Attacking `scrypt` via Cache Timing Side-Channel

Mark Matthew Anderson
Stanford University

Abstract

This paper gives a motivation for the design of memory-hard key derivation functions (KDFs), a summary of a memory-hard password-based key derivation function called `scrypt`, and an overview of cache timing attacks. A cache timing attack against `scrypt` is introduced and described in detail. Finally, additional work necessary to implement the attack and measures to prevent the attack are discussed. Since it is an actively-used utility for generating cryptographic keys, constructing a password stealing attack against `scrypt` raises serious security concerns for any applications that make use of it.

1 Introduction

1.1 Memory-Hard KDFs

KDFs are functions that compute random looking cryptographic keys from human-generated passwords. Human-generated passwords are far from random; users often choose passwords that are based on predictable factors, like words or phrases that are familiar or meaningful to the user. While these passwords are still somewhat random, KDFs are used in practice to generate keys with far higher degrees of randomness by applying pseudo-random functions to the user's input. The output of the KDF is intended to appear independent from the user's input and to be impossible to invert. The increased randomness provided by KDFs makes key guessing attacks extremely unlikely to conduct successfully.

In addition to generating an output that is difficult to predict and invert, standard KDFs are also designed to take a relatively large amount of time to execute in order to reduce the practicality of dictionary attacks. If KDFs were quickly computable, an attacker could easily find an output's preimage by evaluating the KDF for a large set of potential passwords and finding which input produces an output that matches the target value. Long-running KDF algorithms make the process of enumerating all possible inputs a very laborious task. For example, if a given KDF takes 1ms to process, it would take an amount of time on the order of 10^{28} years for a single machine to evaluate the KDF on all possible values of a 128-bit input.

Though designed to take a long time to evaluate, an issue with standard KDFs is that attackers are often able to accelerate their computation by means of algorithmic optimizations and highly parallel or specialized hardware. Ruddick and Yan demonstrate such attacks on the popular PBKDF2 [1]. By shortening the execution time of the KDF, an attacker is able to eliminate any protection against a dictionary attack.

As a stronger measure to prevent dictionary attacks, researchers have designed KDFs that are memory intensive in addition to being time intensive. Designing a KDF to be memory-hard is a way to preserve the long runtime of the KDF by impeding any efforts to accelerate its execution. Memory is an expensive and slow piece of hardware. When large amounts of memory are required to execute a function, parallelized hardware (like a GPU) is ineffective at providing any speedup, since all the parallel hardware is competing for a fixed and small amount of shared memory. Likewise, building specialized hardware with enough parallelism and memory to quickly evaluate the memory-hard KDF is impractical because the designs are prohibitively expensive.

1.2 The `scrypt` Algorithm

The `scrypt` algorithm [2, 3] was designed by Colin Percival to be a memory-hard KDF. It takes as input a password (PW), a salt (S), a memory cost parameter (C), an indexing parameter (I), a parallelization parameter (P), and a desired output length parameter (L_s). Noteworthy parameters within the function are an array of blocks to be mixed (B), and a parameter that corresponds to the length of the inputs and outputs of the function

MHMix (L_{MHM}). The `scrypt` algorithm is defined by the following function:

```
**scrypt**$(PW, S, C, I, P, L_s)$ :
  $(B[0], ..., B[P-1]) \leftarrow$ PBKDF2$(PW, S, 1, P \cdot L_{MHM})$
  for $j = 0$ to $P - 1$ do
    $B[j] \leftarrow$ MHMix$(B[j], C, I)$
  end for
  return PBKDF2$(PW, B[0]||...||B[P-1], 1, L_s)$
```

The MHMix function is what gives `scrypt` its memory-hard property. MHMix takes as input a block to be expanded and hashed (B), a memory cost parameter (C), and an indexing parameter (I). Parameters used within the function are a temporary block placeholder (X), a memory-consuming array of hashes (A), and an indexing variable (k). MHMix uses a hashing function Mix whose details are important to the `scrypt` algorithm, but not directly pertinent to the cache timing attack. MHMix takes the input block, hashes it many times while saving the hash results, and computes an output derived from some of the hash results that are chosen by interpreting certain hash values as indices. Since the hash values will be unique to each password input to `scrypt`, the indices of the hash values that are accessed and used in the output will also be unique to each password. Since the hash values that are needed to compute the true `scrypt` output are password-dependent, an adversary conducting a brute force dictionary attack against `scrypt` will be unable to predict which values will contribute to the final output and will be forced to store all values to potentially be used, making array A in MHMix use large amounts of memory. MHMix is defined as:

```
**MHMix**$(B, C, I)$:
  $X \leftarrow B$
  for $j = 0$ to $C - 1$ do
    $A[j] \leftarrow X$
    $X \leftarrow$ Mix$(X, I)$
  end for
  for $j = 0$ to $C - 1$ do
    $k \leftarrow$ ((int)$X$) mod $C$
    $X \leftarrow$ Mix$(X \oplus A[k], I)$
  end for
  return $X$
```

1.3 Cache Timing Attacks

Cache timing attacks are a class of side-channel attacks that extract information based on the availability or unavailability of data in a processor's cache. These attacks are used to determine when certain data or instructions are being used. Knowledge of when particular data or instructions are in use can disclose details about secret information when data accesses or function calls in a program are dependent on this secret information.

Yuval Yarom's PRIME+PROBE method [4] is an example of a cache timing attack, and is the best technique to use against `scrypt`. The PRIME+PROBE method is used to track the access patterns of data over time by intentionally thrashing target data in the cache and observing when the victim puts that data back in cache. First, the attacker flushes the victim's data from cache (the PRIME stage) by accessing data that it knows will evict the victim's data from cache. Then, after waiting to give the victim an opportunity to access its data, the attacker attempts to access the data it put in cache (the PROBE stage). If the time to retrieve the data is relatively short, that means that the victim did not access its data. If the time to retrieve the data is relatively long, the victim accessed the target data, which caused the attacker's data to be evicted from cache, forcing the attacker to wait for its data to be retrieved from a lower-level memory.

The PRIME+PROBE attack can be used any time the attacker shares some level of processor cache with the victim. This can happen when the attacker and victim processes are two processes running on the same machine or when the attacker and victim are working on separate virtual machines that are hosted on the same machine. Note that it is not necessary that the attacker and victim be using the same core on a multi-core processor, they only need to be sharing cache memory on some level.

2 Attack on `scrypt`

2.1 Vulnerability

What makes the cache timing attack on `scrypt` possible is the following code from the MHMix function:

$$k \leftarrow (\texttt{(int)}X) \bmod C$$
$$X \leftarrow \text{Mix}(X \oplus A[k], I)$$

As previously mentioned, A is the array that gives `scrypt` its memory-hard property. A stores all the repeated hashes of B, such that $A[n] = \text{Mix}^n(B)$, where $\text{Mix}^n(B)$ is the result of hashing B n times (e.g. $\text{Mix}^2(B) = \text{Mix}(\text{Mix}(B, I), I)$). In the above lines of code, elements of A are chosen to be used in calculation of the `scrypt` result by interpreting hash results, which are unique to and derived from secret information (the evaluation of PBKDF2 on the user's password), as integers and using them to index through A (with the variable k). Since these memory accesses are dependent on the password, patterning the memory accesses observed during an execution of `scrypt` will give information about the result of the evaluation of PBKDF2 in the first step of the `scrypt` algorithm.

Learning enough information about the PBKDF2 hash of the victim's password allows an adversary to reduce an attack on scrypt to an attack on PBKDF2, thereby bypassing the memory-hardness of scrypt. More specifically, once the memory access pattern of scrypt is observed, the attacker can construct a dictionary of PBKDF2 hashes of potential passwords, compute what their access patterns will be, and compare the observed memory access patterns to the access patterns of the hashes in the dictionary. Bypassing the memory-hardness of scrypt by shifting the attack to an attack on PBKDF2 allows the attacker to take advantage of all the attack shortcuts scrypt was designed to prevent.

2.2 Procedure

When computing each entry of the dictionary of candidate passwords, the attacker will have to compute the PBKDF2 hash of each candidate with the same parameters as used by scrypt. For each block of the PBKDF2 hash of a candidate password ($B[0],...,B[P-1]$ in the first line of the scrypt algorithm), the attacker will compute and store the resulting derived index $DI[i] = \text{Mix}^m(B[i])$ where m is equal to the value of C being used by the scrypt call that is being attacked.

After constructing the dictionary of candidate passwords and their arrays of derived indices, the attacker will monitor a victim's execution of scrypt. For each iteration of the loop in the scrypt function, the attacker will use a cache timing attack method like the PRIME+PROBE procedure to observe the memory accesses that happen when MHMix is called.

In the MHMix function, the attacker will first notice a series of temporally sequential memory accesses that correspond to the hashes calculated and stored in A. Between these accesses will be other accesses that result from the call to Mix that will follow a predetermined and constant pattern. By knowing what this memory access behavior will be, and seeing it repeat as each hash is calculated, the attacker will be able to distinguish these memory accesses from the important ones, and will thus be able to discard or ignore them during future analysis. For simplicity, these Mix memory accesses are ignored in this paper. The second set of memory accesses the attacker will notice will be a product of the second loop in MHMix where the contents of A are used to construct the function's output. These are the memory accesses that are password-dependent. The attacker will note which memory location is accessed first in this stage of the computation.

By seeing where this memory location falls in relation to the memory locations accessed during the hashing loop, the attacker will be able to reason as to which index of A the memory location corresponds to. By learning the index of the first element accessed in the output construction stage of MHMix, the attacker learns the value of $\text{Mix}^C(\cdot) \bmod C$ evaluated on the input to MHMix. With knowledge of this value, the attacker can reduce the set of potential passwords to the subset whose corresponding DI are equal to this value. From here, the attacker can either continue to observe evaluations of MHMix in this manner and further reduce the set of potential passwords, or they can brute force through the reduced candidate password set to find out which is the victim's password.

A simple, contrived example of anticipated memory access patterns observed during MHMix execution on a block B_{ex} is given in Figure 1. From time slot 0 to time slot 20, the hashing loop is being executed. This is where the attacker will observe sequential memory accesses as the hashing results are stored in A. Knowing this access structure, the attacker can deduce that the first access corresponds to $A[0]$, the second access corresponds to $A[1]$, and so on. Next, at time slot 21, the attacker sees an access to a memory location that was previously accessed during the hashing stage. This marks the end of the hashing stage and the beginning of the construction of the MHMix output. The first access of this stage is what the attacker is interested in. In the example, the first element accessed in the output construction stage corresponds to the fourteenth element accessed in the hashing stage, which in turn corresponds to $A[13]$. This means that $\text{Mix}^{21}(B_{ex}) = 13$ when interpreted as an integer mod 21. With this knowledge, the attacker can narrow the set of potential passwords to the set of candidates whose PBKDF2 hash block corresponding to B_{ex} equals 13 mod 21 after being passed to $\text{Mix}^{21}(\cdot)$.

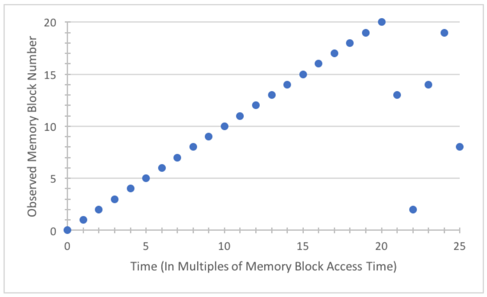

Figure 1: Example MHMix Memory Access Pattern

The size of the PBKDF2 output blocks are, at minimum, 16 bytes. When run under recommended conditions [2], the output block size will be 128 bytes. These sizes are on the order of the size of standard cache lines, which provides good resolution for this attack. The attack would become more difficult if the output blocks were much smaller than cache line sizes because the attacker only observes cache line access. In the case where output blocks are much smaller than cache lines, an access to a cache line means that one of the many output blocks in that cache line is being accessed. Having less of an idea of which specific output block is being accessed means the attacker is not able to rule out as many candidate passwords as they would be able to otherwise.

3 Attack Prevention

3.1 Skew-Associative Cache

A PRIME+PROBE-style attack on `scrypt` would likely be unsuccessful if conducted on a processor that has a skew-associative cache architecture. Standard, unskewed set-associative caches map cache lines to the same index of each way of the cache, where skew-associative caches map cache lines differently to each cache way based on functions that are unique to each way. This is meant to decrease conflicts in the cache, which is exactly why the PRIME+PROBE attack becomes more difficult to conduct.

The PRIME+PROBE attack makes use of eviction sets, which are sets of cache lines that all map to the same index within each way. If a processor has an n-way set-associative cache that holds k cache lines in each way, the attacker will construct k eviction sets consisting of n cache lines. When the attacker wants to evict a specific cache line that exists at index i in one of the cache ways, they will access each of the n elements of the eviction set for index i. As the elements of the eviction set are mapped into the cache, they will evict the cache line that is currently residing at index i. By accessing n such elements, the attacker evicts the data at index i in each of the n ways in the cache, assuming an eviction policy similar to least-recently used.

The attack relies on constructing a set of cache lines that will be mapped to the same index in each way of the cache in order for the eviction just described to work. It is hard, if not impossible, to construct the eviction sets in a skew-associative cache due to the fact that a cache line will evict from a different index depending on which way in the cache it is mapped to. If unable to construct eviction sets, the attacker will be unable to conduct a PRIME+PROBE cache timing attack.

3.2 Exclusive Cache Hierarchy

Another assumption that the attack relies on is the use of an inclusive cache hierarchy. When the cache hierarchy is inclusive, any data stored in higher levels of cache must also be stored in lower levels. This is done in an effort to maintain data coherence and availability between processor cores.

A cross-core attack would not work on a multi-core machine that implemented an exclusive cache hierarchy. When the attacker and victim processes are running on separate cores of a processor, the PRIME+PROBE attack method depends on the fact that an inclusive cache evicts a cache line from all cache levels it resides in when the cache line is evicted from the shared lower level. The attacker evicts victim data from the shared cache, completely removing the data from the entire cache. The victim is then forced to access lower levels of memory when it tries to look up the data that got evicted, which then evicts some of the attacker's data when it is brought into cache. In an exclusive cache architecture, the attacker can evict victim data from the shared level of cache, but doing so does not evict the data from higher levels of cache. If the data still exists in the higher levels of cache, it is still available to the victim core's operation, and the victim is not forced to bring the data back into the shared cache. If the victim does not have to bring data into shared cache, the attacker cannot observe when the data is accessed, making the attack impossible to carry out.

In the case of a single-core machine, the attack practicality holds whether the cache is inclusive or exclusive. This is due to the fact that the attacker and victim processes will be running on the same processor and have access to and control of the same cache memory.

3.3 Parallel Execution

The `scrypt` algorithm currently iterates through a loop and makes sequential calls to MHMix, which an attacker can then attack by observing memory accesses. If `scrypt` were to make these calls to MHMix in parallel rather than sequentially, making sense of the observed memory accesses would be much harder for the attacker.

With multiple parallel calls to MHMix executing simultaneously, it would be difficult to decide which memory accesses belong to which call of the function. This is not to say it would be impossible, though, because if the attacker is able to get information that would help differentiate the function calls, it may be able to correctly group the memory accesses. Such information would include a thread scheduling policy, physical memory locations, or memory access timing relationships. When making parallel calls to MHMix, the `scrypt` user needs to be careful that sufficient memory is available to handle these parallel calls.

4 Conclusion

4.1 Future Work

At the time of writing this paper, the author's efforts to implement the attack described by modifying the operation of existing tools have not been successful. Though some open-source cache timing analysis tools exist, further efforts are required to alter them or build new tools in order to realize the attack in practice.

Further research and experimentation will also need to be conducted to identify and resolve difficulties in the practical implementation of the attack. Such difficulties may include how to detect when the victim begins executing `scrypt`, how to filter out observed memory accesses due to processes unrelated to `scrypt`, and how to deal with otherwise noisy or incomplete memory access observations.

4.2 Summary

This paper gives the motivation for memory-hard KDFs, an overview of the `scrypt` memory-hard KDF, and an introduction to cache timing attacks. A cache timing attack against `scrypt` is then proposed and discussed, along with techniques and policies to prevent it. Finally, ideas for future work and ways to further develop the attack are presented.

References

[1] A. Ruddick and J. Yan. (2016) *Acceleration Attacks on PBKDF2: Or, what is inside the black-box of oclHashcat?* [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/ruddick

[2] C. Percival. (2009) *Stronger Key Derivation Via Sequential Memory-Hard Functions* [Online]. Available: http://www.tarsnap.com/scrypt/scrypt.pdf

[3] C. Percival *et al.* (2017) *The scrypt Key Derivation Function* (Version 1.2.1) [Source Code]. Available: https://github.com/Tarsnap/scrypt

[4] Y. Yarom *et al.* (2015) *Last-Level Cache Side-Channel Attacks are Practical* [Online]. Available: http://ieeexplore.ieee.org/document/7163050/