Rowhammering: a physical approach to gaining unauthorized access

Niccolas A. Ricci
Division of Computer Science
University of Minnesota, Morris
Morris, Minnesota, USA 56267
ricc0082@morris.umn.edu

ABSTRACT

As the information density of DRAM increases, the problems faced by natural decay and cell leakage have become increasingly prevalent. As cells become more closely packed they may leak their charge into adjacent cells, changing their state, and producing memory error. Researchers attempted to intentionally produce memory error by repeatedly accessing cell rows adjacent to each other, a technique which was later labeled "rowhammering". Breakthroughs in research demonstrate working examples of a rowhammer used to exploit memory for unauthorized access. In this paper, I will describe various approaches to rowhammering, discuss potential approaches for protection, and will demonstrate some of the methods described herein.

Keywords

Rowhammer, DRAM, Rowhammer.js, Side-channel attacks, Privilege escalation

1. INTRODUCTION

Cells, known within computer science as *bits*, store all information within memory. Over time, *Dynamic Random Access Memory (DRAM)* has scaled towards greater information density, which decreases the distance between cells. Although information density is beneficial for reducing the cost-per-bit of memory [7], there are side-effects for placing cells within close proximity. As the distance between cells become less than 100 nanometers they experience the *short channel effect* [6]. The short channel effect, within memory, will cause cells to leak their charge at an accelerated rate (i.e. change their bit-state), and reduce their threshold voltage. Threshold voltage determines the voltage differential needed to create a conducting path between a *source* and a *drain terminal*. Within DRAM, a source and drain terminal would be two separate cells; with a reduced threshold voltage, two cells may experience *electromagnetic coupling*. When electromagnetic coupling occurs, the charge of coupled cells is "shared", and their charge attempts to equalize [7, 6].

Issues involving the density of DRAM have been known since the first commercial DRAM chip [7]. In 2014, Yoong Kim et al [7] showed that through frequently alternating the charge of specific memory locations, electromagnetic coupling can be used intentionally to affect the charge of adjacent cells [7]. The effect that Yoong et al demonstrate has been dubbed "rowhammer", and is a form of attack on a physical system. However, the exploits used within Yoong et al's paper are written in assembly, and use commands that a would-be attacker would not have access to [3]. Rowhammering has since been researched in more detail, and has been shown to work without the use of assembly [3, 2, 11].

In this paper, I will detail a variety of methods used to successfully execute rowhammering. I will begin with Yoong et al's approach in Section 3, followed by more applied methods of rowhammering in Sections 3.1 and 3.2. Afterwards I will discuss detected vulnerabilities in Section 3.3, leading into potential solutions to rowhammering as proposed by Yoong et al in Section 3.4.

2. BACKGROUND

2.1 DRAM

Dynamic random-access memory (DRAM) is a system of capacitors which is used to store information. Capacitors, cells, and bits, all refer to the same thing, and can be used interchangeably. A DRAM system can consist of single or multiple *channels*. Channels are the means of communication between DRAM and the *memory controller*, described in Section 2.2. The physical "stick" of memory is plugged into the motherboard through Dual Inline Memory Modules (DIMMs). Each DIMM contains ranks, placed on either side of the physical stick of DRAM. Each rank is comprised of chips, and each chip contains around 8 banks. Inside of each bank is a composition of cells structured as a two-dimensional array of cell rows [7, 3]. In order to better visualize the structural hierarchy of a DIMM see Figure 1. Each bank has *row buffer*, which is used as a temporary storage for all read and write operations. Operations within the row buffer are conducted by the memory controller, both of which are described in Section 2.2.

Information within DRAM is comprised entirely by the charge of capacitor cells, which makes it a much faster way to read and write information than a hard drive. The state

UMM CSci Senior Seminar Conference, December 2015 Morris, MN.

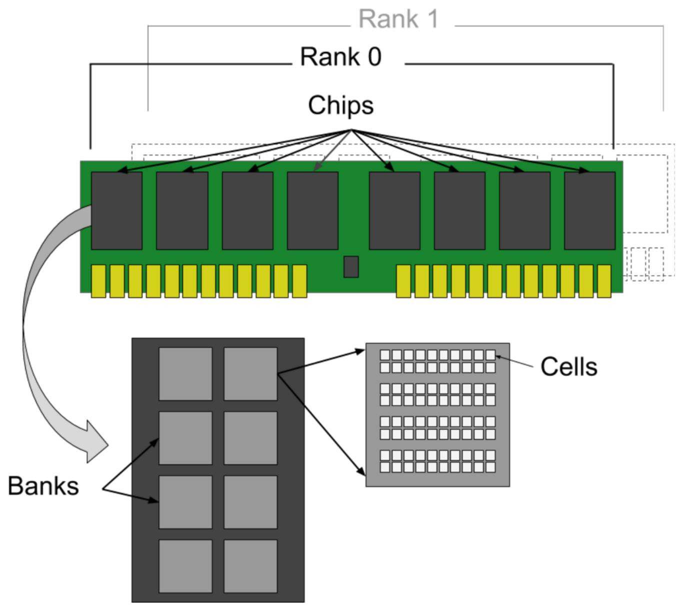

Figure 1: My graph of a DIMM. Ranks 0 and 1 are placed on opposite sides of the memory stick.

of a capacitor cell is determined by its voltage: If a cell is at half capacity or more it is in the 1 state, if a cell is at less than half capacity it is a 0. Cell charge can be thought of like a bucket full of water, where the water represents how many electrons are within a capacitor. In order to determine the charge of a row of cells, all cells within a row are collapsed into the memory buffer (the row of buckets is poured into a temporary row of buckets). The voltage (fullness) of each capacitor determines its state. Since the only moving components within DRAM are electrons, operations in memory happen within nanoseconds.

Due to how closely packed together capacitor cells are within DRAM, cells experience the short channel effect, which a cell‘s charge will leak. This is called natural decay. In order to combat this, every cell has its charge refreshed at rate greater than the rate of natural decay [7, 6, 1]. The short channel effect also reduces the voltage differential, which will cause each cells to “share” their charge with adjacent cells. If a conducting path is made between two cells, the charge of the two cells will change. If a cell‘s charge has a great enough change, its bit-state will change. In other words, the buckets have tiny holes which drain into a system of pipes leading to all other buckets. Over time, the water pressure will try to equalize within a row of buckets, making “full” buckets leak their volume into adjacent empty buckets. Thus, when measured, buckets which were meant to be full might not be, and buckets which weren’t meant to be full might be. In order to retain which buckets are supposed to be full, all buckets need to be refilled at a rate faster than they leak. Thus, to combat both natural decay and charge leakage, the *memory controller* refreshes the charge of a row of cells at a rate faster than either phenomenon [7, 6].

2.2 Memory Controller

All operations within memory are conducted by the memory controller, and take place within the memory buffer. The most fundamental function of a memory controller is to read the states of each cell, and write any changes to a cell‘s charge. Read and write operations are done as follows:

- Open a row: Transfer information to the bank‘s row-buffer (i.e. copy the cell row into the buffer).
- Read states and write changes: interpret state, encode any changes (i.e. preform any changes within the buffer, then copy the array back to the original row).
- Close row: clear buffer.

Reading cell rows is destructive; when a cell row has its charge measured, the charge of each cell is lost (or destroyed). In order to preserve the state of the cell row, each cells charge within the row is copied into the buffer, then copied back into the original cell row. Note that any changes within a row of memory are made after reading a row, and are applied within the buffer before being written to the original cell row. [7, 5]

The process of copying out and copying back in will “refresh” the charge of a cell row. In other words, the buckets full of water will be poured into a temporary row of buckets within the buffer, buckets measured to be half full or more will be filled to capacity within the buffer, and finally poured back into their original buckets. Cell rows are usually not accessed frequently enough to completely hinder the affects natural decay and cell leakage. In order to retain the states of cells, the memory controller will preform “dummy” operations on each row. In other words, the memory controller will frequently open each row and then close them, without making any changes to the copied row within the buffer.

As memory becomes more dense, the leakage will happen at an accelerated rate. Likewise, when more states within a cell row are “full”, there will be more possibilities for electromagnetic coupling. Thus, the memory controller has to refresh more frequently when more capacitor cells are full, and when cells are in closer proximity. [7, 6]

The rowhammer attack is made possible through the short channel effect phenomenon. Rowhammering, in effect, exacerbates the leakage of charge within cells to intentionally alter the state of adjacent cell rows [6].

3. ROWHAMMMERING

Manufacturers of DRAM have known of rowhammering since at least 2012. Initially, it was used as a quality assurance test for DRAM. There is cost to refreshing, so upon discovering the problem manufacturers worked to find the minimum refresh frequency necessary in order for a 0% probability of a bit being flipped by mistake as a result of the short channel effect [3]. Since the discovery of rowhammering, the question arose if it could be used to deliberately change the charge of cell rows without accessing them. Researchers Yoong Kim et al from Carnegie Mellon University demonstrated that they were able to deliberately induce bit flips in their paper *Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors.* Yoong et al show that they were able to induce “disturbance errors” (flipped bits) through their algorithm Code1a (see Algorithm 1).

Algorithm 1 Code1a

```
1: mov (X), %eax // Read from address X
2: mov (Y), %ebx // Read from address Y
3: clflush (X) // Flush cache for address X
4: clflush (Y) // Flush cache for address Y
5: mfence
6: jmp code1a.
```

Code1a will perform all actions on every data access. At steps (1) and (2) the addresses for the rows X and Y are read and filled with data. (3) and (4) run clflush, which will flush data from the *cache*. The cache is a small amount of memory that the CPU uses for efficient repeat instructions, and must be flushed in order to force the CPU to grab the next memory access from the DRAM and not the cache [10]. The mfence command at (5) is stated to ensure that the data within rows X and Y is removed. (6) shows a call to the code1a itself, which starts another iteration of the program. Yoong et al found that when they ran their algorithm on a single row no disturbance errors were produced, they labeled this algorithm as Code1b.

Algorithm 2 Code1b

```
1: mov (X)
2: clflush (X)
3: mfence
4: jmp code1a.
```

Yoong et al discovered that not only must their be two rows, X and Y must map to different rows within the same memory bank [7]. Outside of the research by Yoong et al, it was discovered that rowhammering is much more effective when the *victim* cell row is directly in between two rows that are being being hammered: the victim row X, which is the row intended to be exploited, has bits flipped more frequently when rows X+1 (the row above) and X-1 (the row below) are being hammered. This process was labeled "double-sided hammering" [2]. In order to better visualize the process, look to Figure 2. In Figure 2, as the number of activations are increased, more cells within the victim cell row will have their bit-states flipped.

Double-sided rowhammering is effective, but the effectiveness of rowhammering overall is limited by the refresh rate of the memory controller, and by the maximum number of operations that can be conducted within a set amount of time. Rowhammering happens at the time in between when the memory controller refreshes cells. So, if the memory does not allow a sufficient amount of operations in between each cell refresh, then rowhammering will not occur.

In the article *Architectural Support for Mitigating Row Hammering in DRAM Memories*, by Dae-Hyun Kim et al., the formula for a successful rowhammer is given as:

$$I_{leak-RH} = \alpha \cdot I_{leak-GB}. \tag{1}$$

Where I_{leak} is the *leakage current*: the rate at which a cell will leak its charge. GB is the *guard-band*, a scalar, or a constant multiplier, that multiplies the base memory refresh rate for extra security. The guard-band is chosen by manufacturers as a safety precaution in order to ensure cell leakage does not cause cells to lose their bit-state naturally. α is the hammering rate, which works as a scalar multiplying the natural leakage current with a guard-band: $I_{leak\text{-}GB}$. So, the guard-band (GB) reduces the rate of natural leakage to a new rate $I_{leak\text{-}GB}$, which is multiplied by the hammering rate α, to give the increased leakage of a cell under rowhammering $I_{leak\text{-}RH}$. [6]

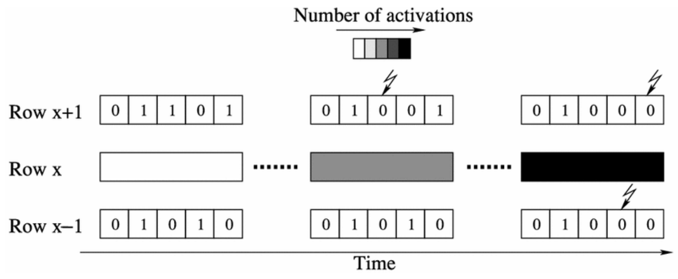

Figure 2: Double-sided hammering [6].

In order to calculate the increased leakage, we will first need the formula for the leakage current (I_{leak}), which is given as:

$$I_{leak} = \frac{Q}{t} = \frac{C \cdot V}{t} \Rightarrow C \cdot V = I_{leak} \cdot t \tag{2}$$

where C is the capacitance of a cell, i.e. the maximum capacity of a cell. V is the "driving voltage", or how much voltage is being applied to a capacitor in order to fill it to capacitance. Q represents the total charge of a capacitor when at capacitance (C · V) [1]. Finally, t represents a period of time. The maximum amount of rowhammering which can be applied over a time period t, is called the rowhammering threshold (RH_{th}). [6]

$$RH_{th} = \frac{\beta - 1}{\alpha - 1} \times M_{max}, \tag{3}$$

β represents the reduced leakage current after the guard-band is applied. α again, represents the scalar multiplier to the natural leakage current. And M_{max} is the maximum operations within the memory over the time period given. In order to better understand these formulas, consider an example of the equation:

$$RH_{th}@64\,\text{ms} = \frac{\beta - 1}{\alpha - 1} \times 1.3\ M. \tag{4}$$

This represents the rowhammering threshold within 64 milliseconds. There are 1.3 million total possible operations in this time period. α has a range in value from 4.0 to 11.7. At $\alpha = 11$, and $\beta = 2$, the rowhammering threshold will be $1/10 \times 1.3M = 130{,}000$ possible hammers. [6]

Though, as mentioned before, the maximum possible hammers is not the only factor into successful rowhammering: the location of X and Y directly affect the success rate. In the proof-of-concept testing that was conducted by Yoong et al, the researchers specifically chose where to conduct the rowhammer test. In other words, they had the knowledge of where X and Y were within a bank. By default, they were satisfying the "two rows, same bank" constraint. Whereas, to someone wanting to run rowhammering as a exploitation tool, this information will not be known. Yoong et al describe that there are techniques an attacker could use in order to learn where they are within memory, and state further that the task is for future researchers. Google Project Zero took on this task. [4]

3.1 Google Project Zero

Mark Seaborn from Google Project Zero in his blog post *Exploiting the DRAM rowhammer bug to gain kernel privileges* describes how the use of the Native Client within the Google Chrome web browser can allow a would-be attacker to gain information about where a cell row is within a bank of memory. Which means, if two rows are chosen at random within an allotted block of memory, their addresses can be compared, and if they point to the same bank then the rowhammer can be conducted successfully. [2]

The rowhammering algorithm that Seaborn uses is a modified version of Yoong et al.

Algorithm 3 Code1a (Modified)

```
1: mov (X), %eax // Read from address X
2: mov (Y), %ebx // Read from address Y
3: clflush (X) // Flush cache for address X
4: clflush (Y) // Flush cache for address Y
5: jmp code1a.
```

Within their testing, they found that the use of the procedure mfence actually reduced the amount of bits flipped, and so they removed it from their algorithm. [2]

In order to satisfy the "two rows same bank" requirement, Seaborn chose to pick memory pairs at random. In order to do so, their algorithm would need access to a memory block. So, first, they would allocate a large block of memory (1GB) then would pick pairs at random within the allocated block. Their test machine had 16 DRAM banks, made of 2 DIMMS each with 8 banks, which gave them a 1/16 chance of two rows being within the same bank if chosen at random on every iteration of Code1a. [2]

The odds of being within the same cell row are high enough even when chosen at random. As mentioned in the previous section, the chances of producing a bit flip are greatly increased by having two rows adjacent to the victim row, it was Seaborn who discovered this (see Figure 2). In some cases, double-sided hammering was necessary to induce any flip at all, which led them to find more solutions to the different row same bank constraint.

Another method for causing bit flips was explored through "escaping the sandbox" of the Native Client within chrome [2]. The Native Client works as a buffer for a browser and a machine for low level scripts. The *sandbox* allows a website to use low-level code to increase the efficiency of a web program. For the sake of security, the sandbox will run the low-level code, measure its effects, and deem them to be safe or unsafe. If deemed safe, chrome will execute the program on the machine, otherwise it will prevent any further actions. Since the sandbox allows the use of low-level code, low level procedures like clflush, and mov, can be run. Therefore, Seaborn ran his Code1a within the sandbox.

In order to find a victim cell row, Seaborn maps a large block of memory with the linux command *mmap()*. With mmap(), the entire memory block is allocated without interruptions, i.e. no other applications have memory addresses inside the mapped block. Allocations without interruptions are crucial for rowhammering, as at least three rows in sequence are necessary for double sided hammering a victim row. After the large block allocation, a random pair would be selected and hammered. If there any errors were found, a victim row has been discovered, otherwise mmap() is run again. Once a victim is exposed, careful rowhammering can be done in order to grant write access to their block of memory. The specifics of how write access is granted was not included, likely for security purposes.

Computer	Iterations	time	bit flipped?
eva01	340	2838 sec	yes
multivac	326	359 sec	yes
falcon	3979	4092 sec	yes
tang	1873	>3 hours	no
zytel	13638	>3 hours	no
reliant	1824	>3 hours	no
neoprene	1670	>3 hours	no
mylar	7541	>3 hours	no
cobar	6253	>3 hours	no
acrylic	5819	>3 hours	no
tedlar	6124	>3 hours	no
rynite	120619	>3 hours	no

Table 1: Table of Seaborn's rowhammer_test.sh run on University of Minnesota Morris machines

Seaborn has given public access to his rowhammer test on GitHub. The rowhammer test does not grant itself kernel privileges, but instead tests the vulnerability of the DRAM where it is being executed. I ran the test on twelve different machines within a computer lab of the University of Minnesota Morris. Three of the twelve machines produced bit flips. A table detailing the results of this test is shown above on Table 1.

Within the list of possible routes for conducting the rowhammer exploit, Seaborn describes how it may be possible to induce a rowhammer without low-level operations. If this is possible, Seaborn states it could be a serious issue, as bit flips could be generated on the open web with JavaScript. Three months later, Daniel Grus, Clémentine Maurice, and Stefan Mangard, published their paper *Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript*, featuring a rowhammer exploit done within Javascript.

3.2 Rowhammer.js

The first thing to note about the uniqueness of rowhammer.js is that JavaScript does not have virtual addresses or pointers. This means, for a rowhammer to be conducted within the JavaScript language, none of the techniques used by Yoong et al and Seaborn can be used; Javascript does not allow the assembly within Code1a. Instead, in order to induce a rowhammer, Daniel Gruss et al use JavaScript typed arrays running inside Firefox 39 on Linux. [3]

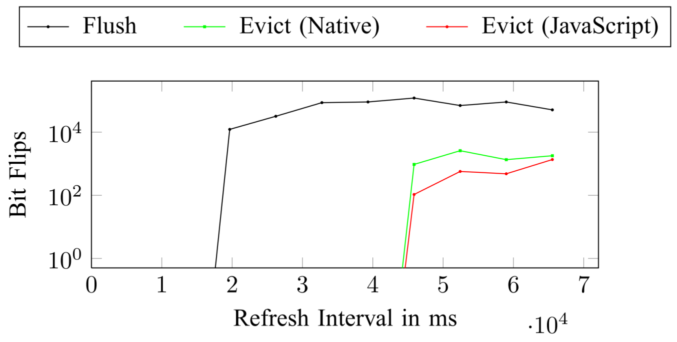

Figure 3: Figure depicting flip rates of methods within [3].

Gruss et al state how large typed arrays in Firefox are "allocated on anonymous 2MB pages" otherwise known as large pages [3]. Since JavaScript does not allow assembly, Gruss et al developed a tool to monitor the time it takes for JavaScript to allocate large pages. They use this information to determine the offset of the physical memory (e.g. the distance from the start of the memory allocation). Next, using the offset information, their program will determine which indices should be hammered. Grus et al states that in a large page the allocation will be divided into "16 row offsets of size 128kb" [3]. Not all rows within the allocation can be hammered, as they may not have two neighboring cell rows: one atop and one below. Therefore, within an offset of size 16, there will be 14 possible row offsets to be hammered. These are the only two steps necessary in order to induce a rowhammer through JavaScript. [3]

As shown in Figure 3, Gruss et al uses three variations of the rowhammer exploit: the "Flush" rowhammer emulated Code1a within Yoong et al's publication, "Evict (Native)" a rowhammer using native procedures, and "Evict (JavaScript)" the JavaScript rowhammer using the methods described above. Figure 3 illustrates that the clflush method is the best overall. However, the native and JavaScript rowhammers still produce a significant number of bit flips, and the difference between them is negligible.

Gruss et al mentioned that in future work, rowhammer.js may be able to simulate a sandbox escape akin to what Seaborn has demonstrated. The result of Gruss et al's work may be cause for alarm as it produces the first ever remote access rowhammer attack. Since it has been demonstrated that remote rowhammering is possible within the most common language used on webpages, an "[arbitrarily] large number of victim machines" could simultaneously and stealthily be hammered [3].

3.3 Vulnerabilities

Since Yoong et al's publication, many different researchers have implemented a working rowhammer through many different methods. Research into rowhammering has provided an insight into what causes some machines to be more vulnerable than others.

First and foremost, the refresh rate within the memory controller plays a crucial role to the success rate of a rowhammer [7, 6, 3, 2, 4]. Increasing the refresh rate does provide more security, however rowhammering is still possible. Seaborn states his suspicion that DRAM manufacturers have already increased the refresh rate of their memory controllers in a silent update. While conducting his research, he updated a specific brand laptop that he was testing, which resulted in a reduced speed of rowhammering [2]. Note however that the speed was only *reduced*, the rowhammering on those machines would still occur, just at a slower rate.

Mobile technology is a field in which rowhammering remains untested. Manufacturers of mobile devices place information density as one of their top priorities. These types of devices also trend towards a reduced size. As a result, mobile devices may always be a high priority target for a rowhammer attack. [8]

3.4 Potential Solutions

Six potential solutions to rowhammering are given by Yoong Kim et al.:

1. Make better chips. (Improve circuitry)

As Yoong et all state themselves, the solution does not scale into the future as we're always seeking memory to be more dense. Furthermore, this would do nothing to resolve the currently existing chips, some of which are likely to remain in use beyond the next decade.

2. Correct errors. (ECC modules)

ECC modules fix single-bit error through the use of a *parity*. Parity, in memory, is an extra chip that checks for errors during read and write operations. ECC modules are used when single bit errors are not tolerable, such as for scientific research. Unfortunately, they are expensive and are therefore not applicable to consumer products, nor the bottom-line for large companies. ECC modules *would* reduce the frequency of rowhammering, but the problem would persist despite the decreased probability of a successful attack.

3. Refresh all rows frequently.

Even doubling the refresh rate within the memory controller would not reduce the probability of a rowhammer to 0. In order to reach a 0% probability, the refresh rate would need to be increased to 8 times its current value [3, 7]. As of now, DRAM is busy refreshing 4.5% of the time, at 8 times this value the memory controller would be refreshing values 36% of the time [3, 7]. While rows are being refreshed, their values cannot be read, which means DRAM would be reduced to 64% efficiency, and as a result of constant refreshing would consume more power.

4. Retire cells (manufacturer). Before DRAM chips are sold, the manufacturer could identify victim cells and re-map them to spare cells

Yoong et al contest this solution by describing how it may take several days *per chip* to be thoroughly tested during manufacturing. Worse still, even if all victim cells were discovered, there may not be enough space to relocate them.

5. Retire cells (end-user). The end-users themselves could test the modules and employ system-level techniques for handling DRAM reliability problems.

End users cannot be expected to know – or learn – how to do this; even those who know about an existing security issue may not bother to update their BIOS. In order to illustrate this point, consider how in a study by [9], it was found that only 30% of Windows systems are up-to-date [3].

6. Identify "hot" rows and refresh neighbors. "PARA"

Yoong et al present a system titled "PARA", which could be implemented within the memory controller. Under the PARA system, each time a cell row is opened all adjacent rows have a low probability of being refreshed. Therefore, during hammering, since rows are being opened and closed repeatedly (labeled "hot" rows), the victim rows would have a high chance of also being refreshed. Although the PARA system – or similar systems – might actually work, they have not yet been implemented. PARA will need to be shown to work before it can be stated to be an actual solution.

Gruss et al considered the PARA solution, and state their critique that, like many proposed solutions, PARA could only be implemented in future DRAM chips, and would provide no benefit to the millions DRAM chips within currently existing computers.

4. CONCLUSION

The trend towards information density continues today as we move toward more applications of mobile computers, and wearable tech. For this reason, the rowhammer attack is likely to persist until a practical solution, if any, is shown to work. Although rowhammering has been known to exist for over three years, the most significant research has been conducted within the last 6 months. The research community has shown an increased interest, which may give an increased exposure to the attack. If the interest persists there will be more opportunities for new researchers to provide solutions to the issue.

Rowhammering used for exploitative purposes appears to be nascent. As the work from Seaborn shows, the bug can be used for privilege escalation on existing systems. With the advent of remote JavaScript rowhammering, malicious websites could use embedded JavaScript to induce the rowhammering exploit for privilege escalation on an arbitrarily large number of persons.

Most of the solutions being presented rely on hardware updates for machines. In the event that a preventative solution within hardware is discovered, there are millions of existing machines that will remain vulnerable until replaced. Until any solution to rowhammering is discovered, malicious use of the exploit is a cause for concern, as there are no means of protection from it.

5. ACKNOWLEDGEMENTS

Peter Dolan for working with me on the electronic engineering. Kevin Arhelger and Leonid Scott for providing insightful feedback. Thomas Hagen for being my "rubber ducky". Haley Gill for assisting in editorial work.

6. REFERENCES

[1] J. Becker. Capacitors and dielectrics (insulators) - chapter 24., 2009. [Online; accessed 7-November-2015].

[2] Google. Exploiting the DRAM rowhammer bug to gain kernel privileges, 2015. [Online; accessed 6-November-2015].

[3] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: a remote software-induced fault attack in javascript. *arXiv preprint arXiv:1507.06955*, 2015.

[4] 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, pages 1012–1017, New York, NY, USA, 2012. ACM.

[5] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. *SIGARCH Comput. Archit. News*, 36(3):39–50, June 2008.

[6] D.-H. Kim, P. Nair, and M. Qureshi. Architectural support for mitigating row hammering in dram memories. *Computer Architecture Letters*, 14(1):9–12, Jan 2015.

[7] 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. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014.

[8] A. F. Nishad Herath. These are Not Your Grand Daddy's CPU Performance Counters, 2015. [Online; accessed 8-November-2015].

[9] OPSWAT. Antivirus and Operating System Report: October 2014, 2015. [Online; accessed 8-November-2015].

[10] M. Rouse. Cache memory definition, 2014. [Online; accessed 6-November-2015].

[11] M. Seaborn. How physical addresses map to rows and banks in DRAM, 2015. [Online; accessed 6-November-2015].