

Cache attacks on symmetric key ciphers, and the effectiveness of defenses

Brian M. Sanders

Abstract—Side channel attacks focusing on cache usage are a relatively new, and dangerous form of attack. I will explain how the cache can leak useful information to an attack, and how this can lead to the discovery of symmetric keys. Using knowledge of how these attacks work, I give a brief analysis of some common ciphers and how these attacks relate to them. Following this I will discuss some proposed methods of defense, both software and hardware, which can address these attacks. In the discussion of software defenses, I will discuss the defensive choices of the open source libraries OpenSSL and Crypto++, discussing what they do and do not provide protection against, and what negative affects they have chosen to accept. None of the proposed defenses provide protection with out some negative affects. In the case of software defenses, it is often at the cost of performance. Hardware can generally avoid the performance issues, but is much more expensive to implement, and takes longer to deploy. Finally I will discuss real world situations in which these attacks might be successful, and what currently proposed defenses might best be used in the situation.

Index Terms—Data Security, Cache memories, Cryptography, Side Channel attacks

I. INTRODUCTION

Cryptographic algorithms have quickly matured to both avoid, and provide defensive measures against, many types of algorithmic flaws. It has become common practice to use contests to help define standards in the field, allowing multiple experts to both submit and analyze new algorithms. These algorithms take into consideration advanced attack techniques such as linear and differential cryptanalysis, and generally provide additional security due to the threat of future unknown attack methods. As the algorithm it self becomes less of an available source of attack, many researchers have begun to look at other methods of attack, besides algorithmic flaws. A category of attack known as side channel attacks specifically focuses on the implementation of algorithms, rather than their mathematical properties. Many times the implementation of these algorithms can provide much easier attack vectors than the well understood and studied mathematical problems the algorithms are based on.

Side channel attacks can not be detected by mathematical

analysis of the underlying algorithms, but rather require an understanding of security beyond that provided by the algorithm. It is as important to understand these attacks as it is to understand weaknesses in the mathematical principles of the algorithm. As the understanding of these new attacks progress, so will the defenses against them.

One of the most recent proposed side channel attacks involves using the cache access of the processor to determine information which was not intended to be available outside the algorithm. As with any side channel, when additional information can be determined, it opens up a great risk which may not be defended against. This particular version is especially dangerous as it contains one aspect which previous side channels have not had, it can be realized in software alone. This allows any attacks based on it to be quickly deployed to a large number of targets.

The first demonstrated attack to make use of these methods was published by D. J. Bernstein [1]. This helped demonstrate that the cache side channel was more than theoretical. More advanced methods based on the cache as a side channel was later published by Osvik et al [2]. Although the later does not provide source code allowing additional testing, the descriptions of the attacks demonstrate how the techniques can be much more dangerous than the previous timing method of Bernstein.

II. HOW CACHE ATTACKS WORK

A. CPU caching

To understand how the CPU can leak information through cache usage, it is important to first understand how the cache is used. Although its implementation can vary according to the hardware used, the concept of caching remains the same. All caches are implemented to allow for quicker access to information. As processes run, they require information which is stored in memory. This memory access can take a long time relative to the CPU's clock; therefore as information is used, it is stored locally in the cache to provide access to it again at much quicker rates. The difference between a cache access and a memory access will depend on the architecture, but generally it is more than 30 times faster to access the data in cache. This gap is continuing to widen as CPU's increase in speed at a faster rate than memory. When data is found in the cache, avoiding a costly memory lookup, it is known as a cache hit. Conversely, if the data is not found in the cache, it

Manuscript received May 8, 2008. Manuscript revised May 12, 2008
B. M. Sanders is with Applied Security, Reston, VA 20109 USA. (e-mail: bsander4@gmu.edu).

is called a cache miss. Due to the high probability of accessing memory near a location which was recently accessed, data is pulled into the cache in larger segments known as lines. By bringing in lines the cache hit rate can be increased, providing better performance. The size of a cache line can vary according to the architecture.

Locations in memory can be mapped to cache locations in multiple ways. The most basic method of doing this is known as a direct mapped cache. This method maps each memory location to a single cache location. Other methods, known as N way set associative cache, map memory locations to a set containing N cache lines, where N is generally 2, 4 or 8. When data is pulled into the cache, it can go into any one of the cache lines in its set.

Since the size of the cache is much smaller than main memory, many memory locations map to the same cache location, or set. As conflicting lines are accessed by programs, they must replace the data which was already in the shared cache line. The next time an access is made to the data which was removed; it will need to be pulled back into the cache.

B. Software implementations of Algorithms

Along with an understanding of how the cache works, it will also be necessary to understand software implementations of common algorithms. The security of the algorithm alone does not create a good standard. It is important to understand that to be useful in applications, it must be able to be implemented in such a way as to provide quick execution in both software and hardware implementations. Cache based side channel attacks use a combination of software optimizations, and knowledge of the cache, to make their attack.

The main software optimization technique which can be used in combination with the cache structure, are tables. Many cryptographic algorithms make use of primitives known as substitution boxes, or s-boxes. As its name suggest, an s-box takes an input, looks up a location referenced by the input, and outputs the value of that location. Therefore the output for any specific input will always be the same, but sequences of inputs can produce unrelated, random outputs. When implementing these structures in software, the logical implementation is to store the values of the s-box in a large table. Inputs are used to reference locations in the table, and the substituted value can be quickly found. Although the values could be calculated each time, the increased complexity would greatly decrease the performance of the algorithm.

A second method of speeding up software implementations is to extend the speed of a table lookup to cover additional steps in the algorithm. If multiple calculations are to be done in sequence, it is possible to make a table of all possible values those calculations can produce, with a given input. Rather than performing multiple operations, the table can be referenced and a value quickly determined.

To demonstrate some of these techniques we will take a look at a very common symmetric key cipher, AES. The AES algorithm consists of four major operations: 1) SubBytes; 2) ShiftRows; 3) MixColumns; 4) Add Round Key. Input to the

algorithm is in 128 bit blocks arranged into a square matrix consisting of four rows and 4 columns. The SubBytes operation is simply a single 256 entry s-box, which is applied to all values in the input matrix. The ShiftRows operation shifts the order of the data inside each row by a constant shift amount specified per row. The MixColumns operation requires multiplying each column of the input data by a matrix in the Galois Field, replacing the original column with the output of this multiplication. Finally the Add Round Key operation applies an XOR of the input data with the key for the current round.

The execution of AES can be broken down into an initial transformation, a loop of 10, 12, or 14 cipher rounds depending on the chosen key size, and then a final transformation. The initial round only performs one Add Round Key operation. The cipher rounds apply SubBytes, followed by ShiftRows, MixColumns, and then Add Round Key. After looping through the cipher rounds the designated number of times, the final transformation is applied, during which the same transformations are applied as the cipher rounds, minus the MixColumns operation.

It is possible to implement the SubBytes operation as suggested using a table lookup. Due to the s-box always generating the same output for the same input, it is possible to perform the shift before or after the s-box lookup, with the same results. Therefore it is possible to implement the shift by adjusting the input to the s-box as needed. Since the MixColumns multiplies each row in a column differently, it can be included in a table lookup by creating four tables, with each one applying the appropriate multiplication function for each row. For each column processed this would then require four table lookups (one from each rows table). The results of the table lookups would then need to be XOR-ed together along with the key, resulting in 4 XOR operations. Considering that each round will consist of 4 columns to operate on, the entire round can be reduced to 20 table lookups (assuming the round key has been setup), and 16 XOR operations. This optimization produces four tables, each still with 256 entries, but each entry is 4 bytes (due to the vector multiplication) rather than 1. This makes each table 1KB, and all four tables together require 4KB of storage. Using this technique produces a speed increase between 100 and 1000 times over non-optimized methods. This technique was the optimized code submitted during the AES contest for the Rijndael cipher.

Combining the large difference in cache hit and miss time, with the use of tables, provides the basis for cache based side channel attacks. How long it takes to look up a location in a table will be measurably different depending on if it is a cache hit or a cache miss. In algorithms where a table index is data dependent, this dependency may give an attacker additional information about the algorithm. In the case of AES, the initial transformation applies the key to the input text, followed by an index into the tables discussed above. Therefore the index being used is a function of the key, and the input. An

attacker who is able to provide the input, and discover the index used, is then able to discover what key was applied.

C. Timing Attacks

The first cache attack methods I will discuss are known as timing attacks. These attacks focus on the time required to perform an entire encryption. Due to timing variances caused by cache hits and misses, they are able to discern information which can be used to discover the symmetric key in use. The main advantage of this form of attack is the possibility of a remote attack.

Bernstein's attack [1,3] consists of four phases. The first phase of his attack is simply a learning phase. During this time the attacker needs to know the key used on the server. For his attack, Bernstein used an all zero key, making later correlations simpler to perform. The attacker then sends large amounts of packets of known but random texts to the server. A large matrix stores the average timing data for each byte, and each value of each byte. This timing data is gathered for a large sample of packets, spanning multiple packet sizes.

A full study by Neve et al [4] helps to define exactly what is being learned during this phase and why the timing differences are relevant. Timing differences arise even though the algorithm is performing many encryptions, due to the other processes and applications running on the same machine. These other functions, such as sending and receiving packets, are also accessing the cache as they run. Over a large sample of accesses, these other processes become constant. Therefore for any given machine, certain cache lines will be consistently used, and any AES table indexes which map to those locations, will be evicted quickly. Therefore what has been learned during the first phase of the attack is a host specific timing profile. The encryptions which take longer are the ones which require access to table indexes that are consistently evicted from the cache by frequently running processes. Since the key of all zeros was used, the input bytes are used during the first round of AES for the table index lookups. Therefore the fingerprint found shows which indexes in the first round, cause longer encryption times.

During the second phase of his attack, the same server is used, although the key is randomly selected. The attacker gathers data in a similar fashion to the learning phase, by sending large amounts of randomly chosen plain texts. The same matrix of timings is created for each byte as was done in the learning phase. After sufficient samples have been sent this phase is complete.

Finally the data is compared in a correlation phase which is done offline. During this phase the results from the learning phase are compared to the attacking phase. Some bytes will correlate very closely, and only these bytes are used. Since in the first attack all variables are known, it becomes simple to take the attack phases correlations, and deduce the key that must have been applied to produce the same index value as seen in the learning phase. For example, during the learn phase the first byte with value 10011100 took on a very high average timing value. Then during the attack phase, it is found

that the first byte with value 01111100 is the only strong correlation to this timing, the XOR of these two values will reveal a value of 01100000 must have been applied during the attack phase to generate the same index as in the learning phase. This reveals some bits of the key used during encryption. Some correlations may not be strong enough to narrow down the key exactly, but can still be used to greatly reduce the values which need to be used during the next step. Using these strongly correlated bits to reduce the key space, a brute force attack is then applied to determine the remaining key bits. In Bernstein's case his offline attack required only about 1 minute of computations, before the correct key was revealed.

Extensions of this type of timing attack have also been devised. After analyzing why Bernstein's attack works, Neve et al go on to describe how to extend this attack to take into consideration the second round of AES encryptions [4]. Although this still requires large numbers of samples to correlate data, the attack extended into the second round was shown to discover more key bits than an equivalent first round only attack. Therefore less data may need to be gathered before a meaningful amount of key bits may be determined.

Another form of timing attack was realized by J. Bonneau and I. Mironov [5]. They suggest looking at the final round of AES rather than the first round for collisions. The advantage being that the final round does not use the MixColumns, and therefore the table lookup is a standard s-box. This means that the final round of encryption does all lookups against the same table. The output of these table lookups is then XOR-ed with the final key bits to produce the output cipher text. This allows his attack to compare the output cipher text values, with the time taken for encryption. This is compared against the average time for encryption. As times decrease below the average, guesses can be made at key bits due to cache collisions in the final round. Using an expansion of this method to consider cache hits from similar line reads he was able to produce full key recovery with up to four times less samples than that of Bernstein's first round attack. Not only does this method produce quicker results, but there is no need for a learning phase, making the attack available to a wider range of situations.

The timing attacks presented here are not necessarily the only attacks which may be realized, but do provide a wide range of how timing attacks can be implemented. The main feature of all timing attacks is the exploitation of the effect cache hits have on the run time of the encryption process. To make use of this fact, large number of samples must be gathered, and analysis applied to filter out noise. The main benefit of this type of attack is the possibility of making an attack remotely.

D. Trace Attacks

The second category of cache attacks are known as trace attacks. These attacks must be initiated on the target machine, allowing for the cache to be directly manipulated. They still use the difference in timing between a cache hit and a cache

miss, but due to their shared usage of the cache with the attacked process, they can be carried out with much fewer operations. Osvik et al [2] suggest two methods to use the shared property of the cache to mount a side channel attack, Evict + Time, and Prime + Probe.

The first suggested method, Evict + Time, focuses on the encryption time of a plain text, to detect if a specific memory location has been accessed. It accomplishes this in three basic steps: 1) Encrypt known text x 2) Evict a specific memory location from cache 3) Encrypt know text x again, timing the results. The first step takes the known plaintext and ensures that all values are in the cache by initiating an encryption on it. Secondly the attacking program will access areas of memory it controls to cause an eviction of the cache lines that are to be tested. Finally the same known text is encrypted again, this time taking a time measurement. This timing score is then used to determine if a cache miss has taken place. By sampling this value for multiple locations, and plaintexts, the attacker can determine what sections of the table are being accessed for each given plain text. As discussed earlier, knowing the references into this table gives the attacker the ability to determine the key used during encryption. This method is susceptible to variations in timing due to interference, but it is valid none the less.

The second suggested method, Prime + Probe, discovers memory accesses by timing access time to memory locations, rather than depending on the encryption process timings. It accomplishes this in the three basic steps as well. 1) Read memory locations in the attacker’s memory that map to each cache location. 2) Encrypt a know plaintext x. 3) Read memory locations which map to all cache locations used by the encryption process, and record the access time. In the first step the attacker fills all cache locations with his own data, by accessing enough locations to fill the cache. Then the attacker initiates an encryption of a known plaintext. As all cache locations have recently been replaced, every table lookup will cause an eviction and it’s self be stored in cache. Finally the attacker times how long it takes to access his own memory locations. Any location in which there was a conflict, the attacker should notice a much longer access time. This will reveal to him what locations where used while performing the encryption. This tactic provides multiple benefits over the Evict + Prime method. The largest advantage is that it bases the timing off simple processes it owns. This reduces noise and gives much better results. Secondly, it provides data about all tables accessed from one encryption, providing more data per encryption analyzed than the Evict + Prime method.

Both the above described methods can be used to extract table information using the cache as a side channel. They require knowledge of when an encryption has been initiated; these are known as synchronous attacks [2]. To address this, Osvik et al discuss a method of asynchronous attack [2] which does not have this restriction. Using a slight variation of the Prime + Probe method, an attacker can continually access his memory locations. As the encryption process runs, it will

cause evictions. Asynchronous attacks use these evictions to form a frequency score, which can then be used help determine what areas of the memory the encryption process is using the most. This form of attack is more restricted on the hardware and architectures on which it will run, but is still effective.

Building on these methods, Neve and Seifert [6] discuss how to extend cache trace attacks to machines with out multi-threading. In the case of AES, the encryption process runs very quickly, and the spy process needs to be run again before the next encryption takes place. On a multi-threading machine this is easier, as the hardware allows the spy process to execute. To achieve this with out the multi-threading hardware, Neve and Seifert propose a method which is operating system dependent. By taking advantage of the scheduling algorithm of the operating system, they spy process may release the CPU with very short time quantum remaining. The crypto process may then run, but will very quickly be interrupted, and the spy process will again be allowed to run. By providing a possible software means of executing these attacks on an expanded set of hardware, they have expanded the effectiveness of this attack to a large set of possible machines.

The methods described here outline how a trace driven cache attack works. These attacks specialized to attack specific algorithm implementations, such as the final round of AES, and can achieve results with much fewer samples than the previous discussed timing attacks. Trace attacks do add the restriction of having the ability to execute code on the same box as the attacked process.

III. CIPHER ANALYSIS

After studying the principles behind cache attacks, I was able to analyze some common ciphers which might be in use today. By looking at how these ciphers are generally implemented, it can be determined if a cache attack would reveal information which could break the cipher. Table 1 shows the results of this analysis. The lower five were the top five candidates in the AES contest.

| Cipher | Timing Attack | Trace attack |
|----------------|---------------|--------------|
| DES/3DES | Yes | Yes |
| Blowfish | Yes | Yes |
| RC4 | No | No |
| MARS | Yes | Yes |
| RC6 | No | No |
| Twofish | Yes | Yes |
| Serpent | No | No |
| AES (Rijndael) | Yes | Yes |

Table 1. Common ciphers and their resistance to cache attacks.

The main feature shared by all susceptible algorithms is a dependence on tables, either s-boxes or lookup tables to speed up execution, which depends on the input data. As described in the software implementation section above, these tables are logical ways of implementing these algorithms. When input data and key material is used to reference them, they become

vulnerable to cache related attacks.

The DES/3DES, Blowfish, MARS, Twofish, and AES algorithms use s-boxes through. The indexes of these s-boxes are data dependent on both the input data and the key. Therefore the index into the table provides information about the key, and the input can be used as a means of adjusting which locations are accessed. Although tables were considered secure from timing attacks due to their constant time of execution, when implemented in this way, they can provide the discussed leaks through the cache.

The RC4 algorithm would seem at first to be vulnerable to this attack for the same reasons as the previous algorithms. It makes use of a very large table when implemented in software. The critical difference with RC4 is that the table is not indexed based on the input. The key, which is combined with the input text, is stored in the table. The data in each location is then consistently swapped during execution as they are used, helping to randomize which value is in which location. Although the encryption depends on the value at each location, the table is not data dependent for its lookups. Therefore a cache analysis could provide some idea as to which locations in the table were used, but without some knowledge of what was in each location, this does not break the cipher.

The RC6 algorithm is not susceptible for differing reasons from its predecessor RC4. The RC6 algorithm does not use s-box lookups or large tables. It depends on multiplications and shifts as its main functions. These do not require any specific cache properties and therefore do not leak any unwanted information. During analysis of the final five candidates in the AES contest, this algorithm was ranked among the bottom two in software efficiency, and about third for hardware implementations. So although the algorithm's structure does provide it additional security, it does not provide the top end performance that may be required by some applications.

Finally the Serpent algorithm is resistant to this attack due to the basic design used while creating it. The Serpent algorithm was designed to work in a mode known as bit slice. A bit slice implementation uses large registers to achieve parallel execution of multiple blocks at a time. In the first round, the first bit of each block to be encrypted is affected. During the second round, the second bit of each block will be affected. By working in this manner a form of parallelism can be achieved. Some functions, such as addition and multiplication, may become slower, but others, such as permutations become very fast. Due to this, the Serpent algorithm does not make use of any tables or s-boxes. This independence from tables also ensures that cache usage does not leak any information about the algorithm. Similar to the RC6 algorithm, when performance was considered for the AES contest five finalists the Serpent algorithm was the slowest in software. It was however among the top performers in hardware.

IV. ATTACK DEFENSES

Defending against the above discussed attacks can prove difficult. It has been shown that the attack relies on a combination of both software and hardware features, to be successful. Proposed defenses can therefore be classified into one of two categories, software and hardware. Although either method can provide effective defenses, there is generally a drawback associated with each. Below I will discuss some of the proposed defenses, and their tradeoffs.

A. Software Defenses

Generically software defenses share one common advantage; they can be deployed on most currently running systems. This allows for quicker implementations, and should keep the cost from being a large factor. Along with this many of them suffer the same drawbacks, most commonly a decrease in performance.

1) Constant Time Code

The first proposed method to prevent cache attacks was discussed by Bernstein in the same document publishing his cache timing results [1]. His attack focused on the execution time variance, and therefore he discusses what it would take to write constant time code. He proposes four main considerations that would need to be addressed. 1) Align relevant pieces of code in memory, 2) Pre-load code after any non-relevant code has been run, 3) Consideration for interrupts, 4) Handling of architecture specific variances.

As the timing differences in the cache timing attack are related to evictions in the cache, it is important to consider where sensitive code is stored in memory. Structures such as tables, the stack frame, and the key schedule, all should be aligned in memory to prevent them from mapping to the same cache locations. This would prevent the algorithm from causing evictions of its own code, and reduce the timing variances. The down side to this is that the code would be architecture dependent. If a device does not have a large enough cache to store all required information without conflicts, then it is not possible to remove this variation. On systems which do have a large enough cache, the code will be architecture dependent, as different architectures have varying structures. Finally this defense would only provide protection against a timing attack, as some of the discussed trace methods do not require timing of the encryption software.

The second suggestion, pre-fetching code, can help prevent variances by removing cache misses. By loading all tables into the cache, every access to one will result in a hit, and therefore will execute very quickly. This could also provide some added defense against trace drive attacks, as they would not be able to determine which location was accessed if they are all accessed. It can not be said to guarantee protection against trace attacks, as interrupts can always occur after the pre-loading and the attacking process could still perform its attack. Additionally this measure does provide a significant level of performance degradation.

To test the level of performance decrease pre-fetching can cause, I have run a simulation using the OpenSSL library and

the AES encryption algorithm. To perform this test I compiled OpenSSL version 0.9.8g with out compiler optimizations, and with out the assembly language optimizations. The purpose of the test was to determine the increase in run time, and therefore the overall speed is not relevant, rather the change in speed between tests. I will later discuss the library used in greater detail. For the test I performed five speed tests with the AES algorithm at all three valid input bit sizes. These numbers were recorded, and then I added a short for loop before the execution process which accesses one memory location in each cache line, for all four tables used by the AES encryption. I then re-ran the speed tests for all three input bit sizes. The results for the 128 bit input size can be seen in Fig. 1. The full results as well as the modified section of code are included in the appendix.

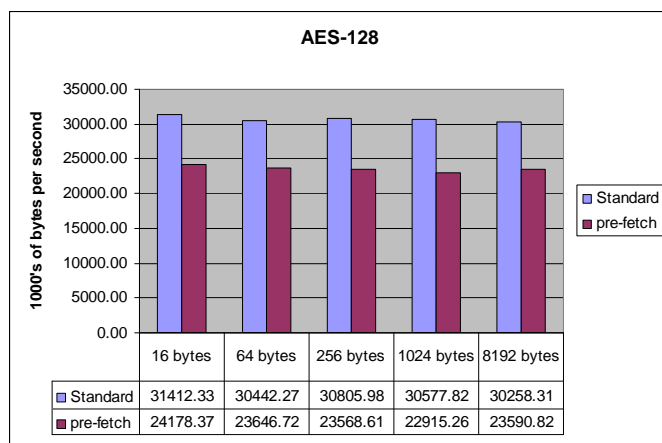


Figure 1. AES timing comparison of standard execution vs. pre-fetching of data.

The results varied slightly between the different tests, and with a sample size of only five runs some variances were present. I averaged the difference across all three input bit sizes and saw approximately a 20% decrease in performance by simply executing a fetch before the encryption operation.

The third suggestion for constant time code suggest that interrupts must be accounted for not necessarily possible in software alone. Some techniques from the following section on operating system assistance may need to be applied. If interrupts are not considered, then operations such as the pre-fetching discussed earlier could not be guaranteed to have any affect at all.

Finally there must be specific considerations for the exact architecture on which the application will be run. Bernstein gives two examples of architectures which can cause timing variations. The first are load-store conflicts. This is a documented behavior of Pentium III and Pentium M processors. When loading from a cache line, which aliases to a recently written line, the load will stall until the write has finished. This can cause small timing variances. The other example involves cache bank loading variances, in which a 1 cycle delay can be caused under specific loading requests which conflict. These small variances which he discusses

might be too small to provide useful data at this time. Even so, they do illustrate the point that specific architectures can have their own influences which a programmer would need to consider if they believe their code to be constant time.

2) Operating system integration

When looking at defenses to cache attacks, the operating system has more control than standard user space applications. With access to privileged commands and control, it is worth discussing steps that can be implemented to help defend against cache based attacks. As with any proposed defense, the operating system providing the defensive measures can still have its own drawbacks.

The first such suggestion would be to implement cryptographic primitives as operating system services. When an application needs to execute these more sensitive operations, it would make a system call to have them executed. This would then allow the operating system to guard against attacks in ways user applications can not provide. For example the operating system may be able to prevent interruptions from occurring. The drawback for this sort of implementation is a loss of flexibility. If the operating system must have support for secure execution, any change in algorithms would require an operating system upgrade or patch. It would also limit the security to algorithms which have been in use long enough, to justify the operating system integrating calls for it.

A second possible method of providing security through the operating system would be to have the operating system provide a set of guarantees that sensitive code could be executed under. For example, the operating system could guarantee that the cache will be flushed after sensitive code has been executed, or that interrupts will not be allowed, etc. Programs could execute their sensitive code under these guarantees, and be notified if the requested guarantee was not satisfied. If the guarantee failed, the program could take appropriate steps, and possibly try again for a successful secure execution. This would avoid the need to integrate specific code into the operating system, but still can have some draw backs. Some of the steps taken, such as a cache flush on context switch, can be detrimental to performance beyond just the cryptographic application. Other services using the cache can be degraded by having the cache flushed every time the cryptographic application runs. Secondly, if code is guaranteed execution with out interrupts, this opens up the possibility of starving other applications for time on the CPU. As the cryptographic application's usage increases, other applications on the same machine will suffer.

3) Avoiding Memory Access

As demonstrated earlier, the cache attack relies not only on the sharing of the cache, but the relationship of the addresses accessed in the cache. Therefore it should follow that the cache functionality can be left alone, and the algorithms can avoid accessing memory for cryptographic primitives.

During my algorithm analysis, I discussed RC6, and serpent. Both of these algorithms avoid accessing memory in such a way as to provide data leakage. It was also noted that neither

of them performed very well in software tests. Therefore an algorithm switch can be viewed as a performance decrease. If further work on either of these, or future algorithms, can provide the speed equal or greater to current tests, this side channel could be avoided.

Algorithms which do use s-boxes or other structures generally implemented as large tables in software could be re-written. S-box lookups can be seen coded as a series of functions which would prevent the memory access used during cache attacks. This also provides the side effect of requiring less memory to run, since the large tables will not be stored in memory. The problem with this method is that it generally provides a very large decrease in performance, as operations which were previously very quick lookups will be replaced with multiple arithmetic operations.

There is another method to avoid memory lookups, which may help avoid the high costs of replacing s-boxes with function calls. A method known as bit slice can operate on one bit of input at a time, while operating on a number of inputs simultaneously. For example, a register will be filled with the first bit of multiple blocks to be encrypted. Transformations are done on this register, which in working on the first bit from all the blocks. A second operation will be done on a register which contains the second bit from all the input blocks being encrypted. By operating in this way, the processor is achieving a sort of parallel execution, similar to using pipe lining in hardware to increase through put.

The bit slice method does incur some additional over head, as the input and output must be re-formatted to fit the expected formats. Also, there would need to be a high enough rate of input to actually take advantage of operating on multiple blocks at once. If small amounts of data are encrypted at a time, the algorithm could end up being very inefficient. One such implementation was created for AES by R. Konighofer. In his implementation he demonstrates a method for executing the AES algorithm, with a bit slice implementation. His fastest implementation tests show a performance decrease of only 5%, while using 8% less code memory, and 93% less data memory [7]. This option does seem promising, but there are still considerations which must be considered. The bit slice implementation takes into account available registers, and the size of them. Therefore the code and performance is architecture dependent. As mentioned before, to achieve full speeds, there must be enough data to allow for parallel operations to be under gone at the same time. Even with these considerations, this method could provide increased security for certain AES implementations for which these conditions are all meet.

4) *Data Oblivious Memory Access Patterns*

If memory must be used in the algorithms, then another suggested defense method would be to remove the data relationship to the memory access pattern. As long as memory access does not provide useful information to an attacker, there is no need to avoid it.

To avoid this, algorithms could be created which can still use tables, but with the added requirement that the table

lookups are not data dependent. A good example of this would be the previously discussed RC4 algorithm. Although RC4 does use a large table structure for key storage and producing pseudo-random output, these references are not dependent on the input data or the key in use. Therefore what locations are accessed does not leak information about the key. It also avoids providing an attacker any method of affecting which table entry is accessed, removing the ability to correlate memory locations with know plain text inputs. The RC4 algorithm might not be acceptable for applications which currently use AES, but it does provide an example of how tables can be used and not cause leak data.

A second method of removing memory access patterns from the attack would be to run dummy encryptions along side the actual encryption. This would generate seemingly random memory accesses to the same tables as the real encryption, making it much more difficult to correlate data. The problem with this method is the same with any such method which merely provides noise, with enough samples the noise can be factored out. Therefore this method could be used to increase the difficulty, but can not provide guaranteed protection. With some of the more efficient attacks from the trace driven category, this time increase could be so insignificant as to not be considered worth the over head of the additional encryption processes.

Finally, as discussed under constant time code, an algorithm could access every location of a table, every time it accesses the table. This would prevent any knowledge of what section of the table was actually used. As shown before with a single table access for the entire encryption of AES, the performance decrease would be far to great for this to be a useful defense.

5) *Dynamic Table Storage*

In cases where the memory access can not have the data dependency removed from it, another option is to remove correlations between these access and the memory address, and therefore the cache mappings. There two methods are discussed in [2] on how this might be achieved.

First, multiple copies of the required tables can be used. Each copy can be stored in different memory locations, and accesses to the tables can be randomly distributed between them. This would provide varying memory addresses for accesses and remove the ability to discover what locations are being accessed. Similarly, rather than multiple copies of tables, one can be used, but it can be randomly moved around in memory. This would still provide the effect of using multiple memory addresses. The problem with this sort of solution lies in the cache hit rate. As it was shown a cache hit is much quicker than a miss. If the data location changes, then the cache will not produce a hit and the much longer miss time will be required. Therefore this implementation decreases performance by increasing the number of misses during execution of the code.

Secondly it would be possible to randomly permute the data inside the table, preventing a memory address from always correlating to the same table location. As the table would need to be permuted multiple times, this would still generate the

same cache hit and miss issues as the previous method. It would also require some form of entropy to provide the permutation of the table. If the permutation followed a set algorithm, the cache accesses could still be correlated by using knowledge of the new locations after each permutation.

6) *Combinations of Defenses*

It has thus far been shown that many software defenses could provide some level of defense against cache based attacks. As none of them seem fit to solve the problem on their own, it may be possible to combine some of the above methods to provide better protection, while decreasing some of the drawbacks.

Brickell et al propose a method [8] which combines multiple defense mechanisms for the AES algorithm. Their method consists of three basic defenses 1) compact s-box implementation 2) pre-loading of relevant cache lines 3) frequently permuting tables. A key to their method is the flexibility achieved when combining all three of these methods. The number of rounds the compact s-box are used during, as well as how frequently permutations are done, can be adjusted according to the level of security that is actually required.

To start with, they suggest using the small 256byte s-box for AES, during the first and last rounds. These are the most vulnerable to attack, and by using the smaller s-box for only these rounds the middle rounds can still achieve faster execution using the larger tables. It does require additional calculations when these smaller tables are used, as they are not built into the tables.

A second advantage to using the smaller tables is a greatly decreased cost of pre-fetching. With the tables being only 256 bytes, they require much less overhead to load into memory. Therefore the tables lookups can be removed from leaking any information.

Thirdly, they suggest permuting tables as often as is necessary by the threat model. The rate of permutation would depend on the threat profile. As discussed above the greater the rate of permutation the more the performance is affected by them.

They do provide simulations of their method with multiple threat profiles considered. Their results show somewhere between 1.35 and 2.85 performance loss. This is still a considerable loss of performance, and if attacks become advanced enough to require the more secure method, the decrease in performance becomes rather drastic.

B. *Hardware Defenses*

A second method of providing defense against cache attacks is to ensure that the underlying hardware prevents such an attack. If the cache can prevent this information from leaking, all algorithms would be defended against such attacks. This could also provide a means of defense which does not add a performance decrease, as many of the software defenses. The tradeoff with hardware is one of cost. New hardware would have to be developed, and any machine which needed protection would need to be upgraded to new hardware. This

is a very large cost, and one which might not be justifiable if software defenses can provide sufficient protection.

1) *Partitioned Cache*

The first such hardware defense comes from D. Page, and is called a partitioned cache [9]. The partitioned cache provides security to applications by removing the shared property of the cache. The CPU's cache is divided up between running processes into partitions. A process can not access cache from another processes partition. This provides protection from trace driven attacks by not allowing the cache to be tested for hits and misses. Timing attacks would not be guaranteed to be mitigated by this technique. Although evictions would not occur due to other running processes on the box, the encryption algorithm could still cause evictions of its own space while running. Additional measures would need to be taken to ensure that the specific software was aware of this and did not cause evictions internally.

Implementing this technique would require additional hardware, to support the process ID of each cache location. The major problem with this design is the performance impact can be high. As processes lock down sections of the cache, other processes will start to have an increased miss rate. The cache can be used inefficiently, as some processes might need more cache than they have available, while others have unused lines locked in their partition.

2) *Partition Locked Cache*

The next hardware suggestion is similar to that of a partitioned cache, but addresses some of the issues caused by cache partitioning. Z. Wang and R. Lee [10] suggest a cache scheme which locks lines of cache to a process, but without the separate partitions. When a sensitive piece of code is run, either by accessing memory locations marked as sensitive, or by a program designating what is sensitive, a lock bit is added to the cache line along with the process ID. Lines which have been locked can not be evicted by other processes. This scheme still would need software measures to guard against applications causing evictions of their own lines in cache, which might be used for timing attacks.

By only locking for processes which require the security and only locking lines which are actually in use, the partition locked cache prevents much of the inefficiency in the basic partitioned cache. It does still suffer large penalties in direct mapped caches, or situations where the secured code takes up a larger percent of the available cache. In either case, many cache accesses of other processes, besides the secured application, will end up being a miss, and require pulling data from main memory. In their testing, the smallest cache size, direct mapped, performed at about 2/3 that of a standard cache [10]. As the cache size increased, and the associativity was increased, the results reached the same performance as a standard cache.

3) *Random Permutation Cache*

The second method detailed by Z. Wang and R. Lee operates in a completely new fashion than the previous methods. They suggest a cache scheme which randomized interference, known as the random permutation cache. This

cache architecture uses permutation tables to randomize the mappings between memory addresses and cache locations for processes which require protection. When cache lines are replaced, if the application is not protected, they are replaced as normal. When a protected application needs to replace a cache line, a random cache line at a random location is replaced, and the permutation table updated so the protected application can find the data. Therefore an attacker, who fills the cache with his own data, will see random lines replaced, and not be able to infer any information from it.

To provide protection for timing attacks, there must also be considerations for internal conflicts within a protected application. This is done in a similar way, both a protected bit and an ID field are added to the cache. When a process replaces one of its own cache locations, if the protection differs between the two lines, a random line is selected and replaced. This provides randomization of conflicts even when they are internal to the protected process. Therefore any timing differences will vary randomly and not provide any information as to which lines the process was actually accessing.

In simulations performed by the authors of this cache scheme, it performs very well. When compared to a standard cache scheme, they report less than a 2% difference in the two schemes. The random permutation cache would require additional hardware for the ID and the protection fields, which will increase costs of the architecture slightly over current builds. Also the suggested layout does add additional gates to assist with the comparisons, and therefore increases power usage of the system. Their results showed about a 10% increase in power consumption using this scheme [10].

The random permutation cache seems to be the most effective hardware solution I evaluated. Although it does come at the cost of some additional power consumption, the protection would allow all current code to be run securely with out any modifications.

V. THE TRADEOFF OF SECURITY AND PERFORMANCE

After reviewing many of the possible defenses which can be deployed against this side channel attack, it is clear that none of them provide perfect protection with out some tradeoff. What is appropriate for an application may vary by situation, as some defenses might work better than others.

It is important to understand the problem, and find measures which ensure that applications are secure, but remain useable. Some situations might be fulfilled by a bitslice implementation of AES. If a software implementation handles enough encryptions to realize the full speed, this could allow it to remain compliant with devices already running the AES algorithm. Another situation might involve hardware device, with smaller amounts of data to be encrypted. In such a situation using the Serpent algorithm could prove very beneficial, as it performed very well in hardware while still protecting against this type of attack.

To take a look at some of the trade offs that have been

made, I have reviewed two open source libraries which implement the AES algorithm, OpenSSL and Crypto++. These libraries provide a good demonstration of how measures can be taken to provide defense, while still trying to keep performance.

A. *OpenSSL 0.9.8g*

The OpenSSL project provides many cryptographic implementations in an open source library. Since the publication of the cache based side channel attacks, some defenses have been applied to the library to provide some protection against these attacks.

It should first be noted that the OpenSSL implementation provides all the defenses in the assembly language code for the AES algorithm. If the library is compiled with the assembly flag removed, the standard code will be used. This code does not provide defenses of any kind to these attacks. The test I previously ran using the OpenSSL library where done with out the assembly language optimizations, to ensure that only the specific defense was tested.

The first defense mechanism is the use of compressed lookup tables, which are pre-loaded. This provides a defense against timing attacks, trace attacks could still possibly use interrupts, but it becomes much more difficult. These defenses are only implemented for CBC mode of execution. On Pentium architectures this resulted in a 30% speed decrease, while a Pentium III saw a 10% speed decrease.

Secondly considerations are taken to align the stack, key schedule, and s-boxes in memory so they will not conflict with each other. This should help reduce timing variances caused by the application it's self. The implementation of this alignment greatly decreases the performance of the 16 byte blocks in CBC mode. To prevent this, data passed to the program is checked to determine if any aliasing exists. If the user program has aligned the memory, this performance decrease can be avoided.

The OpenSSL project has decided to implement defenses in specific cases of their functions. They have taken the steps to provide defenses against timing attacks, while providing some trace defenses. As discussed earlier, interrupts can theoretically still allow trace attacks to be initiated against this library, although the level of difficulty is much higher than even a standard trace attack with no defenses provided. It is worth noting that time was taken to code the most common AES function in assembly language, indicating that every bit of performance matters. Even so, the implemented defenses do come with a performance decrease, but the trade off is consider necessary for the increased security. The library mainly provides protections for two modes of operation, which reflects the difficulty in writing specialized code to handle the attacks.

B. *Crypto++ 5.5.2*

The Crypto++ project provides libraries for many cryptographic functions in a C++ format. The library for the AES function does have some defenses added to try and

reduce the threat of a cache side channel attack.

Similar to the OpenSSL library, compressed tables are used. These tables are only used for the first and last rounds of the AES encryption, as these are the most vulnerable to attack. These smaller tables are pre-fetched before executing the encryption. It is noted that steps are taken to prevent compilers from optimizing the code away, but no guarantee can be made. These smaller tables, and the pre-fetching will cause a decrease in performance, over an unprotected algorithm. The threat from the side channel is great enough that the protections are considered necessary.

The middle encryption rounds use the larger and faster fully expanded tables. This library also implements assembly code if supported, to increase performance as much as possible.

VI. CONCLUSION

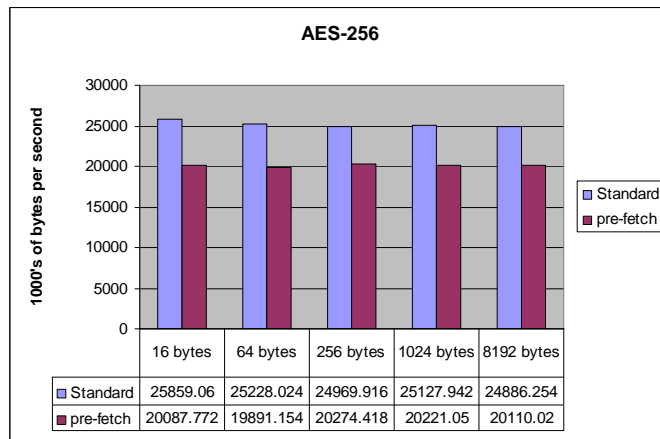
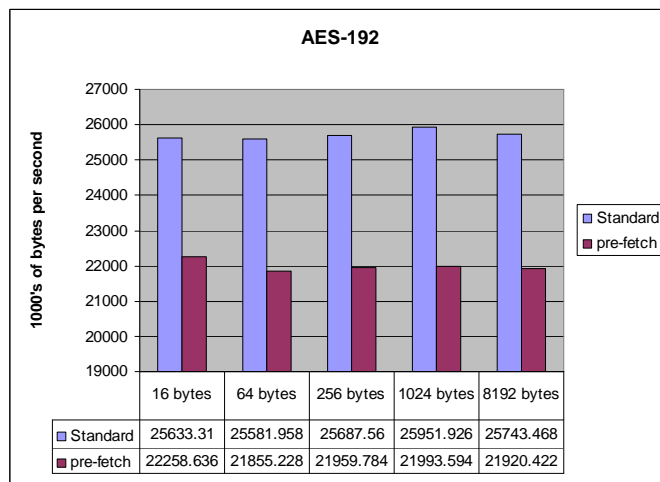
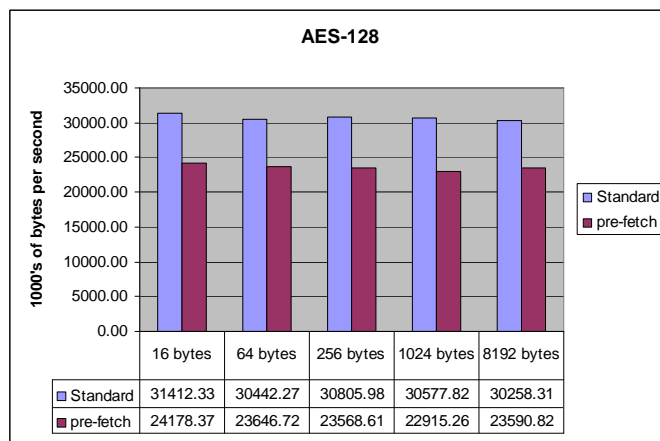
Side channel attacks based on the processors usage of cache have been shown to be a real threat to cryptographic applications today. Although the attacks are generally complicated, they are recently new addition to side channel methods, and as they are researched and understood they will continue to mature. As no proposed defense can be applied to all situations where cryptographic algorithms may be in use, it is important to understand what threat they provide, and how proposed defenses work. By analyzing how the application will be used, the defensive measure might be correctly selected to provide the most benefit with the least amount of performance trade off. Many of the defensive suggestions have potential to grow, and may eventually provide a general solution which protects for most cases.

In both hardware and software there have already been some potentially promising suggestions for defensive mechanism. With the great cost of manufacturing new hardware to defend against this attack, I would expect that software measures will be the front line of defense. If the security threat reaches a high enough level, hardware options may become more viable.

APPENDIX

The following charts show the test run with OpenSSL to determine an average decrease by pre-fetching data in the cache. Each chart shows data for a specific key size of AES. The code was written for a 32 bit processor with 64 byte cache lines. Since every location in the table has 4 bytes, this loop only loads one location out of every 16. This should pull in the entire cache line, with out any additional reads.

```
int location, load=0;
for(location=0;location <= 255;location+=15){
    load = Te0[location];
    load = Te1[location];
    load = Te2[location];
    load = Te3[location];
}
```



REFERENCES

- [1] D. J. Bernstein., (2005, Apr) Cache-timing attacks on AES. [Online]. Available: <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>
- [2] D. A. Osvik, A. Shamir, E. Tromer, "Cache Attacks and Countermeasures: the Case of AES (extended version)," presented at The Cryptographers' Track at the RSA Conference 2006, San Jose, CA
- [3] D. Page, (2003, Mar.). Defending against cache-based side-channel attacks. *Information Security Technical Report*. 8(1), pp. 30-44.

- [4] M. Neve, J. Seifert, Z. Wang, "Cache time-behavior analysis on AES," unpublished
- [5] J. Bonneau, I. Mironov, "Cache-collision Timing Attacks Against AES," in *Lecture Notes in Computer Science*, Heidelberg, Germany, Springer, 2006, Volume 4249/2006, pp. 201-215
- [6] M. Neve, J. Seifert, "Advances on access-drive cache attacks on AES," in *13th International Workshop on Selected Areas in Cryptography*, Montreal, 2006, pp. 147-162
- [7] R. Konighofer, "A Fast and Cache-Timing Resistant Implementation of the AES*," in *Lecture Notes in Computer Science*, Heidelberg, Germany, Springer, 2008, Volume 4264/2008, pp. 187-202
- [8] E. Brickell, G. Graunke, M. Neve, J. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," Intel Corp. Hillsbro, OR, Cryptology ePrint Archive, Rep 2006/052
- [9] D. Page, "Partitioned cache architecture as a side-channel defense mechanism," Dept. Of Computer Science, University of Bristol, United Kingdom, Cryptology ePrint Archive, Rep 2005/280
- [10] Z. Wang and R. B. Lee. (2007, May). New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*. 35(2) pages 494-505