Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript

Daniel Gruss
Graz University of Technology, Austria
daniel.gruss@iaik.tugraz.at

Clémentine Maurice
Technicolor, Rennes, France
Eurecom, Sophia-Antipolis, France
clementine.maurice@technicolor.com

Stefan Mangard
Graz University of Technology, Austria
stefan.mangard@tugraz.at

***Abstract*—As DRAM has been scaling to increase in density, the cells are less isolated from each other. Recent studies have found that repeated accesses to DRAM rows can cause random bit flips in an adjacent row, resulting in the so called *Rowhammer bug*. This bug has already been exploited to gain root privileges and to evade a sandbox, showing the severity of faulting single bits for security. However, these exploits are written in native code and use special instructions to flush data from the cache.**

In this paper we present Rowhammer.js, a JavaScript-based implementation of the Rowhammer attack. Our attack uses an eviction strategy found by a generic algorithm that improves the eviction rate compared to existing eviction strategies from 95.2% to 99.99%. Rowhammer.js is the first remote software-induced hardware-fault attack. In contrast to other fault attacks it does not require physical access to the machine, or the execution of native code or access to special instructions. As JavaScript-based fault attacks can be performed on millions of users stealthily and simultaneously, we propose countermeasures that can be implemented immediately.

I. Introduction

Hardware-fault attacks have been a security threat since the first attacks in 1997 by Boneh et al. [1] and Biham et al. [2]. Fault attacks typically require physical access to the device under attack to expose it to physical conditions which are outside the device specification. This includes high or low temperature, radiation, as well as laser on dismantled microchips. However, software-induced hardware faults are also possible, if the device under attack can be brought to the border or out of the specified operation conditions using software.

Recently, Kim et al. [3] showed that frequently accessing specific memory locations can cause random bit flips in DRAM chips. These random bit flips can even be triggered by software by flushing the memory location from the cache and reloading it. In early 2015, Seaborn [4] demonstrated how an attacker can exploit such bit flips to get access to the physical memory of the machine from inside the Chromium sandbox or from an unprivileged user program. However, these exploits are written in native code and use special instructions to flush data from the cache. A recent tech report by Xuanwu Labs [5] suggests that this is not possible because the attacker can not trigger the flush instruction appropriately from scripting languages.

In this paper, we present an implementation of the Rowhammer attack that is independent of the instruction set of the CPU. Our attack is the first remote software-induced hardware-fault attack. It is implemented in JavaScript in Firefox 39, but our attack technique is generic and can be applied to any architecture, programming language and runtime environment that allows producing an efficient stream of memory access instructions. The main challenges to perform this attack are finding an optimal eviction strategy as replacement for the flush instruction and retrieving sufficient information on the physical addresses of data structures in JavaScript to find address pairs efficiently.

We describe an algorithm to find an optimal eviction strategy for an unknown cache replacement policy. Existing eviction strategies focus on the pseudo-LRU cache replacement policy as implemented in Sandy Bridge [6], [7]. On Haswell and Ivy Bridge CPUs these eviction strategies show a significantly lower eviction rate. While the LRU eviction strategy achieves an eviction rate of 95.2% on our Haswell test machine, our optimal eviction strategy improves the eviction rate to 99.99%. Furthermore, our eviction strategy is more efficient in terms of additional memory accesses and time consumption. Both are crucial to successfully exploit the Rowhammer bug.

As a proof of concept we implement our attack in JavaScript. JavaScript is a scripting language implemented in all modern browsers to create interactive elements on websites. We do not exploit any weaknesses in JavaScript or the browser. Rowhammer.js is possible because today's JavaScript implementations are well optimized and achieve almost native code performance for our use case. JavaScript is strictly sandboxed and the language provides no possibility to retrieve virtual or physical addresses. However, the usage of large pages by the malloc implementation that is used by the JavaScript engine in Firefox on Linux allows determining parts of the physical address.

We compared our implementations of the Rowhammer attack on the three different machines shown in Table I. While two are susceptible to the Rowhammer attack using the flush instruction, one of them was also vulnerable against the Rowhammer attack without the flush instruction. The third machine was only vulnerable to the Rowhammer attack with an increased refresh rate. The probability of a bit flip is significantly higher when using the flush instruction on all three machines. However, comparing our native code (without the flush instruction) and JavaScript implementation we found a negligible difference in the probability of a bit flip. This result suggests that if bit flips can be observed using our native code implementation the system is vulnerable to remote JavaScript-

TABLE I. EXPERIMENTAL SETUPS.

Platform	CPU	Architecture	RAM
Lenovo T420	i5-2540M	Sandy Bridge	Corsair DDR3-1333 8 GB
Lenovo x230	i5-3320M	Ivy Bridge	Samsung DDR3-1600 4 GB (2×)
Asus H97-Pro	i7-4790	Haswell	Kingston DDR3-1600 8 GB

based attacks.

We still need to investigate how many systems are vulnerable to Rowhammer.js. Since the attack can be performed on an arbitrary number of victim machines simultaneously and stealthily it poses an enormous security threat. Furthermore, future work has to show whether Rowhammer.js can be exploited for sandbox escaping or privilege escalation. We suggest countermeasures to be implemented immediately to prevent possible attacks.

Summarizing, our key contributions are:

- We describe an algorithm to find an optimal eviction strategy for an unknown cache replacement policy. We verified that it finds an optimal eviction strategy on Sandy Bridge, Ivy Bridge and Haswell CPUs.
- We built a native code implementation of the Rowhammer attack which only uses memory accesses. We verified that the attack is successful on Ivy Bridge and Haswell CPUs.
- We developed a tool to translate JavaScript array indices to physical addresses in order to trigger bit flips on known physical memory locations from JavaScript.
- We built a pure JavaScript implementation which scans the memory efficiently for vulnerable addresses.
- We discuss how a JavaScript-based Rowhammer attack can be used by a remote attacker to gain access to the physical memory of a system.
- We propose countermeasures to be implemented immediately to prevent attacks on millions of users.

The remaining paper is organized as follows. In Section II, we provide background information on DRAM and the Rowhammer bug, as well as modern CPU caches and cache attacks. We describe the adaptive eviction strategy algorithm in Section III. In Section IV, we evaluate the optimal eviction strategy on Haswell by analyzing the eviction rate and performing a Rowhammer attack in native code without the flush instruction. In Section V, we describe the JavaScript implementation and compare it to the native code implementation and the original implementation with the flush instruction. We discuss countermeasures against our attack in Section VII and future work in Section VIII. Finally, we provide conclusions in Section IX.

II. BACKGROUND AND RELATED WORK

In this section, we give an introduction to DRAM and the Rowhammer bug. Furthermore, we describe how CPU caches in modern Intel CPUs work and related work on cache attacks.

A. DRAM

A memory system can have a single *channel* or multiple channels, that are physical links between the DRAM memory and the memory controller. Multi-channel memory architecture increases the transfer speed of data. A channel consists of multiple *Dual Inline Memory Modules* (*DIMMs*), that are the physical modules plugged into the motherboard. Each DIMM contains one or two *ranks*, that correspond to the sides of the physical module. Each rank is a collection of *chips*, that are further composed of *banks*, typically 8 on recent DRAM. Accesses to different banks can be served concurrently. Each bank is a two-dimensional array of capacitor-based cells. A cell is either in a charged or discharged state, which represents a binary data value. The bank is thus represented as a collection of rows, typically 2^{14} to 2^{17}.

Each bank has a row buffer where the memory accesses are served from. The charge from the cells is read into the row buffer on request and written back as soon as another row is requested. Thus, access to the DRAM is done in three steps:

1) opening a row,
2) accessing the data by reading and/or writing any column,
3) closing the row, which also clears the row buffer before a new row can be opened.

DRAM is volatile memory and discharges over time. To prevent data corruption, each row is refreshed in a certain interval, called the *refresh interval*. Refreshing a row is the same operation as opening the row, *i.e.*, it reads and restores the charge of the cells. DDR3 DRAM specifications require refreshing all rows in a 64ms time window [3].

The selection of channel, rank, bank and row is done by a subset of physical address bits. AMD documents the addressing function used by its processors, but Intel does not. This function can vary between different systems and system configurations. The mapping for one Intel Sandy Bridge machine in one configuration has been reverse engineered by Seaborn [8].

B. The Rowhammer bug

The increase of DRAM density has led to physically smaller cells, thus capable of storing smaller charges. As a result, the cells have a lower noise margin, and cells can interact electrically with each other although they should be isolated. The so called *Rowhammer bug* consists in the corruption of data, not in rows that are directly accessed, but rather in rows nearby the one accessed.

DRAM and CPU manufacturers have known the Rowhammer bug since at least 2012, date of the filing of several patent applications by Intel [9], [10]. In fact, hammering a DRAM chip is one of the quality assurance tests applied to modules. As refreshing DRAM cells consumes time, DRAM manufacturers optimize the refresh rate to the lowest value that still has a probability of virtually zero for bit flips to occur accidentally.

The Rowhammer bug has only been studied recently in academic research [3], [11], [12]. In particular, Kim et al. [3] studied the vulnerability of off-the-shelf DRAM modules to bit flips, on Intel and AMD CPUs. They built a program that induces bit flips by software using the `clflush` instruction. The `clflush` instruction flushes data from all cache levels, forcing the CPU to serve the next memory access from the DRAM instead of cache. Their proof-of-concept implementation frequently accesses and flushes two different memory locations in a loop, causing bit flips in a third memory location.

Although they discussed how to exploit these bit flips for privilege escalation attacks, they did not provide an exploit.

Seaborn implemented two attacks that exploit the Rowhammer bug [4]. The first one is a kernel privilege escalation on a Linux system, caused by a bit flip in a page table entry. By spraying the physical memory with page tables, a bit flip in a page table has a high probability to replace the previous mapping with the mapping of a process owned page table. If a process can modify its own page table, it can access the whole physical memory of the system. The second one targets Native Client in Google Chrome [13]. Native Client allows executing sandboxed native code – that included the `clflush` instruction – through a website. The attack is an escape of Native Client sandbox, from which an attacker could take control over the system, caused by a bit flip in an instruction sequence for indirect jumps.

At the hardware level, Kim et al. [14] proposed two architectural mitigations to the Rowhammer bug. The first solution uses a counter for the number of row activations. A dummy activation is sent to neighboring rows to refresh the cells when the counter exceeds a threshold. The second solution performs, for each activation of a given row, a row activation of the neighboring rows with a small probability. However, these solutions can only be implemented in future DRAM chips, and are not portable to currently deployed chips.

At the software level, the `clflush` instruction was removed from the set of allowed instructions in Google Chrome Native Client [15]. The removal of the `clflush` instruction from the instruction set was already suggested as a countermeasure to cache attacks in native and virtualized environments [16]. However, this instruction is not privileged, thus any unprivileged and possibly sandboxed process can use it. Moreover, it is unknown whether it is possible to disable the instruction at least for unprivileged processes through a microcode update and so far Intel has not released one.

However, as we show in this paper, cache eviction can be used as an efficient replacement to `clflush`. Seaborn and Birhanu have reported to be able to flip bits on Sandy Bridge CPUs by means of cache eviction [17]. In August, Seaborn [18] has published his implementation. However, we provide an adaptive eviction strategy that works on more recent CPUs like Haswell and Ivy Bridge as well. This shows that the Rowhammer attack using `clflush` was only one attack path, while the underlying problem is a timing issue and therefore independent of access to specific instructions.

C. CPU cache addressing

A modern CPU cache is a small and fast memory inside the CPU. The CPU hides the latency of main memory by keeping copies of frequently used data in the cache. In this document we only discuss Intel processors, although parts of it apply to other processors as well. Intel processors have 1 to 4 levels of cache, where L1 is the smallest and fastest cache and L4 the slowest and largest cache. The L3 cache is an inclusive cache, *i.e.*, all data in L1 and L2 cache is also present in the L3 cache. The L3 cache is shared among all cores, thus memory accesses and cache evictions directly influence the other cores. This effect is exploited in cache side-channel attacks [16]. The L4 cache is present in some Haswell and Broadwell CPUs, used for video memory, as well as to hold data evicted from the L3 cache.

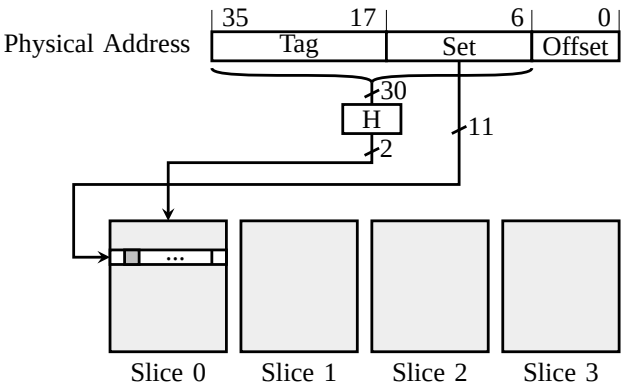

Fig. 1. Complex addressing scheme used in the LLC. The slice is given by a hash function that takes as input a part of the physical address. The set is directly addressed.

TABLE II. COMPLEX ADDRESSING FUNCTION.

		Address Bit														
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
2 cores	o_0		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕
4 cores	o_0	⊕	⊕	⊕		⊕		⊕		⊕		⊕		⊕	⊕	
	o_1	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕

All caches are organized in coherent blocks of 64 bytes, called lines. As the CPU cache is much smaller than the main memory, not all data can fit in the cache. The replacement policy predicts which cache line is least likely to be accessed in the near future. It thus decides which line to evict, and where to place the new line. Current CPUs implement n-way associative caches. It means that a line is loaded in a specific set depending on its address only, in any of the n lines. Common replacement policies are random, Least Recently Used (LRU) or modified variants of LRU. Intel has not disclosed the cache replacement policy of their CPUs. However, the replacement policies for some architectures have been reverse-engineered. Sandy Bridge has a pseudo-LRU replacement policy and Ivy Bridge a modification of the pseudo-LRU replacement policy [19].

Since the Nehalem microarchitecture, the L3 cache is divided into as many slices as CPU cores. However, each core can access every slice. The mapping from physical addresses to cache slices is performed by a so called *complex addressing* function, that takes as an input part of the physical address (see Figure 1). The complex addressing function is undocumented, however researchers have worked towards its reverse engineering. We have combined the results by Hund et al. [20] and Seaborn [21] in Table II. The table shows how address bits 17 to 31 are xor'd into one or two output bits o_0 and o_1. In case of a dual-core CPU, output bit o_0 determines to which of the two cache slices the physical address maps. In case of a quad-core CPU, output bits o_1 and o_0 are used to determine to which of the four cache slices the physical address maps.

D. Cache attacks and cache eviction

Cache eviction has ever been a means to perform cache side-channel attacks. Cache side-channel attacks exploit timing

differences between cache hits and cache misses. Cache attacks were first mentioned by Kocher [22] and Kelsey et al. [23]. Later practical attacks on cryptographic algorithms have been explored thoroughly [24]–[27]. They evict data from the cache by accessing large memory buffers. In 2006 Osvik et al. [28] proposed Prime+Probe, an attack technique that allows determining which specific cache sets have been accessed by a victim program. In order to do so, they determine which addresses are congruent to each other. Priming a cache set evicts all victim data stored on congruent addresses. The time taken by the prime step is directly proportional to the number of ways that have been replaced by other processes including the victim process.

Attacks by Gullasch et al. [29] and Yarom and Falkner [16] exploit shared memory and can thereby determine more accurately the memory locations accessed by a victim program. Their attack called Flush+Reload, works by frequently flushing a cache line using the `clflush` instruction instead of evicting it by means of memory accesses. By measuring the time it takes to reload the data, they determine whether a process under attack has reloaded the data in the meantime. Gruss et al. [30] have demonstrated that the Flush+Reload attack is possible without the `clflush` instruction with only a small loss in accuracy.

As shared memory is not always available between attacker and victim process, recent cache attacks use the Prime+Probe technique again. This allows performing attacks across virtual machine borders, even if memory sharing between guests is disabled. Powerful cross-VM side-channel attacks [6], [31] and covert channels [32] have been presented in the last year. At the same time, Oren et al. [7] successfully implemented a Prime+Probe cache attack from within sandboxed JavaScript to attack user-specific data like network traffic or mouse movements. Both propose eviction strategies for pseudo-LRU cache replacement policies. However, more recent CPUs like Ivy Bridge and Haswell use adaptive cache replacement policies which are only pseudo-LRU in corner cases. In the following Section, we will show that our adaptive eviction achieves a significantly higher accuracy on these new architectures.

III. Adaptive Cache Eviction Strategy

Evicting data from the cache is crucial to cache attacks like Flush+Reload. It is also crucial for the Rowhammer attack, to serve memory accesses from the DRAM instead of cache. Both use the `clflush` instruction for this purpose. Hund et al. [20] described that it is possible to evict data from the cache by filling a large memory buffer the size of the cache. However, this takes significantly more time and memory accesses than are necessary for the eviction of one specific cache line.

Our goal is to optimize cache eviction in terms of the execution time of the eviction function, thus in terms of the number of memory accesses (cache misses). That is, we want to find a set of addresses which are congruent in cache, as well as an access pattern which accesses a minimal number of these addresses. This is done by first optimizing regarding eviction and when the eviction rate is close to 100% reducing of the eviction addresses that do not lower the eviction rate.

The replacement policy influences the number and the patterns of accesses needed for the eviction strategy. Sandy Bridge uses a pseudo-LRU replacement policy. Consequently, after accessing as many congruent locations as the number of ways of the L3 cache (typically 12 or 16), the physical address to evict is evicted with a high probability. However, Ivy Bridge and following microarchitectures use a slightly different replacement policy called Quad-Age LRU [33]. Wong [19] observed that dual-pointer chasing can help to cause eviction on Ivy Bridge with a similar probability as on Sandy Bridge. However, his approach uses chained lists which have the drawback to be slow. Moreover, the algorithm is not applicable to Haswell CPUs. Our measurements indeed show that accessing a memory location twice is not sufficient to trick the CPU to always keep it in the cache as long as possible. Both, on Ivy Bridge and on Haswell CPUs the address is more likely to be evicted if it is accessed three times.

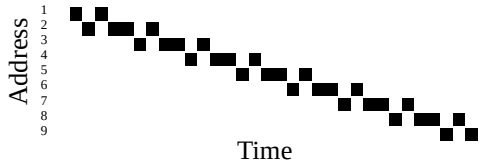

Fig. 2. Excerpt of optimal access pattern on Haswell.

Figure 2 shows the best eviction strategy we found for Haswell CPUs. Although the adaptive algorithm finds similar strategies with a similar access pattern, it results in more randomized access patterns. The pattern uses two accesses to the same physical address, then one access to a second physical address and then a third access to the first physical address again. The first two accesses per memory location avoid data to be evicted immediately. The third access tricks the CPU to keep the data in the cache, as it is frequently used. The first address and the last address are not accessed as often, so they are more likely to be evicted. Therefore, some ways of the cache set remain occupied by the same data all the time and a smaller number of cache misses occurs.

We found that this strategy works on Ivy Bridge and Sandy Bridge as well, but it is not optimal for Sandy Bridge, as a single access per address would already suffice. However, as all accesses are cached it does not incur a significant performance penalty compared to single accesses either. Liu et al. [6] and Oren et al. [7] presented algorithms to find optimal eviction sets on Sandy Bridge microarchitecture, by means of a timing attack. Although these algorithms can work on Ivy Bridge and Haswell under certain circumstances, they are not optimal in the numbers of accesses (cache misses) as they do not consider accessing addresses more than once to trick the cache replacement policy. However, the probability of eviction is significantly lower in case of the LRU eviction strategy than with our adaptive eviction strategy, due to the new replacement policy used since Ivy Bridge.

To quantify the advantage of our adaptive eviction strategy we performed 12 million memory access on a fixed address which is supposedly evicted. We compared eviction using the LRU eviction strategy, our adaptive eviction strategy and for comparison, the `clflush` instruction. Figure 3 shows the access time histogram. On our test machine it takes more than 200 cycles to fetch data from memory. Most accesses in our test were fetched from memory, as can be seen from

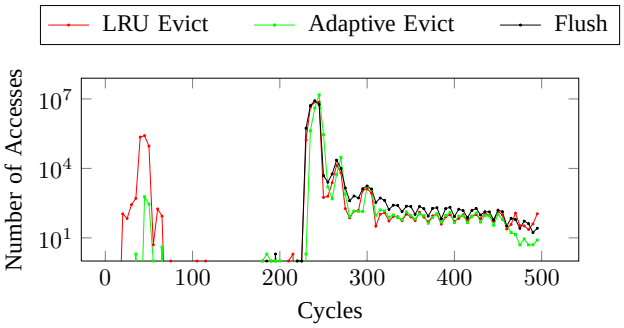

Fig. 3. Comparison of different eviction strategies on Haswell.

the peak at 230 cycles and the access times above. We see that in case of the flush instruction no memory accesses are below 230 cycles. That is, all accesses are fetched from memory, no accesses are cached. In case of the LRU eviction strategy 577319 accesses (4.8%) had a timing significantly below 230 cycles and therefore must have been cached. With our adaptive eviction strategy, we measured 891 accesses (less than 0.0001%) with a timing significantly below 230 cycles. Thus, the LRU eviction strategy yields 648 times as many cache hits than our adaptive eviction strategy.

Algorithm 1: Adaptive Eviction Strategy Algorithm.

Input: Target address p

Access p
while *cached(p)* **do**
  Choose random 64-byte-aligned address r from eviction buffer
  Insert r into eviction set twice e at two random positions
end
Set $c = 0$
while $c < |e|$ *and not cached(p)* **do**
  Choose random index r in e
  Store all indices r_i where $e[r] = e[r_i]$
  Replace all $e[r_i]$ with $e[r_i - 1]$
  if *cached(p)* **then**
    Undo changes for all indices r_i
    Set $c = c + 1$
  else
    Set $c = 0$
  end
end
Set $c = 0$
while $c < |e|$ *and not cached(p)* **do**
  Remove random element from e
  if *cached(p)* **then**
    Undo changes
    Set $c = c + 1$
  else
    Set $c = 0$
  end
end

Therefore, we designed the algorithm shown in Algorithm 1 that finds an adaptive eviction set, regardless of the replacement policy, by performing a timing attack. The `cached(p)` function tries to evict using the current eviction set and decides whether an access was cached or not based on the access time. To achieve a high accuracy, between 16 and 1 million times tests (eviction and access) are performed. If a single cache hit was measured, the function returns true and false otherwise. While consuming more time, more tests increase the accuracy of the binary decision, as eviction can also happen accidentally and thus the function would return a wrong result. Another benefit of performing a higher number of tests is that lower accuracy timers can be used when using the overall time consumed by the tests. Thus, countermeasures proposed by Oren et al. [7], that have been adopted by the W3 consortium [34] cannot prevent successful execution of our algorithm.

In the first step the adaptive eviction set finding algorithm continuously adds addresses to the eviction set. Each memory address is accessed multiple times, to reduce the probability of immediate eviction. We know that the eviction set is large enough as soon as we can clearly measure the eviction of the target physical address. We now seek to minimize this eviction set. We replace all accesses to addresses which have no influence on eviction with accesses to other addresses from the eviction set. Thus, the number of memory accesses does not decrease, but the number of different addresses decreases to the minimum. This decreases the number of cache misses and the execution time. In the final step we randomly remove all accesses which do not influence eviction. The result of the algorithm is an optimal access pattern for the system under attack. As we find the access pattern dynamically, based on the specific system, the strategy is adaptive to all cache eviction schemes.

Using this algorithm, we can compute an optimal eviction strategy and subsequently evict cache lines with a high probability without using the `clflush` instruction. This allows improving existing cache attacks which are based on Prime+Probe, like the JavaScript-based cache attack by Oren et al. [7] or the cross-VM cache attack by Liu et al. [6] on newer CPUs significantly. Furthermore, we are able to exploit the Rowhammer bug using this eviction strategy, as we discuss in the next Section.

The current implementation of our algorithm is significantly slower than the ones by Oren et al. [7] and Liu et al. [6]. However, it is necessary to achieve optimal eviction and as it is executed before the hammering loop it does not influence the attack itself. Therefore, in our proof-of-concept attack we use it only to find optimal eviction strategies in a preprocessing step.

IV. Implementation of Rowhammer without clflush in Native Code

We found on all our test machines a significantly higher probability for bit flips in a row N when hammering its neighbor rows $N - 1$ and $N + 1$. This technique is dubbed "double-sided hammering". We extended the `double_sided_rowhammer` program by Dullien [35] by using the best eviction strategies we have found as well as the adaptive eviction strategy. The two `clflush` instructions were first replaced by the eviction code shown in Listing 1 on Haswell. The eviction sets are precomputed in the

```
1  for (size_t i = 1; i < COUNT(f_evict); i += 1)
2  {
3    *f_evict[i];
4    *s_evict[i];
5    *f_evict[i+1];
6    *s_evict[i+1];
7    *f_evict[i];
8    *s_evict[i];
9    *f_evict[i+1];
10   *s_evict[i+1];
11 }
```

Listing 1. Hand-crafted eviction loop on Haswell

```
1  for (size_t i = 1; i < COUNT(f_evict); i += 1)
2  {
3    *f_evict[i];
4    *s_evict[i];
5  }
```

Listing 2. Generic eviction loop on Haswell

`f_evict` and `s_evict` variables using the physical address mapping and the complex addressing function published by Maurice et al. [36]. We verified that the loop works on Ivy Bridge equivalently.

We verified the probability of eviction by measuring the access times on the two addresses to evict. The second variant is shown in Listing 2. The variables `f_evict` and `s_evict` are computed by Algorithm 1 instead of physical addresses. The memory access pattern stays similar to the one described before, but is now represented in the array instead of the loop code.

Using the adaptive eviction strategy, we were able to reproducibly flip bits on our Ivy Bridge test machine. The machine was operated in default configuration for the DRAM and was mostly idle (Internet surfing) during our tests. However, we only observed bit flips when the CPU was running at the maximum frequency of 3.3GHz. We still have to do more experiments on this machine to exactly find the preconditions in terms of temperate ranges of the environment, the DRAM chips and the CPU. As it is not possible to set the DRAM refresh rate on this machine in the BIOS settings, we could not examine the influence of the refresh rate on the number of bit flips. Lenovo has released a BIOS update for this machine doubling the refresh rate of the DDR3 module [37]. Future work includes investigating whether the update successfully prevents bit flips on this test machine. There are also reports of users that a BIOS update did not solve the Rowhammer bug on their machine [38].

On our Haswell test machine we were not able to reproducibly flip bits with the default settings, not even with the `clflush` instruction. However, as the BIOS configuration allows setting a custom refresh rate, we were able to analyze its influence. We reproducibly flipped bits using the optimal Haswell eviction strategy at lower refresh rates. We used this setup as a development platform for the proof of concept Rowhammer implementation in JavaScript, as described in the next Section.

V. ROWHAMMER IN JAVASCRIPT

Several challenges have to be addressed to trigger the Rowhammmer bug from JavaScript. First, JavaScript has no concept of virtual addresses or pointers. As a work around we used large typed arrays. We observed that large typed arrays in JavaScript in Firefox 39 on Linux are allocated on anonymous 2MB pages. The reason for this lies in the memory allocation mechanism used by Firefox in this case. Only by knowing the offset in the array we know the lowest 21 bits of the virtual and physical address. However, this leads to the second problem: we cannot compute eviction sets based on physical addresses in JavaScript. Instead we use our adaptive algorithm from Section III. Third, memory access have to be implemented in a way that they cannot be optimized out by the just-in-time-compiler. However, we found that the access pattern from native code is not optimized out.

As a first proof of concept we tried to reproduce bit flips we found in JavaScript in Firefox 39. In order to do that we built a tool which monitors the virtual address space of Firefox. Each time a 2MB page is allocated we store the virtual address and the time difference to the last allocation. This way we can detect the beginning of the allocation in JavaScript. In a second step we build an inverted page table for the Firefox process. We then resolve the physical addresses we want to hammer to offsets within the JavaScript array. These offsets are then pasted into a field in the webpage to start hammering on the JavaScript array.

The final JavaScript-based attack does not require any outside computation and thus, runs entirely without user interaction in the browser. It exploits the fact that large typed arrays are allocated on 2MB pages. Thus, we know that each 2MB region of our array is divided into 16 row offsets of size 128KB (single channel). Some rows at the page borders cannot be hammered as they have no neighbored rows on the same 2MB page. For each of the 14 possible row offsets all page offset combinations are hammered. With 32 pages per row and channel there are $32 \cdot 32 = 256$ combinations to hammer pages per row and channel. The eviction set is computed using the adaptive eviction strategy finding algorithm from Section III. Although it is slower than the ones by Liu et al. [6] and Oren et al. [7], it finds a significantly better eviction strategy, which is necessary for the attack. These are the only two building blocks, resulting in the first remote JavaScript-based hardware-fault attack.

VI. ATTACK EVALUATION

We have evaluated how different refresh rates influence the number of bit flips (expected value based on our measurements) in Figure 4 for a fixed address pair and a fixed time interval in different setups. During the tests the system was under slight usage (browsing, typing in an editor, etc.). We see that the `clflush` instruction yields the highest number of bit flips. Native code eviction and JavaScript eviction are very close together and the difference between those two is negligible. The number of rounds that have fit into the fixed 15 minutes interval have varied a bit for the test runs at different refresh rates. The bit flips per round are shown in Figure 5. We can see that the expected number of bit flips per round is significantly lower in case of the eviction-based tests than in the tests with `clflush`.

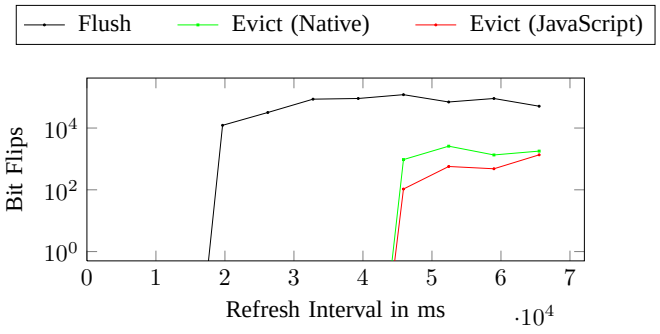

Fig. 4. Number of bit flips within 15 minutes (expected value based on our measurements) on a fixed address pair for different refresh rates on Haswell in the three different setups: `clflush`, native code eviction, JavaScript eviction.

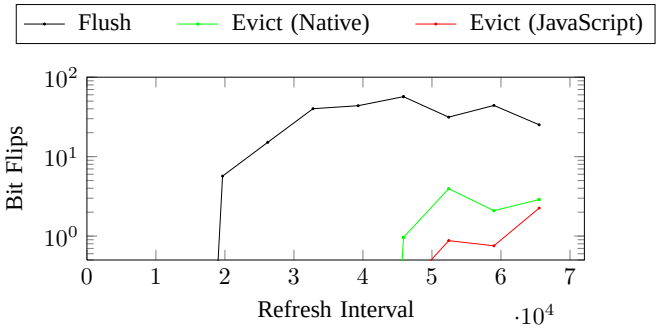

Fig. 5. Number of a bit flips per round (expected value based on our measurements) on a fixed address pair for different refresh rates on Haswell in the three different setups: `clflush`, native code eviction, JavaScript eviction.

We have observed that adding dummy instructions – that only consume time – while hammering sometimes increases the number of bit flips significantly. This effect occurred in both JavaScript and native code. These instructions increase the execution time of each hammering loop, which is counterintuitive with the fact that the bit flips should occur more often with a higher hammering rate. More attention should be given to this phenomenon, and to whether it can be used for countermeasures against the Rowhammer bug.

The probability for bit flips in JavaScript and in native code is almost the same. Thus, if a machine is vulnerable using our native code implementation it is vulnerable using our JavaScript implementation as well. We performed most tests on the Haswell machine as we were able to change the refresh interval. However, a successful remote attack will only be possible on the Ivy Bridge laptop as it has a significantly higher probability of bit flips without `clflush` in default settings.

VII. Countermeasures

Kim et al. [3] proposed several countermeasures which should be implemented for new DRAM modules. One countermeasure they propose is increasing the refresh rate. However, to prevent bit flips caused by row hammering, the refresh rate would need to be increased to 8 times its current value. While the DRAM is currently busy refreshing up to 4.5% of the time, it would then consume up to 35% of time, according to Kim et al. [3]. Some hardware manufacturers have already started to distribute BIOS updates. Most of these updates only double the refresh rate to reduce the probability of a successful attack. This solves the problem for most machines, however on some machines the probability will still be high enough to exploit. There is also the issue of whether end users will actually patch their machines. A report shows that only 30% of Windows systems are up-to-date, although Windows comes with an auto-update feature, which is enabled by default [39]. We assume that comparatively complicated BIOS updates will only be deployed by a diminutive group of users.

We found several guides suggesting to decrease refresh rates to gain higher system performance. Until now this might have affected the stability of the system. Now, users cannot do this anymore without risking a remote attacker performing fault attacks against their systems.

To prevent attacks on millions of users, browsers need to detect and prevent Rowhammer-based attacks. Part of this can be detection using hardware performance counters as suggested by Herath and Fogh [40] to stop suspicious scripts or detection whether the system is susceptible to rowhammering with `clflush` and subsequently slowing down JavaScript execution to prevent an attack. These countermeasures need further evaluation before they can be brought to practice.

VIII. Future Work

Although we demonstrated how the Rowhammer bug can be triggered from JavaScript, we do not provide a full root exploit. One way to get root privileges is by spraying the physical memory with page tables. Seaborn [4] built a root exploit which first tries to find a memory location causing a reproducible bit flip and then tries to fill the memory with its own page tables. If now a bit flips in one of the page tables the exploit notices that the file is not mapped anymore and as it has filled the memory with its own page tables, it probably has one of our its page tables mapped. If this is the case the program has full access to physical memory.

To exploit bit flips in page tables using double sided hammering, it is necessary to allocate 4KB pages. If this is the case page tables can be in a row between the two rows being hammered. As it is possible in JavaScript to get 4KB pages and 2MB pages, we know that a root exploit must be possible and we will investigate this next.

While we cannot memory map files in JavaScript, we observed that if the operating system or hypervisor deduplicates zero pages we can achieve a similar situation. By acquiring a large number of zero pages we could fill the memory with our own page tables like in the native code exploit. Having access to our own page tables from JavaScript would enable us to access and modify all physical memory, as in the native code exploit by Seaborn [4].

Although our eviction-strategy-finding algorithm (presented in Section III) works on different Intel CPUs, it is necessary to evaluate how it performs on non-Intel platforms, e.g., AMD x86 or ARM CPUs. Modern smartphones have fast DRAM modules integrated and should be examined for potential security risks.

While we implemented a successful attack in JavaScript in Firefox 39, the problem is bigger than that. First, we expect the

attack to work in other browsers as well, if the JavaScript just-in-time compilation is as efficient. More general, the presented approach can be applied to any programming language and any runtime environment. If the memory is accessed in the same pattern and the frequency is high enough it will cause bit flips as well. We chose JavaScript because it can easily be executed by a remote attacker. However, other languages like ActionScript or server-side scripting languages could also be interesting targets for an attacker if they are translated to a native instruction stream by a compiler or JIT-compiler.

IX. Conclusions

In this paper, we presented Rowhammer.js, an implementation of the Rowhammer attack using optimal cache eviction through memory accesses. Although implemented in JavaScript, the attack technique is independent of the specific CPU microarchitecture, programming language and runtime environment, as long as the stream of memory accesses is executed fast enough. Rowhammer.js is the first remote hardware-fault attack.

The optimal eviction strategy we used can be found for an unknown CPU by a generic algorithm we propose. The optimal eviction strategy we found improves the eviction rate from 95.2% to 99.99% on our Haswell test system compared to existing eviction strategies. This high accuracy is a crucial to exploit the Rowhammer bug without the flush instruction. Although the probability of a bit flip is significantly lower than with the flush instruction on our test machines, we still found that one machine is vulnerable to our attack with default settings.

Our attack runs in sandboxed JavaScript which is present and enabled by default in every modern browser. Therefore, it can be launched from any website. We expect that a remote attacker can gain root privileges by adapting existing Rowhammer bug exploits to our attack. Efficient countermeasures are known but seem unlikely to be practical. Additionally, we propose practical countermeasures which can be deployed in web browsers immediately to avoid remote attacks on millions of users.

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology - EUROCRYPT '97*, ser. LNCS, vol. 1233. Springer, 1997, pp. 37–51.

[2] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference*, ser. LNCS, vol. 1294. Springer, 1997, pp. 513–525.

[3] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *International Symposium on Computer Architecture – ISCA*, 2014, pp. 361–372.

[4] M. Seaborn, "Exploiting the DRAM rowhammer bug to gain kernel privileges," http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, March 2015, retrieved on June 26, 2015.

[5] XuanwuLab, "Research report on using JIT to trigger RowHammer ," https://xuanwulab.github.io/2015/06/09/Research-report-on-using-JIT-to-trigger-RowHammer/, June 2015, retrieved on July 16, 2015.

[6] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *IEEE Symposium on Security and Privacy – S&P*, 2015.

[7] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.

[8] M. Seaborn, "How physical addresses map to rows and banks in DRAM," http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-a May 2015, retrieved on July 20, 2015.

[9] K. Bains and J. Halbert, "Row hammer monitoring based on stored row hammer threshold value," Jun. 5 2014, US Patent App. 13/690,523. [Online]. Available: https://google.com/patents/US20140156923

[10] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row hammer refresh command," Jan. 2 2014, US Patent App. 13/539,415. [Online]. Available: https://google.com/patents/US20140006703

[11] R.-F. Huang, H.-Y. Yang, M. C.-T. Chao, and S.-C. Lin, "Alternate hammering test for application-specific DRAMs and an industrial case study," in *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, 2012, pp. 1012–1017.

[12] K. Park, S. Baeg, S. Wen, and R. Wong, "Active-Precharge Hammering on a Row Induced Failure in DDR3 SDRAMs under 3x nm Technology," in *Proceedings of the 2014 IEEE International Integrated Reliability Workshop Final Report (IIRW'14)*, 2014, pp. 82–85.

[13] Chrome Developers, "Native Client," https://developer.chrome.com/native-client, retrieved on June 26, 2015.

[14] D.-h. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2014.

[15] Chrome Developers, "Native Client Revision 13809," http://src.chromium.org/viewvc/native_client?revision=13809&view=revision, September 2014, retrieved on June 26, 2015.

[16] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*. USENIX Association, 2014, pp. 719–732.

[17] Zelalem Birhanu Aweke, "Rowhammer without CLFLUSH," https://groups.google.com/forum/#!topic/rowhammer-discuss/ojgTgLr4q_M, May 2015, retrieved on July 16, 2015.

[18] M. Seaborn, "cached_rowhammer.cc," https://github.com/google/rowhammer-test/blob/c1d2bd9f629281402c10bb10e52bc1 Aug. 2015.

[19] H. Wong, "Intel Ivy Bridge Cache Replacement Policy," http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, retrieved on July 16, 2015.

[20] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *IEEE Symposium on Security and Privacy – S&P*. IEEE, 2013, pp. 191–205.

[21] M. Seaborn, "L3 cache mapping on Sandy Bridge CPUs," http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpu April 2015, retrieved on June 26, 2015.

[22] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology – CRYPTO*, ser. LNCS, vol. 1109. Springer, 1996, pp. 104–113.

[23] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *Journal of Computer Security*, vol. 8, no. 2/3, pp. 141–158, 2000.

[24] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," *IACR Cryptology ePrint Archive 2002/169*. [Online]. Available: http://eprint.iacr.org/2002/169

[25] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES Implemented on Computers with Cache," in *Cryptographic Hardware and Embedded Systems – CHES*, ser. LNCS, vol. 2779. Springer, 2003, pp. 62–76.

[26] D. J. Bernstein, "Cache-Timing Attacks on AES," 2004, uRL: `http://cr.yp.to/papers.html#cachetiming`.

[27] C. Percival, "Cache Missing for Fun and Profit," 2005, uRL: http://daemonology.net/hyperthreading-considered-harmful/.

[28] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA*, ser. LNCS, vol. 3860. Springer, 2006, pp. 1–20.

[29] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *IEEE Symposium on Security and Privacy – S&P*. IEEE Computer Society, 2011, pp. 490–505.

[30] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015.

[31] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *IEEE Symposium on Security and Privacy – S&P*. IEEE Computer Society, 2015.

[32] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'15)*, July 2015.

[33] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, "Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge," Hot Chips 2012, http://hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf, retrieved on July 16, 2015.

[34] W3C, "High Resolution Time Level 2 - W3C Working Draft 21 July 2015," http://www.w3.org/TR/2015/WD-hr-time-2-20150721/#privacy-security, Jul. 2015.

[35] M. Seaborn and T. Dullien, "Test DRAM for bit flips caused by the rowhammer problem," https://github.com/google/rowhammer-test, 2015, retrieved on July 27, 2015.

[36] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[37] Lenovo, "Row Hammer Privilege Escalation," https://support.lenovo.com/us/en/product_security/row_hammer, 2015.

[38] Wikipedia, "Talk:Row hammer - Refresh rate," https://en.wikipedia.org/w/index.php?title=Talk:Row_hammer&oldid=674939880, Aug. 2015.

[39] OPSWAT, "Antivirus and Operating System Report: October 2014," https://opswat.com/resources/reports/antivirus-october-2014, retrieved on July 16, 2015.

[40] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters - CPU Hardware Performance Counters for Security," *Black Hat 2015 Briefings*, 2015, [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-G