12-31-2016

Investigating the Spatial Complexity of Various PKE-PEKS Schematics

Jacob Patterson
Rose-Hulman Institute of Technology

Investigating the Spatial Complexity of Various PKE-PEKS Schematics

Introduction and Context:

With the advent of cloud storage, people upload all sorts of information to third party servers. However, uploading plaintext does not seem like a good idea for users who wish to keep their data private. For instance, an employer may wish to upload their payroll information to some cloud storage service in case their local storage is lost due to an on-site accident. If we uploaded the payroll as plaintext, everyone would know how much everyone else is getting paid. This clearly isn't in the best interests of the company, and so the boss may encrypt what they upload so that only they can gain access to the actual payroll.

This works great for information possessed only by the boss. Let's explore the scenario in which everyone has little bits of information that they need to give the boss, but nobody else. Say part of our company's policy was to have every employee submit evaluations of other employees. We could use the same idea where we encrypt everything using the same key. However, if an employee is given access to decrypt their part of the message, they can decrypt everyone else's as well. This compromises the idea of keeping evaluations known between the boss and the employee who wrote them. To solve this dilemma, we could allow the boss to generate a key pair consisting of a public encryption key and a private decryption key for their use only. We can envision this as giving everyone an envelope in which to store the evaluation, but only the boss has the letter opener. Now our boss can privately decrypt any evaluations uploaded to the cloud storage service using the public encryption key.

Side Note: Another thought would simply be to have everyone email the boss their evaluation. This skips over the cloud storage and encryption completely. However, there is a significant complication that may arise from use of this method. We may run into an instance of the boss' local storage being too small to hold all of the documents. In this scenario, we'd need the evaluations uploaded to the cloud storage service, and we couldn't ask the boss to be a sort of middle man in which they gather them, then encrypts and uploads them.

Now let's say the company in question has several thousand employees. The boss will be unable to keep all of the information taken from these evaluations in their memory, and will need some form of sorting system. For example, one may sort magazine subscriptions by their title. If we were to add a sort of tag at the end of each evaluation that identifies the document, the boss could sort them by department, name, or whatever criteria is provided on these tags. We could encrypt these tags in the same way we encrypted the documents and just claim it as a part of the document. However, this doesn't allow us to sort the documents any more effectively than before as we'd still have to decrypt the document and then somehow mark or store it locally. On the other hand, if we didn't encrypt these tags at all, everyone who has access to the data in the cloud will also know the information provided by the tags. This could be particularly compromising if the tags subjects were something along the lines of Evaluator and Evaluated. Suddenly, other employee's would know that Jane, Alice, Bob, and John were somehow involved because all of their names were found on the same evaluation paper. Frequency patterns based on name could be derived and every person in the company would know who evaluated who. The problem, then, is that we need some unique way to encrypt these information tags so that the server can still sort the documents but those who have access to the data in the cloud do not know the information of the tags. Current solutions to this problem in literature involves integrating Public Key Encryption and Public key encryption with keyword search techniques. The intent of this paper is to analyze the spatial complexities of various PKE-PEKS schemes at various levels of security and discuss potential avenues for improvement. We start this investigation by identifying and explaining the more technical information relevant to these schemes.

Evaluating the technical aspects of a generalized PKE-PEKS Schematic:

Suppose we have a message we wish to encrypt and upload to the cloud storage service. We first need a way to jumble up our message into something unintelligible. Since the computer interprets all words as a string of numbers, let's think about this problem in terms of numbers. When we have a prime number, it's not divisible by anything, aside from itself and the number one. So this could be a sort of secret number that only the user knows. Even though there are a lot of numbers, a computer can iterate through millions of numbers a second, which could easily lead to a brute force security breach where they figure out your number. We can't just infinitely randomly generate a string of numbers, stick them together and call that your number either, as that does not guarantee it to be prime. That means we're going to need a bigger number that still possesses the security through uniqueness of a prime number. If we multiply two primes together we create a potentially enormous number that is only divisible by 1, the two primes, and itself. Not only does this drastically increase the size of our number, we hardly make a dent in the security aspect of said number. In fact, we can even make that publicly known. For instance, say I spewed out the number 4767691. Could you guess which two numbers it was made of? If you were using a computer, you could figure out the answer of 1777 and 2683 by dividing 4767691 by 1, 2, 3, 4,… until we hit 1777. You could even speed this up to only look at odd primes for your list to check (3, 5, 7, 11, … , 1777). However, if you were solving this by hand it would take quite a while, don't you think? This can be paralleled to using ridiculously large primes (millions of bits long) as the parameters for our product. If a computer were to brute force this by checking each number starting from 1, current computers wouldn't solve this until after the heat death of the universe. This is actually the basis for the RSA algorithm, which can be used as an encryption function to jumble up a message using our primes p and q. Integer factorization and the RSA challenge were fundamental aspects of proving the computational security of these large semi-primes. As it stands, RSA-768 is the largest semi-prime to have successfully been factored, and industry standard currently recommends public key lengths of 2048 bits. Of course, some additional logarithmic and exponential operations occur, but these sorts of details are out of the scope of this paper. The key point here is that these types of algorithms serve as the basis for PKE schemes. We're going to have to keep track of a few things when using RSA. Firstly, employing the RSA algorithm gives us a public key and private key pair, which we will call PubKey and PrivKey respectively. When we encrypt our plaintext with our PubKey, we'll get an output of ciphertext, which is really a technical term for our jumbled up message. We can use our PrivKey to decipher any jumbled up message we receive back into plaintext. However, we still need to attach our keywords to our encrypted text.

The PEKS portion of this system is a bit more complicated. To start, we will discuss a simple version of PEKS to develop a basic understanding for the reader. If we have a message we wish to encrypt with keywords, we have a few goals in mind. First, we want these keywords and their functions separated from our original message. In order to do this, we can create a separate public/private key pair and encrypt some random messages that don't really mean anything using our new keys. Second, we want a way to filter out all of the input that is not a keyword. Basically, if I say my message will have a keyword of 'apple,' I don't want bumblebee to return a positive result. This would be nonsensical, since I really only wanted my data to be related to the word 'apple.' We call this a trapdoor function, as any words that fail the test will fall through the trapdoor. Keeping these goals in mind, here is a simple implementation.

Boneh et al.'s trapdoor permutation schematic:

Create a public/private key pair for each keyword in our set of keywords or 'keyring' if you will. Now we will encrypt a random gibberish message using each of the public keys we generated from our keywords. The trapdoor function will simply be decrypting our jumbled message with a private key generated from the keyword we wish to test. If the unjumbled message matches up with the plaintext we sent the server, then we will have a keyword match and thus pass the trapdoor test. To get this to work, we have to send the server both the encrypted gibberish and the regular gibberish. We then have to give our sender the specific keyword. The sender feeds this keyword to the server, it generates a test private key using a stored keyword generator, and if the decrypted gibberish is the same as the regular gibberish, then our keyword is a match and the PKE encrypted message can be sorted appropriately.

Costs:

Before we begin with the Cost analysis, it is important to note that the RSA encryption algorithm has a strong weakness in that an attacker can encrypt large libraries of known plaintext messages and store its corresponding ciphertext as well. When we have a deterministic algorithm an attacker could encrypt millions of plaintext messages to generate patterns and then determine the decryption method by running a frequency analysis with the English language, slowly developing a plaintext-ciphertext dictionary. This is particularly dangerous because it could take place offline, as our encryption key is publicly known.

To fix this vulnerability, 11 bytes of random text is added for every 245 bytes of message. This is derived because the maximum size of message that can be encrypted using the RSA encryption technique with a key size of 2048 bits is 245 bytes according to PCKS#1. While this is an archaic security measure, we use it as an example and refer to more effective security measures in the Related Works section. After the padding technique is encrypted, which will not be elaborated upon, the message size's end result will be the absolute maximum size of 256 bytes for each 2048 bit key. This means our message will automatically take up 256/245 or roughly 1.0449 times the space of a plaintext message of the same length.

Hypothesis:

We will now observe the practicality of such a system, by looking at how the spatial complexity grows with modification to the number of keywords. We will have a few static costs and assumptions.

1. The PKE encryption of our message component will be a constant based on the size of the message set forth by the user.
2. Any private/public key pair will cost 4096 bits total, 2048 for each key.
3. Our entire chunk of encrypted information will consist of the PKE encrypted message and the PEKS encrypted message and message chain.
4. The keyword generation function and trapdoor function computational costs are not considered, as they are a onetime constant for the server of negligible size.

Based on these observations, I believe there may be some form of optimizing ratio keywords to message size that is most spatially efficient.

Method:

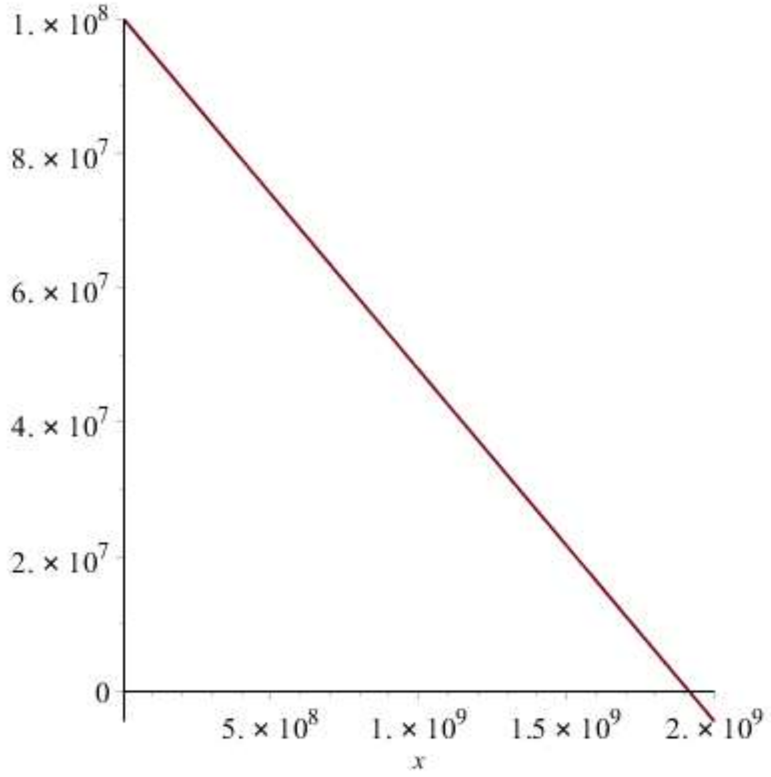
To test this claim, a function was constructed using the above baseline assumptions. If the total package size is restricted to 2GB, an inequality presents itself, as all of the information inside the uploaded package must be less than or equal to 2 billion bytes. Now the question becomes what must be sent to the server and the relative costs of each of these items. The PKE encrypted message must be sent, and there is a slight additional overhead for encrypting a package of m bytes using the RSA encryption technique, as described above. Looking at the PEKS portion of the package, we have to upload both a junk message and the encrypted version of said junk message. This relates our message size x to the total cost f(x) as follows:

$$f(x) = 256\cdot\left(1 + \left\lfloor\frac{x}{245}\right\rfloor\right)$$

If we fix keyword cost at a reasonable 10 bytes per message, we have a 20 byte cost scalable to the number of keywords we wish to include. We must then have a constant- $20\cdot y$ added to our function. When we make an inequality out of these expression with our 2 GB cap in mind, we acquire the following linear function:

$$20\cdot y + 256\cdot\left(1 + \left\lfloor\frac{x}{245}\right\rfloor\right) \leq 2000000000$$

This expression can then be graphed as follows:



Since we were dealing with an inequality, any values below the line and above the axes would be a valid sized package to send to the server. However, because this relation is linear, there is no clear optimal ratio between keywords and message size or any other form of spatial optimization. We also note here that the message size restriction on x has negligible effect as our message package becomes large. While this may seem like a disappointing result, it is actually quite important. Research in other fields such as search engine optimization and language construction have independently come up with the idea that the number of keywords has no significant impact on the ability of said keywords to relate to the given message. That is to say we won't necessarily summarize the message any better with 5% of our words being translated into keywords than 95% of our keywords being translated into keywords. This idea in tandem with the linear relation shown above gives significant flexibility to the uploader of the package. They can arbitrarily pick a set of keywords they deem fitting for the project at hand without having to sacrifice any spatial efficiency.

Boneh et. al's Bilinear Maps analysis:

Generating the PEKS system here uses bilinear maps, relying on the complexity of the Diffie-Hellman problem for its security. In this instance, Boneh et. al describe the construction as "two groups G_1,G_2 of prime order p and a bilinear map e : G_1 X G_1 $\rightarrow$ G_2 between them." Groups are simply sets of elements closed under any binary operation. In this instance, we want our G_1,G_2 to be closed under the cross product as given by the definition of our function e. Another part of the set-up for this schematic are the two hash functions. The first, H_1, takes a binary string of any length and maps it to an element in G_1. The second hash function, H_2, takes an element from G_2 and maps it to a binary string of length log p.

In order to generate the public key, compute some integer α between 0 and $p – 1$, inclusive. We also create a generator g from G_1. The public key consists of our generator and the generator raised to α. For ease of use, we'll call this $h = g^{\alpha}$. The private key is our α in question.

Now that the public private key pair is generated, we compute the encrypted output. To start, we define $t = e(H_1(m),h^r)$ where r is some random integer between 0 and $p – 1$, inclusive and m is our message. This formula basically runs our message through the first hash function, mapping it to a string of length p, and then computes the cross product between the string and our h raised to the random r. The output to be stored on the server, then, will be g^r and $H_2(t)$. The trapdoor in this schematic is $H_1(m)^{\alpha}$. To see if a keyword is within the encrypted message, simply run the second hash on the cross product of our trapdoor and the test word. If this lines up with the expected test output, which is the second string stored on the server, we've got a match.

In Comparison to Trapdoor Permutation Schematic:

Spatially, the benefits of hash functions are two-fold. The first benefit is that any message of length n will map to another message of fixed length, which was defined by our group G_1. Naturally, there will be an issue of aliasing, where multiple inputs map to the same string in G_1, but a cryptographically secure hash function will not allow this aliasing to be easily predictable. Secondly, this fixed length message can be scaled up or down depending on the level of security requested by the user. However, there is a time tradeoff in that larger messages take longer to produce. These hash functions help with the spatial complexity of the decryption/encryption computations. While we are only concerned with the size of storage of these functions, it is important to note the spatial benefits in computation as described above.

The information stored on the server consists of our generator raised to the r, which is within G_1, and the second hash result of our message. This can be expressed by the expression:

$$x + \log x$$

for the chosen prime order of x. In this case, we see that our keywords stored take $x + \log x$ as opposed to simply x. However, this is of very little effect, which is obvious when realizing an example. Take, for instance, the log of 100,000 is 5. The comparison, then, would be 100,000 units with the trapdoor permutation schematic versus 100,005 units for the completed bilinear map schematic.

Chen et al's Generic IBE Construction:

Context:

The work of Yu Chen, Jiang Zhang, Dongdai Lin, and Zhenfeng Zhang contribute a general purpose solution that allows implementation using any Identity Based Encryption, abbreviated IBE, scheme. These schemes have a private key generator that hands them out to various users. In our original analogy, the manager would generate a private key and pass that to any new employees. These keys typically either involve unique information or are generated using unique information of the employees and are then certain to be unique to the user. One other important system Chen's PKE-PEKS Construction takes advantage of is a one time signature. One time signatures can be used by message senders to verify they actually did send the message. Since each signature is only used once, the algorithms and functions used to generate the signatures can be vulnerable to methods of attack that rely on a multiple use case, such as frequency analysis. Any additional details about IBE or one time signatures can be found in outside resources.

Implementation:

In order to encrypt our message, we generate our one time signature using whatever the IBE private key was created. We then encrypt our message m and keyword w using unique identities. However, we use the same private key generated earlier in order to link the keyword with the message. Lastly, the encrypted keyword message pair is signed and the set of verification key, encrypted message, encrypted keyword, and signature comprises the ciphertext.

For decryption, you'd first use the verification key and the encrypted keyword message pair to check that the signature is valid for the ciphertext. If the signature is valid, then use the plaintext's identity and pull out a partial key. Use this partial key to then decrypt your ciphertext into the plaintext message. The same type of extraction can be used to decrypt keywords given its identity and private key.

Generalizations:

Based on this generic construction, we can break down spatial costs of any IBE based PKE-PEKS system in the following way. You will generate a ciphertext that consists of the encrypted message, encrypted keyword, signature, and the public verification key. The verification key and encrypted message will both be of length x. The keyword length will be $k < x$, and the signature will be some value, typically hashed to a variable length determined by the implementer. In that sense, we can generalize this to $2x + k + h(s)$, which flows nicely out of the results we received looking at Boneh et al.'s specific implementations.

Conclusion:

There exist several variations and improvements of integrated PKE-PEKS systems beyond the three previously analyzed. We find most of the improvements involve more theory and models, such as El Gamal, which are beyond the scope of this paper. Expanding on this spatial analysis requires looking further into these more complicated constructions. Based on the analysis provided, there exists a growing spatial cost, albeit limited in scope, as one adds security and functionality to the PKE-PEKS system. One important open question remains to find such a system that produces output no larger than a standard encryption system's ciphertext. However, this problem in itself involves finding some way to ignore block cipher standards, such as having high confusion and diffusion. Furthermore, the temporal complexity of these systems should also be investigated and tradeoffs between space, time, and security examined. The proposed expansions should realize a practical assortment of various PKE-PEKS schemes that can be selected, depending on design specifications and project needs.

References

[1] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, “Public Key Encryption with keyword Search,” In proceedings of Eurocrypt 2004, LNCS 3027, pp. 506-522, 2004

[2] M. Bellare and P. Rogaway, “Optimal asymmetric encryption – How to encrypt with RSA,” Eurocrypt 94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santis ed, Springer-Verlag, 1995.

[3] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2014. A Survey of Provably Secure Searchable Encryption. ACM Comput. Surv. 47, 2, Article 18 (August 2014), 51 pages. DOI=http://dx.doi.org/10.1145/2636328

[4] Aiden Bruen and Mario A. Forcinito. 2004. *Cryptography, Information Theory, and Error-Correction: A Handbook for the 21st Century*. Wiley-Interscience.

[5] Yu Chen, Jiang Zhang, Dongdai Lin, and Zhenfeng Zhang. 2016. Generic constructions of integrated PKE and PEKS. *Des. Codes Cryptography* 78, 2 (February 2016), 493-526. DOI=http://dx.doi.org/10.1007/s10623-014-0014-x

[6] Dan Boneh and Matthew Franklin. 2003. Identity-Based Encryption from the Weil Pairing. *SIAM J. Comput.* 32, 3 (March 2003), 586-615. DOI=http://dx.doi.org/10.1137/S0097539701398521

[7] Dan Boneh, Ran Canetti, Shai Halevi, and Jonathan Katz. 2006. Chosen-Ciphertext Security from Identity-Based Encryption. *SIAM J. Comput.* 36, 5 (December 2006), 1301-1328. DOI=http://dx.doi.org/10.1137/S009753970544713X