Cache-Timing Attacks and Shared Contexts*

Billy Bob Brumley and Nicola Tuveri

Aalto University School of Science and Technology, Finland
{bbrumley,ntuveri}@tcs.hut.fi

Abstract. Cache-timing attacks recover algorithm state by exploiting the fact that the latency of retrieving data from memory is essentially governed by the availability of said data in the processor's cache. Efficient and effective countermeasures to these attacks are needed. A shared memory context is a mechanism for reusing dynamically allocated memory. Focusing on public key cryptography within OpenSSL and its implementation of shared contexts, this paper examines the ability of a shared context to aid in mitigation of cache-timing attacks. The results are pessimistic towards this approach.

Keywords: cache-timing attacks, side-channel attacks, countermeasures, memory allocation

1 Introduction

Caches are used in modern computer architectures to improve memory hierarchy performances exploiting the principle of locality, both in the data and in the instruction flows. The registers used to store data onboard the CPU are very fast but limited in number and capacity, while on the other hand main memory has a huge capacity but is several orders of magnitude slower. One or more levels of cache are hence placed between the CPU and the main memory, reducing the load on the memory hierarchy and the average latency of memory references. We refer to [2, Ch. 5] for an extensive reference on cache architectures and related terminology.

Different virtual address spaces reside simultaneously inside the cache and, although protection of the cache contents among different processes is supported by the hardware logic and the operating system, it still remains a shared resource which can be used as a side-channel to leak information through timing of events. As foreboded by Kocher [5, Sect. 11] and Kelsey et al. [4, Sect. 5], cache-timing attacks against cryptosystem implementations are able to recover key material using this side-channel. We refer to [3, Ch. 18] for a good summary of work in this field. The scope of this paper includes trace-based attacks [7, Sect. 2] where the attacker is able to obtain a cache trace detailing hits and misses for memory accesses during execution.

For the purposes of this paper, two specific results from the literature are particularly relevant.

* Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

1. Percival describes an attack on the RSA implementation in OpenSSL 0.9.7c [8]. The cache trace is obtained through a "spy process" which simply loads its own contents from memory, filling the data cache, and then reads them back, measuring, for each set, the time required to read all its lines. The thus obtained trace is then analyzed to identify patterns caused by data dependent memory accesses: i.e. in the case of RSA the key dependent lookups into the precomputation table used by the sliding window exponentiation algorithm.
2. Brumley and Hakala present a framework for processing cache-timing data and use it to attack the ECDSA implementation in OpenSSL 0.9.8k [1]. The attack results suggest that the most significant time variances present in the trace are not due to data dependent memory accesses, but to differences in memory-access footprints among different operations: i.e. in the case of ECDSA the difference in usage of temporary variables between a point doubling and point addition step during a scalar multiplication algorithm.

Dynamic memory allocation is a costly operation for software. To reduce this cost, many software libraries (including [9], [10], [11]) implement mechanisms that allow reusing of dynamically allocated memory across different function calls: this improves performance. We focus exclusively on shared contexts, the solution adopted by OpenSSL [12]. In light of cache-timing attacks, one countermeasure in [1] proposes (but does not implement or evaluate) that the context randomize allocation of its resources.

To this end, this paper examines the ability of shared contexts to act as a countermeasure against cache-timing attacks. This involves the implementation and evaluation of a simple data alignment countermeasure. The results suggest that, in our experiment environment, the allocation policy enforced by the shared context cannot in isolation render the side-channel useless. That is, surprisingly in this case the shared context can do little to mitigate the known attacks.

We structure this paper as follows. Sect. 2 contains an outline of dynamic memory allocation in OpenSSL and relevant data structures. Sect. 3 describes our implementation of a cache-timing attack countermeasure enforced by the shared context. We discuss the evaluation of this countermeasure in Sect. 4 as well as provide some sample side-channel data. We close in Sect. 4.

2 Dynamic memory in OpenSSL BigNum

The core of many cryptosystems implemented by OpenSSL is the BigNum module, which provides arbitrary precision arithmetic. Dynamic memory is handled through `OPENSSL_malloc`, `OPENSSL_realloc` and `OPENSSL_free` which, by default, are mapped to the `malloc`, `realloc` and `free` functions of the standard library. In the following paragraphs we present the basic data unit of the BigNum module and the mechanisms used for allocation of temporary variables.

2.1 BigNum variables

The basic operand type inside the BigNum module is an abstract data type called `BIGNUM` which represents an arbitrary precision integer and whose internal structure is summarized in Fig. 1.

All the provided arithmetic operations automatically handle resizing of the `d` array to prevent overflows. To improve performance and avoid useless sequences of shrink and extend operations, `BIGNUM` variables are not downsized automatically, and an internal value is used to track how many words are actually used among the allocated words for the `d` array.

It is important to note that, while the `BIGNUM` structure holds the information needed for control flow and resizing, the actual binary representation of the number is not contained within the `BIGNUM` structure but resides in separate memory blocks, referenced by the `d` pointer.

2.2 The `BN_CTX` structure

Creation and destruction of `BIGNUM` variables and reallocations of the referenced bit arrays are, in general, quite time-consuming operations therefore the performances of functions using temporary variables and of the whole library may be improved using some sort of caching functionality to minimize the number of these operations. The `BN_CTX` structure is designed to accomplish this goal and simulates a function stack, with the difference that starting and ending of the frames and allocation of the variables are made explicit.

Internally the `BN_CTX` type is defined as the structure depicted in Fig. 2, containing the count of currently assigned `BIGNUM` temporary variables, two variables for internal error handling, and two internal auxiliary structures:

- The `BN_POOL` structure (depicted in Fig. 3), which provides the caching functionality, preallocating clusters of `BIGNUM` variables and keeping track of unused variables to be reassigned.
- The `BN_STACK` structure which provides the per-function stack frame abstraction.

The `BN_CTX` object is meant to be created once with `BN_CTX_new()` and then passed to every function that may need to allocate temporary `BIGNUM` variables, which in turn will:

1. start a new frame inside the `BN_CTX` through `BN_CTX_start()`, which pushes the current value of the `used` variable into the stack through `BN_STACK_push()`.
2. request all the needed temporary `BIGNUM` variables using `BN_CTX_get()` which increases the `used` counter and returns a new temporary `BIGNUM` variable obtained through `BN_POOL_get()` after setting it to zero.
   `BN_POOL_get()`, in turn, returns the next unused `BIGNUM` variable from the pool: the `used` variable and the `current` pointer are used to track the first unused `BIGNUM` variable in the pool, and when no unused variables are available a new `BN_POOL_ITEM` is automatically created and added to the tail of the internal list.

3. operate on the temporary variables, potentially calling other functions which may require the `BN_CTX` for their own temporary variables.
4. end the frame started at the first step, using `BN_CTX_end()`, before returning to the calling function.
   `BN_CTX_end()` retrieves the previous "frame pointer" from the stack through `BN_STACK_pop()` and, if its value is different from the current `used` counter, all the temporary variables acquired from the pool through `BN_POOL_get()` in the scope of the last frame are released calling `BN_POOL_release(used - fp)`; `used` is then set to `fp`.
   `BN_POOL_release(num)`, in turn, releases the last `num` used `BIGNUM` variables in the pool, decreasing the `used` counter and accordingly updating the `current` pointer, without removing any `BN_POOL_ITEM` from the list.

The `BN_CTX` object is then destroyed, releasing all the allocated memory, with `BN_CTX_free()`, before the process is terminated.

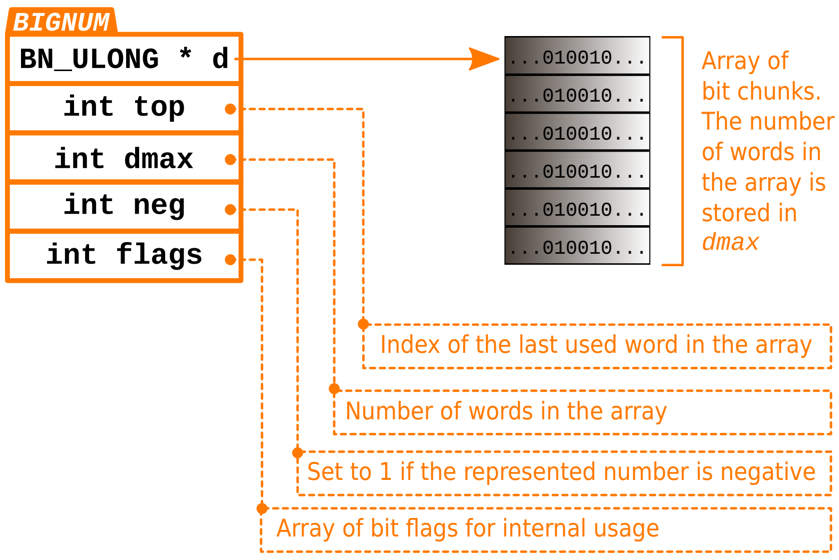
BIGNUM
BN_ULONG * d
int top
int dmax
int neg
int flags
...010010...
...010010...
...010010...
...010010...
...010010...
...010010...
Array of bit chunks. The number of words in the array is stored in dmax
Index of the last used word in the array
Number of words in the array
Set to 1 if the represented number is negative
Array of bit flags for internal usage

Fig. 1. The `BIGNUM` structure

3 Shared contexts and cache attack countermeasures

An attack on OpenSSL's implementation of ECC appears in [1]. The scalar multiplication algorithm uses a modified windowed NAF representation with a precomputation phase that potentially exposes key material in each iteration due to table lookups of precomputed points. However, the authors conjecture that the most visible pattern in the collected traces is due instead to dynamic

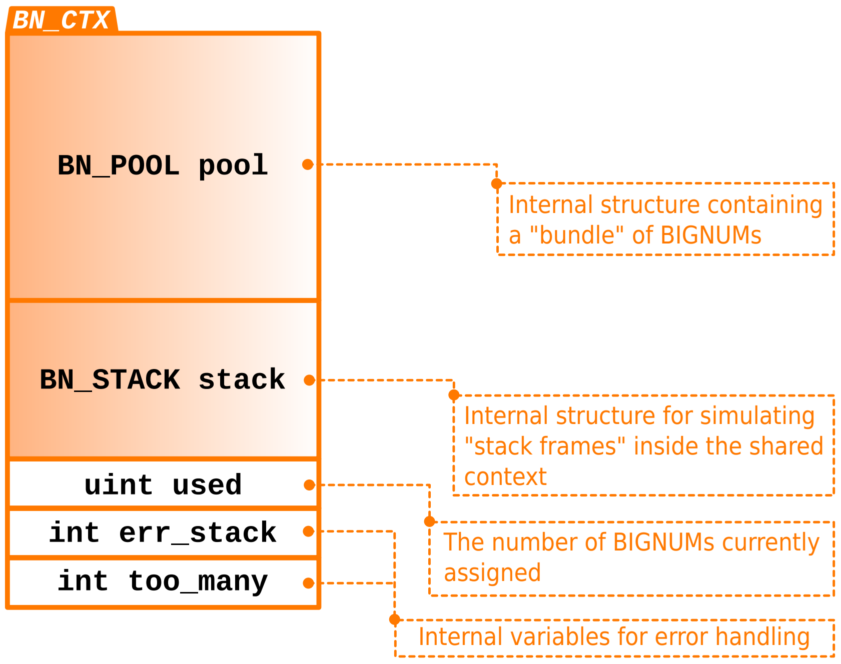
BN_CTX
BN_POOL pool
BN_STACK stack
uint used
int err_stack
int too_many
Internal structure containing a "bundle" of BIGNUMs
Internal structure for simulating "stack frames" inside the shared context
The number of BIGNUMs currently assigned
Internal variables for error handling

Fig. 2. The BN_CTX structure

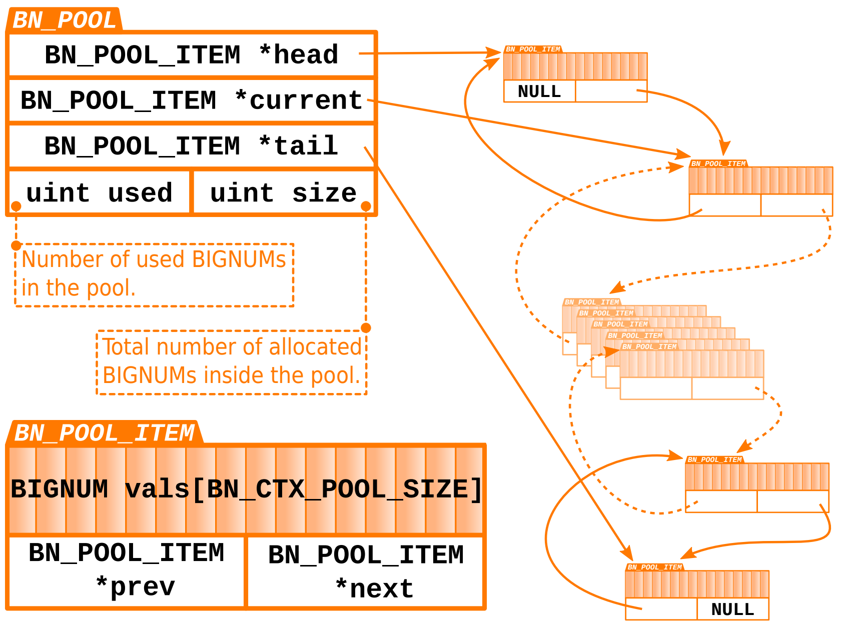

Fig. 3. The BN_POOL structure and the related double-linked list of BN_POOL_ITEMs

memory for variables in the curve arithmetic functions. For example, the unbalanced memory footprint of the point doubling and point addition formulas. This can potentially permeate all the way down to the field arithmetic level, e.g. temporary variables used in field multiplication and field squaring. It is possible that the number and order of these operations can lead to a noticeable pattern in the trace, which can allow an attacker to infer a series of higher level elliptic curve group operations. As such, they propose that the OpenSSL shared context should randomize the allocation of its `BIGNUM` variables to deter cache-timing attacks. While the approach sounds reasonable, they did not implement the proposed countermeasure to evaluate its effectiveness.

The analysis of the shared context summarized in Sect. 2 reveals that it is not possible to implement this countermeasure while limiting changes only to the `BN_CTX` internals: the only relevant addresses in the scope of the `BN_CTX` structure are those of the returned `BIGNUM` objects and, while it might be possible to arrange mechanisms and structures to randomly serve `BIGNUM` addresses associated with different cache sets, the addresses of the memory blocks storing the actual binary representation referenced within each `BIGNUM` cannot be forced by the `BN_CTX` facility as they are handled by the internal `BIGNUM` functions that deals with creation, deletion and resizing of the `BIGNUM` internal binary representation.

We then worked on the hypothesis that aligning all the dynamically allocated memory to the same cache set would be an effective, straightforward countermeasure in this case: we redefined the default `OPENSSL_malloc()` (alongside the associated `free` and `realloc` functions) to use a custom implementation returning addresses aligned to the same cache set and analyzed the resulting cache traces.

3.1 Targeted system

For this paper we focus on a system based on a Intel Pentium 4 processor so that results may be compared with those available in the literature. This processor implements Intel's Hyper-Threading Technology (HTT), a form of Simultaneous Multithreading (SMT) where the single CPU expose two logical processors, sharing between them the physical execution resources while duplicating the architecture state, thus allowing execution of multiple threads concurrently [6]. HTT is not exclusive to the Intel Pentium 4 family and is featured by most recent Intel processor families for mobile, desktop and server systems (i.e. Atom, Core i3, Core i5, Core i7, Itanium and Xeon). Although HTT is not a requirement for trace-based cache attacks, it relaxes the need to force context switches among the spy process and the victim process since, during execution, the two threads naturally compete for the shared resources, including the data cache.

The L1 data cache geometry of the Intel Pentium 4 processor is as follows: 64 B-long lines, 8 KiB total size, 4-way set associative, and 128 lines divided into 32 associative sets. Therefore in the targeted system each virtual address is split in three sections as depicted in Fig. 4: the offset contains the 6 least significant bits used to address one of the 64 bytes in a cache line, the set index contains the adjacent 5 bits used to select one of the 32 associative sets of the cache, and the tag contains the remaining bits used to identify the requested address by distinguishing among different addresses that may be associated with the same set.

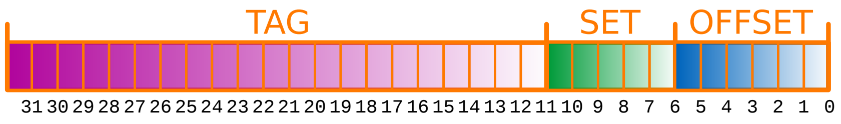

Fig. 4. Logical structure of a virtual address for the targeted system

3.2 Single-set aligned addresses wrapper

For the `malloc()` wrapper we used the function `posix_memalign()` (defined in the POSIX.1d standard) to allocate a memory block aligned to a multiple of `0x00000800` so that the set index portion of the returned address is always set to zero. The first $sizeof(size\_t)\, bytes$ of the address returned by `posix_memalign()` are used to hold the $request_{size}$ value, which is required for the `realloc()` wrapper, hence the length of the actually allocated block is $request_{size} + sizeof(size\_t)$ and the offset portion of the addresses returned by `CRYPTO_amono_malloc()` is set to $sizeof(size\_t)$.

The `free()` wrapper simply calculates the address originally returned by `posix_memalign()` and calls the standard C `free()` function on it (the POSIX.1d standard mandates that addresses returned by `posix_memalign()` can be freed by `free()`). The `realloc()` wrapper simply allocates a new memory block of the desired size through the `malloc()` wrapper, copies the contents of the old memory block into the new one and then frees the old block through the `free()` wrapper.

4 Results

To analyze the effects of the changes described in Sect. 3 we need a spy process to collect the cache-timing traces. We outline our spy process in Fig. 5; it is adapted from [8] and we refer the reader accordingly for a discussion of its origin and operation. The `movnti` instruction is a move with non-temporal hint; it seeks to avoid cache interference by bypassing the cache when writing the obtained timings to memory. The set-0 aligned input buffer at `ecx` contains 8192/4 copies of `0x00000001` which causes all `imul` instructions to carry out nothing more than a high latency NOP.

The illustrations in Fig. 6 depict the typical data cache traces obtained by the described spy process running concurrently with OpenSSL performing an ECDSA signature operation: the gradient of each cell measures the cache set access time, hence time moves within each cell, then from bottom to top through

```
mov $8192,%edi
LOOPA:
sub $4,%edi
mov $1,(%ecx,%edi)
jnz LOOPA
xor %edi,%edi
rdtsc
mov %eax,%esi
LOOPB:
; cache set 00
imul 0x0000(%ecx),%ecx
imul 0x0800(%ecx),%ecx
imul 0x1000(%ecx),%ecx
imul 0x1800(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x00(%ebx,%edi)
add %eax,%esi
; cache set 01
imul 0x0040(%ecx),%ecx
imul 0x0840(%ecx),%ecx
imul 0x1040(%ecx),%ecx
imul 0x1840(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x01(%ebx,%edi)
add %eax,%esi
; cache set 02
imul 0x0080(%ecx),%ecx
imul 0x0880(%ecx),%ecx
imul 0x1080(%ecx),%ecx
imul 0x1880(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x02(%ebx,%edi)
add %eax,%esi
...
; cache set 30
imul 0x0780(%ecx),%ecx
imul 0x0f80(%ecx),%ecx
imul 0x1780(%ecx),%ecx
imul 0x1f80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1e(%ebx,%edi)
add %eax,%esi
; cache set 31
imul 0x07c0(%ecx),%ecx
imul 0x0fc0(%ecx),%ecx
imul 0x17c0(%ecx),%ecx
imul 0x1fc0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1f(%ebx,%edi)
add %eax,%esi
add $32,%edi
cmp <buffer len>,%edi
jge END
jmp LOOPB
END:
```

Fig. 5. Pentium 4 data cache spy process. `ecx` holds the input buffer address and `ebx` the output buffer address.

consecutive cache sets and finally from left to right when the measurements are repeated. Although the time flow is thus distributed, it may be easier to consider the data as vectors which length is equal to the number of cache sets and with time simply moving from left to right. That is, one column vector is the output of one iteration of `LOOPB` in the spy process. High (low) latencies suggest cache misses (hits).

In Fig. 6 we compare a typical trace obtained through the standard version of OpenSSL 0.9.8o with a typical trace obtained through the same code patched to use the described memory wrappers.

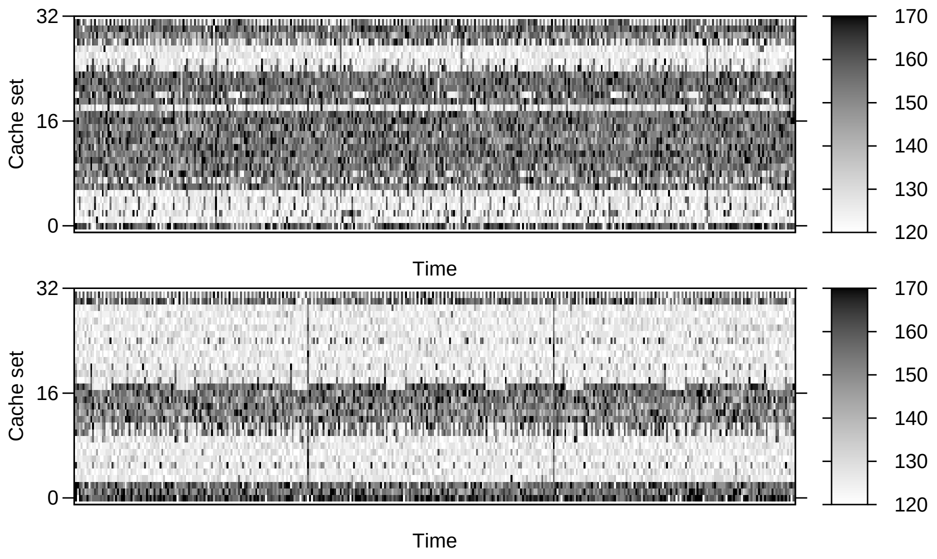

Fig. 6. Cache-timing data from a spy process running concurrently with an OpenSSL ECDSA signature operation, using the unmodified OpenSSL 0.9.8o code (Top) and the patched version implementing the memory wrappers described in Sect. 3.2 (Bottom).

In the unmodified OpenSSL case (top), there is clear correlation between the cache-timing trace and the state of the scalar multiplication algorithm, i.e. whether it is performing a point doubling or point addition step. Identified by manual inspection using known inputs, sets 6, 7, 20, and 31 are good indicators. There are eight point additions separated by a number of point doublings. This is to be expected as no countermeasures have since been implemented in any OpenSSL version to prevent the attacks in [1].

Deploying the countermeasure described in Sect. 3.2, we expected the resulting traces to contain mostly cache misses in set 0 and mostly cache hits in all other sets. The intuition is that, if all dynamic memory allocated within OpenSSL is aligned at different addresses mapping to cache set 0, then the remainder of the cache should not show any interesting activity. Essentially the countermeasure seeks to disable all but a single cache set.

Surprisingly, the results disagree with this intuition. That is, in the patched OpenSSL case (bottom), there is still clear correlation between the cache-timing trace and the algorithm state. Set 17 is a good indicator. There are eight point additions separated by a number of point doublings.

5 Conclusion

Cache-timing attacks represent a serious threat to security-critical software. This is evidenced by bustling activity in academic research in the area over the past decade, as well as the response in the software development community. For example, a number of patches to the OpenSSL library are available attempting to mitigate cache-timing attacks and, more generally, microarchitecture attacks. Proposing and evaluating countermeasures to these attacks is an important topic that can yield insight into improving the security of cryptosystem software implementations, or more generally any software with state that should remain secret.

In this paper, we focused on one such proposed countermeasure related to OpenSSL's handling of dynamically allocated memory using shared contexts. After reviewing OpenSSL's implementation of shared contexts, we implemented a countermeasure that aligns all dynamically allocated memory within OpenSSL at a single cache set. In contrast to the randomly aligned countermeasure proposed, but not implemented, in [1], our intention was to start with this basic countermeasure and, based on the implementation results, build up to a more secure, efficient, and robust one.

After patching the OpenSSL source code with our implementation and evaluating the countermeasure with respect to OpenSSL's implementation of the ECDSA, the resulting cache-timing traces reveal that, contrary to our initial assumption, even when aligning all dynamically allocated memory to the same boundary, there is still a significant amount of correlation between the cache-timing data and the algorithm state, particularly outside of said boundaries. These results suggest that the viability of shared contexts in cache-timing attack countermeasures is, at best, limited.

Excluding the shared context, it remains to identify what mechanism is ultimately responsible for said behavior. This could be a software or microarchitecture mechanism, or even a combination of multiple mechanisms. Suspects worth investigating are the stack, the trace cache, and higher level data caches. We defer this task to future work.

References

1. Brumley, B.B., Hakala, R.M.: Cache-Timing Template Attacks. In: Matsui, M. (ed.) Advances in Cryptology - ASIACRYPT 2009. Lecture Notes in Computer Science, vol. 5912, pp. 667–684. Springer-Verlag, Berlin Germany (2009)
2. Hennessy, J.L., Patterson, D.A., et al.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 4th edn. (2006)
3. Çetin Kaya Koç (ed.): Cryptographic Engineering. Springer (2009)
4. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side channel cryptanalysis of product ciphers. In: Quisquater, J.J., Deswarte, Y., Meadows, C., Gollmann, D. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 1485, pp. 97–110. Springer (1998)
5. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)
6. Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technology Journal 6(1), 4–15 (2002)
7. Page, D.: Defending against cache-based side-channel attacks. Information Security Technical Report 8(1), 30–44 (2003)
8. Percival, C.: Cache missing for fun and profit (2005), `http://www.daemonology.net/papers/cachemissing.pdf`
9. The GNU Project: GMP: The GNU Multiple Precision Arithmetic Library, `http://www.gmplib.org`
10. The GNU Project: GnuPG: libgcrypt, `http://www.gnupg.org`
11. The Mozilla Foundation: Network Security Services (NSS), `http://www.mozilla.org/projects/security/pki/nss/`
12. The OpenSSL Project: OpenSSL: The open source toolkit for SSL/TLS, `http://www.openssl.org`