46.

[2] Eytan Adar. "Why I hate Mechanical Turk research (and workshops)". In: *Proc. CHI Workshop on Crowdsourcing and Human Computation*. 2011.

[3] Jean-Philippe Aumasson et al. *Password Hashing Competition*. https://password-hashing.net/. 2015.

[4] Joël Alwen and Jeremiah Blocki. "Efficiently Computing Data-Independent Memory-Hard Functions". In: *Advances in Cryptology CRYPTO'16*. Springer, 2016.

[5] Joël Alwen and Jeremiah Blocki. "Towards Practical Attacks on Argon2i and Balloon Hashing". In: *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*. http://eprint.iacr.org/2016/759. IEEE. 2017, pp. 142–157.

[6] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. "Depth-Robust Graphs and Their Cumulative Memory Complexity". In: *Advances in Cryptology-EUROCRYPT 2017*. Springer, 2017.

[7] Joël Alwen and Vladimir Serbinenko. "High Parallel Complexity Graphs and Memory-Hard Functions". In: *47th Annual ACM Symposium on Theory of Computing*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. Portland, OR, USA: ACM Press, 2015, pp. 595–603.

[8] Joël Alwen et al. *On the Memory-Hardness of Data-Independent Password-Hashing Functions*. Cryptology ePrint Archive, Report 2016/783. http://eprint.iacr.org/2016/783. 2016.

[9] Joël Alwen et al. "Scrypt Is Maximally Memory-Hard". In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10211. Lecture Notes in Computer Science. Paris, France: Springer, Heidelberg, Germany, 2017, pp. 33–62.

[10] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[11] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[12] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P 2016)*. IEEE. 2016, pp. 292–302.

[13] Alex Biryukov and Dmitry Khovratovich. "Tradeoff cryptanalysis of memory-hard functions". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2014, pp. 633–657.

[14] Alex Biryukov et al. *The memory-hard Argon2 password hash and proof-of-work function*. Internet-Draft draft-irtf-cfrg-argon2-00. Internet Engineering Task Force, Mar. 2016. 23 pp.

[15] *BitLocker*. URL: https://docs.microsoft.com/en-us/windows/device-security/bitlocker/bitlocker-overview.

[16] Jeremiah Blocki, Manuel Blum, and Anupam Datta. "GOTCHA Password Hackers!" In: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM. 2013, pp. 25–34.

[17] Jeremiah Blocki and Anupam Datta. "CASH: A Cost Asymmetric Secure Hash Algorithm for Optimal Password Protection". In: *IEEE 29th Computer Security Foundations Symposium*. 2016, pp. 371–386.

[18] Jeremiah Blocki, Ben Harsha, and Samson Zhou. "On the Economics of Offline Password Cracking". In: *IEEE Security and Privacy* (2018).

[19] Jeremiah Blocki and Hong-Sheng Zhou. "Designing Proof of Human-Work Puzzles for Cryptocurrency and Beyond". In: *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9986. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, 2016, pp. 517–546. DOI: 10.1007/978-3-662-53644-5_20.

[20] Jeremiah Blocki et al. "Optimizing password composition policies". In: *Proceedings of the fourteenth ACM conference on Electronic commerce*. ACM. 2013, pp. 105–122.

[21] Jeremiah Blocki et al. "Spaced Repetition and Mnemonics Enable Recall of Multiple Strong Passwords". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. 2015.

[22] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. "Balloon Hashing: Provably Space-Hard Hash Functions with Data-Independent Access Patterns". In: *Advances in Cryptology-ASIACRYPT 2013*. 2016.

[23] Joseph Bonneau and Stuart Schechter. ""Toward reliable storage of 56-bit keys in human memory"". In: *Proceedings of the 23rd USENIX Security Symposium*. "San Diego, CA, USA", 2014.

[24] John G Brainard et al. "A New Two-Server Approach for Authentication with Short Secrets." In: *USENIX Security*. Vol. 3. 2003, pp. 201–214.

[25] LastPass Breach. *LastPass Security Notice*. https://blog.lastpass.com/2015/06/lastpass-security-notice.html/ (retrieved 11/10/2016).

[26] M. Buhrmeister, T. Kwang, and S. D. Gosling. "AMazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data?" In: *Perp. Psych. Sci.* 6.1 (2011), pp. 3–5.

[27] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. "Practical yet universally composable two-server password-authenticated secret sharing". In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. Raleigh, NC, USA: ACM Press, 2012, pp. 525–536.

[28] John Campbell, Wanli Ma, and Dale Kleeman. "Impact of restrictive composition policy on user password choices". In: *Behaviour & Information Technology* 30.3 (2011), pp. 379–388.

[29] Ran Canetti, Shai Halevi, and Michael Steiner. "Mitigating Dictionary Attacks on Password-Protected Local Storage". In: *Advances in Cryptology - CRYPTO 2006: 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006. Proceedings*. Ed. by Cynthia Dwork. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 160–179. ISBN: 978-3-540-37433-6. DOI: 10.1007/11818175_10. URL: http://dx.doi.org/10.1007/11818175_10.

[30] Xavier de Carné de Carnavalet and Mohammad Mannan. "From Very Weak to Very Strong: Analyzing Password-Strength Meters". In: *ISOC Network and Distributed System Security Symposium – NDSS 2014*. San Diego, CA, USA: The Internet Society, 2014.

[31] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. "Adaptive Password-Strength Meters from Markov Models". In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*. San Diego, CA, USA: The Internet Society, 2012.

[32] Claude Castelluccia et al. "When Privacy meets Security: Leveraging personal information for password cracking". In: *arXiv preprint arXiv:1304.6584* (2013).

[33] Ashok K. Chandra. "Efficient Compilation of Linear Recursive Programs". In: *SWAT (FOCS)*. 1973, pp. 16–25.

[34] Rahul Chatterjee et al. "pASSWORD tYPOS and How to Correct Them Securely". In: *2016 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, 2016, pp. 799–818. DOI: 10.1109/SP.2016.53.

[35] Stephen A. Cook. "An Observation on Time-storage Trade off". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA: ACM, 1973, pp. 29–33. DOI: 10.1145/800125.804032.

[36] Nick Cubrilovic. *RockYou Hack: From Bad To Worse*. 2009. URL: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/.

[37] Solar Designer. *John the Ripper password cracker*. 2006.

[38] *Diskcryptor*. URL: https://diskcryptor.net/.

[39] Cynthia Dwork, Moni Naor, and Hoeteck Wee. "Pebbling and Proofs of Work". In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2005, pp. 37–54.

[40] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. "Key-Evolution Schemes Resilient to Space-Bounded Leakage". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2011, pp. 335–353.

[41] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. "One-Time Computable Self-erasing Functions". In: *TCC 2011: 8th Theory of Cryptography Conference*. Ed. by Yuval Ishai. Vol. 6597. Lecture Notes in Computer Science. Providence, RI, USA: Springer, Heidelberg, Germany, 2011, pp. 125–143.

[42] Stefan Dziembowski et al. "Proofs of Space". In: *Advances in Cryptology – CRYPTO 2015, Part II*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9216. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2015, pp. 585–605. DOI: 10.1007/978-3-662-48000-7_29.

[43] Paul Erdoes, Ronald L. Graham, and Endre Szemeredi. *On Sparse Graphs with Dense Long Paths*. Tech. rep. Stanford, CA, USA, 1975.

[44] Antminer Distribution Europe. *Antminer S9*. http://www.antminerdistribution.com/antminer-s9/ (Retrieved November 13, 2016).

[45] Adam Everspaugh et al. "The pythia prf service". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 547–562.

[46] Sascha Fahl et al. "On the ecological validity of a password study". In: *Proceedings of the Ninth Symposium on Usable Privacy and Security*. ACM. 2013, p. 13.

[47] Niels Ferguson. *AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista*. 2006.

[48] Dinei Florêncio, Cormac Herley, and Paul C. Van Oorschot. "An Administrator's Guide to Internet Password Research". In: *Proceedings of the 28th USENIX Conference on Large Installation System Administration*. LISA'14. 2014, pp. 35–52.

[49] Christian Forler, Stefan Lucks, and Jakob Wenzel. *Catena: A Memory-Consuming Password Scrambler*. Cryptology ePrint Archive, Report 2013/525. http://eprint.iacr.org/2013/525. 2013.

[50] J Alex Halderman, Brent Waters, and Edward W Felten. "A convenient method for securely managing passwords". In: *Proceedings of the 14th international conference on World Wide Web*. ACM. 2005, pp. 471–479.

[51] Brett Hemenway et al. "Adaptively Secure Garbled Circuits from One-Way Functions". In: *Advances in Cryptology – CRYPTO 2016, Part III*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9816. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2016, pp. 149–178. DOI: 10.1007/978-3-662-53015-3_6.

[52] Cormac Herley and Paul Van Oorschot. "A research agenda acknowledging the persistence of passwords". In: *IEEE Security & Privacy* 10.1 (2012), pp. 28–36.

[53] Carl E. Hewitt and Michael S. Paterson. "Record of the Project MAC Conference on Concurrent Systems and Parallel Computation". In: ed. by Jack B. Dennis. New York, NY, USA: ACM, 1970. Chap. Comparative Schematology, pp. 119–127. DOI: 10.1145/1344551.1344563. URL: http://doi.acm.org/10.1145/1344551.1344563.

[54] Tyler Hicks and Dustin Kirkland. *Ecryptfs*. URL: http://ecryptfs.org/.

[55] Philip G. Inglesant and M. Angela Sasse. "The True Cost of Unusable Password Policies: Password Use in the Wild". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: ACM, 2010, pp. 383–392. ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753384. URL: http://doi.acm.org/10.1145/1753326.1753384.

[56] P. G. Ipeirotis. "Demographics of Mechanical Turk". In: *Tech. Rep. CeDER-10-01* (2010).

[57] Zahra Jafargholi and Daniel Wichs. "Adaptive Security of Yao's Garbled Circuits". In: *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9985. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, 2016, pp. 433–458. DOI: 10.1007/978-3-662-53641-4_17.

[58] Ari Juels and Ronald L. Rivest. "Honeywords: making password-cracking detectable". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. Berlin, Germany: ACM Press, 2013, pp. 145–160.

[59] Burt Kaliski. "PKCS# 5: Password-based cryptography specification version 2.0". In: (2000).

[60] Patrick Gage Kelley et al. "Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms". In: *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, 2012, pp. 523–537.

[61] Kevin Killourhy and Roy Maxion. "Comparing anomaly-detection algorithms for keystroke dynamics". In: *DNS* (2009).

[62] Saranga Komanduri et al. “Of passwords and people: measuring the effect of password-composition policies”. In: *CHI*. 2011, pp. 2595–2604. URL: http://dl.acm.org/citation.cfm?id=1979321.

[63] Saranga Komanduri et al. “Telepathwords: Preventing Weak Passwords by Reading Users’ Minds”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 591–606. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/komanduri.

[64] Zhiwei Li et al. “The Emperor’s New Password Manager: Security Analysis of Web-based Password Managers.” In: *USENIX Security*. 2014, pp. 465–479.

[65] Jerry Ma et al. “A Study of Probabilistic Password Models”. In: *2014 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, 2014, pp. 689–704. DOI: 10.1109/SP.2014.50.

[66] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. “Publicly verifiable proofs of sequential work”. In: *ITCS 2013: 4th Innovations in Theoretical Computer Science*. Ed. by Robert D. Kleinberg. Berkeley, CA, USA: Association for Computing Machinery, 2013, pp. 373–388.

[67] Udi Manber. “A simple scheme to make passwords based on one-way functions much harder to crack”. In: *Computers & Security* 15.2 (1996), pp. 171–176.

[68] William Melicher et al. “Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks”. In: *USENIX Security Symposium*. 2016, pp. 175–191.

[69] David Meyer. *How to Check If You Were Caught Up In the Dropbox Breach*. http://fortune.com/2016/08/31/dropbox-breach-passwords/ (retrieved 11/10/2016).

[70] Robert Morris and Ken Thompson. “Password security: A case history”. In: *Communications of the ACM* 22.11 (1979), pp. 594–597. URL: http://dl.acm.org/citation.cfm?id=359172.

[71] Randall Munroe. *Password Strength*. 2011. URL: https://xkcd.com/936/.

[72] Arvind Narayanan and Vitaly Shmatikov. “Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff”. In: *ACM CCS 05: 12th Conference on Computer and Communications Security*. Ed. by Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels. Alexandria, Virginia, USA: ACM Press, 2005, pp. 364–372.

[73] C. Percival. “Stronger key derivation via sequential memory-hard functions”. In: *BSDCan 2009*. 2009.

[74] Colin Percival and Simon Josefsson. *The scrypt password-based key derivation function*. Tech. rep. 2016.

[75] Niels Provos and David Mazieres. “Bcrypt algorithm”. In: USENIX. 1999.

[76] Ling Ren and Srinivas Devadas. “Proof of Space from Stacked Expanders”. In: *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9985. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, 2016, pp. 262–285. DOI: 10.1007/978-3-662-53641-4_11.

[77] Blake Ross et al. “Stronger Password Authentication Using Browser Extensions.” In: *Usenix security*. Baltimore, MD, USA. 2005, pp. 17–32.

[78] John E. Savage and Sowmitri Swamy. “Space-time trade-offs on the FFT algorithm”. In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 563–568.

[79] John E. Savage and Sowmitri Swamy. “Space-Time Tradeoffs for Oblivious Interger Multiplications”. In: *ICALP*. 1979, pp. 498–504.

[80] Richard Shay et al. “Can Long Passwords Be Secure and Usable?” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. Toronto, Ontario, Canada: ACM, 2014, pp. 2927–2936. ISBN: 978-1-4503-2473-1. DOI: 10.1145/2556288.2557377. URL: http://doi.acm.org/10.1145/2556288.2557377.

[81] Richard Shay et al. “Designing Password Policies for Strength and Usability”. In: *ACM Trans. Inf. Syst. Secur.* 18.4 (2016), p. 13.

[82] Richard Shay et al. “Encountering stronger password requirements: user attitudes and behaviors”. In: *Proceedings of the Sixth Symposium on Usable Privacy and Security*. SOUPS ’10. Redmond, Washington: ACM, 2010, 2:1–2:20. ISBN: 978-1-4503-0264-7. DOI: 10.1145/1837110.1837113. URL: http://doi.acm.org/10.1145/1837110.1837113.

[83] Jeffrey M. Stanton et al. “Analysis of End User Security Behaviors”. In: *Comput. Secur.* 24.2 (Mar. 2005), pp. 124–133.

[84] Michelle Steves et al. *Report: Authentication Diary Study*. Tech. rep. NISTIR 7983. National Institute of Standards and Technology (NIST), 2014.

[85] Sowmitri Swamy and John E. Savage. “Space-Time Tradeoffs for Linear Recursion”. In: *POPL*. 1979, pp. 135–142.

[86] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: https://www.R-project.org.

[87] Martin Tompa. “Time-space Tradeoffs for Computing Functions, Using Connectivity Properties of Their Circuits”. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC ’78. San Diego, California, USA: ACM, 1978, pp. 196–204. DOI: 10.1145/800133.804348.

[88] Blase Ur et al. “How does your password measure up? The effect of strength meters on password creation”. In: *Proceedings of USENIX Security Symposium*. 2012.

[89] Blase Ur et al. “Measuring Real-World Accuracies and Biases in Modeling Password Guessability”. In: *Proceedings of the 24th USENIX Security Symposium*. USENIX, Aug. 2015. URL: http://www.ece.

cmu.edu/~lbauer/papers/2015/usenix2015-guessing.pdf.

[90] Rafael Veras, Christopher Collins, and Julie Thorpe. "On the semantic patterns of passwords and their security impact". In: *Network and Distributed System Security Symposium (NDSS'14)*. 2014.

[91] Matt Weir et al. "Password Cracking Using Probabilistic Context-Free Grammars". In: *2009 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Computer Society Press, 2009, pp. 391–405.

[92] Jeff Yan et al. "Password Memorability and Security: Empirical Results". In: *IEEE Security and Privacy* 2.5 (Sept. 2004), pp. 25–31. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.81.

[93] Jianxin Yan et al. "The memorability and security of passwords: some empirical results". In: *Technical Report-University Of Cambridge Computer Laboratory* (2000), p. 1.