Evolution of FLUSH + RELOAD

Preston James Miller
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
mill6528@morris.umn.edu

ABSTRACT

Side channel attacks have been developing and growing more powerful to work around the fixes for them. The FLUSH + RELOAD attack is an excellent example of how attacks evolve over time to become more powerful. Originally developed in 2011 by Gullasch et al, the FLUSH + RELOAD is still getting variations to this day. The original attack was designed to gain the secret key of AES encryption, however it was very slow. A faster version of the attack targeting Square-and-Multiply algorithms was made by Yarom et al. This attack was much faster and less error prone than the original. Another attack came to light building upon Yarom et al's attack that attacked the OpenSSL implementation of ECDSA. This attack used lattice techniques to discover the secret key. The last attack that is discussed is an adaptation that targets Platform-as-a-service cloud systems to obtain user data.

Keywords

Side channel attack, timing attack, FLUSH + RELOAD

1. INTRODUCTION

Data encryption is a practice that all companies do if they need to store sensitive information. Typically if there is not a way to bypass the encryption algorithm itself, an attacker will resort to brute force attacks to try to guess the key. The other way to go about breaking encryption is through side channel attacks. Side channel attacks are powerful attacks that target the physical implementation of a system rather than the code; this makes side channel attacks difficult to prevent. The FLUSH + RELOAD attack is an attack that was made to find the secret key of a cryptographic system using the CPU cache. With the secret key, any encrypted data can be easily decrypted, meaning it is easier for the attacker to retrieve sensitive data from a machine. Making sensitive data easier to access is obviously a big issue with how much sensitive data people have out on the web currently. Knowing how these side channel attacks work can help bring stronger security in the future. The original FLUSH + RELOAD attack was developed in 2011 by Gullasch et al. Yarom et al extended this attack so that it did not have to run on the same processor. Benger et al extended the attack by Yarom et al by adding lattice techniques to obtain the private key from the OpenSSL implementation of ECDSA. The last attack talked about in this paper will be the adaptation by Zhang et al. The attack by Zhang et al can gain user information from Platform-as-a-service cloud systems. These attacks will be discussed in the order they were introduced.

UMM CSci Senior Seminar Conference, April 2016 Morris, MN.

2. BACKGROUND

2.1 Cache

The cache is a part of the CPU that stores data for faster access later. When an operation is preformed, most of the time it gets stored in a cache so it does not have to be recomputed. When a computer is processing data, it will check in the cache to see if it has accessed the given data before. If the operation has been preformed before and was found in the cache, it is called a cache *hit* otherwise, it is called a *miss* [5]. When a cache miss occurs, the data is added into the cache for future access. Caches are in several places on a computer, but for the remainder of this paper the cache will refer to the cache on the CPU. Some CPUs have multi-level caches. In most multi-level cache architecture, the *last level cache* is shared between all levels. This means that once an item is removed from the last level, it will be flushed from all of the levels above it. A command that is used for flushing the cache is the $clflush$ command. $clflush$ is an unprivileged instruction, meaning that it does not require administrative privileges to execute, that tells the CPU to remove items from its cache.

2.2 Cryptography

Cryptography is the practice of encrypting data for secure storage and transport. Encrypting involves a *secret key* to transform the data into something that is essentially random until you use the correct key, which could be different from the one that did the original encryption, to decrypt it. A *block cipher* is an algorithm used for encryption. A block cipher encrypts a short fixed-length block of data. To use the cipher on a larger data set, it is used repeatedly on small blocks of the larger data set.

2.3 AES

The *advanced encryption standard*, commonly known as

AES, is a block cipher that was developed in 1998. AES can have three different key sizes given to it. It can have a 128 bit key, a 192 bit key, or a 256 bit key. The size of the key determines the number of times the input will be transformed by the algorithm. With a 128 bit key, the data goes through 10 iterations, while the 192 bit key gets 12 iterations and the 256 bit key gets 14 iterations [4]. Before AES starts iterating, it breaks the input into 16 byte segments and treats them as a 4×4 matrix, this matrix will be referred to as A. AES then preforms the *AddRoundKey* step. The *AddRoundKey* step takes a subkey of the secret key, which is defined by a scheduler, and converts it into a 4×4 matrix of its bytes, this matrix will be referred to as K. *AddRoundKey* then takes A and K and adds each corresponding entry together using XOR, which is equivalent to saying it adds them and then takes the result modulo 2 [4]. AES then begins its cycles. Each cycle starts with the *SubBytes* step. The *SubBytes* step has a fixed lookup table that it will use to modify the entries in A. Once all of the entries are replaced, the *ShiftRows* step begins. The *ShiftRows* step cyclically shifts each row by a given offset. After the matrices are cycled, the *MixColumns* step is applied. The *MixColumns* step multiplies each column by the value of a function $c(x)$. After that the *AddRoundKey* step is preformed again. These steps are repeated until the last iteration of the cycle where the *MixColumns* step is dropped [4]. AES is the only publicly accessible cipher that is approved by the National Security Agency, it is used in many commonly used encryption libraries such as OpenSSL [4].

2.4 Square-and-multiply

The *square-and-multiply* exponentiation algorithm is a commonly used component in encryption algorithms. It is used in many commonly used encryption packages such as GnuPG in versions before 4.1.14 [7]. This algorithm is a fast way to compute $x^e \bmod m$ where e, m, x are integers. The square-and-multiply algorithm starts with some value of x. The algorithm is passed in an exponent e in its binary form. It then loops over the length of e. In the loop it starts out by squaring x to replace the previous value of x. After x is squared, x gets replaced by the modulus of x and m, where m is a value passed into the algorithm. If the bit that is being looped over is 1, them x is multiplied by the original value of x. Finally x becomes the modulus of x and m and the loop continues [7]. This method is able to find large powers of numbers fairly quickly. Table 1 shows the steps the algorithm takes. The first column is the result to be used in the next operation, and the operation is what is being done at the current step. The exponent column is the bit representation of the exponent and the least significant bit determines what operation is done. Every row adds one bit onto the key or changes the least significant bit to a one. The end result of gives 58, which is also the solution to 5^{83} mod 101.

2.5 Side Channel Attacks

Side channel attacks are attacks that target the physical implementation of a system to get information out of it. There are many different kinds of side channel attacks that use different information to get the information the attacker desires [6]. Some examples of things can can be used in a side channel attack are power consumption, checking the electromagnetic radiation coming out of the machine, and even the sounds that a machine makes can be used for a side channel attack.

x	Operation	Exponent
5 mod 101	None	1
5^2 mod 101	Square	10
25^2 mod 101	Square	100
$19 * 5$ mod 101	Multiply	101
95^2 mod 101	Square	1010
36^2 mod 101	Square	10100
84^2 mod 101	Square	101000
$87 * 5$ mod 101	Multiply	101001
31^2 mod 101	Square	1010010
$52 * 5$ mod 101	Multiply	1010011

Table 1: Square-and-Multiply with $x = 5, e = 83, m = 101, e = 1010011_2$

2.6 Timing Attacks

A type of side channel attack are timing attacks. Timing attacks monitor how long different operations take on a machine and use that information to guess the secret key. Since every operation by the machine takes a certain amount of time, the attackers can work backwards to recreate each step until they get the desired input [3]. *White noise* is commonly used to make it harder for timing attacks to be implemented. White noise puts random operations inside the algorithm to increase the number of steps and throw off systems that might be trying to look in onto the process.

3. FLUSH + RELOAD

The FLUSH + RELOAD attack was developed by Gullasch et al in 2011. The attack is a variation on a timing attack and also uses the CPU cache to gain access to information. The attack uses four stages, one to spy on the client to get sections of the secret key, another to cut out the white noise from the machine, the next to find candidates for parts of they key, and the last one to generate the secret key.

3.1 Setup

The setup for this attack in practice would most likely be a focused malware attack. It is also assumed that the attacker would have a machine with the same configuration as the victim machine so they could set up the attack and make sure it works correctly. Since the attack does not require administrative privileges, the malware can affect any user [2]. This setup for the attack also assumes that the victim is encrypting with the AES block cipher. This attack requires that it runs on the same core as the AES process, this can be done by exploiting a bug with processors that use the completely fair scheduler. It needs to run on the same core because it accesses the first levels of the cache that are not shared between cores.

3.2 Spy Process

The spy process is the first stage of FLUSH + RELOAD. The purpose of it is to make it so the CPU can only run small sections of code so that parts of they key can be discovered. The spy process is the only process where an end user experiences any delay. Once the spy process is activated, it releases many threads in order to clog the system

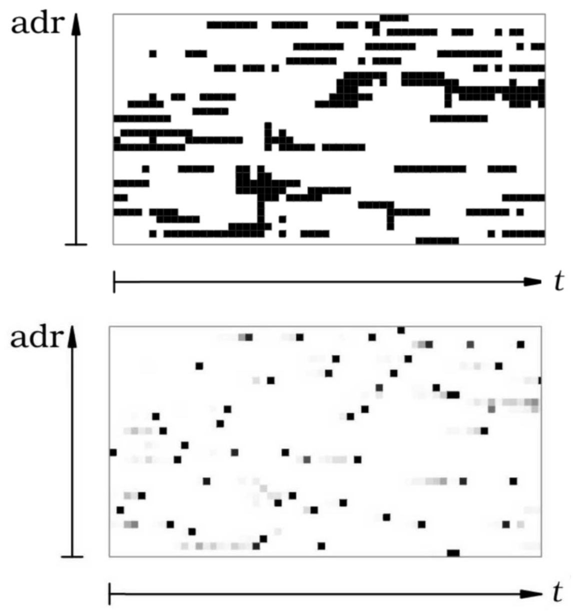

Figure 1: The top picture is all of the noise the CPU produced during the spy process, the bottom is the most likely candidates to be from part of the encryption process, with darker squares having higher probability (taken from [2])

and make processes run one step at a time. Each thread monitors the time and all of the data access that happened before it. Every so often, these threads will execute a section of code that will load the the values of the table for the *AddRoundKey* step of AES. If one of these values create a cache hit, then it stores when these hits and misses occur for later use [2]. After this process is completed, the information gets passed onto another process to analyze the data. When testing this method Gullasch et al discovered a jump from 10ms for 100 encryptions to 2.8 seconds on average, due to the threads clogging up the system, which they say most people would just assume their operating system was doing something else and slowing down their computer [2].

3.3 Cutting Out White Noise

The next process uses the data found in the spy process and figures out which discovered cache hits were produced by the encryption process and what data is from other processes running on the machine. This step and the steps after it can be run on a different machine if the data is exported to the different machine. However, they can also be run on the victim machine. This process is done using neural networks, which give weights to different values to learn what data is useful and what data is not. The neural network is trained to figure out with high probability when references to the table where actually used. Figure 1 shows just how effective neural networks are at finding when the hits were used in the encryption process. Once most of the cache hits are discovered, the data is sent to the next part of the attack. Neural networks can lead to some false positives or they could miss some data accesses, however, since there is no real definitive way to prove a cache hit was from AES and not another process.

3.4 Searching For Key Parts

Searching for key parts requires finding the start of each *SubBytes* step. This is done by assuming every entry is the start of the *SubBytes* step. It then calculates how often the value lies in a predetermined set. From the discovered frequencies, the probability of the element being part of the key can be discovered [2]. These probabilities are sent over to the last step in the process.

3.5 Assembling The Key

This is the part that actually assembles the secret key. Each partial key fragment is given a score based on how many times it came up in section 3.4. The attack then tries to generate a key with the highest possible score. It starts out by fixing one part of the key with a high probability, then adding more on until the score gets better. To figure out if it has a better score, the process takes the standard AES key scheduler and compares the part of the key to each of the different sections. Once the unused element with the highest score has a lower score than the entire generated key, the process quits because it found the best candidate for the key.

3.6 Countermeasures

The best and most obvious countermeasure to this attack would be to disable the CPU cache since that is how the data is obtained; however this would fail in practice because getting memory from RAM instead of the cache is 10 to 100 times slower [2]. A less obvious countermeasure would be taking away high resolution timers on the CPU [2]. This would make the attack very noticeable because it would not be able to stop certain threads from executing if nothing has happened since the last spy thread was executed. This would mean that every thread would time loading the AES table from cache then flush out the table, even when only spy threads have been executed since the last spy thread. This would slow down the attack to the point where it would be easily noticed because a simple encryption could take minutes to complete. The downside to this would be that no other processes could get this information, so it could break some existing programs relying on this information.

4. YAROM ET AL FLUSH + RELOAD

Yarom et al developed their own adaptation to the FLUSH + RELOAD attack in 2014. This attack uses the last level cache, which is the cache level that is shared between all the cores of a CPU. This attack extends the original by being able to attack systems running on separate virtual machines as well.

4.1 RSA

This attack is implemented to steal the private key from the RSA encryption system. *RSA* is a public key encryption system, meaning it has one key to encrypt data, which is public and another private key that will decrypt the data that is returned. RSA is used in GnuPG, meaning it uses the square-and-multiply algorithm discussed in section 2.4.

4.2 Spy Process

The spy process is much faster than in the original attack. The spy process no longer has to start up threads to spy on the victim. Instead it shares memory with the victim. It then flushes the data from the cache and waits. It tries to

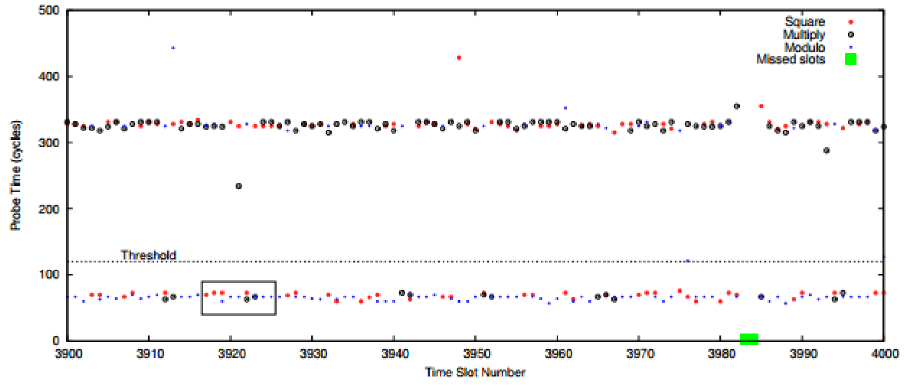

Figure 2: Time measurements taken form the spy process, visualizing how it decides what the victim process used. (taken from [7])

read the data that it recently flushed from the cache and times it to see how long it takes. If the read was fast, then it knows that the victim used that block of memory because it had to have been put back in the cache, while if it was slow it knows that that portion was not used [7]. Figure 2 shows the time difference in CPU cycles between values that were found in the cache and values that were not. If an operation was below the threshold line, then it is assumed to have been used since it was last flushed from the cache.

4.3 Finding The Key

With the data gathered from the spy process, the attack can figure out the bits of they key. Since the spy process can detect when the victim does certain operations finding the bits of the exponent are relatively easy. If the discovered processes does not multiply, then the bit is a 0, otherwise it is a 1. An example of finding the bits would be in Table1, without looking at the exponent column, you can figure out when a 1 came up based on when the multiplication step is used. The problem comes when there are missing bits that the spy was unable to catch. These sections can be found by observing another process most of the time. Since it is very unlikely that the same bit will be missing on multiple executions, having the spy observe multiple cycles will help the find all of the missing bits. Inspecting the data manually also helps find missing bits. The program is not smart enough to decide what happened when there is a bit missing, but if a person observes the data, they can reduce the missing bits by 25% to 50% [7].

4.4 Comparisons

The version of FLUSH + RELOAD Yarom et al created is much more powerful than the original. With this attack it is possible to find the key within one encryption without it being apparent to a user on the victim machine. This is because it uses the last level cache, so it can and will run on a different core than the victim process. Being on a different core means that the attack is much more flexible as well because it doesn't rely on luck or bugs in the processors scheduler in order to execute [7]. This attack also cuts out the use of neural networks, which was a very slow and complicated part of the original attack. Neural networks caused a few false positives as well.

4.5 Countermeasures

This attack actually had a countermeasure implemented in GnuPG version 1.4.14 [7]. The implemented countermeasure was the square-and-multiply-always algorithm. This algorithm is almost identical to the square-and-multiply The difference is it always multiplies, however the value is only assigned to x if the current bit of e is 1 [7]. Similarly to the original attack, restricting the use of the $clflush$ command would also make the attack harder to impossible to use. Even restricting it to an administrative command only would make the attack harder to use. This is because the victim account would have to have administrative privileges, and even then the victim could have to give explicit permission through a popup or other means in order for the code to run. This attack also does not work on certain cores where when data gets flushed from the last level cache it does not get flushed from the other higher level caches [7]. This is because there is no way to know how and when the victim called a function with great certainty, so the data would be very sporadic and inaccurate.

5. FLUSH + RELOAD WITH LATTICE

This attack is built on the attack described in section 4. This attack is used to attack the OpenSSL implementation of the ECDSA algorithm [1]. This attack was also developed in 2014 with help from one of the developers of Yarom et al's FLUSH + RELOAD attack.

5.1 ECDSA

Elliptic Curve Digital Signature Algorithm, or*ECDSA* for short, is an algorithm that is somewhat similar to square-and-multiply. Instead of encryption, ECDSA is used to sign messages and verify signatures, meaning that it can verify identity for something like bit coin to make sure the coins go into your wallet and not into anther person's wallet. One difference between ECDSA and square-and-multiply is that ECDSA works using elliptic curves. The specifics of this algorithm are outside the scope of this paper. The important part of this attack in the scope of this paper is that the OpenSSL implementation of ECDSA uses wNEF to calculate a given point, (x, y) on the elliptic curve, which is where this version of FLUSH + RELOAD will be attacking.

5.2 wNEF

wNEF is the algorithm used by OpenSSL to calculate the (x, y) pair, which is a point on the curve that ECDSA uses. The wNEF uses small amounts of preprocessed information and the fact that addition and subtraction have the same cost on the curve to improve performance over the binary method of point multiplication [1]. This method can be spied on to see when the least significant bits of the variable d is 0. Using this information the attack can get at least one bit of d, which can be expanded to more bits, with $\frac{1}{2x}$ probability [1]. These are found when a *group double* or a *group add* happens. A group double happens when the observed bit of d is zero, this doubles the value of the point Q. A group add happens when the observed bit of d is non zero. When this happens Q gets doubled and added to some computed value. Since these operations take different amounts of time, a program can find when a double or an add is completed by timing these operations.

5.3 Spy Process

The spy process is almost the same as the spy process used in the attack by Yarom et al. The spy only differs in what it is looking for; instead of looking for the square-and-multiply algorithm, it is looking for the group adds and doubles from the wNEF algorithm. Once the spy process gets the bits of d, the information is passed onto the lattice attack to process the information.

5.4 Lattice Attack

Since wNEF only leaks two bits per run on average, the lattice attack figures out the key algebraically [1]. To set up the attack, they set a fixed public key. Using the information from the spy process they get that $k_i \equiv c_i - n (\bmod 2^{l_i})$ where k_i is the private key l_i is the number of known bits, and c_i is some constant. They know that $k_i + n$ is the length of the known run of zeros in the least significant bits, which will be denoted as z_i [1]. Since $k_i + n$ is divisible by 2^Z for some Z, they pick values so that $z_i \geq Z$. From the information discovered by the spy attack, the lattice attack is able to find out when a group double or a group add happens with only a $.55\% - .65\%$ error rate where the value is unknown. They can then put this information into a vector and solve it using the closest vector problem to get the secret key [1].

5.5 Countermeasures

One simple countermeasure to this attack is to use a new key every transaction, which is currently the recommended approach [1]. This would work because then the attack could not get enough information of the constantly changing key, but since not everybody follows these guidelines, it leaves them vulnerable to the attack. Another way would be to not use some precomputed data in wNEF and instead use another algorithm that would be able to compute the values so that the static values would not always be in cache.

6. FLUSH + RELOAD IN THE CLOUD

The FLUSH + RELOAD attack was extended by Zhang et al in 2014 to work on a *Platform-as-a-service (PaaS)* cloud. A PaaS cloud is a cloud that allows tenants to execute interpreted source code, such as PHP, Ruby, Java, or any other language [8]. This attack does not target any particular encryption algorithm, however, it does target data from users using other services on the cloud. Using this attack Zhang et al were able to discover how many items were in a victim's shopping cart, gain password reset links to hijack a user's account, and to break XML encryption schemes [8]. Only obtaining password reset links will be discussed in this paper in depth.

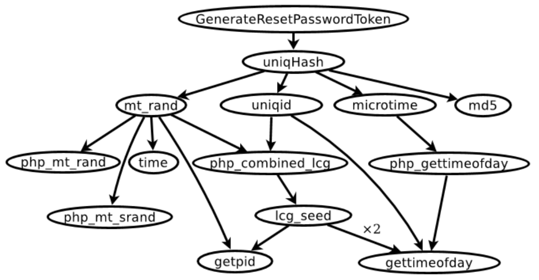

Figure 3: Common function calls in generating password resets links in PHP(Taken from [8])

6.1 PaaS Cloud

PaaS cloud services usually have more than one application running on them at a time, usually from different customers. To isolate each instance, the PaaS provider provides one of the following services. *Runtime based isolation* is an isolation technique that allows services to share the same runtime environment, but have different programs running. For example if two different Java programs are running on a machine, they share the same JVM instance, but they do not share the same values for variables. In this method the security would mostly be provided by the systems running the code, such as the JVM [8]. Another from of isolation is *user based isolation.* In this method each instance of an application has a non privileged user account that runs the applications. This is basically the same as having a public computer with many different accounts, such as a library. Files that are owned by another account cannot be run or even viewed. In this method the operating system takes care of memory protection. *Container based isolation* is a method where each application would get its own container. A container is an isolated group of processes that are separated from others on a deep operating system level. They can be thought of as Tupperware with a name written on it in on office fridge with polite employees, it will not be opened or looked into by anybody but the owner. The last isolation technique is *virtual machine based isolation.* In this method each instance operates as if it is a full computer with everything at its disposal, but it will not be able to access memory outside what the outside operating system lets it have [8]. The FLUSH + RELOAD attack was tested using container based isolation, however, it could be modified to run on other isolation types.

6.2 Spy Process

The spy process for this attack is similar to the others. The spy has a list of instructions that it loads into the cache. If any of these processes result in a cache hit, then the spy process can execute a program depending on what triggered the hit [8]. After it tests for cache hits, it will flush all of the instructions out of cache using *clflush.* After a set interval the spy will reload all of the data and test it again.

6.3 PRNG

*PRNG*s are pseudorandom number generators that are used by many applications for authenticating password reset requests [8]. For the purpose of explanation, the paper will assume that it is using the PHP implementation of PRNG. PRNG usually relies on a system call to get a seed for the generator, which is usually the time of day [8]. In PHP there are many function calls that happen in order to generate a password reset link, shown in figure 3. However the only sources of randomness are $gettimeofday()$, $time()$, and $getpid()$ [8]. This means that if a process can discover when one of these functions is called, it can reproduce the link once it finds the value of $getpid()$. A spy process can view when the $gettimeofday()$ is called and call it on its own and store the value. Since both systems are running on the same machine, it should be extremely close to, if not the same as the victim's call. The same can be said about the $time()$ function. The $getpid()$ call returns a number less than or equal to 2^{16}.

6.4 Attacking The PRNG

Currently, these password reset links are used in many off the shelf eCommerce applications as well as in WordPress, which makes up 21.9% of the top 10 million websites [8]. In order to do this, two password reset requests are sent out, one to an account owned by the attacker and another to the one owned by the victim. The program stores information form the $gettimeofday()$ call. Using that information and the password reset token given by the application, the result of the $getpid()$ call can be found by a brute force attack that takes $O(2^{20}), 2^{16}$ for guessing the $getpid()$ value and 2^4 to account for differences in the $gettimeofday()$ call [8]. Once the value for $getpid()$ is found, it only takes four requests to find the correct link to reset the victim's password [8].

6.5 Countermeasures

This attack can also be countered by disabling the $clflush$ instruction. Disabling it would result in the same consequences as before, other processes that rely on it, such as the operating system could not use it. A way around this could be to put each application in a sandbox where $clflush$ is disabled, but there are ways to get around this by using different commands [8]. Another way to counter this attack would be do disallow any resource sharing. In order to disallow resource sharing, each application would have to be given their own binary files. This would have a huge impact on memory, meaning that less applications would be able to be stored on each machine due to the memory overhead of each instance [8]. Because of that fact, disallowing resource sharing most likely will not be implemented because it is not cost efficient.

7. CONCLUSION

Overall side channel attacks are very powerful attacks and the FLUSH + RELOAD attack is no exception. With how fast FLUSH + RELOAD is developing and the information it is able to obtain, it currently poses a massive threat to security. While FLUSH + RELOAD used to be a slow attack that was only possible under very specific circumstances. Now somebody can purchase a section of a PaaS cloud and spy on the other applications on that box, including getting the user data. As time goes on, unless a way to stop the FLUSH + RELOAD attack is discovered, user data will be at greater risk of being spied on by this attack.

8. ACKNOWLEDGMENTS

I would like to thank Elena Machkasova for the excellent feedback she gave to the drafts of this paper. I would also like to thank Kevin Arhelger for the feedback he gave on my draft.

9. REFERENCES

[1] N. Benger, J. Pol, N. P. Smart, and Y. Yarom. *Cryptographic Hardware and Embedded Systems – CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, chapter "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way, pages 75–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[2] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[3] Wikipedia. Timing attack — wikipedia, the free encyclopedia, 2015. [Online; accessed 28-February-2016].

[4] Wikipedia. Advanced encryption standard — wikipedia, the free encyclopedia, 2016. [Online; accessed 5-April-2016].

[5] Wikipedia. Cache (computing) — wikipedia, the free encyclopedia, 2016. [Online; accessed 19-March-2016].

[6] Wikipedia. Side-channel attack — wikipedia, the free encyclopedia, 2016. [Online; accessed 28-February-2016].

[7] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, Aug. 2014. USENIX Association.

[8] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.