Partitioned Cache Architecture as a Side-Channel Defence Mechanism

D. Page

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
page@cs.bris.ac.uk

Abstract. Recent research has produced a number of viable side-channel attack methods based on the data-dependant behaviour of microprocessor cache memory. Most proposed defence mechanisms are software based and mainly act to increase the attackers workload rather than obviate the attack entirely. In this paper we investigate the use of a configurable cache architecture to provide hardware assisted defence. By exposing the cache to the processor and allowing it to be dynamically configured to match the needs of a given application, we provide opportunity for higher performance as well as security.

1 Introduction

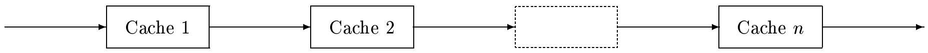
State of the art cryptanalysis has conventionally resided in the realm of mathematicians who seek techniques to unravel the hard problems on which modern cryptosystems are based. Side-channel analysis moves the art of cryptanalysis from the mathematical domain into the practical domain of implementation. By considering the implementation of cryptosystems rather than purely their specification, researchers have found they can mount attacks which are of low cost in terms of time and equipment and are highly successful in extracting useful results.

Side-channel attacks are based on the assumption that one can observe an algorithm being executed on a processing device and infer details about the internal state of computation from the features that occur. Such observation is typically performed by passive monitoring execution features such as timing variations [15], power consumption [16] or electromagnetic emission [1, 2]. Attacks usually consist of a collection phase which provides the attacker with profiles of execution, and an analysis phase which recovers the secret information from the profiles. Considering power consumption as the collection medium for example, attack methods can be split into two main classes. Simple power analysis (SPA) is where the attacker is given only one profile and is required to recover the secret information by focusing mainly on the operation being executed. In contrast, differential power analysis (DPA) uses statistical methods to form a correlation between a number of profiles and the secret information by focusing mainly on the data items being processed. Both these techniques present a clear danger to security sensitive applications, especially since attacks can be mounted with low cost, commodity signal processing equipment.

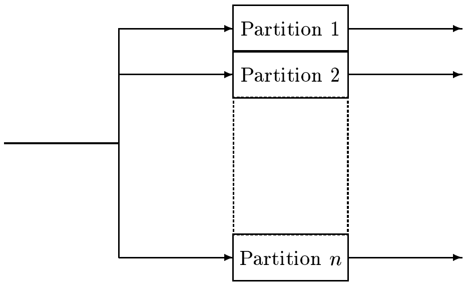
As understanding side-channel attack and defence has evolved, new methods of inferring secret information from execution profiles have emerged. One such method is monitoring the data dependent behaviour of the processor memory hierarchy and, in particular, any cache memories present. The concept of using cache behaviour as a side-channel was first mooted by Kocher [15] who noted the effect of memory access on execution time, and then Kelsey et al. [12] who predicted that the timing dependent behaviour of S-Box access in Blowfish, CAST and Khufu could leak information to an attacker. This was followed with more concrete attacks on DES by Page [21], who assumed cache behaviour would be visible in a profile of power consumption, and Tsunoo et al. [28, 27] who simply required that an attacker timed the cipher over many executions. Further breakthroughs were made by Bertoni et al. [4] and Bernstein [3] who applied power and timing analysis attacks to AES. The former work shows cache behaviour is observable in a power trace, the latter shows that attacks can be mounted remotely; both further magnify the danger of cache attacks in an operational context. Finally and most recently, Percival [22] demonstrated an attack against CRT based RSA utilising the Hyper-Threading capability of Intel Pentium 4 processors but essentially relying on cache behaviour as a means of leaking information. Subsequently, such inter-process attacks have been extensively investigated by Osvik et al. [19].

In this paper we investigate the use of partitioned cache architecture as an aid to defending against cache based side-channel attack. Such designs dynamically split the cache memory into protected regions. As a result, the level of cache interference is drastically reduced and the cache can be configured specifically for an application rather than optimising for the average case. Traditionally, such partitioned caches have been proposed as ideal for embedded and media processors due to their size, performance and power characteristics. This alone provides a compelling reason for their use in the same computational environments which are most vulnerable to conventional side-channel style attacks. However, features such as dynamic configuration of the address translation function and protection or locking of data in the cache also provide a number of opportunities for countermeasure against both profile and timing driven cache based side-channel attacks. To restrict our focus, we primarily consider block ciphers, and access through the cache to S-box style tables, since the majority of attacks exist in this context. Further, we concentrate only on data caches by assuming instruction and data access is segregated. We try to take a processor agnostic approach by leaving open several implementation choices which do not otherwise effect our work.

The paper is organised as follows. In Section 2 we give an introduction to cache partitioning before describing the experimental cache architecture used subsequently. In Section 3 we recap on cache based side-channel attack methods; we describe proposals for software based countermeasures before outlining



(a) A conventional memory hierarchy.



(b) A partitioned cache.

Fig. 1. Using filters to describe conventional and partitioned cache hierarchies.

how partitioned caches can be utilised to provide hardware based alternatives. Finally, we present some concluding remarks and areas for further work in Section 4.

2 Caches and Cache Partitioning

A cache is a small area of fast RAM and associated control logic which is placed between the processor and main memory; for an in depth description of cache design and operation see [9, Chapter 5]. The area of RAM is typically organised as a number of cache lines, each of which comprise a number of sub-words that are used to store contiguous addresses from main memory. Since the cache is smaller than main memory, it stores a sub-set of the memory content. As a result of locality in the incoming address stream, the cache reduces the load on the rest of the memory hierarchy by holding the current working set of data and instructions. Accesses that are serviced by the cache are termed cache-hits and are completed very quickly; accesses that are not held by the cache are termed cache-misses and take much longer to complete since main memory must be accessed. Since locality guarantees we should get more cache-hits than cache-misses, performance of the average case application is improved. However, given that many addresses in memory can map to the same location in the cache, data items can compete for space and evict each other; this is termed cache interference or contention.

As an aid to understanding complex cache designs, Weikle et al. [30] use the concept of optical filters as a metaphor for how such systems operate. As shown in Figure 2a, each level of the memory hierarchy acts as a filter which translates a input stream of memory references into an output stream that is dependent on the properties of that level. Utilising the concepts of temporal and spacial locality, the hope is to combine these filters so that they remove as many references as possible, reduce the load on slower parts of the memory hierarchy and maximise performance. One draw back of this approach is that the references for all data objects in all running processes are conglomerated into one monolithic stream. The memory hierarchy must be optimised for the average case program and can thus fall foul of interference patterns as a result.

Although specific designs differ slightly, a partition cache is a direct-mapped style cache that can be dynamically partitioned into protected regions by the use of specialised cache management instructions. By modifying the instruction set architecture (ISA) and tagging memory accesses with partition identifiers, each access is hashed into a partition dedicated to dealing with it. Phrased using the metaphor of caches as filters, the idea of a partitioned cache is to act as a refraction lens or prism by separating the stream of input references into a number of sub-streams; see Figure 2b. A coarse grained example of this technique in action is where a segregated, Harvard style instruction and data cache architecture is used. Although on a smaller and less configurable scale, this choice performs the same function as cache partitioning by splitting the reference stream into instruction and data streams.

The decision of how to split the resources into instruction and data caches in a Harvard style architecture is performed at design-time by optimisation for an average case which is unlikely to suit all application programs. Unlike conventional caches, the partitioned cache is visible to software running on the host processor. This allows one to utilise the cache management instructions and load/store mechanism to allocate partitions of the cache to specific data objects and streams of instructions so as to control persistence and eliminate interference at run-time. Typically, one would expect such cache management instructions to only be available when the processor is in protected mode and hence managed by the operating system; this ensures user processes cannot examine or alter each others cache configuration. For example, the act of one process accessing a partition owned by another would result in an exception.

2.1 Previous Work

Enforcing segregation of processes cache content from each other is not a new idea. The cheapest way to implement such a scheme, and one typically used to improve performance, is by using software based layout rules to place instructions and data in the process image so they do not interfere with each other in the cache. For example, among a vast amount of work in the area Mueller [18] presents a software based partitioning method for direct-mapped caches with applications in real-time computing, while Calder et al. [5, 8] provide further mechanisms for cache conscious layout. Alternatively, one can consider allowing control of the cache by exposing it to the programmer; one might view this as a less general form of scratch-pad memory. For example, Wagner [29] allows the programmer to control the address translation mechanism to facilitate cache conscious loop blocking in kernels such as matrix multiplication. Zhang et al. [31]

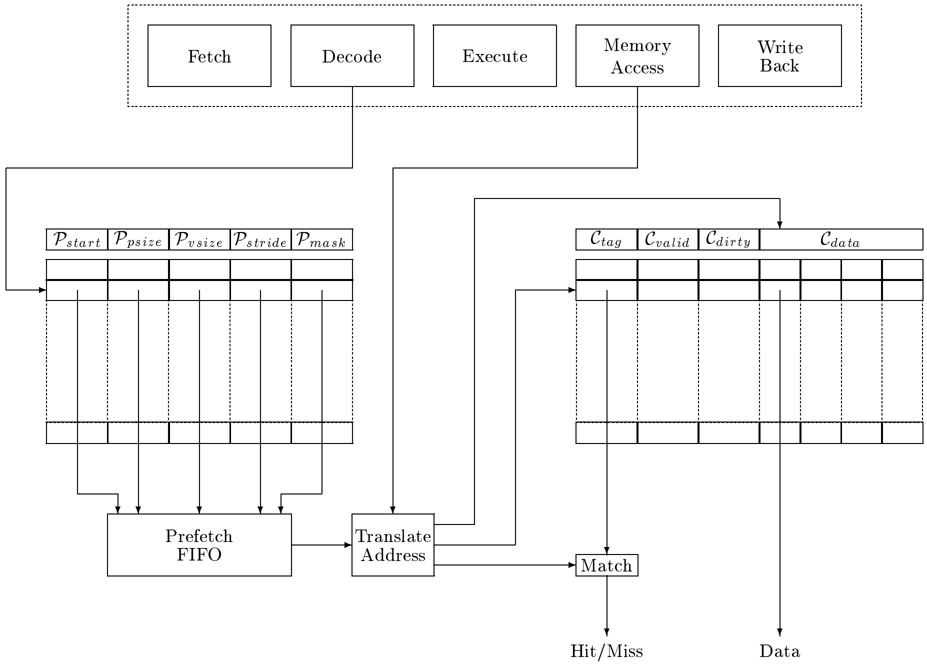


Fig. 2. A block diagram describing a partitioned cache.

describe a cache with configurable levels of associativity that be controlled by the programmer.

Perhaps the first hardware assisted cache partitioning designs were presented by Juan et al. [11] and González et al. [6] in the context of high performance computing. As interest in the area has increased, further designs have included those of Page [20] and Irwin [10], who focus on multi-threaded architectures and compiler directed partitioning; Ranganathan et al. [24], who focus on use in media applications; and both Kim et al. [13] and Petrov and Orailoglu [23], who focus on low-power implementations. Although implementing such a device is clearly a problem in conventional commodity processors, within domain-specific and high-volume markets as is the case with embedded processors, it has already been investigated. For example, a precursor to the SH5 was produced by ST Microelectronics that utilised a degree of cache partitioning detailed in later patents on partitioning hardware [25]. This media-oriented processor used a fixed number of partitions to segregate memory accesses produced by different system constituents.

2.2 Cache Design

Assuming a RISC style processor architecture, Figure 2.2 describes the operation of a simple partitioned cache. From here on we assume each cache line is composed from a number of sub-words that are each one byte, that is 8-bits, in size and the memory system is byte addressed; clearly this might differ depending on the exact architecture. A conventional, direct-mapped storage structure is coupled with a second structure which houses the cache configuration. Each access to the cache is tagged with a partition identifier which is used to access this configuration data. The resulting information is used by the address translation function to map the address into the correct line and sub-word the storage structure. One of the perceived disadvantages of this design is the potential for an increased critical path length as a result of the extra look-up into the cache configuration. In processor designs that use a high clock speed, this is certainly a problem. However, aside from simply accommodating this increase with a slower clock speed we can somewhat reduce the overhead in a pipelined design. Depending on the processor architecture, the partition identifier could be either a register or immediate operand; in either case we can are-fetch the associated configuration data in the decode phase and hide the cost of the extra look-up.

We augment this basic design with three features which further enhance the degree of configurability, and hence flexibility, of the cache:

- Firstly, we include the concept of strided cache lines. Instead of sub-words in a cache line being contiguous addresses in memory, they are permitted to be spaced apart by a fixed distance; this distance is the stride which can be configured on a per-partition basis. The benefit of strided cache lines is realised, for example, during execution of media applications who often access image data in this manner: the resulting cache behaviour exhibits less interference and higher performance as a result of catering for it.
- Next we introduce the concept of adaptive line size; see for example the work of Tang et al. [26]. Essentially, we allow each partition to be configured so that the line size, and hence size of transfer between the cache and memory, can be set at run-time. This is implemented by defining a fixed physical line size and allowing a virtual line to span a number of physical lines. A cache-hit is serviced in the same way normal; a cache-miss causes an entire virtual line to be retrieved from memory.
- Finally we define a mask, or offset value for each partition which acts to perturb the address translation. This essentially allows the address translation to be randomised on a per-partition basis and is similar in concept to work on XOR based placement strategies; see for example the work of González et al. [7]. Adding the mask to the original address allows virtual movement of addresses in memory with respect to cache operation but without the cost of actually moving them. Clearly this perturbed address is only used for cache operation: when addressing memory during loads and stores the cache uses the original address.

Given an access to address $\mathcal{A}'$ using partition $\mathcal{P}$, the cache first retrieves the configuration data yielding:

- $\mathcal{P}_{start}$, the line at which the partition begins.

- $\mathcal{P}_{psize}$, the number of physical lines in the partition.
- $\mathcal{P}_{vsize}$, the number of physical lines per virtual line.
- $\mathcal{P}_{stride}$, the stride between sub-words in the partition.
- $\mathcal{P}_{mask}$, the address perturbation mask for the partition.

We assume that there are $\mathcal{C}_{lines}$ cache lines in total and that each line has $\mathcal{C}_{words}$ sub-words in it. The address is perturbed using the mask to give $\mathcal{A} = \mathcal{A}' + \mathcal{P}_{mask}$. From this, the address translation function computes the physical line and sub-word, denoted by $\mathcal{A}_{pline}$ and $\mathcal{A}_{pword}$, as follows:

$$\begin{aligned}\mathcal{A}_{pline} &= ((\mathcal{A} \gg \mathrm{lsb}(\mathcal{P}_{stride}))/\mathcal{C}_{words}) \bmod \mathcal{P}_{size} \\ \mathcal{A}_{pword} &= (\mathcal{A} \gg \mathrm{lsb}(\mathcal{P}_{stride})) \bmod \mathcal{C}_{words}\end{aligned}$$

The required data is therefore located in line $\mathcal{P}_{start} + \mathcal{A}_{pline}$ at sub-word $\mathcal{A}_{pword}$. The virtual line, denoted by $\mathcal{A}_{vline}$, associated with $\mathcal{A}_{pline}$ can be calculated as

$$\mathcal{A}_{vline} = \mathcal{A}_{pline}/\mathcal{P}_{vsize}.$$

Note that in the above $\mathrm{lsb}(x)$ returns the position of the least significant bit of x, $x \gg y$ denotes a logical left shift of x by y bits, and all division is integer division. Indeed, we required that $\mathcal{P}_{stride}$, $\mathcal{P}_{psize}$, $\mathcal{P}_{vsize}$ and $\mathcal{C}_{words}$ be powers-of-two so the scheme is realistically implementable.

As an example, consider a cache with $\mathcal{C}_{lines} = 128$ and $\mathcal{C}_{words} = 4$. We create a partition $\mathcal{P}$ in this cache and configure it with $\mathcal{P}_{start} = 8$, $\mathcal{P}_{psize} = 4$, $\mathcal{P}_{vsize} = 2$, $\mathcal{P}_{stride} = 2$ and $\mathcal{P}_{mask} = 0$. Addresses $0, 2, 4, \ldots, 18$ map into the cache structure as follows:

$$\begin{array}{llll}
\mathcal{A}' = 0 & \mathcal{A} = 0 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 0 \longrightarrow \text{Miss} \\
\mathcal{A}' = 2 & \mathcal{A} = 2 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 1 \longrightarrow \text{Hit} \\
\mathcal{A}' = 4 & \mathcal{A} = 4 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 2 \longrightarrow \text{Hit} \\
\mathcal{A}' = 6 & \mathcal{A} = 6 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 3 \longrightarrow \text{Hit} \\
\mathcal{A}' = 8 & \mathcal{A} = 8 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 0 \longrightarrow \text{Hit} \\
\mathcal{A}' = 10 & \mathcal{A} = 10 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 1 \longrightarrow \text{Hit} \\
\mathcal{A}' = 12 & \mathcal{A} = 12 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 2 \longrightarrow \text{Hit} \\
\mathcal{A}' = 14 & \mathcal{A} = 14 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 3 \longrightarrow \text{Hit} \\
\mathcal{A}' = 16 & \mathcal{A} = 16 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 0 \longrightarrow \text{Miss} \\
\mathcal{A}' = 18 & \mathcal{A} = 18 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 1 \longrightarrow \text{Hit}
\end{array}$$

Notice that our performance is good; the strided cache lines are increasing the density of used data. This, coupled with the fact that our partition is protected from address streams that access other objects, means both spacial and temporal locality have a better chance of being capitalised on. Useful data is likely to be more persistent due to the lack of interference; any pre-fetching will be more accurate as a result. Also notice that as a result of our virtual line size, we remove the miss that address 8 would have otherwise caused: when the miss from address 0 occurred, we fetched lines 8 and 9 rather than just 8. This sequence is dependent on the mask however, for example setting $\mathcal{P}_{mask} = 4$ produces different usage of lines:

$$
\begin{array}{llll}
\mathcal{A}' = 0 & \mathcal{A} = 4 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 2 \longrightarrow \text{Miss} \\
\mathcal{A}' = 2 & \mathcal{A} = 6 & \mathcal{A}_{line} = 8 & \mathcal{A}_{word} = 3 \longrightarrow \text{Hit} \\
\mathcal{A}' = 4 & \mathcal{A} = 8 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 0 \longrightarrow \text{Hit} \\
\mathcal{A}' = 6 & \mathcal{A} = 10 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 1 \longrightarrow \text{Hit} \\
\mathcal{A}' = 8 & \mathcal{A} = 12 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 2 \longrightarrow \text{Hit} \\
\mathcal{A}' = 10 & \mathcal{A} = 14 & \mathcal{A}_{line} = 9 & \mathcal{A}_{word} = 3 \longrightarrow \text{Hit} \\
\mathcal{A}' = 12 & \mathcal{A} = 16 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 0 \longrightarrow \text{Miss} \\
\mathcal{A}' = 14 & \mathcal{A} = 18 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 1 \longrightarrow \text{Hit} \\
\mathcal{A}' = 16 & \mathcal{A} = 20 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 2 \longrightarrow \text{Hit} \\
\mathcal{A}' = 18 & \mathcal{A} = 22 & \mathcal{A}_{line} = 10 & \mathcal{A}_{word} = 3 \longrightarrow \text{Hit}
\end{array}
$$

2.3 ISA Design

Equipping a processor with partitioned cache hardware demands changes to the ISA so that the cache is a visible part of the architecture. These changes need to include how partition identifiers are passed to the memory hierarchy with normal loads and stores, and include extra instructions which manage the cache configuration.

Passing the partition identifier to the memory hierarchy can be achieved in several ways: by adding an extra register or immediate operand to instructions; via a dedicated register which specifies the active partition; or even by using out-of-band address bits to specify the partition. For our purposes, the mechanism is irrelevant: we simply assume the partition identifier is an extra operation to each load and store instruction. We also assume only a basic of cache management interface which must be exposed somehow as instructions:

- `ADDPAR pid, start, psize, vsize, stride, mask`
  Add a partition with identifier `pid` to the configuration, allocating it `psize` physical lines starting at line `start`. Also set the virtual line size of the partition to `vsize` and the stride and mask values to `stride` and `mask` respectively.
- `DELPAR pid`
  Delete the partition with identifier `pid` from the configuration.
- `INVPAR pid`
  Flush the partition with identifier `pid` so that any data stored in it is evicted and potentially written back to the next level of the memory hierarchy.

3 Cache Based Side-Channel Attacks

Consider a theoretical block cipher E_K which encrypts plaintexts using the key K and uses a single S-box S during execution. We assume that all memory access during execution is due to the S-box; this is a vast simplification but somewhat reasonably from the point of view of block ciphers since most of the working data set can be held in registers. Say there are two accesses to the S-box, $S[i]$ and

$S[j]$, using indices i and j to provoke access to addresses $\mathcal{A}_i$ and $\mathcal{A}_j$ in memory. These accesses will usually have little or no locality since by design, the indicies will be randomly distributed throughout the S-box.

As a simple example attack, if the cache is initially empty and the second access results in a cache-hit, we can deduce that up to the bits that select the sub-word $\mathcal{A}_i = \mathcal{A}_j$ and hence $i = j$. Typically, i and j are computed using the secret information K and a plaintext P, for example from the result of a key addition. Roughly speaking, the attacker can use the details of the cipher and a number of collected relationships to recover K using an adaptive plaintext attack.

3.1 Attack Methods

Trace based methods assume that the attacker, by observing a side-channel such as a power consumption, is able to recover traces of cache behaviour. One might view this as analogous to an SPA style attack: each access the executing algorithm makes to memory will be visible in the trace as either a cache-hit or cache-miss depending on how the cache serviced the access. For example, denoting a cache-hit by H and a cache-miss by M, the trace

$$MMHMHH\ldots$$

tells the attacker that accesses one and two were cache-misses while access three was a cache-hit and so on. After matching features in the trace to operations in the algorithm, the the required relationships between indices can be recovered and hence offer a point of attack [21, 4].

Timing based methods use a more statistical, DPA style approach to attack. Since they require a much easier form of monitoring, simple timing of the algorithm rather than a profile of power consumption, are more realistically mounted both locally and remotely. Essentially, such attacks work by assuming more cache-hits means shorter execution time; hence shorter execution time means it is more probable that indices used for any given S-box access are the same. Given this correlation and numerous plaintexts which provoke a short execution time, the attacker can form probabilistic relationships between the indices. In the same sense as above, if the indices are derived from secret information, the skew in probability gives a point of attack [28, 27, 3].

3.2 Defence Methods

Since security in some applications is a high priority, the above attack methods have naturally provoked several countermeasures; see the work of Bernstein [3] and Osvik et al. [19] for a number of for an extensive and modern approaches. Given that instrumenting new a cache architecture can be an expensive and disruptive task, such countermeasures have typically been software based. One option is implement the block cipher such that the execution is somehow constant in terms of cache behaviour:

- Most drastically, one can consider turning off the cache for S-box access, essentially employing cache-bypass to always load data directly from memory. By eliminating the potential for cache-hits and cache-misses we reduce performance significantly but ensure that each access takes the same length of time.
- For small S-boxes, we can consider pre-fetching or warming their content into the cache before execution begins. This essentially makes all S-box accesses cache-hits and hence constant time. However, this is only true if the S-box content is never evicted by other data or instructions and the S-box fits entirely into the cache: neither of these assumptions are guaranteed and hence the method can only be described as statistically sound.
- As proposed by Bernstein [3] and many others, a good approach is to avoid S-box tables altogether and use some form of computed non-linear transformation instead. This not only offers greater assurance of constant time access, but allows the potential for parallel execution of such transformations which are denied by the need for sequential memory access. As an example, one might consider the transformations described by Klimov and Shamir [14].

Alternatively, one can randomise execution in order to at least partly mask features in any collected side-channel profile:

- In the most simple case, one can randomly insert dummy load operations in the execution so that the execution time is randomised to some extent. Realistically, this method is not sound since the randomisation is simply noise that can be statistically removed. Additionally, since extra operations need to be serviced, the overall average execution time might increase by an unattractive factor. However, the approach has some value when considering attacks which operate on behaviour traces rather than timing information: with enough dummy loads inserted one cannot be sure if a given cache-hit or cache-miss is produced by real or faked execution.
- In a similar vein, random reordering of memory accesses will reduce the correlation between a captured behaviour trace or execution timing and the input and algorithm. This can be achieved, for example, by using a non-deterministic processor architecture [17] but must be careful not to introduce potential hazards from the reordering.
- Alternatively, one can insert actual random delays in the execution to randomise the overall execution time. This suffers from the same drawbacks of inserting random load operations in the sense that the statistical noise can be removed and will potentially increases the average execution time.

3.3 Using a Partitioned Cache

The benefits of adding a partitioned cache to devices which are vulnerable to side-channel attack are two-fold. Firstly, embedded processors at the heart of such devices are typically constrained in both computational and storage ability. Any method of masking such deficiency is hence valuable; the proposed architecture has a number of well studied advantages in terms of size, performance and power characteristics, particularly when operating with small, kernel sized applications.

More importantly in terms of the current work, one would hope to use the high degree of flexibility and configurability to combat side-channel attacks against more conventional designs. With this in mind, and although a perfect defence mechanism is somewhat unrealistic, we propose three areas in which a partitioned cache can at least improve on current cache designs:

Remark 1. Since a partitioned cache segregates the cache behaviour of one process from another, it seems to totally prevent inter-process style attacks such as that of Percival. This is essentially achieved by removing the cache as a shared resource: although the cache hardware is still shared, access by a process to partitions of another process is invalid. Further, the segregation mechanism prevents intra-process interference in the sense that if one has enough space to store the S-box entirely in the cache, partitioning offers a mechanism to lock it once pre-loaded. This offers a similar method of defence as some existing processors already provide, but in a more flexible format.

Segregation has a secondary benefit in that it is no longer possible to forcibly flush the cache, of for example S-box data, by churning through large dummy arrays. Some attacks require the cache to be initially empty with respect to such data. This flushing technique is denied them by a partitioned cache, although simply powering down the device is clearly still possible.

Remark 2. The authors of several cache based attacks have noted that with longer cache lines, attack is more difficult. This is intuitively easy to see: with longer lines more bits will be used to determine the sub-word and hence one can infer less information about addresses that provoke a given cache miss and resulting fetch operation.

The use of virtual line sizes within our design allows one to configure the fetch size, and hence in some sense the line size, on a per-partition basis. Hence, by allowing larger fetch sizes for partitions that store S-box data, we can make the attackers task much harder. Our design has the marked benefit that since each partition is independently configurable, one can select large fetch sizes where required but revert to normal sizes where not. Therefore, one need not pay any price by optimising for the average case: the partitions owned by each process can match the exact requirements.

Some unanswered issues arising from this fact are how long the cache lines must be before an attack is infeasible and how the length of the lines effects the hit-ratio for a partition containing S-box data.

Remark 3. Using a perturbation mask, our design essentially allows any address to map to any line and sub-word in the cache depending on the mask value. Selecting a random mask prior to execution hence introduces a level of non-determinism in the cache operation. Although the number of cache-hits and cache-misses is not necessarily altered, the examples in Section 2.2 show that the order or such features and the addresses that provoke them does change.

This non-determinism is probably not enough to prevent attacks which can filter out such noise statistically. However, if one is willing to pay the price of flushing and re-configuring a partition, it seems useful in decorrelating the results of one execution from another. As natural consequence, one would expect a trade-off between the increase in workload for the attacker and the algorithm performance. As above, a key unanswered issue is where this trade-off lies and whether it can be acceptable from both performance and security perspectives.

4 Conclusion

Instrumenting a new cache architecture, especially one which changes the way caches are viewed by the processor, is an architecturally disruptive process. However, altering standardised cryptographic primitives is also a disruptive and unattractive option. In order to provide sound defence against cache based attack methods at least one of these options seems a vital step. Such defence methods are ultimately going to be a trade-off between cost, in terms of either time or space, and security: one cannot hope to utilise conventionally designed caches, get conventional performance and still be secure. In high volume markets where bespoke processor designs are permitted, we posit that using a novel cache architecture produces a number of benefits.

Beyond the well known size, performance and power characteristics, we have investigated how a partitioned cache can assist in providing defences against side-channel attack methods. Use a partitioned cache architecture, and in doing so exposing the cache to the processor, offers a number of advantages in this respect. Optimising for the average case application will inherently produce problems; allowing application specific configuration offers a better degree of control over the cache behaviour. Likewise, treating the cache as a shared resource between potentially adversarial processes is flawed in a secure context; partitioning allows a flexible means of segregating the cache so this danger is removed. Although there is unlikely to be one single mechanism that offers defence against all cache based attacks, the proposed architecture seems to go some way toward helping and offers some attractive options for embedded processor designs.

There are many areas in which this work could be extended Firstly and most importantly, we need to experimentally investigate the observations from Section 3.3. This includes verification that our proposed architecture does not introduce new vulnerabilities not yet considered. For example by essentially making the cache behaviour more deterministic by decreasing interference, it is possible we have made attacks easier by simplifying many of the attackers assumptions. It also seems vital to investigate the physical implementation of such cache architecture: size and cost are clearly as important as security in terms of realistic deployment. It would be additionally interesting to see if such a device could be implemented using modern, side-channel resistant technologies such as dual-rail logic.

References

1. D. Agrawal, B. Archambeault, J.R. Rao and P. Rohatgi. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 29–45, 2002.
2. D. Agrawal, J.R. Rao and P. Rohatgi. Multi-channel Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2779, 2–16, 2003.
3. D.J. Bernstein. Cache-timing Attacks on AES. Available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
4. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *IEEE Conference on Information Technology: Coding and Computing (ITCC)*, 2005.
5. B. Calder, C. Krintz, S. John and T. Austin. Cache-Conscious Data Placement. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 139–149, 1998.
6. A. González, C. Aliagas and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *ACM International Conference on Supercomputing (ICS)*, 338–347, 1995.
7. A. González, M. Valero, N. Topham and J.M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-Based Placement Functions. In *ACM International Conference on Supercomputing (ICS)*, 76–83, 1997.
8. N. Gloy, T. Blockwell, M.D. Smith and B. Calder. Procedure Placement Using Temporal Ordering Information. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 303–313, 1997.
9. D.A. Patterson and J.L. Hennessy. *Computer Architecture: A Qualitative Approach*, Morgan Kaufmann, 1996.
10. J.P.J. Irwin. Systems With Predictable Caching. PhD Thesis, University of Bristol, 2002.
11. T. Juan and D. Royo and J.J. Navarro. Dynamic Cache Splitting. In *International Conference of the Chilean Computational Society*, 1995.
12. J. Kelsey and B. Schneier and D. Wagner and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In *Journal of Computer Security*, **8** (2-3), 141–158, 2000.
13. S. Kim, N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam, M.J. Irwin and E. Geethanjali. Power-aware Partitioned Cache Architectures. In *International Symposium on Low Power Electronics and Design (ISLEPD)*, 64–67, 2001.
14. A. Klimov and A. Shamir. A New Class of Invertible Mappings. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 471–484, 2002.
15. P.C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1109, 104–113, 1996.
16. P.C. Kocher, J. Jaffe and B. Jun. Differential Power Analysis. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1666, 388–397, 1999.
17. D. May, H.L. Muller and N.P. Smart. Non-deterministic Processors. In *Information Security and Privacy (ACISP)*, Springer-Verlag LNCS 2119, 115–129, 2001.
18. F. Mueller. Compiler Support for Software-Based Cache Partitioning. In *ACM Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 137–145, 1995.

19. D.A. Osvik, A. Shamir and E. Tromer. Cache attacks and Countermeasures: the Case of AES. In *Cryptology ePrint Archive*, Report 2005/271, 2005.
20. D. Page. Effective Use of Partitioned Cache Memories. PhD Thesis, University of Bristol, 2001.
21. D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. In *Cryptology ePrint Archive*, Report 2002/169, 2002.
22. C. Percival. Cache Missing For Fun And Profit. Available at: http://www.daemonology.net/papers/htt.pdf.
23. P. Petrov and A. Orailoglu. Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches. In *International Symposium on Hardware/Software Codesign*, 79–84, 2001.
24. P. Ranganathan, S.V. Adve and N.P. Jouppi. Reconfigurable Caches and their Application to Media Processing. In *International Symposium on Computer Architecture (ISCA)*, 214–224, 2000.
25. A. Sturges and D. May. A Cache System. ST Microelectronics, US Patent Number 6,871,266, 2005.
26. W. Tang, A. Veidenbaum and R. Gupta. Architectural Adaptation for Power and Performance. In *ACM International Conference on Supercomputing (ICS)*, 145–154, 1999.
27. Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2779, 62–76, 2003.
28. Y. Tsunoo and E. Tsujihara and K. Minematsu and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications (ISITA)*, 2002.
29. R.A. Wagner Compiler-Controlled Cache Mapping Rules Technical Report CS-1995-31, Duke University, 1995.
30. D.A.B. Weikle and S.A. McKee and W.A. Wulf. Caches As Filters: A New Approach to Cache Analysis. In *International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1998.
31. C. Zhang, F. Vahid and W. Najjar. A Highly Configurable Cache Architecture For Embedded Systems. In *International Symposium on Computer Architecture (ISCA)*, 136–146, 2003.