Drive-By Key-Extraction Cache Attacks from Portable Code

Daniel Genkin[1,2], Lev Pachmanov[3], Eran Tromer[3,4](✉), and Yuval Yarom[5,6]

[1] University of Pennsylvania, Philadelphia, PA, USA
danielg3@cis.upenn.edu
[2] University of Maryland, College Park, MD, USA
[3] Tel Aviv University, Tel Aviv, Israel
{levp,tromer}@tau.ac.il
[4] Columbia University, New York, NY, USA
[5] University of Adelaide, Adelaide, Australia
yval@cs.adelaide.edu.au
[6] Data61, Sydney, Australia

Abstract. We show how malicious web content can extract cryptographic secret keys from the user's computer. The attack uses portable scripting languages supported by modern browsers to induce contention for CPU cache resources, and thereby gleans information about the memory accesses of other programs running on the user's computer. We show how this side-channel attack can be realized in WebAssembly and PNaCl; how to attain fine-grained measurements; and how to extract ElGamal, ECDH and RSA decryption keys from various cryptographic libraries.

The attack does not rely on bugs in the browser's nominal sandboxing mechanisms, or on fooling users. It applies even to locked-down platforms with strong confinement mechanisms and browser-only functionality, such as Chromebook devices.

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

1 Introduction

Since their introduction [5,29,30,36], 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 execute code on the target machine. Even without special privileges, such code can contend with concurrently-executing target code for the use of low-level microarchitectural resources; and by measuring the thus-induced timing variability, an attacker can glean information from the target code. Many such resources have been analyzed and exploited, including branch predictors and arithmetic units, but contention for cache resources has been

© Springer International Publishing AG, part of Springer Nature 2018
B. Preneel and F. Vercauteren (Eds.): ACNS 2018, LNCS 10892, pp. 83–102, 2018.
https://doi.org/10.1007/978-3-319-93387-0_5

proven to be particularly devastating. Cache attacks allow fine grained monitoring of memory access patterns, and can extract cryptographic keys [5,29,30], website fingerprints [28], and keystrokes [15]; see [11] 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 [17,31,37], 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, run 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 untrusted code.

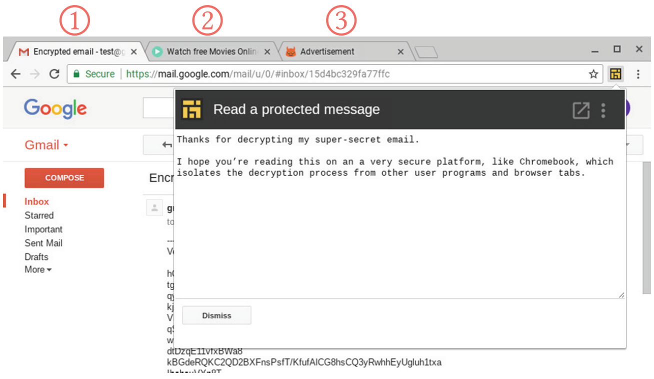
Recent works [13,28] made progress towards effective cache attacks on end-user devices, using JavaScript code running in the target's browser and without requiring native code execution. However, since JavaScript is far-removed from the native platform, the information obtained by a JavaScript attacker is severely degraded. Indeed compared to 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)?** In particular, do such attacks apply to locked-down platforms, such as Chromebook running Chrome OS, where functionality is restricted to sandboxed web browsing?

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 low-level details of the executed code—first and foremost, to make it constant-time. This picture changes when cryptographic software is deployed as portable high-level code, where the final code and memory layout are left to the whims of a just-in-time compiler. On the 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: **(b) Do portable program representations compromise the side-channel resilience of (supposedly) constant-time algorithms?**

1.1 Our Results

We answer both questions in the affirmative. (a) We present cache side-channel attacks which can be executed from a web page loaded in a sandboxed browser



```
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.
```

```
dtDzqE11vfxBWa8
kBGdeRQKC2QD2BXFnsPsfT/KfufAICG8hsCQ3yRwhhEyUgluh1txa
```

Fig. 1. Attack scenario screenshot. The targeted user opens an online streaming website in Tab 2. Clicking within this tab (e.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 (in Tab 1), the malicious advertisement in Tab 3 learns information about the user's secret key.

environment, and are capable of extracting keys from ElGamal and RSA implementations. (b) We demonstrate key extraction even from an implementation of Curve25519 ECDH, 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 do not rely on vulnerabilities in the browser's nominal isolation mechanisms (e.g., they work even if Same Origin Policy and Strict Site Isolation are perfectly enforced). Rather, they glean information from outside the browser's sandbox 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 it 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 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 ads from web ad services that support non-static ads (JavaScript, pop-under or IFRAME ads).

Concretely, we embedded the attack code in an advertisement, which we submitted to a commercial web ad service. Whenever a user navigated to a site that uses that service, and our ad was selected for display, the attack code was triggered (see Fig. 1). This code measured the memory access patterns on the user's machine, and sent it to our server for analysis. When the targeted cryptographic software happens to be repeatedly invoked using some secret key during the time when the ad was shown in some browser tab (even in the background), our code extracted the secret key in as little as 3 min. This works even across processes and browsers (e.g., JavaScript ad in Firefox attacking cryptographic code running in Chrome).

Attacking Curve25519. One of our attacks targets a JavaScript implementation of Curve25519 Elliptic Curve Diffie-Hellman (ECDH) [6]. 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 a portable code-cache side-channel attack.

Measurement Technique. We implement the cache measurement procedure using portable code running within the browser. To achieve the measurement resolution required to mount an attack on ElGamal and ECDH, we used PNaCl or WebAssembly. These are architecture-independent code representations which browsers execute in a sandbox—analogously to JavaScript, but lower-level and more efficient. PNaCl is supported by desktop versions of the Chrome and Chromium browsers since 2013, and automatically executed by the browser without user involvement. WebAssembly is the standardization of the idea behind PNaCl. It is supported by all major browsers and enabled by default since 2017.

Like JavaScript, PNaCl and WebAssembly are sandboxed, subject to Same Origin Policy, and isolated from host resources such as the filesystem and other processes. However, the portable 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 for cache side-channel attacks. Using this, and additional techniques, the portable code can execute a variant of the Prime+Probe attack of [29], to detect which memory addresses are accessed by other processes.

Compared to the two prior works on portable-code cache attacks (see Sect. 1.3), our use of a portable but low-level program representation, as opposed to JavaScript in [28], reduces measurement overheads and provides better timing sources on modern browsers; and by using a precise eviction set construction algorithm (adapting the approach of [23] to the portable setting) we moreover reduce the eviction sets' size by $\times 64$ compared to [13]. Taken together, these attain the requisite temporal resolution for several cryptanalytic attacks.

Challenges. Launching cache attacks involves numerous challenges, such as recovering the mapping between the memory and the cache, and identifying cache sets corresponding to security-critical accesses (see [23] for a detailed list). Mounting the attack from portable code introduces several additional challenges:

1. *Emulated environment:* Both PNaCl and WebAssembly modules run 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) portable architectures incur an overhead compared to native execution, reducing the measurements' temporal resolution.
3. *Inability to flush the CPU pipeline and cache:* PNaCl and WebAssembly do not support instructions for flushing the CPU pipeline, the cache or avoiding out-of-order execution, as needed by many native-code attacks.
4. *Inaccurate time source:* Architecture independence forces PNaCl applications to only use generic interfaces or indirect measurements to measure time. WebAssembly modules can interact with external APIs only using JavaScript, hence they are limited to the time sources available to JavaScript code.

   Moreover, the cryptographic software we attack is implemented in JavaScript, which introduces yet more challenges:
5. *Unpredictable memory layout:* The target's JavaScript code is compiled anew at every page load, and moreover, its memory allocations are done in an unpredictable way at every invocation.
6. *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, not unavailable here.

1.2 Targeted Hardware and Software

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

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

Elliptic. Elliptic [18] is an open-source JavaScript cryptographic library, providing efficient implementations of elliptic-curve cryptographic primitives such as Elliptic Curve Diffie-Hellman (ECDH). Elliptic is widely used (over 20M downloads), and underlies more than a hundred dependent projects including crypto-currency wallets. Elliptic supports state-of-the-art elliptic curve constructions such as Curve25519 [6], which was designed to offer increased resistance to side channel attacks. We show that while Elliptic's implementation does use the Montgomery-ladder method with apparently constant execution time, memory access leakage induced by the JavaScript routines does allow for key extraction.

Google's End-to-End Library. End-to-End is an open-source JavaScript cryptographic library developed by Google for use by websites and browser plug-ins. To facilitate email encryption and signing directly inside the user's browser, End-to-End supports the OpenPGP standard, as documented in RFC 4880. End-to-End is the cryptographic engine for many browser plugins such as E2EMail, Google encrypted email extension, and Yahoo's fork of EndToEnd.

OpenPGP.js. OpenPGP.js is a popular open-source library for browser-based cryptographic operations, and in particular encrypted email. Similarly to End-to-End, OpenPGP.js implements the OpenPGP standard and is widely deployed in web applications and browser plug-ins. These include password managers, encrypted mail clients and other applications. To create seamless user experience, some of those plug-ins (e.g., ProtonMail and CryptUp) automatically decrypt received content upon opening the received email.

1.3 Related Work

Cache Attacks. Cache attacks were introduced over a decade ago [5,29,30,36]. Initial attacks exploited the L1 and L2 data caches [29,35], however later attacks targeted other caches, such as the L1 instruction cache [1,4] the shared last level cache [16,23] and specialized caches including the branch prediction unit [2,3,10] and the return stack buffer [7]. Recent works [13,34] were able to extract information without using huge pages. See [11] for a survey.

Cache Attacks from Portable Code. The first published browser-based cache attack was shown by [28]. 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 [28] web browsers nowadays provide reduced timer precision, making the techniques of [28] inapplicable.

Recently, [13] achieved higher cache-line accuracy, and used it to derandomize the target's ASLR from within it's browser. They 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 Sect. 3).

The Rowhammer attack [20] was also implemented in JavaScript by [14].

Speculative Execution Attacks. Going beyond cryptographic keys, cache attacks can be also leveraged to read memory contents across security domains. The Meltdown [22] and Spectre [21] attacks exploit the CPU's speculative execution to let a process glean memory content to which it does not have access permissions, by accessing that memory directly (Meltdown) or by inducing the valid owner of that memory to access it within a mispredicted branch (Spectre). In both attacks, the read is invalid and the architectural state will eventually be rewound, but the carefully-crafted side effects on the cache can be observed.

These attacks rely on cache covert channels, for which very coarse cache measurements suffice, as opposed to our side-channel setting, which necessitates fine-grained cache measurements. Meltdown further requires the attacker to access a protected memory that is mapped into its own address space; this is inapplicable to portable code. Web-based Spectre does not work across browser processes (of different browsers or tabs).

Side-Channel Attacks on ElGamal Encryption. Several works show side-channel attacks on implementations of ElGamal encryption. [39] show a cross-VM attack on ElGamal that exploits the L1 data cache and the hypervisor's scheduler. Our attack is loosely modeled after [23], who implemented a Prime+Probe attack [29] targeting an implementation of ElGamal. Recently, [12] show a physical (electromagnetic) side-channel attack on ElGamal running on PCs.

2 Preliminaries

2.1 Portable Code Execution

JavaScript is the oldest and most common portable programing language that can be executed inside the web browser. For intensive computational tasks, JavaScript is much slower than native applications; NaCl, PNaCl and WebAssembly are alternative, more efficient solutions.

PNaCl. Modern Chrome browser support Google Native Client (NaCl) [38]. This is a sandboxing technology which enables secure execution of native code as part of untrusted web applications, which can run compiled code at near-native speeds and fine-grained control over the memory usage. While NaCl deploys architecture-dependent (through OS-independent) code, the subsequent Portable Native Client (PNaCl) achieves full cross-platform portability 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. PNaCl is enabled by default on Chrome browsers and does not require user interaction.

WebAssembly. WebAssembly is the standardized successor of PNaCl, standardized by the World Wide Web Consortium (W3C), and supported by all major web browsers on all operating systems, including mobile platforms. Similarly to PNaCl, WebAssembly defines a binary format which can be executed in a sandboxed environment inside the browser. Code is represented in simple stack machine, with a limited set of operations (mostly arithmetical and memory accesses). This is translated, by the browser, to the host's native instruction set, allowing it to be executed in near-native speed.

The simple abstract machine severely limits the environment observable to WebAssembly code. As oppose to PNaCl, the limited instruction set of WebAssembly does not directly expose any of the system's APIs; functionality beyond simple computation is exposed only via call-outs to interpreted JavaScript code, which are relatively slow.

Web Workers and JavaScript's `SharedArrayBuffer`. Web Workers is an API designed to allow JavaScript code to run heavy computational tasks in a separate context, without interfering with the user interface, using multiple threads. The communication between the main JavaScript context and Web Workers threads can be done using an asynchronous messaging system, or via the `SharedArrayBuffer` API which can allocate a shared memory buffer and coordinate access to it using synchronization primitives.

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 both PNaCl and WebAssembly, the mapping from memory addresses to cache sets consists of multiple abstraction layers, as follows. The portable runtime emulates a 32-bit execution environment, which is mapped (by the browser) into the hosting process's virtual address space, which is in turn mapped (by the operating system) into physical memory. Neither mapping is made available to the portable code. Lastly, physical memory addresses are mapped (by the CPU) to cache sets; Intel does not disclose this mapping, but it has been reverse-engineered. Despite two levels of indirections with unknown mapping, and complications introduced by the third one, we can find the mapping of memory blocks to sets.

Past Approaches. Several prior works [14,23,24] describe techniques for creating the eviction sets using *huge pages*: a CPU feature that allows pages in the page table to have a very large size (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 [13,34] were able to avoid huge pages, at the cost of imposing other limitations. The attack of [34] assumes consecutive physical memory allocation and deterministic heap behavior. Those assumptions allows the attacker to find the cache set index up to a fixed offset, providing as much information as using huge pages. Unfortunately, they are generally inapplicable, and for JavaScript code running in a browser environment, due to its complex garbage collection pattern, we empirically did not observe any allocation pattern between different execution of the decryption operations.

Next, the work of [13] 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 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 $\times 64$ (compared to smaller eviction sets which are designed to only evict a single cache set) thus reducing the channel's bandwidth. Large eviction sets also induce higher measurement noise due to unrelated memory accesses. While that method suffices to derandomize ASLR, key extraction attacks requires fine-grained, low-noise measurements of the target's memory access, with temporal resolution on the order of a big-integer multiplication.

3.1 Methodology

We now describe our methodology of constructing eviction sets by recovering the mapping from memory blocks to cache sets. As described above, the mapping consists of several layers. The work of [23] introduced an algorithm for uncovering the mapping between the physical address and cache slices, without the knowledge of the CPU's internals. However, the algorithm assumed knowledge of the cache set index, acquired by using huge pages. This assumption does not hold for PNaCl and WebAssembly since they do not provide access to huge pages. Instead we generalize this algorithm to the portable environment.

Constructing Eviction Sets from Portable Environment. Portable code only has access to the 12 least significant bits of the physical address, due to the fact that "page offset" goes through the mapping between portable environment and physical address space. Thus, the portable code knows the 6 least significant bits of the cache set index, but is missing the 4 or 5 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. Applying the algorithm of [23] on the pool results in initial eviction set for each cache set index with 6 least significant bits equal to 0. Then, by enumerating each of the possible values for the 6 least significant bits, we extend each initial eviction set to 64 eviction sets, each corresponding to a single cache set.

However, for the algorithm to work, we need to modify the eviction testing procedure. This is since when running on a system configured with regular-size 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 if the witness block is successfully evicted.

Handling TLB Noise. *Eviction testing* finds whether accessing a list of memory blocks forces a cache eviction of a specific, *witness*, memory block. 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. We achieve this by accessing another memory block in the same page as the witness. Thus the *eviction testing* algorithm becomes: access the witness to ensure it is in the cache; access each memory block in the list of memory blocks; access a memory block in the same page as the witness (to ensure the TLB entry for the page is updated); and finally measure the access time to the witness (which 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).

Handling Additional Noise. Even after handling the noise from the TLB, the increased footprint of our methodology and the overhead of the portable environment causes high measurement noise. We handle this noise by repeating the contracting stage, randomizing the order of the tested elements each time, and calculating the intersection between the constructed eviction sets.

3.2 Implementation

PNaCl Implementation. The above approach requires several capabilities. In order 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. Conveniently, PNaCl provides a `clock_gettime()` function which provides time at nanosecond accuracy (when called with `clock_realtime` parameter). Next, in order to construct the eviction sets in PNaCl's execution environment we allocate a sufficiently large contiguous buffer (approximately 4 times larger than the size of the LLC). Using this buffer and the aforementioned timing source, we performed the phases outlined above for the construction of the eviction sets.

WebAssembly Implementation. As discussed in Sect. 2.1, PNaCl has been available for a few years, but only on Chrome browser. Using the newer WebAssembly standard, along with Web Workers and `SharedArrayBuffers` allowed us to reimplement the approach without using browser-specific features. Similarly to PNaCl, in order to construct eviction sets we obtain a high-precision timer, and a contiguous allocated memory buffer.

The work of [28] prompted the web browser developers to reduce the precision of the time source available to JavaScript code. Unlike PNaCl, WebAssembly does not have access to system's APIs like `clock_gettime()`. Thus, we use an alternative technique, based on an intentional inter-thread race condition (see [33] for a recent survey of JavaScript timing sources, including this one).

In this approach, we allocate a `SharedArrayBuffer` array within the main JavaScript context, and pass it to a "Timer" Web Worker which iteratively increments the value in the first cell of the array in a tight loop. To learn the current time, the main context reads that cell. The naive implementation, accessing the array directly, did not work due to runtime optimization of supposedly-redundant reads. To overcome this, we used the `Atomics` API to force reading from the array (with sufficiently small performance penalty).

Next, we construct our eviction sets using `WebAssembly.Memory` contiguous buffer accessible both for JavaScript and WebAssembly. Accessing to this buffer from WebAssembly, and using the time source described above, allows us to identify cache misses using the above techniques.

Exprimental Results. On the Chromebook machine described in Sect. 1.2 we used the PNaCl implementation. Out of the 4096 sets, withing less then a minute we were able to construct 4032–4160 eviction sets (some duplicate eviction set was not removed during the collect phase). For the HP EliteBook 8760w laptop equipped with 8192 cache set, constructing the eviction sets took 11 min using the PNaCl and resulted in 7680–8320 eviction sets (with some duplicates as well). Using the WebAssembly implementation we were able to construct eviction sets on Chrome and Firefox as well. Constructing the eviction sets took 60–70 min and yield 7040–7680 eviction sets.

4 Attacking Elliptic

This section shows 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 on Elliptic's Curve25519-based ECDH implementation, which uses the Montgomery ladder method.

4.1 Deployment

Our attack scenario is based on running cache-monitoring portable code, using either of PNaCl or WebAssembly, inside the target's browser. We now describe a 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 portable attack code and submitted it to a web ad service. The targeted user opened a web browser (either Chrome or Firefox, and on either the Chromebook or HP laptops described in Sect. 1.2), accessed a third party web page which uses the ad service, and clicked anywhere within the page. Consequentially (courtesy of the ad service), our advertisement was opened in a background tab and started monitoring the cache access patterns on the target machine. Concurrently, the user opened a third tab, in the Chrome browser, which performed ECDH key-exchange operations using Ellipstic's Curve25519. Neither the website used to trigger the attack, nor the ad service, were controlled by the attacker; and the user never typed or followed a link to an attacker-controlled website.

4.2 Key Extraction

ECDH. Elliptic curve Diffie Hellman (ECDH) is a variant of the Diffie-Hellman key exchange protocol [8] 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 [6] and standardized by RFC 7748. Curve25519 was specifically designed to increase resistance to side channel attacks and other common implementation issues.

Scalar-By-Point Multiplication. In order to increase side channel resistance, implementations of Curve25519-based ECDH often use the Montgomery ladder [26] to perform the scalar-by-point multiplication operation. See Algorithm 1. Notice that the 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 multiplication algorithms [19,27].

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 1 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 does not leak the secret key via key-dependent sequences of double and add operations, or key-dependent memory accesses to a table of precomputed values. As we

Algorithm 1. 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: $b = [k]P$.
1: procedure SCALAR_BY_POINT_MULTIPLICATION($k, P$)
2:     $a \leftarrow P, b \leftarrow \mathcal{O}$                    ▷ $\mathcal{O}$ is the point of infinity
3:     for $i \leftarrow n$ to 1 do
4:         if $k_i = 0$ then
5:             $a \leftarrow a$.ADD($b$)                              ▷ $a + b$
6:             $b \leftarrow b$.DOUBLE()                              ▷ $[2]b$
7:         else
8:             $b \leftarrow a$.ADD($b$)                              ▷ $a + b$
9:             $a \leftarrow a$.DOUBLE()                              ▷ $[2]a$
10:    return $b$
```

have empirically validated, Elliptic's implementation of Algorithm 1, running on Chrome, is almost constant time, without key-dependent timing deviations.

While Algorithm 1 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 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 a and b change with each iteration of the main loop, without any obvious patterns. This makes monitoring memory accesses to a and b difficult, since the attacker has to predict and monitor a different cache set at every iteration of the main loop.

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 approach we identify a key-dependent change in the target's control flow. During the ECDH operation, we monitor the code cache accesses via PNaCl or WebAssembly, 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 1. 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 1 does not reveal any additional key-dependent control flow, we do observe that Algorithm 1 invokes the double operation in Line 6 on variable b, while in Line 9 it is invoked on object a. While

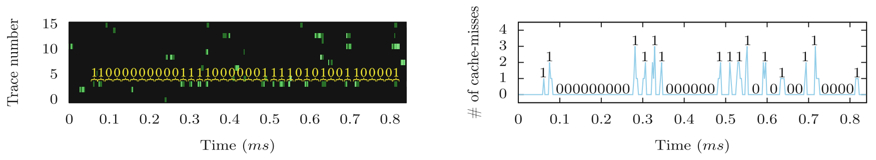


Fig. 2. Cache accesses as detected by the attacker during ECDH key exchange over Curve25519 by Elliptic. Trace 3 (left) contains cache misses observed by the attacker during the scalar-by-point multiplication. On the right, which only shows Trace 3, it can clearly be noticed that the cache-misses corresponds to key bits of 1, while sequence without cache-misses of 20 µs corresponds to bits of 0.

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 with WebAssembly. We demonstrated our WebAssembly attack in a cross-browser, cross-process scenario. We used the HP laptop to launch two separate web browser instances: Chrome, running a page that uses Elliptic's implementation of Curve25519-based ECDH, and Firefox, running a third-party web site presenting advertisements from our advertisement provider. After clicking inside the third-party web site, our WebAssembly attack code was loaded as a pop-under ad, and automatically started the eviction-set construction procedure described in Sect. 3. The CPU of this HP laptop has 8192 cache sets, and each Curve25519 ECDH key exchange lasts 2.5 ms. Hence, after the construction procedure, our code sampled each of the 8192 eviction sets, performing Prime+Probe cycle every 380 µs for a duration of 22 ms, for a total sampling time of about 3 min.

Monitoring Elliptic's Side Channel Leakage with PNaCl. Alternatively, we opened two tabs in the Chromebook's browser: one tab running our PNaCl attack code, and the other running Elliptic's implementation of Curve25519-based ECDH, with each key exchange lasting 4.5 ms. Next, we sampled each of the 4096 eviction sets, performing Prime+Probe cycle every 3 µs for a duration of 35 ms, totally sampling for less than 3 min.

Leakage Analysis. Out of the acquired traces, for each of the sampling methods we identified 5 as containing the side channel leakage described above. Figure 2 shows some out of the acquired traces using PNaCl on the Chromebook machine, Trace 3 (left) contains the information regarding the secret key.

As can be seen from the right part of Fig. 2, showing only Trace 3, a sequence of 10 µs of cache-misses cache-misses followed by 5 µs of cache-hits in the monitored set corresponds to a bit of 1, while 20 µs of cache-hits corresponds to 0 bit.

Using this, we automated the extraction of keys from traces, yielding correct extraction of up to 236 (out of 252) bits of the secret key from *individual* traces. Combining 4 traces of key-exchange operations we were able to extract all the 252 bits of the secret key. For the WebAssembly attacks, the acquired traces and automated algorithm are very similar, and likewise result in full key extraction.

5 Attacking ElGamal

5.1 Attacking End-to-End

ElGamal [9] is a public-key crptosystem based on hardness of computing discrete logarithms. In a nutshell, to decrypt a ciphertext (c_1, c_2), one has to compute the shared secret $s = c_1^x \bmod p$ and then recovers the message by computing $m' = c_2 \cdot s^{-1} \bmod p$. To compute the modular exponentiation during decryption, End-to-End uses a variant of the fixed-window (m-ary) exponentiation algorithm [25, Algorithm 14.109]. The algorithm divides the secret exponent into equal-sized groups of bits called windows, performing a single multiplication for each window using a precomputed value for every possible windows value.

Our attack largely follows the technique of [23] and consists of two phases. In the *online phase* we collect many 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, we recover information on the operands of the multiplications, from which we deduce bits of the exponent and then recover the key.

Monitoring End-to-End's Side Channel Leakage. Following Sect. 4 we opened two tabs in the Chromebook's browser: one running our PNaCl attack code, and the other running End-to-End's ElGamal, where each decryption operation lasts 1.58 s on the Chromebook device. Next, we selected 8 random cache sets and monitored them in parallel, performing a Prime+Probe cycle on each of the cache sets once every 31.5 µs for a duration of 5 s. We repeated this process sequentially for about 74 min, acquiring 7100 traces.

Leakage Analysis. Figure 3 shows the side channel leakage from an End-to-End ElGamal decryption. Traces 3 and 19 contains cache misses observed during the multiplication operations used by the exponentiation algorithm. To extract the key, we applied offline processing: denoising, clustering, merging, conflict resolution and key recovery. This took 90 min (cost: under \$6 on Amazon EC2). We ran our attack on several random ElGamal keys with 3072-bit public primes, both on the Chromebook and the HP laptop, successfully extracting the entire secret exponent in every trial.

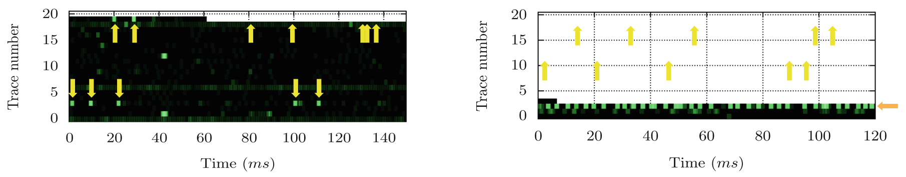


Fig. 3. Cache accesses as detected by the attacker during ElGamal decryption by End-to-End (left) and OpenPGP.js (right). Intensity represents the number of cache misses. Traces 3 and 19 on the left, and trace 11 and 19 on the right, contain cache misses observed by the attacker during the multiplication operations used by the exponentiation algorithm. Trace 2 (right) shows code-cache misses in the execution of the modular multiplication code during an OpenPGP.js decryption operation; the different intervals between the multiplications leak the location of sequences of zero bits.

End-to-End's implementation of RSA [32] decryption operations uses the same fixed-window routine to perform modular exponentiation. Thus, our attack is applicable for extracting RSA keys, even tough End-to-End implemented ciphertext blinding countermeasure against side-channel attacks.

5.2 Attacking OpenPGP.js

OpenPGP.js implements ElGamal decryption using sliding-window exponentiation [25, Algorithm 14.85]. Similarly to fixed-window exponentiation, the sliding window algorithm also use indexes a table of precomputed multipliers, on every multiplication operation. However, for speed, sequences of 0-bits are handled by simply performing corresponding squaring operations. Thus, the sliding-window algorithm leaks the location and length of zero sequences, and has been proven less resistant to side-channel attacks [23].

To measure the leakage we used an analogous setup to the one used in Sect. 5.1. 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. We monitored random cache sets, performing a Prime+Probe cycle on each set every 20 µs for a period of 0.62 s. The cache access patterns observed by the attacker reveal when a specific window value is used during the multiplication operations, Fig. 3 (right) shows the side channel leakage from one ElGamal decryption operation. Finally, the squaring operations performed by the sliding-window algorithm reveal long sequences of zero exponent bits This additional source leakage in Trace 2 of Fig. 3 (right) by monitoring the executions of the modular multiplication code.

6 Conclusion

In this paper we present a method for implementing an LLC-based Prime+Probe attack on an multiple cryptographic libraries ranging from ElGamal to state-of-the-art Curve25519-based ECDH using portable code executing inside a sandboxed web browser. We successfully deployed the attack using a commercial ad service that triggers the attack code from third-party websites, and automatically starts monitoring the memory access patterns when users navigate to the ad. 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 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, Chromebooks are vulnerable to cache based attacks.

Countermeasures. Side-channel resistant code requires constant-time implementation and avoiding secret dependent branches and memory accesses. These approaches are very delicate, and may fail when on different hardware or with different compilers. Using these techniques in JIT-compiled environments is an unexplored area that we leave for future work. Meanwhile, cryptographic operations in JavaScript should to delegated to suitable native implementations, such as (extensions of) WebCrypto API.

Limitations. Constructing eviction sets as described in Sect. 3 depends on the cache structure and eviction policy: in particular, an inclusive LLC, and an LRU (or similar) eviction policy. While both assumptions hold for modern Intel CPUs, other vendors may differ. Some of our attacks (Sect. 4) requires only a few minutes of sampling time (corresponding to about a thousand decryptions), and suggest a realistic threat to affected systems that conduct frequent decryptions. Others (Sect. 5) requires over an hour of sampling time, but should none the less indicate that observable leakage is prevalent across diverse cryptographic algorithms and implementations, and is expoitable by portable code.

Thus, the threat of cache timing side-channel attacks from sandboxed portable code must be considered, and mitigated, in the design of modern systems where such code is trivially controlled by attackers.

Acknowledgments. This work 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. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_8
2. Acıiçmez, O., Gueron, S., Seifert, J.-P.: New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 185–203. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77272-9_12
3. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 225–242. Springer, Heidelberg (2006). https://doi.org/10.1007/11967668_15
4. Acıiçmez, O., Schindler, W.: A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 256–273. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79263-5_16
5. Bernstein, D.J.: Cache-timing attacks on AES (2005). http://cr.yp.to/papers.html#cachetiming
6. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
7. Bulygin, Y.: CPU side-channels vs. virtualization malware: the good, the bad or the ugly. In: ToorCon (2008)
8. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
9. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theory **31**(4), 469–472 (1985)
10. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.B.: Understanding and mitigating covert channels through branch predictors. TACO **13**(1), 10:1–10:23 (2016)
11. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptograph. Eng. **8**(1), 1–27 (2018)
12. Genkin, D., Pachmanov, L., Pipman, I., Tromer, E.: Stealing keys from PCs using a radio: cheap electromagnetic attacks on windowed exponentiation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 207–228. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_11
13. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: practical cache attacks on the MMU. In: NDSS (2017)
14. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in JavaScript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 300–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_15
15. Gruss, D., Spreitzer, R., Mangard, S.: Cache template attacks: automating attacks on inclusive last-level caches. In: USENIX, pp. 897–912 (2015)

16. İnci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Cache attacks enable bulk key recovery on the cloud. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 368–388. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53140-2_18
17. Inci, M.S., Gülmezoglu, B., Apecechea, G.I., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud. IACR Cryptology ePrint Archive, p. 898 (2015)
18. Indutny, F.: Fast elliptic curve cryptography in plain JavaScript (2017). https://github.com/indutny/elliptic
19. Joye, M., Yen, S.-M.: The montgomery powering ladder. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36400-5_22
20. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: ISCA, pp. 361–372 (2014)
21. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: exploiting speculative execution. ArXiv e-prints (2018)
22. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. ArXiv e-prints (2018)
23. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Symposium on Security and Privacy, pp. 605–622 (2015)
24. Maurice, C., Weber, M., Schwartz, M., Giner, L., Gruss, D., Boano, C.A., Römer, K., Mangard, S.: Hello from the other side: SSH over robust cache covert channels in the cloud. In: NDSS (2017)
25. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography, 1st edn. CRC Press, Boca Raton (1996)
26. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243 (1987)
27. Okeya, K., Kurumatani, H., Sakurai, K.: Elliptic curves with the montgomery-form and their cryptographic applications. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 238–257. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46588-1_17
28. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: practical cache attacks in JavaScript and their implications. In: ACM SIGSAC, pp. 1406–1418 (2015)
29. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
30. Percival, C.: Cache missing for fun and profit. In: Presented at BSDCan (2005). http://www.daemonology.net/hyperthreading-considered-harmful
31. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In: CCS, pp. 199–212 (2009)
32. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
33. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 247–267. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_13

34. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: using SGX to conceal cache attacks. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 3–24. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_1
35. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. J. Cryptol. **23**(1), 37–71 (2010)
36. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 62–76. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45238-6_6
37. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX, pp. 719–732 (2014)
38. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: a sandbox for portable, untrusted x86 native code. In: IEEE Symposium on Security and Privacy, pp. 79–93 (2009)
39. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: CCS, pp. 305–316 (2012)