Browser Security with a Grain of Salt: Drive-by Key-Extraction Cache Attack via PNaCl

Daniel Genkin
University of Pennsylvania and
University of Maryland
danielg3@cis.upenn.edu

Lev Pachmanov
Tel Aviv University
levp@tau.ac.il

Eran Tromer
Tel Aviv University and
Columbia University
tromer@tau.ac.il

Yuval Yarom
The University of Adelaide and
Data61
yval@cs.adelaide.edu.au

August 22, 2017

Abstract

We show how malicious web content, served by a webpage or an online ad, can extract cryptographic secret keys from the user's computer. The attack does not rely on bugs in the browser's nominal sandboxing mechanisms, or on user actions, but rather on cache side-channel attacks. The malicious web content uses PNaCl (an efficient alternative to JavaScript, supported by Chrome) to induce contention for the CPU cache resources with other programs running on the user's computer, and makes deduction about those programs' data.

We show that even on Chromebook devices (a secure, locked-down, browser-only platform), a malicious online ad can extract ElGamal and ECDH decryption keys from various cryptographic libraries.

Moreover, on a browser platforms the attacked software too may be written in portable JavaScript; and we show that in this case even supposedly-secure constant-time algorithms, such as Curve25519's, are vulnerable to our attack.

1 Introduction

Since their introduction over a decade ago [65, 7, 58, 57], microarchitectural side channel attacks have become a serious security concern. Contrary to physical side channels, which require physical proximity for exploitation, microarchitectural attacks only require the attacker to have the ability to execute code on the target machine. Even without any special privileges, such code can contend with concurrently-executing target code for the use of low-level microarchitectural resources; and by measuring timing variability induced by this contention, an attacker can glean information from the target code. Many such resources have been analyzed and exploited, including branch prediction units and arithmetic units, but contention for cache resources has been proven to be particularly devastating. Cache attacks allow fine grained monitoring of the target's memory access patterns, and have been demonstrated to successfully extract secret information such as secret cryptographic keys [7, 58, 57], website fingerprinting [56], and keyboard sniffing [32]. See Ge et al. [23] for a survey.

Less is known, however, about realistic attack vectors by which cache attacks (and other microarchitectural attacks) be deployed in practice. Most research has assumed that the attacker has the ability to run native code on the target machine. This makes sense for scenarios such as attacks across virtual machines [59, 35, 68], especially in public compute clouds, or attacks between different users sharing the same PC. But in the typical end-user setting, hardware devices are not shared by multiple mistrusting users. Moreover, native code, ran locally by a user, usually executes in a security context that allows access to that user's data, making security-savvy users reluctant to run such code.[1]

Recent works [56, 30] made progress towards effective cache attacks on end-user devices, using JavaScript code running in the target's browser and without requiring native code execution. However, since JavaScript is far-removed from the native platform, the information obtained by a JavaScript attacker is severely degraded. Indeed compared attacks which are based on native-code execution, those works were only able to detected coarse-scale events (distinguishing between websites loaded in another browser tab or ASLR de-randomization), leaving open the feasibility of monitoring and exploiting fine-grained events.

Thus, in this work we focus on the following question: **(a) Are there practical deployment vectors for microarchitectural attacks on single-user devices, that are capable of extracting fine-grained information (such as cryptographic keys), and do not require privileged user operations (such as software installation or native code execution)?**

A particularly challenging attack setting is that of locked-down platforms, such as Chromebook devices running Chrome OS, where functionality is restricted to sandboxed web browsing, and even the user is not at liberty to install or execute native code. In light of the growing popularity of such platforms, we ask: **(b) Are microarchitectural key-extraction attacks feasible on locked-down, browser-only platforms?**

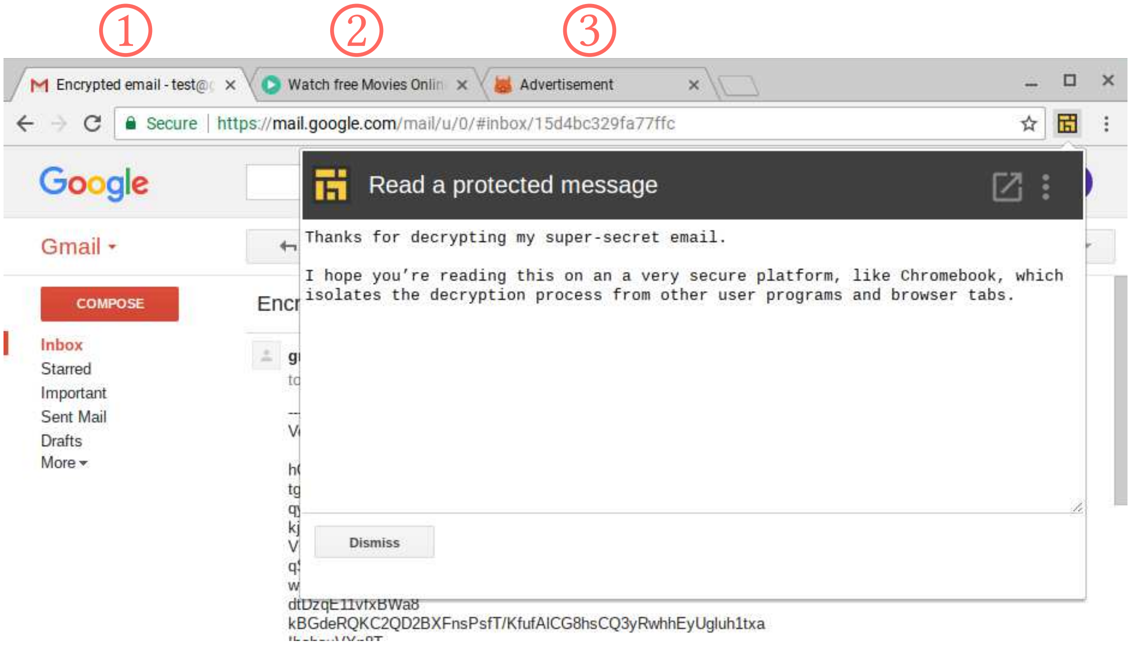
Even when microarchitectural information leakage occurs, its exploitability depends on the implementation of the attacked software. Modern cryptographic software is often designed with side channels in mind, employing mitigation techniques that require the programmer to carefully craft low-level details of the code execution — first and foremost, to make it constant-time. This picture changes when cryptographic software deployed as portable high-level code (as is desired for portability, and indeed necessary in the aforementioned locked-down platforms), where the final code and memory layout are left to the whims of a just-in-time compiler. On one hand, defensively exercising the requisite control becomes more difficult. On the other hand, the attacker too has to cope with increased variability and uncertainty, so it is not obvious that leakage (if any) is at all exploitable. We thus ask: **(c) Do portable program representations compromise the side-channel resilience of (supposedly) constant-time algorithms?**

1.1 Our Results

We answer all three questions in the affirmative. (a) We present cache side-channel attacks which can be executed from a web page loaded in a sandboxed browser environment, and are capable of extracting keys from cryptographic implementations. (b) We demonstrate these empirically on several platforms, including Chromebook. (c) We demonstrate key extraction even from an implementation of Curve25519 Elliptic Curve Diffie-Hellman, which was explicitly designed to minimize side channel leakage, but becomes susceptible due to use of high-level JavaScript.

Our attacks do not require installing any software on the target machine, and does not rely on vulnerabilities in the browser's nominal isolation mechanisms (e.g., they work even if Same Origin Policy is perfectly enforced). Rather, they gleans information from outside the browser's sandbox

[1] One exception is mobile platforms such as Android, where it was shown that a malicious app can use cache attacks spy on touch screen activity and even cryptographic operations, despite the highly constrained security context — but still, only after the user has actively installed the malicious app [45].



```
Thanks for decrypting my super-secret email.

I hope you're reading this on an a very secure platform, like Chromebook, which
isolates the decryption process from other user programs and browser tabs.
```

Figure 1: Attack scenario screenshot. The targeted user opens an online streaming web-site in Tab 2. Clicking within this tab (fe.g., to start a movie) causes a pop-under to open up as Tab 3. The malicious advertisement in Tab 3 then monitors the cache activity on the target machine. When an encrypted email is received and decrypted using Google's encrypted email extension [29] (in Tab 1), the malicious advertisement in Tab 3 learns information about the user's secret key.

purely by inducing and measuring timing variability related to memory accesses outside its sandbox. All the target user has to do in order to trigger the attack is to have its browser execute malicious code embedded in a comprised website.

Drive-by Attack. The main attack scenario we investigate is a "drive-by" web attack, where the attacker's code is embedded in a web page and is automatically activated when this web page is rendered by the user's browser. This can happen when the user explicitly visits the attacker's web page (e.g., enticed by phishing), or a web page into which the attacker can inject HTML code (e.g., by a cross-site scripting attack). Most deviously, the attack may be automatically triggered when the user visits unrelated third-party web sites, if those sites display advertisement from a web advertising service that support non-static ads (whether as pop-under ads, IFRAME ads, or JavaScript).

Concretely, we have embedded the attack code in an advertisement, which we submitted to a commercial web advertisement service. Whenever a user navigated to a web site that uses that service, and our ad was selected for display, the attack code was triggered (see Figure 1). This attack code measured the memory access patterns on the user's machine, and sent it to our server for analysis. If the targeted cryptographic software happens to be repeatedly invoked using some secret key during the time when the advertisement was shown in some browser tab (even in the background), our code extracted the secret key in about one hour.

Attacking Constant-Time Curve25519. One of our attacks targets a JavaScript implementation of Curve25519 Elliptic Curve Diffie-Hellman (ECDH) implementation [8]. The implementation attempts to mitigate side channel leakage by using a nearly constant-time Montgomery-ladder scalar-by-point multiplication, but the in-browser compilation from JavaScript introduces key-dependent control flow, which we can detect and exploit by an in-browser code-cache side-channel attack.

Measurement Technique. We implement the cache measurement code using Portable Native Client (PNaCl) [27]. This is an architecture-independent code representation executed by the browser, in a sandbox, by a corresponding compiler/interpreter — analogously to JavaScript, but lower-level and more efficient. PNaCl is supported by desktop versions of the Chrome and Chromium browsers. PNaCl code is automatically executed by the browser without user involvement.

Like JavaScript, PNaCl is strongly sandboxed, subject to Same Origin Policy, and isolated from host resources such as filesystem and other processes. However, PNaCl code (inevitably) uses the underlying microarchitectural resources of the CPU it is executing on, and in particular the data cache. Thus, it can induce the memory-contention effects required by cache side-channel attacks. Moreover, PNaCl supports arrays, which are naturally implemented as contiguous ranges of virtual memory on the host platform; this offers the attacker's PNaCl code some control over the addresses of the memory accesses induced by its execution, and thus some specificity in the part of the cache where contention is induced. Using these phenomena, and additional techniques, the PNaCl code can execute a variant of the Prime+Probe attack of [57], to detect which memory addresses are accessed by other processes running on the same CPU.

Compared to the two prior works on portable-code cache attacks (see Section 1.3), our use of a portable but low-level program representation, as opposed to JavaScript in [56], reduces measurement overheads and provides better timing sources on modern browsers; and by using a precise eviction set construction algorithm (following the approach of [46] and adapted to the PNaCl setting) we moreover reduce the eviction sets' size by a factor of 64 compared to [30]. Taken together, these let us attain the requisite temporal resolution for implementing several cryptanalytic attacks.

Vulnerable Platforms. We empirically demonstrate our attack on commodity laptop computers as well as on Google Chromebooks. We observe that Chromebooks (running Google's Chrome OS) present a particularly hard target for microarchitectural side channel attacks. This is since Chrome OS is a locked-down operating system where users are essentially constrained to browser functionality with most of the platform administrative capabilities (i.e. as available via the Unix "root" account or the Windows "Administrator") are not available to users. Google claims that "Chromebook is the safest computer one can buy" [22], boasting that Chrome OS is the "first operating system designed with [malware] threat in mind" [26]. The operating system further uses sandboxing, verified boot, data encryption, and automatic updates in order to minimize the attack surface. The advertised security features, together with the low price-tag of Chromebooks has seen an increase in the popularity of the platform, for example in K-12 education [63].

While the above design indeed limits the capabilities of potential attackers, as we show in this paper, none of these measures addresses flaws in the underlying processor hardware thus making software key extraction attacks possible.

Challenges. Launching cache attack involves numerous challenges, such as recovering the mapping between the memory and the cache, and identifying cache sets corresponding to security-critical accesses (see [46] for a detailed list). Launching the attack from PNaCl code introduces several additional challenges:

1. *Emulated environment:* PNaCl modules runs inside an emulated 32-bit environment, preventing access to useful host platform services such as huge pages, `mlock()` and `posix_memalign()`.
2. *Slower memory access:* memory accesses using (current implementations of) PNaCl entail an overhead compared to native execution, reducing the measurements' temporal resolution.

3. *Inability to flush CPU pipeline:* PNaCl does not support instructions for flushing the CPU pipeline or avoiding out-of-order execution, as needed by many native-code attacks.
4. *Inability to flush the cache:* PNaCl does not include instructions for flushing memory blocks from the CPU cache, as used in FLUSH+RELOAD [68].
5. *Inaccurate time source:* Architecture-independence forces PNaCl applications to use only generic interfaces or indirect measurements to measure time.

Moreover, the cryptographic software we attack is implemented in JavaScript, which introduces yet more challenges:

6. *Unpredictable memory layout:* The target's JavaScript code is compiled anew at every page load, and moreover, its memory allocation are done in an unpredictable way at every invocation.
7. *No shared memory:* Many prior cache attacks relied on the attacker code and target code having some shared memory (e.g., AES S-tables or code), due to shared libraries or memory deduplication, which are not unavailable here.

Our attack approach addresses or circumnavigates all of these.

1.2 Targeted Hardware and Software

Chrome OS. Chrome OS is an operating system designed by Google, based on the Linux kernel the Chrome web browser. Its primary functionality is as a thin web client. Chrome OS is a locked-down platform, that not allow even the device owner to run arbitrary native code.[2] Security measures including verified boot, a stripped-down operating system, enforced auto-updates, and numerous process-isolation and hardening mechanisms.

Overall, Chrome OS is arguably the most secure consumer platform available for web browsing with modern HTML5 functionality.

Chromebook. We demonstrate the attack on a Chromebook device (Samsung XE550C22) which is tailored for running Chrome OS, including all of its security measures. This device ran Chrome OS 58.0.3029.112 (the latest version at the time of writing). It is equipped with an Intel Celeron 867 Sandy-bridge 1.3GHz CPU featuring a 2048KB L3 cache divided into 4096 sets and 8 ways.

Laptop. The attack is mostly independent of the operating system, and of the precise CPU model (within a CPU family). To demonstrate this, we also executed the attack on an HP EliteBook 8760w laptop, running Kubuntu 14.04 with Linux kernel 3.19.0-80, with an Intel i7-2820QM Sandy Bridge 2.3GHz CPU featuring a 8192KB L3 cache divided into 8192 sets and 16 ways.

Google's End-to-End Library. End-to-End [29] is an open-source cryptographic library developed by Google. Designed for facilitating end-to-end encryption in web applications, it is written entirely in JavaScript and can be incorporated in any website or browser plugin, for execution inside users' browsers. End-to-End supports both public-key operations, such as digital signatures and asymmetric encryption, and secret-key operations, such as symmetric encryption and hashing. To facilitate email encryption and signing directly inside the user's browser, End-to-End supports the OpenPGP standard, as documented in RFC 4880 [13]. Furthermore, End-to-End is the cryptographic engine for many browser plugins such as E2EMail [18], Google encrypted email extension [29], and Yahoo End-to-End extension [67].

OpenPGP.js. OpenPGP.js [12] is a popular open-source library for browser-based cryptographic operations, and in particular encrypted email. Similarly to End-to-End, OpenPGP.js implements

[2] This can be overriden by switching the system into *developer mode*, which entails deleting all user accounts and their files, voiding the warranty of the device, and on some devices, activating a hidden hardware switch.

the OpenPGP standard and is widely deployed in web-applications and browser plug-ins that require cryptographic functionality. These include password managers [6], secure transport layer provider [17], encrypted mail clients [72, 21] and other applications. To create seamless user experience, some of those plug-ins [72, 21] automatically decrypt received content upon opening the received email.

Elliptic. Elliptic [36] is an open source cryptographic library written in JavaScript, providing efficient implementations of elliptic-curve cryptographic primitives such as Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signatures (ECDSA). Elliptic is widely used (over a million downloads per week according to NPM package manager [53]), and more than a hundred dependent libraries including crypto-currency wallets [37, 47, 52, 73]. Elliptic supports state-of-the-art elliptic curve constructions such as Curve25519 [8], which was designed to offer increased resistance to side channel attacks. In particular, Elliptic uses a Montgomery Ladder based implementation of the scalar-by-point multiplication operation, which performs the same arithmetic operations irrespective of the values of the secret key bits. While our techniques do achieve key extraction, we empirically verified that Elliptic's implementation does not leak the secret key via timing variations. Instead, our techniques recover the key using memory access leakage created form Elliptic's JavaScript implementation of the Montgomery ladder.

Ethical Considerations. When using the commercial advertisement service for demonstrating attack deployment, we took care to avoid impacting unrelated users who were shown our ad. The attack page first checks for an existence of a specific browser cookie (which was manually set only in our intended victim), and exits immediately if the cookie is amiss. On the server side, we indeed did not observe any unexpected receipt of measurement data.

1.3 Related Work

Cache Attacks. Cache attacks were introduced over a decade ago [65, 7, 58, 57]. Initial attacks exploited the L1 data cache (and L2 data cache in [57, 64]), however later attacks targeted other caches, such as the L1 instruction cache [2, 5] the shared last level cache [46, 34] and specialized caches including the branch prediction unit [3, 4, 20] and the return stack buffer [11]. Recent works [61, 30] were able to extract information without assuming huge pages. See [23] for a survey.

Cache Attacks from Portable Code. The first published browser-based cache attack was shown by [56]. Using JavaScript, they detected coarse cache access patterns and used them to classify web sites rendered in other tabs. They did not demonstrate attacks that use fine-grain cache monitoring (such as key extraction attacks). Moreover, following [56] web browsers nowadays provide reduced timer precision, making the techniques of [56] inapplicable.

Recently, [30] achieved higher cache-line accuracy, and used it to derandomize the target's ASLR from within it's browser. However, the technique of [30] relied on constructing very large eviction sets, resulting in low temporal resolution of the memory access detection, well below what is required for key extraction attacks (see Section 3).

JavaScript was also used Gruss et al. [31] for implementing browser-based Rowhammer attack [41].

Side-Channel Attacks on ElGamal Encryption. Several works show side-channel attacks on implementations of ElGamal encryption. Zhang et al. [74] show a cross-VM attack on ElGamal that exploits the L1 data cache and the hypervisor's scheduler. Our attack is loosely modeled after the work of Liu et al. [46] who implement a Prime+Probe attack [57] targeting an implementation of ElGamal. Recently, [24] show a physical (electromagnetic) side-channel attack on an ElGamal implementation running on PCs.

2 Preliminaries

2.1 Portable Code Execution

One of the features of the Chrome browser is the Google Native Client (NaCl) platform [71]. It is a sandboxing technology which enables secure execution of native code as part of untrusted web applications. In particular, NaCl allows for running compiled C and C++ code at near-native speeds and includes support for vectors, parallel processing and fine-grained control over the memory usage. While NaCl provides operating system independence, it requires the developer to generate architecture-specific executables for each hardware platform. Portable Native Client (PNaCl) [27] solves the portability issue by splitting the compilation process into two parts. First, the developer compiles the source code into an intermediate representation, called a *bitcode executable*. Next, as part of loading the code to the host browser, the bitcode executable is automatically translated to the host-specific machine language. This design allows PNaCl to obtain architecture and operating system independence while achieving the same security guarantees as NaCl. PNaCl is enabled by default, does not require any specific plug-ins and executes PNaCl modules without requiring user approval.

2.2 Memory Hierarchy of Intel CPUs

Cache Structure. The execution speed of modern CPUs is higher than the speed of typical memory hardware. To bridge this speed gap, CPUs are equipped with a hierarchy of caches, with each layer in the hierarchy being larger and slower then the previous layer. Intel CPUs typically have two or three levels of caches, ranging from very small (typically 64 KB) and fast level-1 (L1) caches, located close to each core, to a relatively large (a few megabytes) and slow last level cache (LLC), which is shared among all of the cores.

Cache Lookup Policy. When the CPU attempts to access the memory, the cache hierarchy is first consulted to check if a copy of the required data is cached, thereby avoiding a slow memory access. The CPU first tries to find the required memory address inside each successive cache layer. In a *cache hit*, when the requested address is found in the cache, the cached content is made available to the CPU without the need to perform a memory access. Otherwise, in a *cache miss* the main memory is accessed, bringing the contents of the requested memory address to all the cache layers, while evicting older entries from the caches.

Cache Sets. Modern caches are typically *set associative*. They consist of multiple *sets* each containing multiple *ways*. Every memory block (typically 64 bytes) is mapped to a specific set based on its physical address. A memory block can only be stored in a way of the set it maps to. Consequently, whenever a memory block is entered into the cache it can only evict blocks from the same cache set.

Cache Slices. For L1 and L2 caches, the mapping from addresses to cache sets is based on a consecutive range of (physical) address bits. The Intel LLC, however, has a more complex design. The cache is divided into multiple *slices*, each operating as an independent cache. While Intel does not disclose the mapping of memory blocks to cache sets, past works have reverse-engineered it [69, 33, 48, 38]. These show that the processor uses a hash function that converts the physical address to a slice number, and uses a range of consecutive bits of the address (bits 6–16 before Skylake and 6–15 in Skylake) to determine the cache set within the slice.

Cache Attacks. When multiple processes access memory block that map to the sames cache sets, memory block of one process might be evicted from the cache to make room for the memory of the other process. As a result of such evictions, future memory accesses of the first process will be

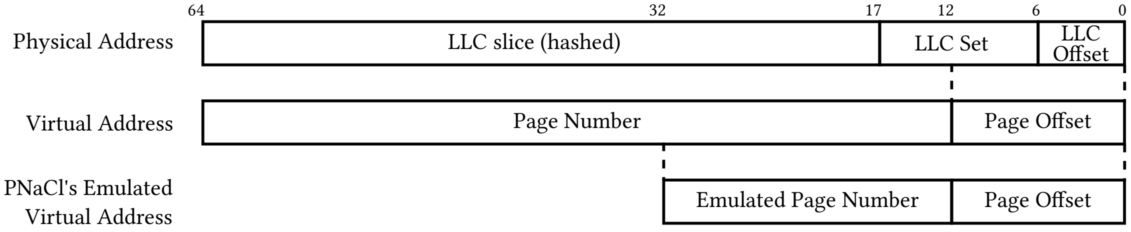


Figure 2: Physical vs. Virtual vs. PNaCl's address spaces on a CPU with 4096 last level cache sets. Dashed lines indicate equal parts. Notice that PNaCl's environment has only the lower 6 bits of the cache-set.

considerably slower. Thus, measuring the time it takes to perform memory access operations may leak information about the memory access patterns of other processes. Starting with [58, 57], this observation has led to numerous side and covert channel attacks utilizing this cache channel. We now proceed to describe the Prime+Probe attack of [57] which we utilize in this paper. At high level, the attack consists of three main phases.

- **Prime.** The attacker process first *primes* the cache by performing memory accesses to some carefully-chosen memory blocks, selected to fill-up some specific cache sets with the attacker's data. We use the name *eviction set* for a collection of memory blocks that all map to the same cache set.
- **Wait.** The attacker process then waits for some time, during which the victim process performs its memory accesses. If the target process accesses memory blocks that map to previously-primed cache sets, the victim's data will evict the attacker's data from these sets. This eviction will cause future cache misses when the attacker attempts to access the previously-cached data again, increasing the time required to complete the attacker's memory accesses.
- **Probe.** The attacker then *probes* the cache by accessing the memory addresses used for priming the cache, while monitoring the time required to perform each access. In case a memory access takes longer to complete, the attacker knows that the victim accessed a memory block that map to the corresponding cache set.

3 Constructing Eviction Sets

The Prime+Probe attack relies on having an eviction set for every targeted cache set. The main obstacle to constructing these sets is the requirement of finding the mapping between the internal addresses used in the attacker's program and the cache sets they map to. In the case of PNaCl, as shown in Figure 2, the mapping from memory addresses to cache sets consists of multiple abstraction layers which we now describe. First, the PNaCl runtime emulates a 32-bit execution environment, mapping the internal addresses to the hosting process's virtual address space. Since a running PNaCl code only handles pointers to its emulated address space, it does not have access to the mapping from the emulated address space to the hosting processes virtual address space. Next, the virtual memory subsystem of the operating system, together with the hardware memory management unit of the processor map the virtual address space into the physical memory. In a nutshell, the virtual address space is divided into fixed size chunks (typically 4096 bytes) called *pages* where each page is mapped to a *frame* in the physical memory. In Linux until kernel version 4.0 processes had access to this mapping. However, due to the Rowhammer attack [41] this option has been removed and in modern operating systems the mapping is no longer available to user processes [15].[3]

The final layer of the mapping converts physical memory addresses to cache sets. As we mentioned, Intel does not disclose this mapping, but it has been reverse-engineered. Yet, without the knowledge of the physical address bits we are unable to use this knowledge to find the mapping of memory blocks to sets.

Past Approaches. Several prior works [46, 31, 49] describe techniques for creating the eviction sets required for implementing microarchitectural attacks using *huge pages*: a CPU feature that allows pages in the page table to have a very large size (in Intel processors, typically 2 MB instead of 4 kB), for improved address translation efficiency (e.g., reduced TLB thrashing).

Because both the physical and the virtual starting addresses of a huge page must be a multiple of a huge page size, the virtual address and its corresponding physical address share the least significant 21 bits. In particular, that means that given a virtual address in a huge page, we know bits 0–20 of the physical address and consequently we know the index within a slice of the cache set that the virtual address maps to.

Avoiding Huge Pages. Recent attacks [61, 30] were able to avoid huge pages, at the cost of imposing other limitations. The attack of [61] assumes consecutive physical memory allocation and deterministic heap behavior. The assumption that physical memory allocations are made consecutively is not generally applicable. Furthermore, assuming consecutive allocation allows the attacker to find the cache set index up to a fixed offset. Consequently, the consecutive allocation provides as much information as using huge pages. Moreover, for JavaScript code running in a browser environment, we empirically did not observe any allocation pattern between different execution of the decryption operations. We conjecture that this is due to JavaScript's complex garbage collection pattern.

Next, the work of [30] avoided huge pages by only using the 6 bits shared between the virtual address and physical address to construct the eviction-sets. In this approach, all the cache-sets sharing the 6 least significant bits are mapped to a single large eviction set. However, using such large eviction sets increases probing time by a factor of ×64 (compared to smaller eviction sets which are are designed to only evict a single cache set) thus reducing the channel's bandwidth. In addition, large eviction sets also increase measurement noise, due to mapping of noise from unrelated memory accesses to larger sets. While [30] was able to use this approach to obtain memory access patterns which are sufficiently accurate to derandomize ASLR, our attack can not be launched using these techniques. This is since, in addition to increasing measurement noise, the reduction in the channel's bandwidth prevents accurate rapid measurements of the target's memory access, which are necessary for key extraction attacks (for temporal resolution on the order of a big-integer-multiplication or curve-point-addition).

Recovering the Mapping Between Virtual Address and Cache Slices. We now turn our attention to recovering the mapping between virtual addresses and cache slices. Past techniques rely on the knowledge of the cache set index, revealed when using huge pages. The crux of these techniques is observing cache collisions between memory blocks that share the cache set index. While there are some technical variations between these techniques, they are mostly similar. We now describe these techniques, mostly following the methodology of Liu et al. [46].

Eviction Testing. The main component of all eviction set construction techniques is *eviction testing*. This operation finds whether accessing a list of memory blocks forces a cache eviction of a specific memory block, to which we shall refer as the *witness* memory block.

[3] Even had this mapping remained available for user processes, the sandoxeded PNaCl environment makes it inaccessible for PNaCl code.

The assumption is that accessing a list of blocks will evict the witness memory block if the list contains enough blocks that map to the same cache set as the witness. Thus, to test a list, we first access the witness to ensure that it is in the cache. We then access all of the memory blocks in the list before performing a final memory access to the witness block. To test whether the witness was evicted from the cache we measure the time this final access takes. If the witness block is still in the cache, this final access would be faster than if the witness block was evicted, where the final access results in a cache miss. This process is typically repeated several times to remove noise from unrelated system activity.

Constructing the Eviction Sets. To construct an eviction set for some witness memory block, we first start with a pool of memory blocks that all have the same cache set index. We repeatedly process the pool to find memory blocks that all map to the same cache set. This process consists of three phases:

Expand: create a list of memory blocks that forces the eviction of a known witness block. We start with an empty list of memory blocks and repeatedly pick a witness from the pool. We then perform eviction testing, i.e. we access the entire list of memory blocks, to check if it evicts the witness block. In case it does then the expand phase is over and we proceed to the next phase. Otherwise, we add the witness block to the list of memory blocks and proceed to pick a new witness block. Assuming the pool of witness blocks is large enough, at some stage the list will have enough memory blocks that map to the cache set of the current witness, at which time the list is found.

Contract: process the above list in order to remove all of the memory blocks that do not map to the witness's cache set. To that aim, we iterate over the memory blocks in the list. We remove a memory block from the list and check whether the list still evicts the witness block. If it does, we proceed to removing the next memory block from the list. Otherwise, we return the removed memory block to the list and proceed to pick a different memory block. At the end of the process we remain with only those memory blocks that are essential for evicting the witness in the list. These form the eviction set for the witness's cache set.

Collect: remove redundancies. To avoid having multiple eviction sets for the same cache set, we remove from the pool all of the memory blocks that map to the same cache set as the witness block. This is done by iterating over the pool and remove any memory block that the eviction set evicts.

Constructing Eviction Sets in PNaCl's Virtual Memory Environment. Unfortunately, PNaCl modules cannot use the technique described above to construct the eviction set. This is since PNaCl code does not support huge pages and consequently we do not have the cache set index information from the least significant bits of the virtual address. Instead, as can be seen from Figure 2, PNaCl code only has access to the 12 least significant bits of the physical address (bits 0–11). Thus, the PNaCl code knows the six least significant bits of the cache set index, but is missing the four or five most significant bits.

To overcome this, we first find eviction sets for all of the cache sets that have indices with 6 least significant bits being zero. To that end, we create a large pool of memory address whose least significant 12 bits are zero. Note that these bits are preserved when mapping to physical addresses, hence all of the addresses in the pool are in the sets we target. We then apply the above described method and find eviction sets in the pool. However, for the method to work, we need to modify the eviction testing described above. This is since when running on a system configured with small memory pages, performing eviction testing as described accesses a large number of memory pages. This stresses the address translation mechanism, and in particular causes evictions from the Translation Lookaside Buffer (TLB), which is a specialized cache used for storing the results of recent address translations. These TLB evictions causes delays in memory accesses even when the accessed memory block is cached. In particular, this introduces noise when checking whether the witness block is successfully evicted.

To address the TLB noise, we modify the eviction testing approach, ensuring that the TLB entry for the witness block is updated before we measure the access time for accessing the witness memory block. We achieve this by accessing another memory block in the same page as the witness. Thus the eviction testing algorithm becomes:

1. access the witness to ensure it is in the cache.
2. access each memory block in the list of memory blocks.
3. access a memory block in the same page as the witness to ensure the TLB entry for the page is updated.
4. measure the access time to the witness. This will be short if the witness is in the cache or long if accessing the list of memory blocks evicts the witness from the cache.

Extending the Eviction Sets. For each cache set value v whose 6 least significant bits are set to 0, we construct an eviction set S_v using the above method, obtaining 64 initial eviction sets. We now describe how to construct the remaining 4032 eviction sets, without additional cache profiling. Indeed, since the 6 least significant bits of the cache set index correspond to bits 6–11 of the PNaCl virtual address (see Figure 2), we can construct the remaining eviction sets by directly modifying bits 6–11 of the addresses within each initial set. That is, let (v_1, v_2) be a cache set such that $v_2 \in \{0{,}1\}^6$, $v_2 \neq 0$, the eviction set $S_{(v_1,v_2)}$ corresponding to (v_1, v_2) can be constructed by modifying each of the addresses in $S_{v_1,000000}$ to have bits 6–11 to be equal to v_2. This results in an eviction set for each of the cache sets.

Handling Additional Noise. We find that even when handling the above-described noise from the TLB, significant noise remains in our measurements. We speculate that this is the result of system activity and the fact that the cache footprint of our algorithm is much larger than the footprint of previous works. To alleviate the effects of the noise, we check that the size of the eviction set produced the contract phase matches the number of ways in the cache. If the size of the eviction set is too large, we attempt to repeat the contract phase several times. If it is too small, or if it is too large after several contractions, we restart the entire attempt.

PNaCl Implementation. In order to implement the above approach we require the following capabilities: First and foremost in order to be able to distinguish between slow memory accesses (corresponding to cache misses) and fast memory accesses (corresponding to cache hits) the attack code must gain accesses to a timing source of sufficient resolution. Luckily, PNaCl provides a `clock_gettime()` function which is when instantiated to the correct timing source (`clock_realtime`) is capable of tracking time with nanosecond accuracy. Next, in order to construct the eviction sets in PNaCl's execution environment we allocate a sufficiently large buffer (approximately 4 times larger than the size of the LLC). Using this buffer and the time source described above, we then performed the phases outlined above for the construction of the eviction sets.

Exprimental Results. On the Chromebook machine described in Section 1.2, constructing the initial 64 eviction sets took about 42 seconds and resulted in 63-65 initial evictions sets constructed in 95% of attempts made. Since the Chromebook's cache contains 64 possible initial eviction-sets, one duplicate eviction set was not removed during the collect phase described above. We than expanded the initial eviction sets using the above-described procedure, obtaining 4032–4160 eviction sets, covering 98%–101% of the Chromebook's cache. For the HP EliteBook 8760w laptop, constructing the eviction sets took 11 minutes and resulted in 120–130 initial eviction-sets constructed in 95% of attempts made (out of 128 possible initial eviction sets). These were again expanded to 7680–8320 eviction sets.

4 Covert Channel

We now show that two collaborating websites are able to use PNaCl code running in the target's browser in order to establish a covert channel via the last level cache. Using this channel, two collaborating websites are able to transfer information between them (such as cookies or login permissions) in a way that violates browser's policy regarding cross-domain communication. We observe that since Chrome browser executes PNaCl code without prompting any notifications or requiring user's permission, the user has virtually no indication that such an exfiltration is taking place.

We now describe our implementation of the covert channel sender and receiver using PNaCl code running in a Chrome browser.

Sender. To send a message, the sender starts by mapping the eviction sets using the process described in Section 3. Next, the sender seeks for an eviction set corresponding to a quiescent cache set, which is not frequent used by system activity. This is done by going over all the eviction sets, and sampling each of them for some time period. Each acquired trace, corresponding to some cache set, contains a timed series where each element is the number of cache-misses occurred during the time period between two consecutive samples. Going over the trace for each cache set, the sender locates a trace containing a relatively low number of cache misses which indicates a quite cache set suitable for cover channel transmission. Once such a cache set found, the sender begins to transmit the message $M = m_1, \cdots, m_n$ as follows. In case $m_i = 0$ the sender accesses the memory blocks inside the chosen eviction set for a predefined period of time t_0. Otherwise, if $m_i = 1$ the sender accesses the memory blocks inside the chosen eviction for a predefined period of time t_1. Finally, to separate between the bits, after each of them the sender waits another period of time t_{wait} before transmitting the next bit.

Receiver. To receive a message the receiver starts by mapping the eviction sets using the process in Section 3. After acquiring the eviction sets, a naive receiver will try to monitor a priorly agreed eviction set in an attempt to measure the sender's cache activity. Unfortunately, since the receiver and sender run in a different browser tabs (and therefore in different processes), the receiver's memory space has a different mapping between the virtual and the physical memory compared to the sender's mapping. Hence, eviction sets mapping procedure yields an independent mapping between the and sender and receiver.

In order to overcome this issue, the sender first transmits a pre-determined message for a duration t_{sync}, while the receiver samples all the available eviction sets looking cache-miss patterns of length t_0 and t_1 separated by cache-hits of length t_{wait} which correspond to the pre-determined message. Once a common eviction set was found, as long as the sender and receiver preserve their mapping of the eviction sets (i.e. the browser tabs were not closed), the receiver can sample only the right eviction set thus increasing the channel's capacity. The receiver then goes over the acquired trace for the common eviction set and decodes the transmitted message by translating the sequences of cache-misses of length t_0 and t_1 to their corresponding message bits.

Experimental Evaluation. We empirically set $t_0 = t_{wait} = 30\text{ms}$, $t_1 = 60\text{ms}$ and, $t_{sync} = 1.5\text{sec}$ which seem to produce the best results in terms of channel capacity. Using these parameters, after the receiver finds the common eviction set chosen the sender (i.e., after 1.5sec) we achieve a transmission rate of 13.3 kilobits per second. Figure 3 portrays the cache accesses detected by the receiver. The sequence of received bits can be clearly observed.

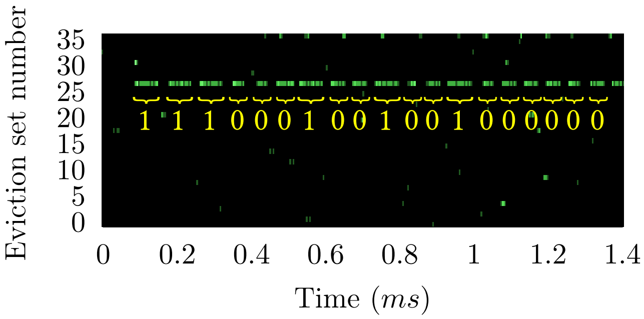


Figure 3: Cache accesses as detected by the receiver during message transferring. The intensity of the green color represents the number of cache misses (estimated by the receiver's measurements).The transmitted message can be clearly seen in the 27-th eviction set.

5 Attacked Portable Cryptographic Libraries

In this section we present End-to-End's and OpenPGP.js's ElGamal implementations as well as Elliptic's implementation of Curve25519-based Elliptic Curve Diffie-Hellman.

5.1 ElGamal Implementation

ElGamal. ElGamal [19] is a public-key crptosystem based on hardness of computing discrete logarithms. To generate a key, Alice chooses a large prime p, a generator $g \in \mathbb{Z}_p^*$ and a private key $x \in \{1, ..., p-2\}$. The public key is the triple (p, g, y) where $y = g^x \bmod p$. To encrypt a message $m \in \mathbb{Z}_p^*$, Bob chooses a random nonce $k \in \{1, ..., p-2\}$ and computes the encrypted message $c = (c_1, c_2)$, where $c_1 = g^k \bmod p, c_2 = m \cdot y^k \bmod p$. To decrypt a ciphertext (c_1, c_2), Alice computes the shared secret $s = c_1^x \bmod p$ and then recovers the message by computing $m' = c_2 \cdot s^{-1} \bmod p$. We observe that Alice can compute $s^{-1} = c_1^{p-x-1} \bmod p$ directly, avoiding the inversion.

Choosing The Secret Exponent. The performance of ElGamal decryption can be improved by using a short private key [55]. This does not affect the security of the scheme as long as brute forcing the secret key is harder than solving the associated discrete logarithm problem. A common way of choosing key size is consulting the so called "Wiener table", which contains recommended secret exponent sizes for popular sizes of public primes. For keys using 3072-bit moduli, Wiener's table recommends using an exponent of size 269 bits. Below, we utilize short keys in conjunction with an implementation choice made by End-to-End to accelerate the offline processing time required for our key extraction attack.

End-to-End's ElGamal Implementation. To compute the modular exponentiation required for decrypting messages, End-to-End uses a variant of the fixed-window (m-ary) modular exponentiation algorithm. (See [50, Algorithm 14.109].) The algorithm divides the exponent to groups of m bits, called *digits*. It processes the digits from the most significant to the least significant, performing m squaring operations and a single multiplication for each digit. For the multiplication, the current exponent digit indexes a table t of precomputed multipliers. See pseudocode in Algorithm 1 for $m = 4$.

End-to-End ElGamal Key Generation. End-to-End does not use the browser in order to generate secret keys. Instead it requires the user to generate the key outside the browser environment and subsequently import it. In all the experiments described in this paper, we used GnuPG [1] in order to randomly generate 3072-bit ElGamal keys.

Algorithm 1 End-to-End's Modular Exponentiation (simplified).

```
Input: Three integers $c$, $x$ and $p$ where the $x = \sum_{i=0}^{n-1} x_i 2^i$ and $n$ is a multiple of 8.
Output: $s = c^x \bmod p$.
procedure MODULAR_EXPONENTIATION($c, x, p$)
    $t[0] \leftarrow 1$
    for $u \leftarrow 1$ to 15 do
        $t[u] \leftarrow c \cdot t[u-1] \bmod p$
    $s \leftarrow c$, $e \leftarrow n-1$
    while $e \geq 0$ do
        for $i \leftarrow 1$ to 4 do
            $s \leftarrow s \cdot s \bmod p$
        $u \leftarrow x_e \cdots x_{e-3}$
        $s \leftarrow t[u] \cdot s \bmod p$
        $e \leftarrow e - 4$
    return $s$
end procedure
```

OpenPGP.js's ElGamal Implementation. OpenPGP.js implements ElGamal decryption using a different modular exponentiation algorithm. More specifically, OpenPGP.js uses a more performant variant of the fixed window exponentiation algorithm, called sliding-window exponentiation. (See [50, Algorithm 14.85].) Unlike the fixed window version (Algorithm 1) which uses windows of fixed length, the sliding window algorithm divides the exponent x into variable-length windows. Each window is either an arbitrary-length sequence of 0-bits or a sequence of at most m bits starting and ending with a bit of 1. Similarly to fixed window, the sliding window algorithm also use indexes a table of precomputed multipliers, on every multiplication operation. Concretely, the windows are processed from the most-significant-bit to the lease-significant-bit. Window values that start and end with 1 are processed like in the fixed window algorithm, by performing squaring operations according to the length of the window and multiplying by the precomputed value from the table, using the window-value as the table index. However, window values which are sequence of 0-bits are processed differently, by simply performing sufficiently many squaring operations. See Algorithm 2 for pseudo-code. Thus, the sliding-window algorithm leaks the location and length of zero sequences, and has been proven less resistant to side-channel attacks [46].

5.2 Elliptic's ECDH Implementation

ECDH. Elliptic curve Diffie Hellman (ECDH) is a variant of the Diffie-Hellman key exchange protocol [16] performed over suitable elliptic curves. Given a curve over a finite field $\mathbb{F}$ and a generator point $G \in (\mathbb{F} \times \mathbb{F})$, in order to generate a key Alice chooses a random scalar k as a private key and computed the public key by $[k]G$ (here and onward, we use additive group notation with and $[k]G$ denoting scalar-by-point multiplication of k and G). In order to compute the shared secret, Bob sends his public key $G' = [k']G$ to Alice (where k' is Bob's secret key). Alice and Bob then recover the shared secret by computing $[k]G'$ and $[k']G$, respectively. Notice that $[k]G' = [k]([k']G) = [k']([k]G) = [k']G$.

Curve25519. Curve25519 is an elliptic curve introduced by [8] and standardized by RFC 7748 [43]. Curve25519 was specifically designed to increase resistance to side channel attacks as well as other common implementation issues.

Algorithm 2 Modular exponentiation in OpenPGP.JS (simplified).

Input: Three integers c, x and p where the $x = \sum_{i=0}^{n-1} x_i 2^i$.

Output: $s = c^x \bmod p$.

```
procedure MODULAR_EXPONENTIATION($(c, x, p)$)
    $t[1] \leftarrow c$
    for $u \leftarrow 1$ to $2^{m-1} - 1$ do
        $t[2 \cdot u + 1] \leftarrow c \cdot (t[2 \cdot u - 1])^2 \bmod p$
    $s \leftarrow 0,\ e \leftarrow n - 1$
    while $e \geq 0$ do
        $n \leftarrow m$
        $u \leftarrow x_e \cdots x_{e-m}$
        while $iu_1 \neq 0$ do
            $n \leftarrow n - 1, u \leftarrow$ RIGHT_SHIFT$(u,1)$
        for $i \leftarrow 1$ to $n$ do
            $s \leftarrow s \cdot s \bmod p$
        $s \leftarrow t[u] \cdot s \bmod p$
        $e \leftarrow e - n$
        while $ix_e = 0$ do
            $s \leftarrow s \cdot s \bmod p$
    return $s$
end procedure
```

Scalar-By-Point Multiplication. In order to increase side channel resistance, implementations of Curve25519-based ECDH often use the Montgomery ladder [51] to perform the scalar-by-point multiplication operation. See Algorithm 3. Notice that algorithm performs the same number and order of addition and double operations, regardless of the value of k_i, making it more side channel resistant compared to other scalar-by-point multiplication algorithms [39, 54].

6 Attacking End-to-End

In this section we describe our attack on the End-to-End implementation of ElGamal encryption. For each multiplication operation performed in Line 10 of Algorithm 1, we aim to find the index u used in order to obtain one of the multiplication's operands. Key extraction is possible using this information since the values of u directly reveal bits of the secret key.

Our attack largely follows the technique of Liu et al. [46] and consists of two phases. In the *online phase* we collect a large number of memory access traces, with the aim of capturing enough samples of accesses to memory locations that store the table of pre-computed multipliers. In the *offline phase* we analyse the collected traces to identify the traces that correspond to memory locations that store pre-computed multipliers. From these traces, we recover the sequence of the indexes u used in order to obtain the operands for the multiplications performed by Algorithm 1. As described above, this information reveals the secret key.

6.1 Deployment

We demonstrated our attack on the Chromebook described in Section 1.2. Our attack scenario is based on running cache-monitoring PNaCl code inside the target's browser. We now describe a

Algorithm 3 Elliptic's Point Multiplication (simplified).

```
Input: A scalar $k$ and a point $P$ where the $k = \sum_{i=0}^{n-1} k_i 2^i$.
Output: $b = [k]P$.
procedure SCALAR_BY_POINT_MULTIPLICATION($k, P$)
    $a \leftarrow P, b \leftarrow \mathcal{O}$                ▷ $\mathcal{O}$ is the point of infinity
    for $i \leftarrow n$ to $1$ do
        if $k_i = 0$ then
            $a \leftarrow a$.ADD($b$)                ▷ $a + b$
            $b \leftarrow b$.DOUBLE()                ▷ $[2]b$
        else
            $b \leftarrow a$.ADD($b$)                ▷ $a + b$
            $a \leftarrow a$.DOUBLE()                ▷ $[2]a$
    return $b$
end procedure
```

specific attack scenario which does not require the user to install any malicious application or even actively browse to the attacker's website.

Pop-Under Advertisement. Pop-Under advertisement is a common technique to circumvent pop-up protection used in modern web browsers. Once the user clicks anywhere inside the web page, a new browser tab containing the requested web page is shown with while the previous tab (which is now hidden) is redirected to an advertisement loaded from the attacker's website.

Attack Scenario. We created an advertisement leading to a web page containing our PNaCl attack code and submitted it to a web advertisement service. When the targeted user accessed a third party web page and clicked anywhere in it, our advertisement was displayed in a background browser tab and started monitoring the cache access patterns on the target machine. Concurrently, a third tab was performing End-to-End ElGamal decryption operations. As mentioned in Section 1.1, the website the attack was launched from is not directly controlled by the attacker and is compromised indirectly by agreeing to present advertisements from a third party advertisement service. Thus, in this scenario there is no need for the user to directly access the attacker's website. See Figure 1.

6.2 The Online Phase

The main aim of the online phase is to collect traces of victim access to the pre-computed multipliers. These traces are later processed to extract the key. The main challenge that the attacker needs to overcome is that the locations of the multipliers in memory changes with each invocation of the decryption. Consequently, the attacker does not know which memory address to monitor in order to capture access to the multipliers.

Following Liu et al. [46], we overcome this challenge using statistical capturing. If we select a random cache set, there is a non-zero probability that one of the multipliers is allocated in a memory location that maps to the selected set. Consequently, by collecting a large number of traces from random cache sets we are likely to collect multiple traces of the accesses to each multiplier.

More specifically, we select 8 random cache sets and monitor them in parallel, performing a Prime+Probe cycle on each of the cache sets once every 31.5μs for a period of 5s. Because the decryption operation lasts 1.58s, a sampling period of 5s period allows enough time for most decryptions that start during the sampling period to complete within the period. We repeat this process sequentially for about 74 minutes, acquiring 7100 traces. Because each multiplier occupies 6 cache

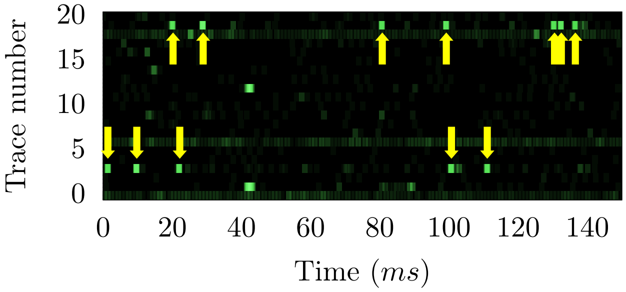


Figure 4: Cache accesses as detected by the attacker during ElGamal decryption by End-to-End. Intensity represents the number of cache misses. Notice traces 3 and 19, which contain cache misses observed by the attacker during the multiplication operations in Line 10 of Algorithm 1.

sets, and since the cache has 4096 different sets, we expect to find on average $7100 \cdot 6/4096 \approx 10.4$ traces of each precomputed multiplier in the traces we acquire.

After the traces are collected, they are sent to a server for offline processing. A sample of collected traces is shown in Figure 4.

6.3 The Offline Phase

The aim of the offline phase is to process the traces collected in the online phase and to recover the key. To achieve that, the attacker first needs to identify which of the traces corresponds to memory access to precomputed multipliers. The attacker then has to match the traces against the algorithm and determine constraints on the possible sequence of multiplications based on the traces. The last step in the process is to combine the constraints and recover the order of accesses to the multipliers. We now describe these steps in greater details.

6.3.1 Identifying Traces of Multipliers

Like Liu et al. [46], we identify the traces of accesses to multipliers by looking for temporal patterns in the data. Recall that Algorithm 1 performs one multiplication for every four exponent bits. Hence, for the 3072 bit exponent we use, the algorithm performs 768 multiplications. Next, since Algorithm 1 uses 16 precomputed multipliers, each of these is accessed 48 times during a decryption, on average. Note, however, that because the exponent is random, the number of times each multiplier is used may vary and usually is slightly higher or slightly lower than 48.

In End-to-End's ElGamal implementation running on the Chromebook machine, a multiplication operation takes about 410μs. In our attack we perform a Prime+Probe cycle once every 31.5μs. Hence, we expect each multiplication to span over about 13 samples.

Combining the information above, we scan the traces, counting the number of sequences that consist of 13 consecutive samples that indicate access to the monitored cache set. If the number of such sequences is below 30, we assume that the monitored cache set is unlikely to correspond to a precomputed multiplier. After rejecting all these we are left with 750 candidates that may contain information about multiplier access in Algorithm 1. We can see examples of two such traces in Figure 4.

Observing Figure 4 we see that many traces, including those that correspond to a precomputed multiplier, contain noise in the form of sporadic memory accesses, lasting well below 13 samples. To remove this noise, we scan each trace sequentially, looking for a memory access. We then look at the window of 13 samples starting at the access and check how many of the samples indicate memory accesses. If nine or more of the samples in the window indicate memory access, we treat all of the samples in the window as access and skip the window. If, however, less than nine samples in the window indicate access, we ignore the first access, treating it as noise. This results in "denoised" traces that consist of sequences of 13 or more consecutive accesses, separated by long stretches of no-access.

6.3.2 Generating Annotated Multiplication Sequences

The next task is to recover, for each u in the range of indexes of the table t of precomputed multipliers, an approximation of the sequence of the multiplications performed by Algorithm 1, where a multiplication is marked if u was the index used to obtain one of the operands in Line 10. We call such sequences *annotated multiplication sequences.*

We first generate annotated multiplication sequences from the captured traces. Recall that each multiplication operation spans over about 13 samples in the trace. We, therefore, convert sequences of samples into sequences of multiplications by simply dividing the length of the sequence of samples by 13, rounding to the nearest integer. Sequences of samples that indicate memory access are converted into marked multiplications and sequences of samples with no memory access are converted to unmarked multiplications.

Next, in order to further reduce measurement noise in the collected sequences, we rely on the observation (due to Liu et al. [46]) that sequences corresponding to the same multiplier are similar. We first cluster the annotated multiplication sequences we collected based on their similarity. We consider two sequences to be in the same cluster if the edit distance [44] between them is less then some threshold. For our purposes, we found that a threshold of 20 produces good clustering results. We then merge each cluster into a single representative sequence that is similar to all of the sequences in the cluster. For that, we use Clustal Omega [62], a solver for the multiple sequence alignment problem [42], originally designed for aligning biological sequences such as protein chains or DNA.

Finally, we adjust the resulting representative annotated multiplication sequences, so that they are consistent with the expected execution of Algorithm 1. Observing Algorithm 1 we note that it consists of two phases. In the first phase, which consists of 16 multiplications, the algorithm constructs the lookup table of the precomputesd multipliers. In the second phase, during the actual exponentiation, the algorithm performs sequences of four multiplications of s by itself (Line 8), followed by a multiplication with one of the precomputed multipliers, that is, every fifth multiplication accesses a precomputed multiplier. We therefore pad or truncate the gaps between marked multiplications in the represenatative annotated multiplication sequences to ensure that marked multiplications are at positions that are congruent modulo 5.

At this stage we have 15 annotated multiplication sequences that are expected to be similar to the sequence of multiplication that access each of the multipliers. However, the information is still incomplete. In particular, we do not know the index value u that corresponds to each of the sequences. Had we known how Algorithm 1 accesses the multiplier in the first 16 multiplications, we could have recovered the multiplier. However, because we can only identify the first marked multiplication, we do not know how many unmarked multiplications precede it. We do know that we do not have the sequence of the 0th multiplier because our algorithm assumes that multiplications take 13 samples, whereas multiplication by the 0th multiplier (i,e, $u = 0$), which is always 1, are much faster. Finally, the sequences we have still contain some errors, and while the number of errors is small (less than one error per sequence, on average), these need to be resolved.

6.3.3 Key Extraction

The last step in the offline phase is to find value of the table index u that corresponds to each of the annotated multiplication sequences and correct the remaining errors. We suggest two methods for assigning multipliers to sequences. The straightforward approach is to brute-force the assignment by trying all combinations. A more efficient, albeit less general is to exploit an implementation issue in End-to-End's ElGamal. We now describe these two approaches.

Brute Force. For the brute-force approach, we try every possible assignment of the values of the table index u to annotated multiplication sequences. Observing Algorithm 1, we see that during the generation of the lookup table, the uth multiplier is only accessed in the uth and the $(u+1)$st multiplications. Hence assigning the table index to an annotated multiplication sequence determines the number of preceding unmarked multiplication for the sequence. Knowing the number of preceding multiplications allows us to align all of the sequences and, from the aligned sequences, determine the table index used in each multiplication. We note that in Algorithm 1 only one table index is used in each multiplication. Consequently, a large number of collisions, where two or more sequences have a marked multiplication at the same position, indicates that the assignment is unlikely to be correct and thus can be ignored. For assignments that result in a small number of collisions, we try multiple combinations of slight variations in the positions of the colliding multiplications and test if they yield the correct key. Using this approach, we brute forced a key on an Amazon m4.16xlarge instance within about 76 minutes at a cost of \$4.10.

Expliting End-to-End's ElGamal Implementation. As presented in Section 5.1, End-to-End performs ElGamal decryption by computing $c_1^{p-x-1} \bmod p$. Next, we observe that ElGamal private keys generated using Wiener's table are significantly shorter than the public prime p. More specifically, for 3072-bit moduli, Wiener's table recommends using a private key whose length is less than 450 bits. Thus, with high probability, the 2500 most significant bits (MSBs) of $p-x-1$ and of p are the same. Since the modulus p is part of the ElGamal public key, the attacker knows the 2500 MSBs of the secret exponent $p-x-1$. We can, therefore, use the 2500 MSBs of p to compute ground truth for the first 85% of each annotated multiplication sequence. We then match the annotated multiplication sequences obtained in Section 6.3.2 with the ground truth calculated from the MSBs of p. This allows us to recover the assignment of multipliers to sequences and to find the key within a few seconds.

We note that neither computing $c_1^{p-x-1} \bmod p$ directly (as opposed to $(c_1^x)^{-1} \bmod p$) nor using Wiener's table to generate short ElGamal private keys pose major side channel weaknesses. However, the combinations of these two optimizations improves the performance of our attack.

6.4 Overall Attack Performance

The attack described above consists of an online phase, which monitors the target's cache for 74 minutes. Offline processing of the data, including denoising, clustering, merging and key recovery, takes at most 90 minutes on an Amazon EC2 m4.16xlarge instance and costs less than \$5.50. We tried our attack on several randomly generated ElGamal keys with 3072-bit public primes, successfully extracting the entire secret exponent in every trial.

6.5 Attacking RSA

RSA [60] is a commonly used public-key cryptosystem based on the hardness of factoring. Key generation is done by selecting two large prime numbers p, q, a (fixed) public exponent e and a secret exponent d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$. The public key is (n, e) where $n = pq$ and the private key is (d, p, q). Encryption of a message m is done by computing $c = m^e \bmod n$ while decryption of a ciphertext c is done by computing $m = c^d \bmod n$. In order to increase side channel resistance against chosen ciphertext attacks [10, 25], End-to-End also implements RSA blinding where m is computed by $m = (r^e \cdot c)^d \bmod n$ where r is generated at random. This has the effect of stopping adversarialy controlled ciphertexts from being used as input to the exponentiation routine.

Attacking End-to-End's RSA Implementation. Our attack utilizes exponent-dependent memory accesses during End-to-End's modular exponentiation routine in order to extract the exponent, which is part of the secret key both for ElGamal and for RSA. Thus, since modular exponentiation is used during decryption of both ElGamal and RSA ciphertext, our attack is applicable for extracting RSA keys in addition to extracting ElGamal keys. We note that since our attack only utilizes exponent-dependent memory access during End-to-End's modular exponentiation routine in order to achieve key extraction, it naturally bypasses End-to-End's ciphertext blinding countermeasure.

6.6 Attacking Other Devices

Our attack can be executed on any Intel machine capable of running PNaCl code in a Chrome browser, and thus is applicable to not only Chromebooks but also to full-fledged laptop and desktop computers. In particular, we performed the attack on an HP EliteBook 8760w laptop running Chrome version 57.0.2987.110 (which is the latest version at the time of writing this paper), Kubuntu 14.04 and equipped with an Intel i7-2820QM processor (see full description in Section 1.2). Due to the different processor used, and in particular the higher speed and the larger cache, we need to collect 9600 traces, resulting in an online phase that lasts 90 minutes. Offline analysis of the traces achieves key extraction within 90 minutes on an Amazon EC2 m4.16xlarge instance, at a cost of $5.50.

7 Attacking OpenPGP.js

The side channel vulnerabilities present in Google's End-to-End exponentiation code are also present in other JavaScript cryptographic libraries. This makes our attack techniques applicable to these libraries, potentially resulting in key extraction attacks. In fact, some libraries use the sliding window exponentiation method which achieves better performance compared to the fixed window method. While this method also produces similar leakage as the fixed window method, it also leaks additional information about consecutive runs of zeros in the secret exponent.

In this section we demonstrate this by attacking OpenPGP.js, which is another popular JavaScript cryptographic library.

Leakage Source. Similarly to Section 6, in order to extract the secret exponent used in the sliding window exponentiation performed during OpenPGP.js's ElGamal decryption operation, we need to identify the value u used as the table index in each multiplication operation performed by OpenPGP.js's variant to Algorithm 1.

Monitoring OpenPGP.js's Side Channel Leakage. To measure the leakage describe above we used an analogous setup to the one used in Section 6. Using the Chromebook, we opened two browser tabs with one tab running our PNaCl attack code while the other tab was performing ElGamal decryption operations using the OpenPGP.js, each lasting 0.25s. Using the methodology of Section 6, we monitored 64 random cache sets, performing a Prime+Probe cycle on each of the cache sets once every 20μs for a period of 0.62s. After acquiring 640 traces we filtered them using the techniques described in Section 6.

Leakage Analysis. Figure 5 shows the side channel leakage from one OpenPGP.js ElGamal decryption operation. Notice the leakage present in traces 11 and 19, where the cache access patterns observed by the attacker reveal when a specific table index u is used during some multiplication

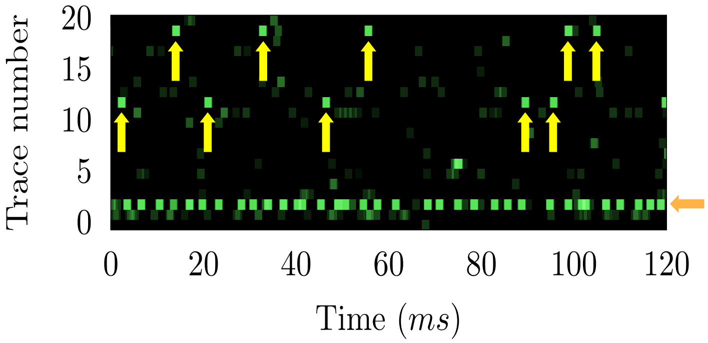


Figure 5: Cache accesses as detected by the attacker during ElGamal decryption by OpenPGP.JS. The intensity of the green color represents the number of cache misses. Traces 11 and 19 contains cache misses observed by the attacker during the multiplication operations of the modular exponentiation. Trace 2 is the result of monitoring (via code-cache misses) the execution of the modular multiplication code during an ElGamal decryption operation. Notice the different intervals between the multiplications leak the location of sequences of zero bits.

operations performed by OpenPGP.js's variant of Algorithm 1. As demonstrated in Section 6, monitoring this leakage for all possible values of u completely reveal the secret exponent. Finally, while the fixed-window exponentiation algorithm performs a constant pattern of squaring and multiplying operations regardless of the value of the secret exponent, the squaring operations performed by the sliding-window algorithm used by OpenPGP.js reveal long sequences of zero exponent bits. We demonstrate this additional source leakage in Trace 2 of Figure 5 by monitoring the executions of the modular multiplication code executed by OpenPGP.js during an ElGamal decryption operation.

8 Attacking Elliptic

The attacks presented in Sections 6 and 7 utilized key-dependent memory accesses in order to recover the secret exponent. In this section we claim that even highly regular algorithms, which do not perform key-dependent operations or memory accesses, can produce exploitable side channel leakage when implemented in high-level programming languages such as JavaScript. We empirically demonstrate this claim using Elliptic's Curve25519-based ECDH implementation, which uses the Montgomery ladder method as the underlying scalar-by-point multiplication routine.

Inapplicability of Data Cache Leakage. The Montgomery ladder scalar-by-point multiplication routine attempts to achieve side channel resistance by being highly regular. Each iteration of the main loop of Algorithm 3 accesses both of the internal variables (a and b) and performs a single elliptic curve add operation followed by a single elliptic curve double operation. In particular, both operations are performed, in the same order, irrespective of the value of the current secret key bit (k_i). Thus, the Montgomery powering ladder avoids many side-channel attacks, such as leaking the secret key via key-dependent sequences of double and add operations, or performing key-dependent memory accesses to a table of precomputed values. As we have empirically validated, Elliptic's implementation of the Montgomery ladder, running on Chrome, is almost constant time, without key-dependent timing deviations.

While Algorithm 3 does leak the secret key via memory accesses performed to the operand of the elliptic curve double operation (Lines 6 and 9) as well as the memory accesses to the result of the elliptic curve add operation (Lines 5 and 8), this leakage is hard to exploit due to JavaScript's memory allocation mechanism. Concretely, since each iteration of the main loop of Algorithm 3 always updates both variables, Elliptic's implementation always allocates new objects for the updated values, at different and changing memory addresses. As we empirically verified, the addresses of the objects pointed by a and b change with each iteration of the main loop of Algorithm 3, without any obvious patterns. This makes monitoring memory accesses to a and b difficult, since the attacker has to predict and subsequently monitor a different cache set at every iteration of the main loop of Algorithm 3.

While the memory re-allocation countermeasure was probably unintentional, this countermeasure combined with the inherent regularity of the Montgomery ladder scalar by point multiplication routine prevent the use of the data cache as a source of side channel leakage.

Finding a Leakage Source. We choose, instead, to conduct a code-cache side-channel attack. In this approachwe identify a key-dependent change in the target's control flow. During the ECDH operation, we monitor the code cache accesses via PNaCl, deduce control flow changes, and from these, recover the key.

An immediate candidate for such key-dependent control flow would be the if-else statement in Line 4 of Algorithm 3. However, distinguishing between different cases of the if-else statement in Line 4 appears to be difficult, since both case are very similar, call the same functions in the same order, have the same length and are relatively small (each consisting of only two code lines).

While a high-level examination of Algorithm 3 does not reveal any additional key-dependent control flow, we do observe that Algorithm 3 invokes the doubleoperation in Line 6 on variable b, while in Line 9 it is invoked on object a. While in a low-level programing language the execution of different code paths is usually explicit, in a high-level language such as JavaScript, the compiler/interpreter is at liberty to select different execution paths for performing identical operations on different data. Empirically, this indeed occurs here. We were able to empirically distinguish, using code cache leakage, between the double operation performed in Line 6 (on variable b) from the double operation in Line 9 (performed on a) — thus attaining key extraction.

Monitoring Elliptic's Side Channel Leakage. Following the experimental approach in Section 6 we opened two tabs in the Chromebook's browser, one tab running our PNaCl attack code while the other tab was running Elliptic's implementation of Curve25519-based ECDH, with each key exchange lasting 4.5ms. Next, we sampled each of the 4096 eviction sets, performing Prime+Probe cycle every 3μs for periods of 35ms, totally sampling less than 3 minutes.

Leakage Analysis. Out of the acquired 4096 traces, we identified 5 as containing the side channel leakage described above. Figure 6 shows some out of the acquired traces, Trace 3 (top) contains the information regarding the secret key. As can be seen from the bottom part of Figure 6, showing only Trace 3, a sequence of 10μs of cache-misses cache-misses followed by 5μs of cache-hits in the monitored set corresponds to a bit of 1, while 20μs of cache-hits corresponds to 0 bit. With this observation, we automated the extraction of keys from the aforementioned traces, yielding correct extraction of up to 236 (out of 252) bits of the secret key from *individual* traces.

9 Conclusions

In this paper we present a method for implementing an LLC-based Prime+Probe attack on an multiple cryptographic implementations ranging from ElGamal to state-of-the-art Curve25519-based ECDH using from portable code executing inside a sandboxed web browser. We successfully deployed the attack using a commercial ad placement service that triggers the attack code from third-party web sites. When users navigate to these web sites, the attack code is executed in the

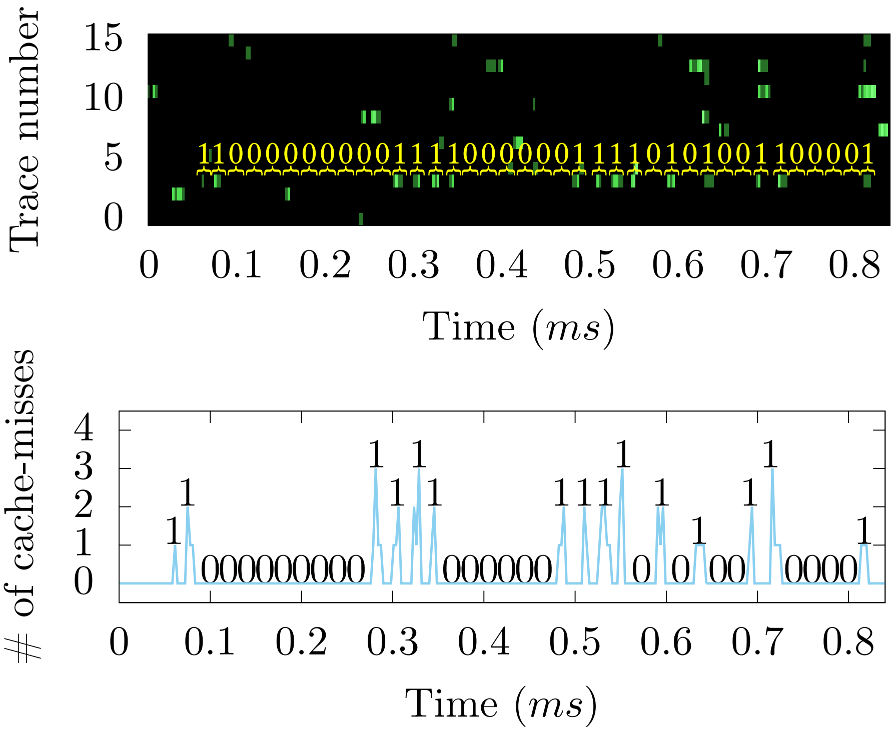


Figure 6: Cache accesses as detected by the attacker during ECDH key exchange over Curve25519 by elliptic. Trace 3 (top) contains cache misses observed by the attacker during the scalar-by-point multiplication. At the bottom, which only shows Trace 3, it can clearly be noticed that the cache-misses corresponds to key bits of 1, while sequence without cache-misses of approximately 20μs corresponds to bits of 0.

users' browsers and starts monitoring the memory access patterns. To our knowledge, this is the first demonstration of a drive-by cache attack, and the first portable cryptographic side channel attack.

Unlike prior works, our attack target is implemented using a portable, non-native code. Yet, even without the knowledge of the target's memory layout, the attack successfully extracts the target's ElGamal and ECDH keys. Finally, we show that in spite of their secure design and the limited control users have, Chromebooks are vulnerable to cache based attacks.

Countermeasures. To write a side-channel resistant code, one has to use a constant-time implementation that does not perform any secret dependent branches and memory accesses. Techniques for developing constant-time implementations have been explored, e.g. in Bernstein et al. [9]. These approaches can be tricky to get right [70], and even when correctly implemented, the implementation is fragile and can fail to achieve protection when used on different hardware [14] or with different compilers [40]. Using constant-time techniques in JIT-compiled environments is an unexplored area that we leave for future work. Until then, the only secure way to perform cryptographic operations in JavaScript is to delegate them to the browser so that they can be executed using a native code implementation. Indeed, modern Browsers are equipped with WebCrypto API [66] that allows JavaScript to execute cryptographic operations using native, carefully designed and deterministically compiled implementations, such as BoringSSL [28] used by Google Chrome.

Limitations. The measurement code is currently implemented using PNaCl, and is thus restricted to browsers that support PNaCl (i.e., Chrome and Chromium). Since we do not use any special feature of PNaCl other than its efficiency and low-level memory model, we expect that the attack can be readily ported to competing low-level portable code technologies like `asm.js`/WebAssembly, thus extending the attack to all popular web browsers.

The process of constructing the eviction sets as described in Section 3 highly depends on the cache structure and eviction policy. More specifically our attack assumes an inclusive LLC and an LRU or similar eviction policy. While both assumptions hold for Intel CPUs, other vendors offer different implementations.

While the attack presented in Section 8 requires only a few minutes of sampling time, the attack in Section 6 requires over an hour of sampling time. Moreover, both attacks require the target to perform thousands of secret key operations before the secret key could be extracted. With such long attack times one may question the practicality of the attack. We accept that the attack as published should not raise immediate security concerns for end users. Yet, the security community should be aware of the possibility and consider the threat of side channel attacks from sandboxed mobile code because, as we demonstrate, sandboxing does not prevent such attacks.

We leave addressing these challenges as future work.

Acknowledgments

This research was partially inspired by unpublished work on portable cache attacks done jointly with Ethan Heilman, Perry Hung, Taesoo Kim and Andrew Meyer.

Daniel Genkin, Lev Pachmanov and Eran Tromer are members of the Check Point Institute for Information Security. Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania.

This work was supported by the Australian Department of Education and Training through an Endeavour Research Fellowship; by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by the Check Point Institute for Information Security; by the Defense Advanced Research Project Agency (DARPA) and Army Research Office (ARO) under Contract #W911NF-15-C-0236; by the Israeli Ministry of Science and Technology; by the Israeli Centers of Research Excellence I-CORE program (center 4/11); by the Leona M. & Harry B. Helmsley Charitable Trust; by NSF awards #CNS-1445424 and #CCF-1423306; by the 2017-2018 Rothschild Postdoctoral Fellowship; by the Warren Center for Network and Data Sciences; by the financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology; and by the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of ARO, DARPA, NSF, the U.S. Government or other sponsors.

References

[1] GNU Privacy Guard. https://www.gnupg.org

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *CHES*. 110–124.

[3] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *IMA Int. Conf.* 185–203.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *CT-RSA*. 225–242.

[5] Onur Acıiçmez and Werner Schindler. 2008. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *CT-RSA*. 256–273.

[6] Cedric Alfonsi, Remy Bertot, Kevin Muller, and Diego Lendoiro. 2016. passbolt: Free, open source, self-hosted, extensible, OpenPGP based password manager. https://www.passbolt.com/. (2016).

[7] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005). http://cr.yp.to/papers.html#cachetiming.

[8] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC*. 207–228.

[9] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *LATINCRYPT*. 159–176.

[10] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.

[11] Yuriy Bulygin. 2008. CPU Side-Channels vs. Virtualization Malware: the Good, the Bad or the Ugly. In *ToorCon*.

[12] Bart Butler. 2017. OpenPGP.js: OpenPGP JavaScript Implementation. https://openpgpjs.org/. (2017).

[13] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. 2007. OpenPGP Message Format. RFC 4880. (Nov. 2007).

[14] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on seL4. In *ACM SIGSAC*. 570–581.

[15] Jonathan Corbet. 2015. Pagemap: security fixes vs. ABI compatibility. https://lwn.net/Articles/642069/. (April 2015).

[16] Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Trans. Information Theory* 22, 6 (1976), 644–654.

[17] Drulac. 2017. wrapper to encrypt and sign socket.io messages. https://github.com/Drulac/socket.io-with-PGP. (2017).

[18] E2EMail Organization. 2016. E2EMail: A Gmail client that exchanges OpenPGP mail. https://github.com/e2email-org/e2email. (2016).

[19] Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 469–472.

[20] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2016. Understanding and Mitigating Covert Channels Through Branch Predictors. *TACO* 13, 1 (2016), 10:1–10:23.

[21] FlowCrypt Limited. 2017. CryptUp PGP browser extension. https://cryptup.org/. (2017).

[22] Sean Gallagher. 2013. Why the NSA loves Google's Chromebook. https://arstechnica.com/information-technology/2013/09/why-the-nsa-loves-googles-chromebook/. (Sept. 2013).

[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* - (Dec. 2016), 1–27.

[24] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2015. Stealing Keys from PCs using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *CHES*. 207–228.

[25] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *CRYPTO*. 444–461.

[26] Google Inc. 2009. Security Built-In. http://static.googleusercontent.com/media/www.google.com/en/us/intl/en/chrome/assets/business/chromebook/downloads/chromebook-security-built-in.pdf. (2009).

[27] Google Inc. 2013. Portable Native Client. https://developer.chrome.com/native-client. (2013).

[28] Google Inc. 2014. BoringSSL - The SSL library in Chrome/Chromium and Android. https://www.w3.org/TR/WebCryptoAPI/. (2014).

[29] Google Inc. 2017. End-To-End: A a crypto library to encrypt, decrypt, digital sign, and verify signed messages. https://github.com/google/end-to-end. (2017).

[30] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. *NDSS* (2017).

[31] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*. 300–321.

[32] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX*. 897–912.

[33] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *NDSS*.

[34] Mehmet Sinan İnci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*. 368–388.

[35] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive* (2015), 898.

[36] Fedor Indutny. 2017. Fast Elliptic Curve Cryptography in plain Javascript. https://github.com/indutny/elliptic. (2017).

[37] Fedor Indutny and Christopher Jeffrey. 2017. Bcoin - JavaScript Bitcoin library for node.js and browsers. http://bcoin.io/. (2017).

[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *DSD*. 629–636.

[39] Marc Joye and Sung-Ming Yen. 2002. The Montgomery Powering Ladder. In *CHES*. 291–302.

[40] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS*. 573–582.

[41] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*. 361–372.

[42] Eric S. Lander, Robert Langridge, and Damian M. Saccocio. 1991. Mapping and Interpreting Biological Information. *Commun. ACM* 34, 11 (Nov. 1991), 32–39.

[43] A. Langley, M. Hamburg, and S. Turner. 2016. Elliptic Curves for Security. RFC 7748. (2016). http://www.ietf.org/rfc/rfc7748.txt

[44] Vladimir Iosifovich Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

[45] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX*. 549–564.

[46] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*. 605–622.

[47] Christian Lundkvist. 2015. Lightweight JS Wallet for Node and the browser. https://github.com/ConsenSys/eth-lightwallet. (2015).

[48] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*. 48–65.

[49] Clémentine Maurice, Manuel Weber, Michael Schwartz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[50] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. 1996. *Handbook of Applied Cryptography* (1st ed.). CRC Press.

[51] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48, 177 (1987), 243–243.

[52] Richard Moore. 2014. Complete Ethereum wallet implementation and library in JavaScript. https://github.com/ethers-io/ethers.js. (2014).

[53] npm Inc. 2017. Package manager for Node.js. https://www.npmjs.com. (2017).

[54] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. 2000. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In *PKC*. 238–257.

[55] Paul C. van Oorschot and Michael J. Wiener. 1996. On Diffie-Hellman Key Agreement with Short Exponents. In *EUROCRYPT*. 332–343.

[56] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *ACM SIGSAC*. 1406–1418.

[57] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*. 1–20.

[58] Colin Percival. 2005. Cache missing for fun and profit. (2005). Presented at BSDCan. http://www.daemonology.net/hyperthreading-considered-harmful.

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*. 199–212.

[60] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[61] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*. 3–24.

[62] Fabian Sievers, Andreas Wilm, David Dineen, Toby J Gibson, Kevin Karplus, Weizhong Li, Rodrigo Lopez, Hamish McWilliam, Michael Remmert, Johannes Söding, and others. 2011. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular systems biology* 7, 1 (2011), 539.

[63] Natasha Singer. 2017. How Google Took Over the Classroom. https://www.nytimes.com/2017/05/13/technology/google-education-chromebooks-schools.html. (May 2017).

[64] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology* 23, 1 (2010), 37–71.

[65] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *CHES*. 62–76.

[66] World Wide Web Consortium (W3C). 2012. Web Cryptography API. https://www.w3.org/TR/WebCryptoAPI/. (2012).

[67] Yahoo! Inc. 2016. Yahoo End-To-End. https://github.com/yahoo/end-to-end. (2016).

[68] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX*. 719–732.

[69] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. IACR Cryptology ePrint Archive 2015/905. (2015).

[70] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptographic Engineering* 7, 2 (2017), 99–112.

[71] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*. 79–93.

[72] Andy Yen and Bart Butler. 2017. ProtonMail: An easy to use secure email service with built-in end-to-end encryption. https://protonmail.com/. (2017).

[73] Yours Inc. 2016. JavaScript implementation of Bitcoin. https://github.com/yoursnetwork/yours-bitcoin. (2016).

[74] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *CCS*. 305–316.