Are AES x86 Cache Timing Attacks Still Feasible?

Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham[*]

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

ABSTRACT

We argue that five recent software and hardware developments — the AES-NI instructions, multicore processors with per-core caches, complex modern software, sophisticated prefetchers, and physically tagged caches — combine to make it substantially more difficult to mount data-cache side-channel attacks on AES than previously realized. We propose ways in which some of the challenges posed by these developments might be overcome. We also consider scenarios where side-channel attacks are attractive, and whether our proposed workarounds might be applicable to these scenarios.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection—*Cryptographic Controls*; E.3 [**Data**]: Data Encryption

Keywords

AES, x86, Cache Timing, Side Channels, AES-NI

1. INTRODUCTION

Side-channel attacks are a classic topic in computer security. But are they still feasible on modern x86 machines?

In a side-channel attack, an attacker recovers secret information from his victim by observing or manipulating a shared resource. The most attractive channels for such attacks are shared hardware resources such as the data cache, and the most devastating attacks recover cryptographic keys. Even as the traditional scenario for side channel attacks — multiuser timesharing on workstation systems — has fallen into decline, other attractive attack scenarios have arisen.

This paper arises from our unsuccessful attempt at exploiting one such scenario: compromising Chrome browser SSL keys from a Native Client control, using AES cache timing as a channel. In our attempt, we ran up against several recent changes to the x86. Some of these changes have already been mentioned in work on side-channel attacks; others are well known by architects but less so in the security community. Taken together, these changes make it much more difficult to mount side-channel attacks on AES.

Our contribution in this paper is to describe the new challenges to AES cache attacks and to propose ways in which they might be overcome. We also consider scenarios where side-channel attacks are attractive, and whether our proposed workarounds might be applicable to these scenarios.

The new challenges to AES side-channel attacks are:

- The AES-NI instruction set, which moves AES data structures out of the cache;
- multicore processors with per-core L1 and L2 caches;
- the complexity of modern software and the pressure that it places on caches;
- the increasingly sophisticated and poorly documented prefetcher units on modern processors; and
- the switch from virtually tagged to physically tagged caches.

The first two of these can make AES cache attacks impossible; the last three increase the difficulty of AES cache attacks and make them inapplicable in some settings.

Architectural side-channel attacks are enabled when shared hardware between two mutually distrusting principals is incompletely virtualized by the supervisor. Traditionally, the principals were users on a timesharing system and the supervisor was the OS kernel. We believe that today there are (at least) three scenarios where architectural side-channel attacks are a threat: (1) infrastructure-as-a-service cloud computing [22], with virtual machines as principals; (2) client-side in the Web browser and its plugins, with Web origins as principals; and (3) smartphones and tablets, with apps as principals. (Mobile devices mostly use ARM chips, not x86.)

Cache side-channel attacks on AES were first demonstrated by Bernstein [6], Tromer, Osvik, and Shamir [23], and Bonneau and Mironov [8]. These attacks were compared and analyzed by Canteaut [11], and were recently improved by Gullasch, Bangerter, and Krenn [15]. Proposed mitigations include cache-oblivious AES algorithms [9], the AES-NI instruction set [16], deterministic computing environments [4], and fuzzy timing channels [24].

In this paper, we argue that mounting AES (data) cache attacks on modern systems is more difficult than previously realized. This does not, of course, mean that they are impossible. In addition, our results do not rule out attacks on other cryptographic primitives, such as RSA [21], nor at-

[*]Supported by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

tacks that rely on other architectural channels, such as the instruction cache [1, 2] or the branch prediction unit [3], nor timing attacks that do not depend on an architectural channel [10, 7]. In addition, while we have considered the x86, other architectures may still be vulnerable. Viewed another way, our results suggest that other cryptosystems than AES, other microarchitectural features than the data cache, and other architectures than the x86 should be considered in future side-channel research. The recent paper of Zhang et al. [26] partly corroborates this positive view of our claims. Zhang et al. mount a cross-VM side-channel attack on cryptographic keys; but they target public-key crypto rather than AES, and use the instruction cache rather than the data cache.

2. COMPLETE MITIGATION

First, we identify two trends in processor development which have the potential to completely prevent AES cache timing attacks: AES-NI and multicore processors. These recent phenomenon represent material change in the hardware capabilities underlying our computer systems, and will almost certainly grow for the forseeable future.

2.1 AES-NI

Cache timing side channel attacks depend solely on measuring the processor's use of memory during encryption. Without these cache-changing accesses, the entire class of attacks is mitigated.

Intel processors that support AES-NI [14] provide hardware implementations of key generation, encryption rounds, and decryption rounds. The cryptographic operations are moved out of RAM and into custom hardware, improving performance and eliminating cache side channels.

2.1.1 Hardware Prevalence

Intel shipped the first AES-NI–supporting processor in Q1 2010 [17]. All Sandy Bridge i7 and all but two Sandy Bridge i5 support AES-NI, while all announced Ivy Bridge i5 and i7 processors support it. Though Intel is still producing a few processors without AES-NI, presumably it will shortly be supported throughout the entire lineup (akin to the MMX or SSE extensions). AMD is introducing AES-NI support as well, with Bulldozer, the first supported microarchitecture, released in October 2011 [13].

Therefore, many consumer-facing systems built in the past two years are inoculated against AES cache timing attacks, and we posit that this number will grow with time.

Even on processors that do not implement AES-NI, alternative AES implementations exist that keep AES data outside of RAM; see, e.g., [19].

2.1.2 Software Support

Of course, all the AES-NI hardware in the world can't protect against cache timing attacks if applications persist in using vulnerable software AES implementations. Fortunately, SSL and crypto libraries are actively providing support for AES-NI: OpenSSL 1.0.1 [20], Microsoft Cryptography API: Next Generation in Windows 7 [16], and Network Security Services 3.12.2 [18] all utilize AES-NI when the processor provides suppport. Therefore, up-to-date versions of Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox are all completely immune to AES cache timing attacks on modern hardware.

2.2 Multicore Processors

In the pursuit of performance, processor designers are increasingly adding multiple physical cores to each die. For example, currently Intel is almost exclusively shipping multicore chips, with a few single core holdouts in the Celeron and Atom lines. The Atom N270 on which we conducted our experiments contains only a single core, but newer processors in the Intel Atom line (designed to be minimal and power–efficient) now contain multiple cores. Even mobile devices are touting dual (or quad) cores as a major selling point. Multicore is here to stay, and, in order to remain a threat, cache timing attacks must be proven to work under the new hardware regime.

The mere inclusion of multiple cores complicates cache attacks immensely. First, the attack must be aware of processor-specific multicore cache behavior. For example, Intel Sandy Bridge processors have a per-core L1 and L2 cache, but all cores share L3. The attacker must also understand the eviction policy: can data remain in a L2 cache if it is evicted from L3 by another core? Further complicating matters, an attacking thread might be rescheduled onto another physical core at any time, physically separating it from its carefully manicured test data. Therefore, the best approach might take cues from Gullash's attack: create many attacker threads, and trust that attacker threads will probabilistically control every core most of the time. Under this scheme, cross-thread communication is vital: the attacker, when communicating with herself, must take care to avoid the cache lines she's trying to measure. While a solution to each of these problems is imaginable, none of the existing cache timing attacks are directly applicable to multicore processors, and further work would definitely be needed to indicate that such attacks remain viable in the multicore paradigm.

Recent work by Xu et al. on L2 cache timing side channels in virtualized environments show that the upper bound on side channel bandwidth in EC2 is just over 10 bits per second [25]. A covert channel consists of communication between two cooperating principals; it is not clear that Xu et al.'s analysis can be applied to adversarial side channels, let alone to the very specific side channels required for attacking AES using the data cache.

Furthermore, core "pinning" is becoming an extremely popular technique to manage load and scheduling behavior. Essentially, the operating system assigns a thread to one or more physical cores, and guarantees that it will never execute anywhere else. With this in mind, we observe that a pinned attacker thread (even if pinned to $n-1$ cores) physically cannot perform cache timing attacks against an encryption thread on the forbidden core, especially when that core's L2 cache vastly exceeds 4 KiB. The AES lookup tables will fit entirely in the encrypting core's L2, which the attacker is unable to manipulate or examine, rendering the technique powerless.

3. ATTACK OUTLINE

The first element needed in a successful cache timing attack is a high resolution timer, able to differentiate the few hundred cycles between a cache hit or miss. For this purpose, our predecessors in cache timing use the x86 instruction `rdtsc`, which provides cycle-accurate count information. To also use `rdtsc`, we need to be able to deliver and execute semi-arbitrary x86 instructions on the target machine. Google Chrome provides such an ability in Native Client.

Native Client (NaCl) allows web developers to run native code alongside web applications, sidestepping the performance penalty of interpreters or JITs. It uses sophisticated software-fault isolation techniques to prevent misbehavior. For example, Native Client executables are restricted from making arbitrary system calls, and developer-provided code is restricted from making syscalls at all (special NaCl code, placed in a certain location in the NaCl application's address space, handles outside communication). Furthermore, writes to memory are region-checked, so applications are prevented from simply writing a stream of instructions and executing them. Control flow is also restricted: jump destinations must always be on an aligned address. Lastly, NaCl disallows the use of certain x86 instructions, such as `ret`.

NaCl is permissive enough that NaCl code can mount cache timing attacks. NaCl allows use of both the x86 instructions `rdtsc` and `cpuid`, a serializing instruction that returns detailed information about processor features. NaCl code can issue an arbitrary sequence of memory reads.

Next, a cache timing attack must cause encryptions to occur, ideally with valuable keys. Assuming the network attacker model, the most attractive use of AES in Google Chrome is to protect TLS traffic. As a network attacker can record encrypted traffic between the victim's browser and the target domain, possessing the AES key protecting that traffic allows the attacker to decrypt and read every packet in the flow. Therefore, malicious NaCl code should attempt to induce traffic in an TLS session to a domain of interest.

The NaCl standard library provides the `URLRequestInfo` and `URLLoader` classes, which, when used in unison, can be used to fetch remote assets for use by NaCl clients. However, to direct traffic to our third-party victim domain, we must bypass the same origin policy as enforced by the browser. Usefully, when fetching remote resources, NaCl implements Cross-Origin Resource Sharing (CORS)[1]. This draft specification lets clients request cross-origin fetches by attaching optional headers to the request, and allows domains to explicitly enable such requests by adding optional headers to the response. Of course, in our attack model, the attacker cannot be certain that the remote domain implements CORS or allows cross-domain fetches from our attacking origin. However, their support (or lack thereof) doesn't impact the attack: we only need to cause AES encryptions and decryptions. To comply with CORS, Chrome must parse the response's headers before deciding if the NaCl client is allowed to see the results. So, a NaCl request for a remote resource will always trigger a fetch to be sent over the network. Therefore, by simply requesting any resource in the victim HTTPS origin, a malicious NaCl client can insert traffic into any SSL stream it chooses (even if it will never receive the results of that fetch).

We chose to investigate the feasibility of cache timing attacks on the Intel Atom N270 processor. Released in 2008 as one of the first Atom chips, the N270 is a relatively simple 32-bit single core hyperthreaded processor with a 512 KiB, 8-way associative L2 cache. Importantly, it lacks AES-NI hardware (as discussed in Section 2.1).

And so, *prima facie*, the possibility for a cache timing attack using Native Client against AES as used by Google Chrome seems quite good. However, in trying to implement such an attack, we have discovered three major roadblocks standing between us and a successful exploit on the N270: modern software engineering, prefetching, and cache indexing. Since we feel that these comprise a material setback against the viability of cache timing attacks in the wild, we present them in the next few sections.

[1]Online: `http://w3.org/TR/2012/WD-cors-20120403/`

4. MODERN SOFTWARE ENGINEERING

Unfortunately for a cache attacker, real–world programs do not perform thousands of AES operations and then immediately cease execution. At the very least, encrypted data will be written to disk or the network, incurring operating system complexity and overhead. Even local functions called before or after the AES operation require the processor to fetch instruction memory touched by the program counter, causing even more cache evictions and noise.

Therefore, to provide a lower bound on the cache noise that our attacker must disregard, we examine the internal architecture and code size of Chromium, the open source version of Google Chrome. First, we acquired and built Chromium r105554 in debug mode, and created a minimal NaCl application which causes a cross-origin HTTPS fetch. Next, we attached GDB to various parts of Chromium as the fetch was triggered, and wrote a script which, through judicious use of the `step` and `disassemble` GDB commands, revealed the size and location of each function called by Chromium during each period of execution. This provides a slight overestimate of actual utilized code size (since portions of each function are undoubtedly skipped by, e.g., an `if` statement), but this measurement completely ignores any data structures, including the stack, heap, global variables, and C++ virtual method tables. It also does not count any operating system overhead, such as process scheduling, memory mapping, handling interrupts, or facilitating interprocess communication. Therefore, we believe that this metric underestimates the total cache-impacting memory fetches during Chromium's execution.

4.1 Chromium Architecture

Chromium, to facilitate security guarantees, uses sandboxing mechanisms to separate responsibilities among several processes. For our purposes, we must inspect three distinct processes: the browser, the renderer, and the NaCl runtime. The browser is the main thread, responsible for managing all network and disk access, while the renderer is a origin-specific sandbox which handles parsing HTML/CSS and executing JavaScript in order to render a particular page. The NaCl runtime exists in a third process, where the NaCl application is loaded and run, and this runtime communicates solely with its parent renderer. When our attacking NaCl code initiates an asynchronous HTTPS fetch, the request first travels via RPC into the renderer, where it is checked for viability. Next, the renderer makes another RPC to the controlling browser process, requesting the fetch. At that point, the browser opens a socket and sleeps. When the OS reports that socket is writable, a browser thread wakes up and finally performs the AES encryption.

4.2 Measurements

We first attached GDB to the renderer process, at `NaClSrpcReceiveAndDispatch()`. While there are several more layers of calls above this one handling other tasks of the RPC, this is the function which initiates the URL loading process. Allowing GDB to run and disassemble all functions until this function returns, we find that the renderer thread calls at least 2288 distinct functions, totaling 282 KiB and spread across 484 memory pages.

Next, we attached GDB to the browser process, at `Http-NetworkTransaction::OnStreamReady()`. We then perform our step-and-disassembly analysis down the entire call stack, recording which functions are called. Note, again, that this underestimates the code size as well, since there are a few more general functions above `OnStreamReady()` in the call stack. Nevertheless, Chromium executes at least 894 distinct functions, encompassing 165 KiB on 254 pages.

Lastly, we examine the overhead of receiving a packet of TLSed data, with the idea that an attacker could initiate a very long download over HTTPS and asynchronously attack AES while the download completes. To do this, we set a breakpoint on `SSLClientSocketNSS::BufferRecvComplete` and executed until it returned. During this operation, 158 functions were executed for 29 KiB on 82 pages.

5. PREFETCHING

Designed to reduce memory latency and increase performance, hardware prefetchers speculate about future memory accesses and issue fetch requests for likely data. Unfortunately for our purposes, these fetches cause replacements and evictions on cache lines that would be otherwise untouched during AES operations.

First described by Baer and Chen in 1991 [5], stride prefetchers attempt to notice memory accesses at constant offsets (such as traversing an array) and issue requests for the next items in the series. This behavior wreaks havoc on naïve cache probing strategies — simply accessing 8 memory locations which map to the same cache index might trigger the ninth to be fetched, evicting one of the original 8. Tromer et al. describe a clever workaround: a linked list is created using the locations of each intended probing address, connecting every address in a random order. As the list is traversed, the stride prefetcher does not detect a regular pattern of memory accesses, and does not trigger fetches.

As processors evolve and gain extra transistors, prefetchers become more complicated. Since prefetchers greatly impact processor performance and interesting new prefetching strategy can be implemented without breaking compatibility, processor manufacturers do not release precise descriptions of their prefetchers' behavior. Security analyses usually assume access to the full specification of the system under attack, but reverse-engineering the prefetcher circuit from a 32 nm process chip would be prohibitively expensive.

Our experiments with the Atom N270 indicate that its hardware prefetcher does, in fact, influence the viability of cache timing attacks. In Figures 1 and 2, some of the prefetching behavior of the processor is revealed. To generate these timings, we first flush the cache, then access memory in a few dis-contiguous cache lines. Then, using `rdtsc`, we measure the access time for an individual cache line. The cache is then reset and the process repeated for the next location. Each location was measured 16 times.

Examining Figure 1, we see that, as expected, accesses to cache lines 0, 4, and 8 always hit in the cache. Surprisingly, lines 1, 5, and 6 always hit as well, indicating that the prefetcher is always active. We also see occasional hits at locations 9 through 12, and 15, indicating further prefetching.

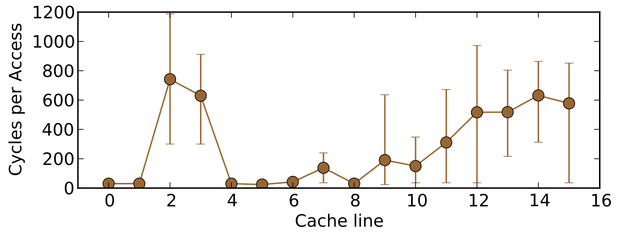

Figure 1: Access times per cache line after cache flush on Atom N270 (min, average, max across 16 runs). Timing done after accesses at lines 0, 4, then 8.

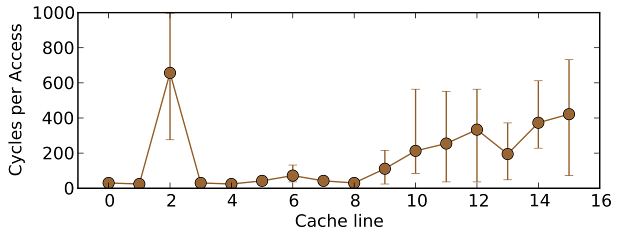

Figure 2: Access times per cache line after cache flush on Atom N270 (min, average, max across 16 runs). Timing done after accesses at lines 0, 3, 4, then 8.

In Figure 2, the prefetcher is even more active. Lines 0, 1, and 3 through 8 always hit, even though we prime only four lines. Cache hits are also occasionally seen at higher locations, indicating unpredictable prefetching behavior.

To avoid this complex prefetching behavior, the attacker can, in theory, access relatively distant cache lines. However, since fetching cache line A always brings in line $A + 1$ (see lines 0 and 1 in figures), our cache timing resolution is limited to 128 bytes, as accessing one 64 B cache line brings in its 64 B neighbor. When probing 1 KiB AES tables on the N270 (even if prefetching behavior is reverse-engineered and understood), cache measuring techniques offer at most 3 bits of table offset, as opposed to the 4 bits supposed by both Tromer and Gullasch.

Interestingly, though, hardware prefetchers can be disabled, and are often shipped disabled on server-class hardware. Note, however, that disabling the prefetcher requires either modifications to the BIOS or an instruction run at privilege level 0, and if an attacker can effect either of these things, they have full control over the hardware; a cache timing attack is superfluous. However, in a server environment, AES cache timing attacks may be able to sidestep the effects of a hardware prefetcher.

6. CACHE INDEXING

The basic operation of any cache timing attack is the cache line probe. To do so, current cache timing attacks load their own data into the cache and measure, through timing repeated accesses, when said data is evicted from the cache.

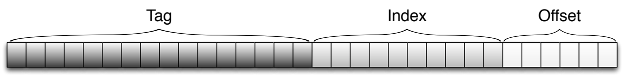

Figure 3: Address bits on Atom N270

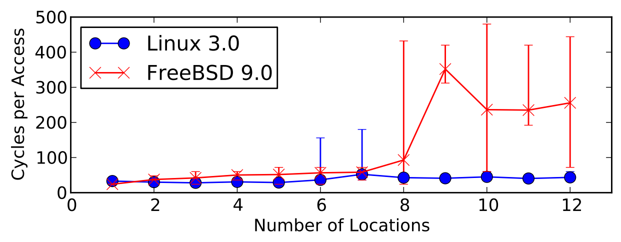

Figure 4: Average access time for virtual addresses which should collide in the cache. Error bars indicate the maximum and minimum time seen.

As one of the first steps towards implementing the attack, then, we tested how this might work on our Atom N270.

The L2 cache in the Atom N270 is 512 KiB, with a line size of 64 B and an 8-way set associativity. With these characteristics, the cache contains 1024 distinct sets, each able to contain 8 lines. Since each line is 64 bytes long, the memory address will be broken down into 6 tag bits, 10 index bits, and 16 tag bits (shown in Figure 3).

Therefore, addresses which differ only in their top 16 bits should map to a single cache set. Also, once we access 9 or more of these addresses, we should begin to see evictions and memory access times should spike.

To test this hypothesis, we created a program which begins by loading 1 MiB of useless data into the cache. This provides a fair baseline from which to start testing, since any memory location that we will test should now be evicted. Next, we access n memory locations at a stride of 16 KiB, taking care to use a non-linear order designed to minimize the effects of a stride prefetcher. Once this is done, we access all n locations again in the same order, measuring each memory access time, in cycles. We then ran this program on Linux 3.0.0 and FreeBSD 9.0. Figure 4 plots the average access time, across four runs, for each location in this second round against the total number of accesses.

As expected, when the number of locations is small, our entire data set fits in the cache, and accesses are fast. However, once we access 9 unique locations, there is a drastic performance difference between Linux and FreeBSD. On Linux, we continue to see access times in line with L2 cache hits; FreeBSD shows the expected behavior of memory fetches.

The explanation for this disparate behavior lies not only in the physical hardware of the N270, but in the virtual memory subsystems of each operating system as well. FreeBSD, unlike most modern operating systems, implements "cache coloring" [12], which attempts to allocate virtual memory so that virtual pages correspond to physical pages. That is, cache coloring allows processes to predict which pages will conflict in the cache, and optimize their memory accesses accordingly. Linux 3.0 and OS X do not implement cache coloring. In order to understand how cache coloring impacts our attack, however, we must discuss how the processor manages its caches.

The N270, along with almost every modern x86 CPU, has "physically-indexed" caches, or, more specifically, caches which slot memory locations into sets using their physical addresses, rather than their virtual ones. Since there is no mechanism for a process to discover the physical addresses of its memory (which, indeed, might change at any point), physically-indexed caches offer no exploitable guarantees about which addresses will map to the same cache set. This uncertainty increases the difficulty and complexity of mounting any sort of cache timing side channel attack.

6.1 Attack Complexity

From the physical capabilities of the N270, we can predict exactly how much more difficult an attack becomes on a physically-tagged processor. The N270, along with most other x86 processors, provides support for 4 KiB memory pages. When a virtual address is mapped to its physical counterpart, the lower $\log_2(4\,\text{KiB}) = 12$ bits remain unchanged. From Figure 3, the N270 uses address bits 15 through 6 to index into a cache set. Therefore, for any given virtual address on the N270, 6 of 10 index bits are already known, leaving 4 unknown index bits.

To fully predict how these unknown bits complicate an attack against Google Chrome's SSL AES implementation, we now examine the position and size of the AES lookup tables. On Ubuntu Linux 11.10, Chromium links against and uses the NSS library provided by the OS. Since Ubuntu Linux implements ASLR, we cannot predict the page's location in virtual memory (or, indeed, its physical tag), but the page offset will always remain the same! The AES lookup tables, when linked into Chromium's address space, begin at a page offset of 0xA0. Since the tables total 4 KiB, the area of interest to our AES cache attack spans two physical pages. As such, on mainstream operating systems, the attacker must now discover the physical positions of both these pages, among the $2^{4+4} = 2^8$ possible configurations. Since there are no direct methods of probing these tables, the attacker is reduced to either triggering AES operations and searching the cache for likely positions, or by simply guessing at the tables' position in physical space.

We note, as well, that when our attacking thread runs on physically-indexed caches, there is no reliable way to determine which addresses in its own virtual address space map to identical cache lines. However, by the pigeonhole principle, once the attacking thread possesses $(1024*8+1) = 8193$ pages of memory (a manageable 32 MiB), at least one cache set is guaranteed to contain at least 9 pages, and the attacker can conceivably discover, by eviction, which of their pages overlap in the cache.

7. CONCLUSION

In conclusion, we present an re-analysis of the computational complexity of a cache timing attack against AES with a key size of 128 bits, modeled after that of Tromer. Their analysis is detailed and precise but, as a full summary will not fit in this paper, we make do with an abbreviated version. First, let δ be the cache line size divided by the size of a table entry (4 bytes). Due to the well-meant meddling of the prefetcher, the effective distinguishable cache line size on the Atom N270 is not the physically-defined 64 B, but rather 128 B, so $\delta = 32$. For comparison, in Tromer's initial analysis, $\delta' = 16$.

Next, Tromer's analysis assumes access to an ideal predicate $Q_k(p, l, y)$, which is defined to be 1 iff, during the AES encryption, the block y in lookup table T_l is accessed at least once, for key k and plaintext p. With this predicate, an attack against the simplest first round of AES reveals the top $\log_2(256/\delta) = 3$ bits of each key byte.

Next, Tromer applies a attack against the second round of AES. From analysis of the AES algorithm, they find that four table lookups in the second round can be directly computed from four key bytes. Applying $Q_k(p, l, y)$ with $l = 2$, they compute the table accesses across every possible key. In their analysis, since $\delta' = 16$, this gives an analysis complexity of $\delta'^4 = 2^{16}$ — easily manageable. In our case, however, $\delta^4 = 2^{20}$. This process must then be repeated four times, since each iteration reveals the lower bits of four bytes, and our key is 16 bytes long. To sample $Q_k(p, l, y)$ enough for this analysis to function requires about $\log \delta^{-4} / \log(1 - \delta/256 \cdot (1 - \delta/256)^{38})$ samples, which for our δ is 17,720.

Therefore, our second round attack has a full computational complexity of $4 \cdot 2^{20} \cdot 17,720 \approx 2^{36}$ to extract the full AES key. Already, this is looking dire.

Compounding the issue further, $Q_k(p, l, y)$ is an ideal predicate — we must make do with an approximation. Furthermore, we must take into account the cost of finding the AES tables in memory, given the fact that modern processors are exclusively physically-tagged. Without some clever trick, we posit that the best an attacker can do is guess, for both pages, where in the cache they lie. Since each page has 4 unknown bits on the Atom N270, we must repeat our analysis $2^{4+4} = 2^8$ times, once with each guess. This brings our total expected work to 2^{44}.

Finally, as if it weren't hard enough, the attack must be able to pick out the AES page table accesses from the avalanche of memory fetches caused by simply executing the target program. Whenever a NaCl client requests a remote resource, approximately 447 KiB of code is accessed. Within a 512 KiB cache, this represents a serious disturbance to any prepared cache timing attacker. Attempting to extract the effects of a few thousand accesses to a region of 4 KiB in an unknown location appears to be a herculean effort.

Therefore, we posit that any data-cache timing attack against x86 processors that does not somehow subvert the prefetcher, physical indexing, and massive memory requirements of modern programs is doomed to fail, to say nothing of the difficulties imposed by multicore processors and hardware AES implementations.

8. REFERENCES

[1] O. Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In R. Sandhu and J. A. Solworth, editors, *Proceedings of CSAW 2007*, pages 11–18. ACM Press, Nov. 2007.

[2] O. Acıiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In S. Mangard and F.-X. Standaert, editors, *Proceedings of CHES 2010*, volume 6225 of *LNCS*, pages 110–24. Springer-Verlag, Aug. 2010.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In M. Abe, editor, *Proceedings of CT-RSA 2007*, volume 4377 of *LNCS*, pages 225–42. Springer-Verlag, Apr. 2007.

[4] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In A. Perrig and R. Sion, editors, *Proceedings of CCSW 2010*. ACM Press, Oct. 2010.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Nov. 1991.

[6] D. J. Bernstein. Cache-timing attacks on AES, Apr. 2005. Online: `http://cr.yp.to/papers.html#cachetiming`.

[7] N. T. Billy Bob Brumley. Remote timing attacks are still practical. In V. Atluri and C. Diaz, editors, *Proceedings of ESORICS 2011*, volume 6879 of *LNCS*, pages 355–71. Springer-Verlag, Sept. 2011.

[8] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, pages 201–215, 2006.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. `http://eprint.iacr.org/`.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of USENIX Security 2003*, 2003.

[11] A. Canteaut, C. Laradoux, and A. Seznec. Understanding cache attacks. Technical report, Inria, Apr. 2006. Online: `http://hal.inria.fr/docs/00/07/13/87/PDF/RR-5881.pdf`.

[12] M. Dillon. Page coloring. Online: `http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html`.

[13] J. Fruehe. Following instructions. Online: `http://blogs.amd.com/work/2010/11/22/following-instructions/`, Nov. 2010.

[14] S. Gueron. Intel advanced encryption standard (aes) instructions set. Online: `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/`, Jan. 2010.

[15] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of IEEE Security and Privacy ("Oakland") 2011*, May 2011.

[16] Intel. Intel advanced encryption standard instructions (aes-ni). Online: `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/`, Oct. 2010.

[17] Intel. Intel ark. Online: `http://ark.intel.com/search/advanced/?AESTech=true`, Jan. 2010.

[18] Mozilla. Bug 459248 - support intel aes extensions. Online: `https://bugzilla.mozilla.org/show_bug.cgi?id=459248`, Feb. 2009.

[19] T. Müller, F. C. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In D. Wagner, editor, *Proceedings of USENIX Security 2011*. USENIX, Aug. 2011.

[20] OpenSSL. openssl/changes 1.1686. Online: `http://cvs.openssl.org/fileview?f=openssl/CHANGES&v=1.1686`, Mar. 2012.

[21] C. Percival. Cache missing for fun and profit. Presented at BSDCan 2005, May 2005. Online: `http://www.daemonology.net/papers/htt.pdf`.

[22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In S. Jha and A. Keromytis, editors, *Proceedings of CCS 2009*, pages 199–212. ACM Press, Nov. 2009.

[23] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, Jan. 2009.

[24] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen (short paper). In T. Ristenpart and C. Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, Oct. 2011.

[25] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of l2 cache covert channels in virtualized environments. In C. Cachin and T. Ristenpart, editors, *Proceedings of CCSW 2011*. ACM Press, Oct. 2011.

[26] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In G. Danezis and V. Gligor, editors, *Proceedings of CCS 2012*. ACM Press, Oct. 2012. To appear.