Cache-Timing Attacks and Shared Contexts

Billy Brumley Nicola Tuveri

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

February 25, 2011

Outline

Cache-timing attacks

Shared contexts

Implementation of a simple countermeasure

Conclusion

Cache-timing attacks

- Cache-timing attacks are software side-channel attacks that recover algorithm state by observing timing differences introduced by the cache, i.e. cache hits and cache misses.
- This work considers trace-driven (access) attacks where an attacker can execute a spy process locally on the same machine (not same space) as the crypto software.

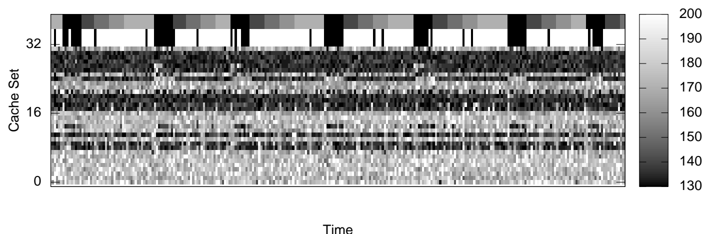
Cache missing for fun and profit

- C. Percival describes how to exploit Intel's (then) recent HyperThreading technology to "spy" on other processes through the data cache. (BSDCan 2005)
- He uses this to attack OpenSSL's (0.9.7c) implementation of RSA and recover keys.
- Published the spy process code.
- One of the most influential works on cache-timing attacks.

Cache-timing template attacks

- Brumley & Hakala develop a framework to analyze cache-timing data. (ASIACRYPT 2009)
- Works by creating templates that represent the algorithms memory access behavior in different states.
- Uses Vector Quantization (VQ) and Hidden Markov Models (HMM) in analyzing the data.
- Use the framework to attack OpenSSL's (0.9.8k) implementation of ECDSA and recover keys.

Sample trace with framework metadata: ECDSA



Shared contexts

- Dynamic allocation of memory is costly (e.g. malloc).
- Sometimes it makes sense to save the allocated memory and reuse it; allocate it in a parent function and pass it to children.
- Lots of arithmetic libraries have some mechanism to do this (OpenSSL, GMP, GnuPG, NSS).
- OpenSSL uses a mechanism called a **shared context**.

This work explores:

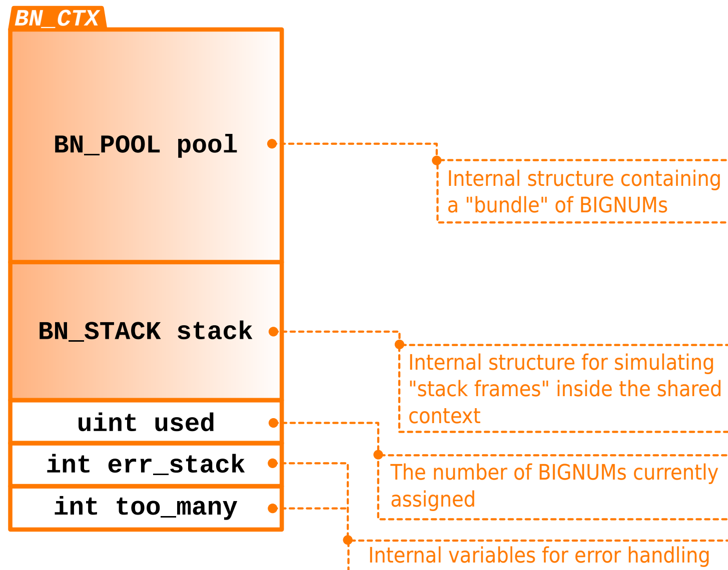
1. What role does a shared context play in cache-timing attacks?
2. Can it be used as a countermeasure?

Example: using the context

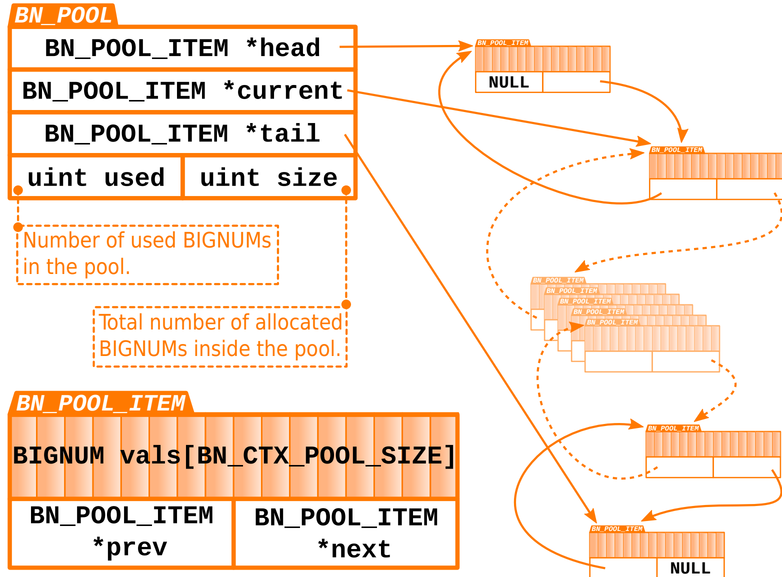
```
void fxn_A(BIGNUM *r, const BIGNUM *a, const BIGNUM *b, const BIGNUM *c,
                                       const BIGNUM *d, BN_CTX *ctx) {
  BIGNUM *x, *y;
  BN_CTX_start(ctx);
  x = BN_CTX_get(ctx);
  y = BN_CTX_get(ctx);
  BN_add(x, a, b);
  BN_add(y, c, d);
  BN_add(r, x, y);
  BN_CTX_end(ctx);
}

void fxn_parent(BIGNUM *r) {
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *a, *b, *c, *d;
  BN_CTX_start(ctx);
  a = BN_CTX_get(ctx);
  b = BN_CTX_get(ctx);
  c = BN_CTX_get(ctx);
  d = BN_CTX_get(ctx);
  fxn_A(r,a,b,c,d,ctx);
  fxn_A(r,a,b,c,d,ctx);
  BN_CTX_end(ctx);
  BN_CTX_free(ctx);
}
```

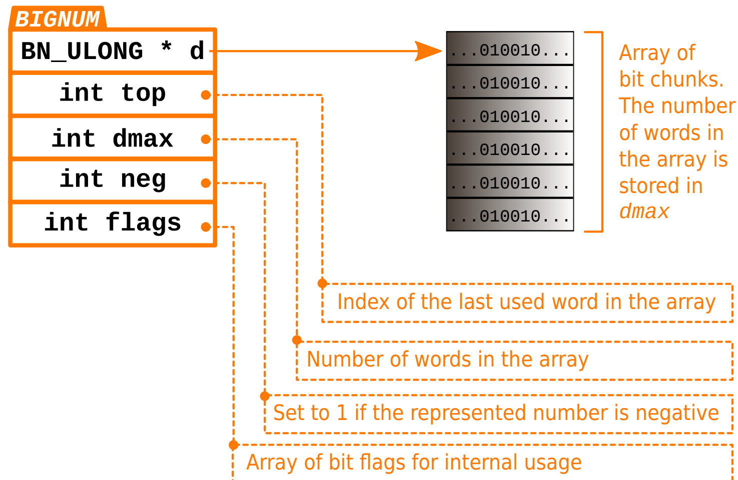
The BN_CTX structure



The BN_POOL structure



The BIGNUM structure



What can possibly go wrong?

- The way a single function accesses temporary variables from the context might depend on how the parameters are passed.
- Different functions that use the same context might use a different number of temporary variables.
- Even with the same number of variables, they might access them differently.

```
void fxn_crypto(BN_CTX *r, unsigned char *k, int len) {
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *a, *b, *c, *d;
  BN_CTX_start(ctx);
  a = BN_CTX_get(ctx);
  b = BN_CTX_get(ctx);
  c = BN_CTX_get(ctx);
  d = BN_CTX_get(ctx);
  for(int i=len-1; i>=0; i--) {
    for(int j=7; j>=0; j--) {
      if(k[i] >> j & 1) fxn_A(r,a,b,c,d,ctx);
      else fxn_A(r,c,d,a,b,ctx);
    }
  }
  BN_CTX_end(ctx);
  BN_CTX_free(ctx);
```

Example: different number of temporary variables

```
void fxn_A(BIGNUM *r, const BIGNUM *a, const BIGNUM *b, const BIGNUM *c,
                                       const BIGNUM *d, BN_CTX *ctx) {
  BIGNUM *x, *y;
  BN_CTX_start(ctx);
  x = BN_CTX_get(ctx);
  y = BN_CTX_get(ctx);
  BN_add(x, a, b);
  BN_add(y, c, d);
  BN_add(r, x, y);
  BN_CTX_end(ctx);
}

void fxn_B(BIGNUM *r, const BIGNUM *a, const BIGNUM *b, const BIGNUM *c,
                                       const BIGNUM *d, BN_CTX *ctx) {
  BIGNUM *x, *y;
  BN_CTX_start(ctx);
  x = BN_CTX_get(ctx);
  BN_add(x, a, b);
  BN_add(x, x, c);
  BN_add(r, x, d);
  BN_CTX_end(ctx);
}
```

Example: different access order

```
void fxn_A(BIGNUM *r, const BIGNUM *a, const BIGNUM *b, const BIGNUM *c,
                                       const BIGNUM *d, BN_CTX *ctx) {
  BIGNUM *x, *y;
  BN_CTX_start(ctx);
  x = BN_CTX_get(ctx);
  y = BN_CTX_get(ctx);
  BN_add(x, a, b);
  BN_add(y, c, d);
  BN_add(r, x, y);
  BN_CTX_end(ctx);
}

void fxn_B(BIGNUM *r, const BIGNUM *a, const BIGNUM *b, const BIGNUM *c,
                                       const BIGNUM *d, BN_CTX *ctx) {
  BIGNUM *x, *y;
  BN_CTX_start(ctx);
  x = BN_CTX_get(ctx);
  y = BN_CTX_get(ctx);
  BN_add(x, a, c);
  BN_add(y, b, d);
  BN_add(r, x, y);
  BN_CTX_end(ctx);
}
```

Using the context as a countermeasure

- Simple solution: context should randomize allocation of its resources. Idea from ASIACRYPT 2009.
- Our angle: start by aligning all the dynamically allocated memory, see how that looks and performs, go from there.

Aligning dynamically allocated memory

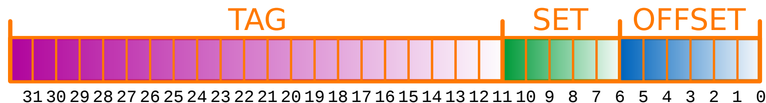
- OpenSSL wraps malloc. You can over-allocate and slide the pointer to align data.
- We used a more standard solution instead:

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);

The function posix_memalign() allocates size bytes and places the
address of the allocated memory in *memptr.  The address of the
allocated memory will be a multiple of alignment, which must be a
power of two and a multiple of sizeof(void *).
```

The spy process

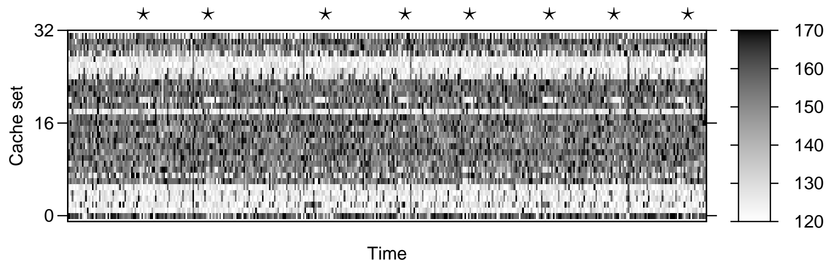


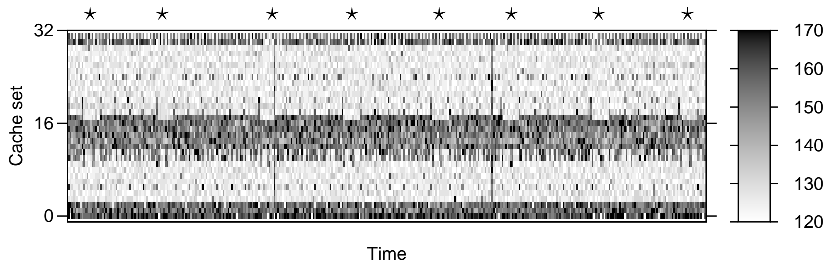
- Spy process runs in parallel with the crypto process, collecting cache-timing data.
- Target architecture is a P4 with HyperThreading: runs two processes in pseudo-parallel.
- 8KiB 4-way set-associative data cache, 64B lines; 32 sets.
- Started with a published spy process (C. Percival), but the signal was too noisy.
- Modified it by:
  1. unrolling all control flow;
  2. instead of one pointer, using two: one each for reading and writing;
  3. serializing things as much as possible.

Example: spy snippet

```
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:
```

Sample traces with and without alignment: ECDSA





Conclusion

- Results suggest that, contrary to the hypothesis, dynamically allocated memory is not responsible for (the bulk of) the signal.
- i.e., context-based countermeasures not immediately worth effort.
- What is responsible? Some suspects:
  1. The stack?
  2. Higher level caching (L2 etc.)?
  3. The trace cache?
  4. (Undocumented) microarchitecture feature x?
- Thanks :) Questions?