Computational Security and The Economics of Password Hacking

Abstract

Given the recent rise of cloud computing at cheap prices and the increase in cheap parallel computing options, brute force attacks against stolen password databases are a new option for attackers who may not have enough computing power on their own. We take a survey of the current availability and cost of cloud computing as it relates to the idea of computational security in the context of breaking password databases. Rather than look at just the increase in computing power available per computer, we look at how computing as a service is raising the barrier for password protections being computationally secure. We look at the set of key stretching functions meant to defeat brute force password attacks with the current cheapest cloud computing service in order to determine what amount of money and effort an attacker would need to compromise a password database.

Michael Phox

Zachary Sherin

Adin Schmahmann

Augusta Niles

Context

In password-based network security systems, there is a general architecture whereby the password is sent from the user device to a service server, which then hashes the password some number of times using a random oracle before storing the password in a database. Authentication is completed by following the same process and checking if the hashed password is correct. If the password is in the database, access permission is granted (See Figure 1).

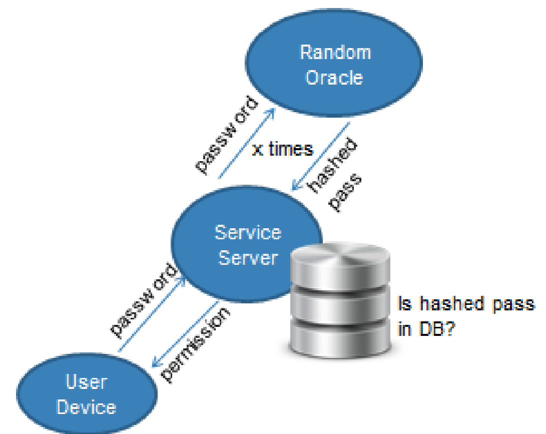


Figure 1 Password-based Security

However, the security system above has been shown to have significant vulnerability depending on the method of password encryption. In contrast to informationally secure (intercepting a ciphertext does not yield any more information to change the probability of any plaintext message. eg. one-time pad), these systems are only secure assuming that any adversaries are computationally limited, as all adversaries are in practice. Because the contents of databases are frequently compromised, if password encryption is not

one-way and collision resistant, decryption is trivial. However, even if both of these conditions are met, passwords are routinely cracked and the accounts exposed using brute force methods. Salting (using random data as an additional input to a one-way hash function) can increase the security of a database by making it more difficult to build and use a dictionary attack against passwords in the database to decrease the amount of time required to crack each password.

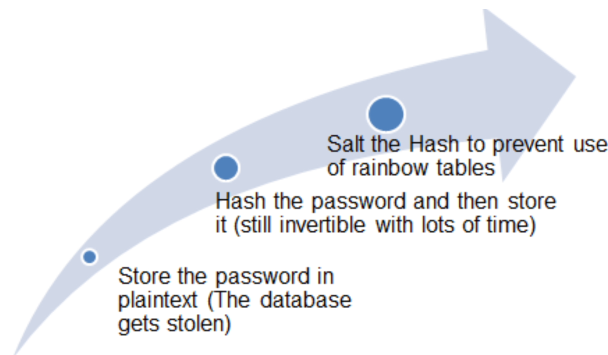


Figure 2 Evolution of Password Storage

If the passwords are being correctly salted and hashes are one-way and collision resistant, there is less risk that an adversary may be able to use a dictionary or rainbow table to make cracking passwords less time-consuming. In this case, the attacker must either store dictionaries for each possible salt or invest the time and money to build them at the time of the attack. The assumed infeasibility of cracking salted passwords remains the standard today, however, salts are not always used and current hashing schemes may not be sufficient to ensure computational security. This problem only gets worse as computing power gets cheaper.

In recent months, the price of rentable computing power has decreased significantly, driven by a price war in cloud computing between Amazon Web Services, Google, and Windows Azure set in motion in order to remove margins and decrease the incentive of new players to enter this market. The following events show the accelerating nature of the price declines in the cloud computing area.

- March 24th, 2014: Cisco joins cloud computing race with $1 billion plan
- March 26th, 2014: Google Slashes Cloud Prices: Google vs AWS Price Comparison
- March 28th, 2014: AWS Responds with Price Cuts: Google vs AWS Pricing Round 2
- March 31st, 2014: Microsoft Azure Matches Amazon's Price Cuts And Introduces New "Basic" Tier

This accelerating price competition comes on top of significant declines in price over the past five years. In this time, the cost of two popular instance sizes at Amazon Web Services, the quad-core 4xlarge (left axis below), and m1.small (right axis) have halved.

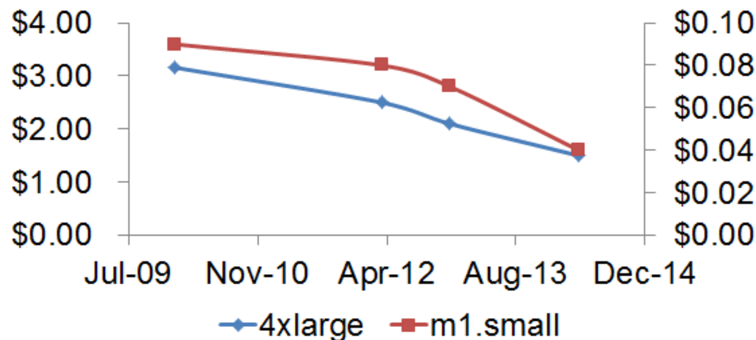
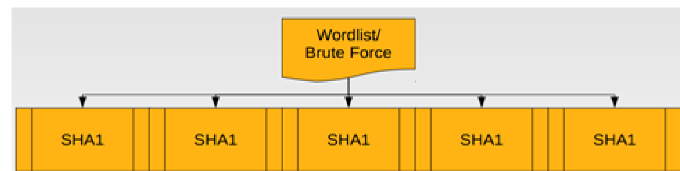


Figure 3 Declining Prices for OnDemand Computation

The growing vulnerability for computationally secure systems has not been overlooked and in January of 2012, a german researcher named Thomas Roth used AWS to crack all hashes from the 560 character SHA1 hash with a password length from one to six in only 49 minutes.

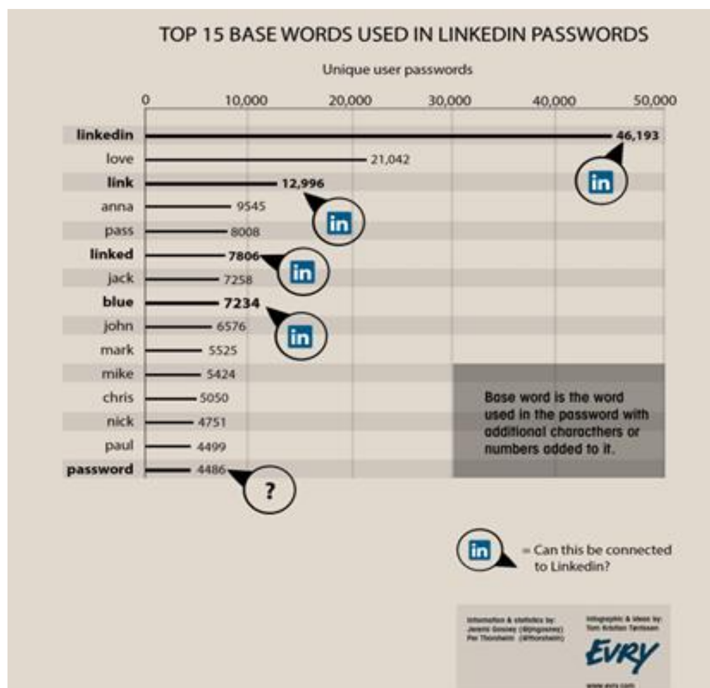


His hacking system architecture was:

- 22GB RAM
- 2 x Intel Xeon X5570
- 2 x NVIDIA Tesla “Fermi” M2050
- $2.10/Hour

Roth has subsequently been deterred from making his code open-source by involvement from the German police, who raided his apartment the evening before he was set to release the took at the Black Hat DC conference. However, another open-source system, Cryptohaze implements network-clustered GPU accelerated password cracking as well and this project seeks to add to this body of research by examining the implications for computation security as computation becomes inexpensive. We lay out a framework for quantifying computational security given the anticipated characteristics of the user’s and adversary’s systems and the quality of a given password.

Quantifying Password Quality



While there are the valid concerns listed above regarding salting and correctly hashing passwords before storing them in a database, these are not the only concerns. Perhaps more troubling is that people tend to choose low entropy passwords. This means that, for example, guessing a 6-letter lowercase letter password is likely to require much much less than 26^6 attempts. In fact, Joseph Bonneau's paper on "The science of guessing" [2012] he describes a statistical method of analyzing passwords that gave results such as most passwords having 15-25 bits of randomness. More specifically, he defines as the mean number of password guesses to before finding the correct one with probability alpha (the equation is given below) and found that was between 15-25 bits (i.e. between 33 thousand and 33 million guesses) for the large sets of data examined. Decreasing the number of guesses required for a sample password can be done by using a dictionary of words tuned towards the audience being examined. For instance, in a break resulting in leaked LinkedIn passwords some of the most popular were related to the website (such as linkedin, link, linked).

Therefore when we examine the quality of passwords and how much entropy they truly have we cannot assume passwords are random within the acceptable password space. Instead we should, if we are to be conservative from the point of view of those trying to protect their passwords, look at the password distributions of similar systems and decide what proportion of the population we care about. In particular it is worth noting that if a hacker can easily attempt a few million guesses of a password that for about 50% of users the hacker will be able to retrieve the password.

$$G_\alpha(\mathcal{X}) = (1 - \lceil \alpha \rceil) \cdot \mu_\alpha(\mathcal{X}) + \sum_{i=1}^{\mu_\alpha(\mathcal{X})} p_i \cdot i$$

$\mu_\alpha(\mathcal{X})$: Minimum dictionary size to succeed with Pr = α

p_i : Probability of password X

Our Research

Background on Key Derivation Functions

Having looked at the state of research on passwords, it is clear that simply hashing a password is not enough to keep it secure, if the password itself is weak. We look to a different class of algorithms that don't just hash a password, but instead turn it into a key suitable for either user identification or data encryption. These algorithms are called key stretching algorithms, as they take a low entropy password and use it to produce a high entropy (usually 256 bit) key. These functions need the following criteria to be considered good key stretching functions:

1. Collision Resistant (no key can be produced by two different passwords)
2. One-Way (impossible to get the password from the key)
3. Resource Intensive (hard to run the algorithm many times at once, to prevent brute-force attacks)

This last point is the most important part of key stretching algorithms. Rather than simply making it impossible to derive the password from the key, they are designed to actively resist brute-force search. By taking up a large amount of an attacker's resources, whether time, computer memory, or cost for custom hardware, these algorithms try to push the time for cracking a hashed password into the area considered computationally secure.

To check this security, we decided to see if given the dropping price of cloud computing and the increase of commercially available, parallel computing graphics hardware these algorithms could be overcome. The idea of computationally secure is being pushed further and further by the availability of cheap computation power. To verify their claims, we look at one existing hash function, SHA256, and two key stretching functions, scrypt and PBKDF2, for comparison.

SHA-2 is a fast hash function commonly used for storing passwords in a database. It makes no attempt to be resource-intensive, making it our baseline for what is an algorithm that can be brute forced. The advent of Bitcoin's enormous value increase has produced a huge

number of algorithms, dedicated hardware, and other ways to quickly brute force SHA-2 hashes.

PBKDF2 is a key stretching function that uses SHA-2 as a starting point for an algorithm that takes more time. It runs for several thousand iterations of the hashing algorithm (in this case SHA-2) and combines the output of each stage into the final key. This algorithm takes a large amount of time, meaning that an honest user would wait on the order of half a second to produce the key from their password for user verification, but an adversary would take much, much longer to brute force a password for an offline attack. However, it doesn't resist parallelism as an attacker can use graphics hardware or a custom-designed ASIC to run many instances of the algorithm at the same time.

Scrypt is a different approach to the key stretching function. Rather than expanding the amount of time that any one iteration of the algorithm takes, it expands the use of memory. It runs PBKDF2 to generate an input key, then uses a large amount of memory to store a number of pseudorandom values that will be used by scrypt to calculate the final output. It resists parallelism by using a large amount of memory space. This makes it hard to use graphics hardware (which has less memory than the CPU) to run many instances of the algorithm at once. An important caveat to this point is that the algorithm can be made to waste less memory space by recalculating certain values over and over. This adds an extremely inefficient time component to the algorithm, similar to PBKDF2.

We looked at each of these algorithms, and compared their speeds on a user device, an android phone, versus an attacker's devices, a mid-grade gaming laptop and Amazon's EC2 GPU computation cloud service. The user device comparison is to see at what point these algorithms become unserviceable for the common user. Though PBKDF2 could theoretically be expanded to take several seconds to compute a user's password verification, a user may not wish to wait the time it takes to log into the service every time. Therefore we keep our device comparisons to time that the average user would not notice for maximum usability.

Why this Matters

Many web services need to store user identification information to allow those users to login. When storing passwords to verify a user at a later time, the web service should store them in such a way that even if the database is compromised user credentials cannot be stolen. As shown above, hashing algorithms such as SHA-2 are not enough to prevent passwords from being stolen or figured out from the hash value. Key stretching functions are meant as a better

way to store passwords in a database securely. We hope to prove or disprove this assumption for each of the two algorithms currently in use, which are scrypt and PBKDF2.

Modeling Hack-Economics

In order to quantify the computational security of a system, the team first defined the effort required to break a password as the amount of time or money required to break a password:

$$Effort = \#Guesses\ Required\ x\ Time\ per\ Guess\ x\ Dollars\ per\ Compute\ Hour$$

The number of guesses has units, guesses. This is multiplied by the time per guess and then the dollars per unit of time to arrive at a value for the effort required to break a password denominated in dollars. This equation depends both on the quality of the password (#Guesses Required), the capability gap between the hacker and user hashing systems which determines the amount of time required to complete one "guess" (hardening from plaintext guess to ciphertext candidate using whichever encryption scheme is selected), and the cost of the computation hour. The cost of computation can either be seen as the hourly rate of a rented system or an allocated rate of return on a purchased system investment.

The amount of time required per guess can either be determined computationally, or can be estimated from the parameters of the algorithm used. For instance for PBKDF2 we can use the equation:

$$Time\ per\ Hacker\ Guess = \frac{Time\ for\ User\ Entry\ x\ Hash\ time\ Hacker}{Hash\ time\ User}$$

The above equation is derived from the equation:

$$Time\ per\ Guess = Hash\ time\ x\ Number\ of\ hashes\ required\ per\ Guess$$

Because the number of hashes required per guess is the same for the hacker as it is for the user for a particular hashing scheme, an equivalency is developed which cancels out this variable and allows us to create the ratio for the time required per hacker guess, a key element of the Effort equation.

In order to translate Effort into system cost, we must consider the economics of the hacker system and ability to parallelize in order to get more compute hours out of a real hour if there is a real hour constraint. The following equation is rounded up to determine the number of whole instances required to hack a password from a user system in a given amount of time:

$$\frac{Effort\ x\ Compute\ Hours\ per\ Dollar}{Real\ Hours\ Constraint} = \#\text{Instances Required}$$

Finally, the cost of breaking a password is calculated by calculating the cost of usage of the instances over the time required for hacking. This is again, rounded up to the nearest feasible quantity of hours.

$$\#Instances\ x\ Real\ Hours\ x\ Computation\ Cost = Cost\ to\ Break$$

The cost to break the password, denominated in dollars is then compared with an estimated value of the password (the possible value associated with hacking an account) as a last step in the analysis. This analysis relies upon the user to provide an estimated value of their account but in system implementation, it may be possible to derive this figure. Computational security is then defined by the test of whether or not the effort and cost to hack is worth the value. The model structure that implements the above equations is given in the figure below:

	Android (Javasript)	Data
User System Variables	Scrypt - phone v comp	
Millions of Guesses Required	4	Guesses
Seconds per Guess	0.03	
Guesses Per Second	33.33333333	
Hacker System Variables		
Dollars / Compute Hour	$ 0.77	$
Time per Guess	5.40541E-06	
Guesses Per Second	185000	
Hacker Constraints		
Hack Value to Hacker	$ 5.00	$
Hack Hours Available	1	Hours
Calculations		
Time per User Entry	0.03	Seconds
Dollars / Compute Second	$ 0.000214	$
Time per Guess	5.40541E-06	Seconds
Effort ($/Hacked Password)	$ 0.15	$
Number of EC2 Instances Required	1	Instances
Cost to Hack in Time Desired	$ 0.77	$
Conclusion		
Password Secure?	NO	

Figure 4 Hacking Economics Model Structure

The above example scenario supposes that a password takes 4 million guesses to hack using an algorithm which uses 1000 hashes per guess. The cost per compute hour is $1.50 and the value of the hack is $5.00. Here, we see that the effort required is $3.70, but the cost to hack in the constrained timeframe is $4.50. In this example case, the password is not economically secure.

In order to analyze the computational security of Scrypt and PBKBF2 algorithms given a realistic password quality and rented computing power, test programs were developed to gather information about the performance of various user and hacking systems.

Our Method

By using the fastest implementation available for each algorithm, we look to see if an adversary could brute force a database of passwords. We first run the algorithm on a device that a user might own, an android phone. By running the algorithm on a user device we can determine what types of algorithm parameters might be acceptable to users. Then, we run the algorithm with the parameters determined from the user machine using an implementation for a computer with a graphics card, Amazon EC2 GPU instances. Since password cracking is embarrassingly parallelizable once we know how many passwords an Amazon EC2 instance can crack per second we can determine, given a budget of time and money, whether a password is likely to be crackable. The model described below allows us to calculate the amount of “effort” an adversary might have to spend in order to defeat each algorithm. This effort is a representation of how much money an adversary might have to spend to acquire a password assuming that the adversary can rent his computers in fractions of an hour and that renting n computers for 1 hour costs the same as renting 1 computer for n hours.

Description of android and scripting algorithms

In order to gain some perspective on password based key derivation, we made an Android application that measures how long it takes to run different hashing algorithms several times on the same key.

For each algorithm, we measured how long it to run that particular hashing algorithm 100000, 200000, and 500000 times on the same key. Since users expect applications to respond fairly quickly when they attempt to log in, we realized we could learn a lot about how much keys can be stretched in certain amounts of time. A pseudocode version of the algorithm is written below, note that the algorithm only measures how long the key stretching takes and doesn’t measure the overhead from initializing the required variables.

```
1. //declare variables
2. key = "hello"
3. start timer
4. for x iterations { // x = 100000, 200000, or 500000
5.   key = hash(key)  // hash = md5, sha1, sha256, sha512
6. end timer
7. return elapsed time in ms
```

The full java code can be found in the appendix at the end. Note that in the table below using PBKDF2 refers to using PBKDF2(SHA1, x, key) instead of simply iterating through SHA1 many times.

The table below shows how long it took each of the algorithms to derive a key through by repeatedly hashing some initial key.

#Iter	md5	sha1	sha256	sha512	pbkdf2
100000	5816	5963	5671	9536	6316
200000	11713	121119	11400	14838	12977
500000	29082	29177	28310	35746	34701

We found that, for the algorithms not using SHA512, that it takes 6 seconds to derive a key by iterating through it 100000 times and we found that as the number of iterations increase, that the amount of time required increases linearly. We believe that these times are likely longer than the average user is willing to wait (they are probably willing to wait around 1 second) for the key to be derived so that they can successfully log in to whatever system they're trying to access. Therefore using SHA512 about 10000 iterations is as long as a user might be willing to wait and for the others 50000 iterations is likely the right number as well.

EC2 Algorithms and Approach

In order to fully utilize EC2 as a way to crack stolen password hashes, we went with a base of the Amazon EC2 instance 'g2.2xlarge' to allow GPU computation. The SHA-2 and PBKDF2 algorithms are straightforward ports to the NVIDIA CUDA programming language. There are no special alterations to each of these algorithms, they are simply run in parallel for

higher throughput of hashes. The scrypt implementation is from an implementation meant for mining Litecoin. As such, it uses certain tricks of parallelism to calculate sections of the algorithm faster than a CPU could alone. The implementation is taken directly from the popular CUDAminer software available for Litecoin cryptocurrency mining. Below is the data for both a middle-cost GPU from a consumer laptop, and the EC2 instance numbers for a single instance of 'g2.2xlarge.'

Hackenomics Results and Conclusions

As seen in the table below, we have given the hacker a time and money budget to spend cracking a password with EC2. Whether the password is crackable will depend on the hacker's budget, the algorithm's parameters and how many password need to be tried (we assume around 4 million for most passwords based on the literature discussed previously). As can be seen the standard parameters for PBKDF2 HMAC SHA512 of 10,000 iterations is insufficiently secure for many hackers. Additionally, more than 10,000 iterations of this PBKDF2 is likely not a good idea from a usability perspective since it takes around a second to compute and users will likely not wait much longer than that.

The parameters we used in experimentation for Scrypt were not sufficiently strong (N=1024, r=1, p=1) unfortunately. In order to correct for this, we have multiplied the effort required by Guesses/second for the user in order to linearly extrapolate the effort required by the adversary to be as if the user had waited a full second for Scrypt to run. While this method is not ideal, the fast and publicly available GPU code for Scrypt was made to work with these parameters.

From our results we can conclude that given a way to quickly verify the correctness of a password stretched into a key using PBKDF2 with HMAC SHA512 for 10,000 iterations is insufficient to protect most users. Nonetheless, there are companies (such as BoxCryptor) whose entire security model relies on these types of passwords being difficult to crack. We hope that our results get people to consider the "effort", a function of both time and money, of cracking average passwords when considering key stretching algorithms and their parameters. Even more so we hope that security researchers keep these measures of "effort" up to date with the developments in cloud computing, customizable hardware, and even crypto-currencies which tend to use hash functions internally.

	Computer (Javasript) Data	Android (Javasript) Data	Android (Java) Data		Computer (Javasript) Data	Android (Javasript) Data	
User System Variables	PBKDF2 - Comp vs EC2	PBKDF2 - phone v comp	PBKDF2 - Android v EC2		Scrypt - Comp vs EC2	Scrypt - phone vs EC2	
Millions of Guesses Required	4	4	4		4	4	Guesses
Seconds per Guess	0.2	0.72	0.6316		0.085	0.03	
Guesses Per Second	5	1.388888889	1.583280557		11.76470588	33.33333333	
Hacker System Variables							
Dollars / Compute Hour	$ 0.77	$ 0.77	$ 0.77		$ 0.77	$ 0.77	$
Time per Guess	0.0001	0.0001	0.0001		5.40541E-06	5.40541E-06	
Guesses Per Second	10000	10000	10000		185000	185000	
Hacker Constraints							
Hack Value to Hacker	$ 1.00	$ 1.00	$ 1.00		$ 1.00	$ 1.00	$
Hack Hours Available	1	1	1		1	1	Hours
Calculations							
Time per User Entry	0.2	0.72	0.6316	0	0.085	0.03	Seconds
Dollars / Compute Second	$ 0.000214	$ 0.000214	$ 0.000214		$ 0.000214	$ 0.000214	$
Time per Guess	0.0001	0.0001	0.0001	0	5.40541E-06	5.40541E-06	Seconds
Effort ($/Hacked Password)	$ 0.09	$ 0.09	$ 0.09		$ 0.05	$ 0.15	$
Number of EC2 Instances Required	1	1	1		1	1	Instances
Cost to Hack in Time Desired	$ 0.77	$ 0.77	$ 0.77		$ 0.77	$ 0.77	$
Conclusion							
Password Secure?	NO	NO	NO		NO	NO	

Recommendations for the future

Since the industry standards for key stretching are PBKDF2 and Scrypt it would be great to apply our results to commonly used parameters for Scrypt and be able to run an efficient GPU implementation on EC2 GPU instances to more accurately determine whether or not Scrypt is sufficiently secure for the majority of users.

While thus far we have determined a basic metric for computational “Effort” and a process for analyzing this metric across different systems, there may be other economical factors (e.g. economies of scale and overhead) that might be relevant to incorporate into the metric. Additionally, most password based key derivation functions like PBKDF2 and scrypt try to make it difficult for an attacker to attempt getting the keys for many passwords by wasting computational resources. As we have mentioned, the issue with this is that valid users have a very limited wait time and also likely have (much) worse hardware than their adversaries. In the case of PBKDF2 GPUs speed up computation time immensely giving the adversary a large advantage. Similarly, while scrypt was designed to be harder for GPUs people have found that it is not necessarily the case. However, in both of these cases the publicly declared algorithm is meant to make particular (non-consumer) hardware slow. Perhaps though another scheme where the resource used by the algorithm is “networking” as opposed to time (PBKDF2) or space/memory (scrypt) would be able to help in addressing detectability of a hacker in addition to simply trying to make it more difficult for the hacker to break the passwords.

Acknowledgements

We would like to thank Gene Itkis at Lincoln Laboratory for his assistance and guidance in making this project happen.

Bibliography

[1] J. Bonneau, "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords".

[2] P. W. Staff, "Cloud Computing Used to Hack Wireless Passwords," PC World, 2011.

[3] C. Herley and P. Oorchot, "A Research Agenda Acknowledging the Persistence of Passwords," *Security&Privacy Magazine,* 2011.

[4] U. Banerjee, "Amazon AWS 19th Price Reduction - a Closer Look," 2012. [Online]. Available: http://setandbma.wordpress.com/2012/03/08/amazon-aws-price-reduction/. [Accessed 2014].

[5] RightScale, "Cloud Pricing Trends," RightScale, Inc., 2014.

[6] K. Higgins, "Cloud-Based Crypto-Cracking Tool To Be Unleashed at Black Hat DC," InformationWeek, 2011.

[7] H. Hosseini, "Google Slashes Cloud Prices: Google vs AWS Price Comparison," Cloud Management Blog, 2014.

[8] J. Bonneau, C. Herley, P. Oorshot and F. Stajanoy, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *IEEE Symp. S&P 2012*, 2012.

[9] "Hash function in PBKDF2," Cryptography Stack Exchange, 2013. [Online]. Available: http://crypto.stackexchange.com/questions/2851/hash-function-in-pbkdf2. [Accessed 2014].

Appendix 1: **Android Code**

```
public void buttonClick(View view){
  String password = "hello";
  byte[] hashed = password.getBytes();
  byte[] salt = new byte[16];
  new Random().nextBytes(salt);
  int iterations = 500000;
  String iteration =
   ((EditText)findViewById(R.id.numiterations)).getText().toString();
  KeySpec spec = new
PBEKeySpec(password.toCharArray(),salt,iterations,160);
  long start = System.currentTimeMillis();
  try{
    SecretKeyFactory f =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    start = System.currentTimeMillis();
    byte[] hash = f.generateSecret(spec).getEncoded();
    ((TextView)findViewById(R.id.moop)).setText(hash.toString());
  } catch(Exception e){
    ((TextView)findViewById(R.id.error)).setText(e.toString());
  }
  long totalTime = System.currentTimeMillis()-start;
  setContentView(R.layout.fragment_main);
((TextView)findViewById(R.id.time)).setText(String.valueOf(totalTime))
;
}
```

Appendix 2 : EC2 Code

http://hashcat.net/oclhashcat/ (used for benchmarking)

https://github.com/dave-andersen/keplerminer (EC2 scrypt implementation derived from this)

Appendix 3 : JavaScript Code

Stanford JavaScript Crypto Library (SJCL) (https://github.com/bitwiseshiftleft/sjcl) for PBKDF2

js-scrypt (https://github.com/tonyg/js-scrypt) for Scrypt

Appendix 4: Historical Cloud Computing Prices for Two Popular Instance Types

	Small – Linux – N. Virginia (m1.small)	*Quadruple Extra Large (/cgl.4xlarge/c3.4xlarge)*
1/1/2010	*$0.09*	*$3.16*
3/12/2012	*$0.08*	*$2.50*
12/1/2012	*$0.07*	*$2.10*
4/14/2014	*$0.04*	*$1.50*