BitMine: An End-to-End Tool for Detecting Rowhammer Vulnerability

Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang,
Yansong Gao, Minghua Wang, Kang Li, Surya Nepal, Yang Xiang (IEEE Fellow)

***Abstract*—Rowhammer is a destructive software-induced DRAM fault, which an attacker can leverage to break system security. Both individual customers and enterprise users (e.g., cloud providers) might refrain from using a computing system if it is vulnerable to rowhammer vulnerability.**

In this paper, we provide the first end-to-end tool, coined BitMine, that systematically assesses a DRAM chip's vulnerability to rowhammer bit flips. BitMine is an extension of DRAMDig. As DRAM address mappings are proprietary techniques and critical in inducing rowhammer bit flips, DRAMDig, our prior work, leverages domain knowledge to efficiently and deterministically reverse-engineer DRAM address mappings on Intel machines. By incorporating DRAMDig, BitMine configures three key parameters, i.e., *hammer methods*, *hammer patterns*, *data patterns*, on the effectiveness of finding rowhammer bit flips. BitMine by default implements 13 hammer methods, 4 hammer patterns and 16 data patterns and is extensible to support more. We evaluate DRAMDig and BitMine against multiple machine models that combine different DRAM chips and Intel microarchitectures. Our experiment results show that DRAMDig efficiently uncovers a deterministic DRAM address mapping for each machine model, and every implemented parameter in BitMine has its distinct effectiveness in triggering bit flips for different machine models.

***Index Terms*—Rowhammer, DRAM Address Mapping, Hammer Method, Hammer Pattern, Data Pattern.**

I. Introduction

DRAM is the main memory unit of modern computing systems and organized into rows. In 2014, Kim et al. [1] reported a software-induced DRAM fault, the so-called "rowhammer", that is, frequently accessing two DRAM rows (*aggressor row*) can cause bit flips in an adjacent row (*victim row*) even without accessing the row. The fault was soon after exploited in many rowhammer attacks (e.g., [2], [3], [4], [5], [6], [7]), posing a serious security threat to our computing systems. As such, individual and enterprise users are concerning whether their systems are susceptible to rowhammer fault.

To address their concerns, understanding how physical addresses are mapped into DRAM forms the necessary base [2], [8], [5]. Although a DRAM address mapping is available in AMD's architectural manual, it is undocumented by another major chip company, Intel. As shown in Table I, Seaborn et al. [2] were the first to uncover a mapping. They first perform a blind rowhammer test, results of which are used to uncover DRAM address mapping of a given machine. Although their methodology is intuitive and simple, the blind test is quite inefficient (within hours) and needs to be performed again if the machine setting changes (e.g., its microarchitecture and/or DRAM chips are replaced). To address the inefficiency issue, Xiao et al. [5] applied a timing channel [9] to uncover DRAM address mappings. However, their algorithm only works for old Intel architectures such as Sandy Bridge. The root cause behind their limitation is that they manually figure out bits of a physical address that index DRAM banks (we have experimented the code they shared and details are in Section VI-A). DRAMA [8] was the first to present a generic reverse-engineering algorithm that can be used in any Intel machines. To decide the bank bits, DRAMA blindly selects physical addresses, enumerates all possible combinations of certain part of the physical-address bits and verify each combination based on the aforementioned timing channel. Due to its blind selection of physical addresses, DRAMA is inefficient (within hours) and often fails to output a deterministic DRAM address mapping (we also tested their code[1] and details are in Sections VI-A and VI-B).

Reverse Engineer	Generic	Efficient	Deterministic
Seaborn et al. [2]	×	× (within hours)	√
Xiao et al. [5]	×	√ (within minutes)	√
DRAMA [8]	√	× (within hours)	×
DRAMDig [10]	√	√ **(within minutes)**	√

TABLE I: A comparison of algorithms that reverse-engineer DRAM address mappings. DRAMDig [10] is the first generic and efficient algorithm to generate deterministic DRAM address mappings. All other algorithms cannot achieve these three properties at the same time.

DRAMDig: In this paper, we revisit the limitations and advantages of the aforementioned reverse-engineering algorithms and observe that none of them made full use of domain knowledge. With this key observation, we propose a generic knowledge-assisted algorithm, DRAMDig [10], that utilizes domain knowledge (see Section IV-A) to produce a deterministic DRAM address mapping for an Intel-based machine. First, DRAMDig detects physical-address bits that index rows and columns, respectively. Second, DRAMDig

[1] https://github.com/IAIK/drama

Z. Zhang and W. He are joint first authors.
Corresponding author: Y. Gao.
Z. Zhang is with Data61, CSIRO. E-mail: zhi.zhang@data61.csiro.au.
W. He is with SKLOIS, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences. E-mail: hewei@iie.ac.cn.
Y. Cheng is with NIO. E-mail: yueqiang.cheng@nio.io.
W. Wang is with Institute of Information Engineering, CAS, China. E-mail: wangwenhao@iie.ac.cn.
Y. Gao is with School of Computer Science and Engineering, NanJing University of Science and Technology, China. E-mail: yansong.gao@njust.edu.cn.
M. Wang is with Ant Group. E-mail: minghua.wmh@antgroup.com.
K. Li is with Baidu. E-mail: kangli01@baidu.com.
S. Nepal is with Data61, CSIRO. E-mail: surya.nepal@data61.csiro.au.
Y. Xiang is with School of Software and Electrical Engineering, Swinburne University of Technology, Australia. E-mail: yxiang@swin.edu.au.

resolves the mappings from remaining bits to banks including channels, DIMMs, and ranks. Third, DRAMDig determines some bank bits that are also row bits or column bits.

DRAMDig is tested against 9 different machine models, results of which show that DRAMDig efficiently and deterministically uncovers DRAM address mapping on each model within only 7.8 minutes on average. On the uncovered mappings, we perform double-sided rowhammer tests on three different machine settings and all results show that DRAMDig is much more effective in inducing rowhammer bit flips than the other generic algorithm, DRAMA [8], justifying the correctness of DRAMDig.

BitMine: With DRAMDig as the base, BitMine is the first end-to-end rowhammer tool to evaluate a computing system's susceptibility to rowhammer. BitMine incorporates three key configurable parameters as follows that have observable effects on finding rowhammer bit flips:

- *hammer method*. A hammer method enables memory accesses to a targeted DRAM row. As CPU cores have multiple levels of caches to reduce memory access latency, a hammer method must bypass the caches, e.g., `clflush` followed by a memory load. By default, BitMine implements 13 hammer methods. We note that we have identified 5 new hammer methods in this work and the remaining 8 ones are summarized from previous works.
- *hammer pattern*. A hammer pattern is to apply a hammer method for hammering one or multiple rows. An effective hammer pattern results in frequent row activations and thus bit flips. Double-sided hammer is corroborated by previous works [11], [1] in inducing the most significant number of bit flips in DDR3-based systems, which however cannot induce a single bit flip in recent DDR4-based systems that support Hardware Target Row Refresh [12], [13] (HardTRR), an in-DRAM rowhammer mitigation technique. BitMine leverages TRRespass [14] and presents 4 hammer patterns that can induce bit flips in both DDR3 and DDR4-based systems.
- *data pattern*. A data pattern is to store specific data values into aggressor and victim rows, e.g., RowStripe [1] where rows padded with '0' are interleaved with rows padded with '1'. Different data patterns can have different effectiveness in inducing bit flips (e.g., the difference of bit-flip number can be in an order of magnitude). Besides, the most effective data pattern for a given computing system may not apply to another system. BitMine provides 16 data patterns across rows' and columns' data transformation.

BitMine is evaluated on 6 different machine models. The experimental results show that the hammer methods, hammer patterns and data patterns have different effectiveness in inducing rowhammer bit flips. Specifically, different hammer methods have distinct time costs and a single hammer method behaves differently in different machines. Besides, double-sided hammer is much more effective in causing bit flips in DDR3-based machine models in comparison with one-location hammer and single-sided hammer. In DDR4-based machine models, only many sided hammer (i.e., more than 2-sided) can induce bit flips due to HardTRR's protection. Particularly, one-location hammer does not induce a single bit flip in all our test machines. For data patterns, different machines exhibit a distinct set of effective data patterns in triggering bit flips.

In summary, we make the following key contributions:

- We present DRAMDig to reverse-engineer DRAM address mappings for different Intel-based computing systems that are equipped with either DDR3 or DDR4 chips.
- We extend DRAMDig to BitMine, the first end-to-end system tool that provides three key user-configurable parameters in effectively inducing rowhammer bit flips.
- BitMine by default implements 13 hammer methods (5 hammer methods are newly identified), 4 hammer patterns and 16 data patterns, and can support more from the three parameters.
- We conduct a comprehensive evaluation of DRAMDig and BitMine, respectively. The experimental results corroborate that DRAMDig is efficient in reverse-engineering a deterministic DRAM address mapping. Besides, each key parameter has distinct effectiveness in inducing bit flips for different machine models.

The rest of the paper is structured as follows. In Section II, we introduce the DRAM organization and rowhammer vulnerability. In Section III, we present the overview of BitMine. Sections IV and V describe DRAMDig and three key configurable parameters of BitMine. In Sections VI and VII, we present a comprehensive evaluation of DRAMDig and BitMine on different CPU microarchitectures with different DRAM types including both DDR3 and DDR4. We discuss the related works and conclude this paper in Sections VIII and IX, respectively.

II. Background

In this section, we describe modern DRAM and rowhammer.

A. Dynamic Random-Access Memory

Dynamic Random-Access Memory (DRAM) modules are produced in the form of Dual Inline Memory Module (DIMM). A DIMM is directly connected to the CPU's memory controller through one of two channels. A DIMM consists of one or two ranks, corresponding to its one or two sides. Each rank is further decomposed of multiple banks. A bank is structured as arrays of cells with rows and columns. For example, we have two 4 GiB DDR3 Samsung modules. Each module has two ranks (2 GiB each), and each rank is vertically partitioned into 8 banks, which in turn consists of 32 K rows of memory (8 KiB each). The DIMM information can be queried using the *decode-dimms* tool on Linux [15]. A cell has two types [16]. One is *true cell* where a charged state represents a bit of "1". The other is *anti cell* where a charged state is "0". When a memory access to a desired bank occurs, this "opens" a specified row by transferring all data in the row to the bank's row buffer and a specified column from the row buffer will be accessed. Any subsequent access to the same row will be served by the row buffer, while opening another row will flush the row buffer.

DRAM Refresh: The charge in the DRAM cell is not persistent and will drain over time due to charge leakage [1].

Target Objects	Privilege Boundary	Attacks
Last-Level PTEs	User-Kernel	[2], [3], [4], [6], [18]
	VM-Hypervisor	[5]
Setuid Opcodes	Inter-Process	[11]
JavaScript Objects	Intra-Process	[19], [20]
DNN Model Weights	Inter-Process	[21], [22]
Memcached Objects	Intra-Process	[23]
RSA Keys	Inter-Process	[24], [25]
	Inter-VM	[26]

TABLE II: Target objects of existing rowhammer attacks.

To prevent data loss, a periodic re-charge or refresh is required for all cells. DRAM specification dictates that the DRAM refresh interval is either 32 or 64 *ms*, during which all cells within a rank will be refreshed. A higher interval indicates better performance and thus 64 *ms* is the default one for DDR3 and DDR4.

DRAM Address Mapping: A CPU's memory controller decides how physical-address bits are mapped to a DRAM address. A DRAM address refers to a 3-tuple of *bank, row, column* (DIMM, channel, and rank are included into the *bank* tuple). As this mapping is not publicly documented on a major processor platform, i.e., Intel, Seaborn et al. [15] observed that only different rows within the same bank can induce rowhammer bit flips. Based on this observation, they made an educated guess on the DRAM address mapping of an Intel Sandy-Bridge CPU. Both DRAMA [8] and Xiao et al. [5] relied on a timing channel [9] to uncover the mapping.

B. Rowhammer

DRAM rows are vulnerable to persistent charge leakage induced by adjacent rows [1]. Specifically, frequently activating one row before the DRAM refresh comes can cause bit flips to its neighboring rows. The activated rows are called *aggressor rows* while the bit-flipped rows are called *victim rows*. Mutlu et al. [17] provides a detailed survey of the rowhammer.

If sensitive data (e.g., page tables) exist in some victim rows, the data can be corrupted by rowhammer and thus the whole system security can be compromised (e.g., privilege escalation). Since 2014, various rowhammer attacks have been proposed, which are listed Table II based on an attack target.

Control Last-Level PTEs: All rowhammer attacks that target kernel or hypervisor focus on manipulating last-level page-table-entries (PTEs), which can be massively created by attackers. Seaborn et al. [2] presented the first rowhammer exploit to compromise last-level PTEs and gain kernel privilege in bare-metal systems. Xiao et al. [5] demonstrated the first attack against host PTEs in paravirtualized Xen hypervisor and gain the hypervisor privilege.

Flip Opcodes in Setuid Binaries: Gruss et al. [11] were the first to show that gaining root privilege by compromising real-world Setuid processes (e.g., `sudo`) is possible.

Corrupt JavaScript Objects: Bosman et al. [19] leveraged memory deduplication to craft counterfeit JavaScript objects and corrupted these objects by the rowhammer, escaping the sandbox environment and gaining the browser privilege.

Flip DNN Model Weights: In 2019, Hong et al. [22] showed that single-bit corruptions in DNN model weights could greatly downgrade the inference accuracy of popular DNN models, opening up a new attack vector for exploitation as the machine learning models have been widely deployed in many sensitive scenarios.

Corrupt Memcached Objects: While all above attacks require unprivileged code execution in the target system, Andrei et al. [23] showed how to trigger and exploit bit flips in memcached objects by only sending network packets.

Leak RSA Keys: Andrew et al. [25] were the first to exploit rowhammer as a `read` side channel. In contrast to all other rowhammer attacks that break integrity of target data, they showed that rowhammer-induced data confidentiality is also a realistic threat, as they observed that data dependence between rowhammer-induced bit flips in one row and the bits of its adjacent rows could be used to deduce these bits.

III. BitMine

In this section, we discuss threat model and assumptions as well as the overview of BitMine.

A. Threat Model and Assumptions

The primary goal of BitMine is to help a user detect the rowhammer vulnerability from his DRAM-based system. In other words, the user leverages the tool to effectively trigger rowhammer bit flips. To run BitMine properly, the user is assumed to be a system administrator and own the root privilege. Without the privilege, DRAMDig will fail in reverse-engineering DRAM address mapping, which is critical for BitMine to perform effective rowhammer detection. We explain more in the following section.

B. Overview

To effectively trigger rowhammer bit flips in vulnerable rows, from a modern CPU, BitMine has four main components that have been marked in Figure 1. As shown in the figure, there are two phases that map a virtual address to a DRAM address in mainstream operating systems. In the first phase, a memory access will trigger an address translation. In response, Memory Management Unit (MMU) translates a virtual address to a physical address. For BitMine users, the virtual to physical mapping can be addressed by accessing `pagemap`, which allows a user-space process to find out the physical page a virtual page is mapped to. To mitigate rowhammer, the Linux kernel only allows root users to access `pagemap` interface since kernel version 4.0 [27]. As such, BitMine, DRAMDig in particular, requires the root privilege for acquiring the first-phase mapping.

Hammer Method: If the physical address is for a read access, a CPU core will first try to find target data/instruction by searching CPU caches. If it is for a write access, the core will search either caches or write-combining (WC) buffers (e.g., non-temporal stores [28]). A *hammer method* ensures that a memory access goes directly into the DRAM for hammer. As such, relevant CPU cache lines or WC buffers must be flushed. Specifically, flushing all-level CPU

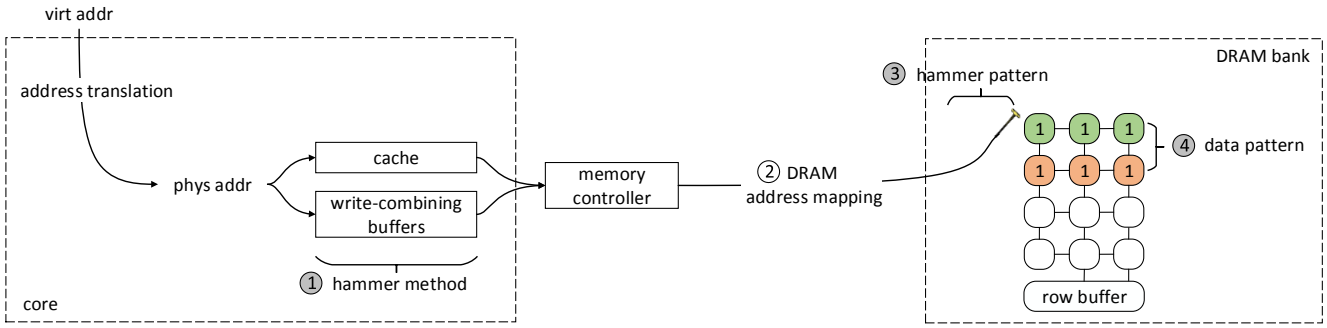


Fig. 1: BitMine has four primary components in effectively triggering rowhammer bit flips, i.e., *hammer method*, *DRAM address mapping*, *hammer pattern*, *data pattern*. We note that the DRAM address mappings are undocumented in Intel architectures and have been reverse-engineered by DRAMDig [10]. (In the DRAM bank, green ovals are in an aggressor row while orange ovals are in a victim row.)

cache lines can be achieved by either unprivileged instructions or eviction sets of physical memory lines. As eviction set-based methods are much less efficient than instruction-based methods [18], BitMine only considers available Intel instructions, i.e., `clflush` and `clflushopt`, already used in prior works [2], [3]. Regarding flushing the WC buffers, prior work [28] only reported two such Intel instructions, i.e., `movnti` and `movntdq`, which write non-temporal data directly into memory. Besides them, we have found that Intel has a string of such instructions, i.e., `movntpd`, `movntps`, `movntq`, `maskmovq` and `maskmovdqu`. Based on them, BitMine by default provides 13 efficient hammer methods, among which, 5 are newly identified in this work and 8 are summarized from previous works (see details in Section V-A).

DRAMDig: In the second phase, a physical address will be mapped by the memory controller to a DRAM address, the so-called *DRAM address mapping*. DRAMDig makes use of domain knowledge to produce a deterministic DRAM address mapping on a machine. In the next section, we talk about how DRAMDig reverse-engineers DRAM address mappings in detail.

Hammer Pattern: We note that Section II-A states that the row buffer serves frequent memory accesses, which will prevent hammering aggressor rows. On top of that, hardware-based Target Row Refresh (HardTRR) [13], [12] is designed to prevent rowhammer bit flips in present LpDDR4 and DDR4 chips. HardTRR identifies possible victim rows by counting rows activations and refreshes rows to suppress bit flips when the counter reaches a predefined threshold. A *hammer pattern* will clear the row buffer and use a selected hammer method for each aggressor row. In our implementation, BitMine provides four hammer patterns that all bypass row buffer and hammer one or multiple aggressor rows. One of them bypasses HardTRR and thus works in DDR4 chips (see details in Section V-B).

Data Pattern: Kim et al. [1] observe that whether a cell in a victim row flips has a dependency on data values stored in victim rows and their adjacent aggressor rows (i.e., *data pattern*), indicating that an individual victim row might not be bit-flipped due to an ineffective data pattern. As such, BitMine provides 16 data patterns where data are organized in terms of rows or columns in order to find as many bit flips as possible (see details in Section V-C).

IV. DRAMDig

As shown in Figure 2, DRAMDig consists of three main steps. In *step 1*, we perform row and column bits detection and produce a coarse-grained results; that is, most row and column bits are uncovered while bits in green boxes are still covered. In *step 2*, we carefully select physical addresses that only differ in bits shown in green boxes, then partition those addresses into different piles (addresses in each pile share the same bank), and identify bank address functions that can work for all piles. In *step 3*, we perform a fine-grained analysis on the resolved bank address functions to detect row or column bits that also play a role in the bank address functions as shared bits (see lined boxes in Figure 2). Note that in each step we need domain knowledge.

In the following paragraphs, we first introduce domain knowledge and a timing primitive, and then talk about the three steps, respectively.

A. Domain Knowledge

We obtain the domain knowledge from the following three sources.

- *Specifications*. From the specifications of DDR3, DDR4 and LPDDR4 [29], [12], [13], we can collect physical-address bits that index banks, rows and columns for a given DRAM chip.
- *System Information*. The Linux system information includes the total number of banks, the size of physical memory, and whether DRAM chips support Error Code Correction (ECC) [30]. We get this information from the output of two Linux bash commands, i.e., *decode-dimms* and *dmidecode*.
- *Empirical observations*. We summarize two key empirical observations from previous works [2], [5], [8]. First, a bank address function on Intel microarchitecture is a tuple of multiple physical address bits, which are XORed to output

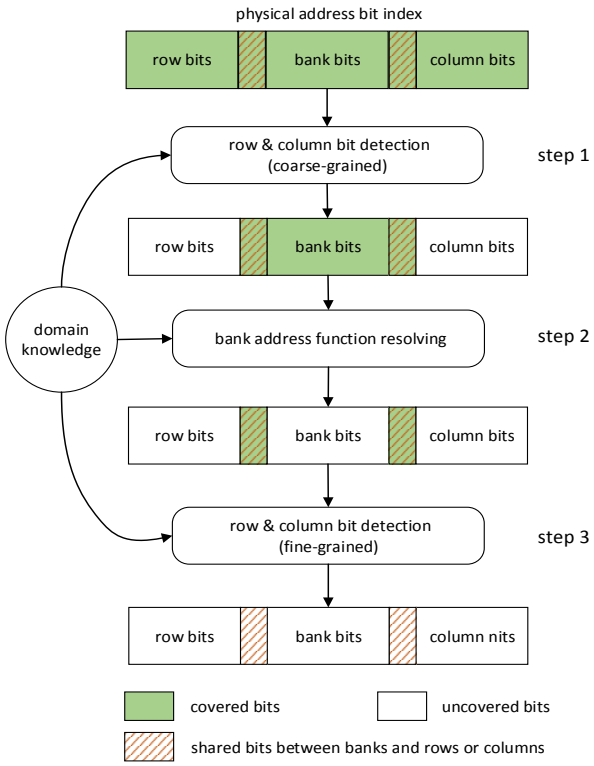


Fig. 2: DRAMDig Workflow.

a single bit. Second, since Ivy Bridge, the lowest bit of a bank address function that owns the most number of bits is not the column bit. For example, there are five bank address functions for M6 shown in Table III of the paper. One of them has the most number of physical-address bits, that is, (8, 9, 12, 13, 18, 19). Based on this observation, this function's first bit (i.e., 8) is not a column bit.

B. A Timing Primitive

We resort to a timing channel [9] to reverse-engineer the DRAM mappings. Specifically, this timing channel is caused by the row-buffer conflicts within the same DRAM bank. As mentioned in Section II, each bank has a row buffer that caches the last accessed row. If a pair of addresses reside in two different rows of the same bank and they are accessed alternately, the row buffer will be repeatedly cleared and reloaded. This causes the so-called row-buffer conflicts. Clearly, row buffer conflicts can lead to higher latency in accessing the two addresses than the case where they are either within the same row or in different banks. As such, we can distinguish whether two addresses are in different rows within the same bank.

C. Row & Column Bit Detection (Coarse-Grained)

We first partition the physical address bits into row, column and bank bits at a coarse-grained level. We use the same approach as the work [5]. Specifically, for row bits, we measure access latency for two physical addresses that have only one bit different. If the latency is high, that means those addresses reside within the same bank but different rows (SBDR), and that bit is the only different bit for the two addresses. Thus, this bit is a row bit.

For column bits, we select two physical addresses with only two different bits. One bit is a detected row bit and the other is a non-row bit. If we get high latency for those two addresses, the non-row bit is a column bit. Given that the two addresses are in the same bank and neither of the two bits is a bank bit, the non-row bit is thus a column bit that determines which column a physical address is mapped to. After finding out row and column bits using the above approaches, we consider remaining bits as bank bits.

Note that the bits detection results in this step are coarse-grained. The row and column bits that have been detected are to index rows and columns. There can be other row bits or column bits that also index banks, and they are not detected in this step. We will discuss how to identify them in section IV-E.

D. Bank Address Function Resolving

We uncover bank address functions based on the above coarse-grained detection results. We have three steps. First, we enumerate a number of possible address ranges based on the bank bits, and select physical addresses within those ranges by allocating a large memory region. Second, we partition the selected addresses into #bank number of piles (#bank can be known from the *System Information* in Section IV-A). Last, we detect the bank address functions by analyzing the address piles. We discuss the three phases in detail as follows.

Physical Address Selection: The main idea is to only select the physical addresses within specified ranges. The ranges reflect all possible values of all bank bits, and thus the selected addresses contain all the bank address functions. As the number of bank bits is determined through coarse-grained detection, the number of selected addresses can be determined. Algorithm 1 presents how we perform physical address selection.

The algorithm first calculates a $range\_mask$, which indicates bank bits positions, and selects the physical pages that cover that range (line 7-15). In order not to miss any bank bits, we require that the selected physical pages are consecutive, and if there are some pages missed in $phys\_pages$, we try again. The last found physical page range is presented by $[P\_start, P\_end]$.

Note that we use b_min and b_max to calculate range mask, but not all the bits in $[b\_min, b\_max]$ will be used in the bank address functions. We use "$miss\_mask$" to represent those bits that have nothing to do with the address functions such that we set the bits to 1. Next, for every investigated address p with $[P\_start, P\_end]$, we make it first mask against $miss\_mask$ and then check whether the new address p' is within $phys\_pages$. If so, we add it to $phys\_pool$, which holds final selected addresses. With "$miss\_mask$", it enables us to only focus on the reasonable number of addresses that does matter in determining the address functions.

Physical Address Partition: We apply Algorithm 2 to partition the selected addresses into #bank numbers of piles.

Algorithm 1 Physical Address Selection

Input: $phys\_pages$: allocated memory pages; B: possible bank bits;
Output: $phys\_pool$

1: $b\_min, b\_max = find\_min\_max(B)$
2: $range\_mask = (1 << (b\_max + 1)) - (1<<b\_min)$
3: $miss\_mask = 0$
4: **for** $b \in [b\_min, b\_max]$ && $b \notin B$ **do**
5: $miss\_mask+=(1 << b)$
6: **end for**
7: **for** $p \in phys\_pages$ **do**
8: **if** $(p\&range\_mask) == range\_mask$ **then**
9: $P\_start = p - range\_mask$
10: $P\_end = p + PAGE\_SIZE$
11: **if** !page_miss($phys\_pages$, P_start, P_end) **then**
12: break
13: **end if**
14: **end if**
15: **end for**
16: $phys\_pool = \{\}$
17: **for** $p = P\_start$; $p < P\_end$; $p+ = (1 << b\_min)$ **do**
18: p' = $p \mid miss\_mask$
19: **if** $page(p') \in phys\_pages$ **then**
20: $phys\_pool.add(p')$
21: **end if**
22: **end for**
23: **return** $phys\_pool$

First, we randomly select one address p from $phys\_pool$ and measure the latency with every other address. If the latency is high, it means we have one address that is SBDR with p. We put it into $piles[p]$ which stores addresses from $phys\_pool$ that are SBDR with p (line 4-9). By doing so, we can collect all addresses from $phys\_pool$ that are SBDR with p and record them in $piles[p]$. Next we verify whether the number of addresses in $piles[p]$ is within a valid range. If so, we consider this round of partition is successful, and then remove all the addresses in $piles[p]$ from $phys\_pool$ and conduct next round of partition (line 10-12). Finally, it stops when enough addresses have been partitioned (line 13-15).

Algorithm 2 Physical Address Partition

Input: $phys\_pool$: selected physical addresses
Output: $piles$: a map $< k, v >$, of which k is an address and v is the addresses that are SBDR with k.

1: $pool\_sz$ = $phys\_pool.size()$
2: $pile\_sz$ = $pool\_sz/\#bank$
3: **while** True **do**
4: randomly select p from $phys\_pool$
5: **for** $p' \in phys\_pool - \{p\}$ **do**
6: **if** $latency(p, p') == high$ **then**
7: $piles[p].add(p')$
8: **end if**
9: **end for**
10: **if** $1 - \delta \leq piles[p].size()/pile\_sz \leq 1 + \delta$ **then**
11: $phys\_pool = phys\_pool - \{piles[p]\} - \{p\}$
12: **end if**
13: **if** $phys\_pool.size() > per\_threshold * pool\_sz$ **then**
14: break
15: **end if**
16: **end while**
17: **return** $piles$

Ideally, all the addresses in $phys\_pool$ will be partitioned into #bank number of piles with each pile having the same number of addresses. However, in practice the partition may be influenced by noises introduced by incorrect results of latency measurement, so it is possible that not all of piles have the same number of addresses, and also there may be some addresses that are not partitioned into any pile. That's why we introduce δ and $per\_threshold$ and they can be adjusted in practice. Empirically, we set δ to 0.2 and $per\_threshold$ to 85% and then the addresses can be successfully partitioned into #bank number of piles.

Bank Address Function Detection: We utilize Algorithm 3 to present how to detect bank address functions based on the address piles obtained.

Algorithm 3 Bank Address Function Detection.

Input: $piles$: a map $< k, v >$, of which k is an address and v is the addresses that have SBDR with k; B: bank bits
Output: $bank\_funcs$: the set that stores bank address functions

1: xor_masks = gen_xor_masks(B)
2: $bank\_funcs$ = { }
3: **for** $pile \in piles$ **do**
4: $func\_set$ = { }
5: **for** $mask \in xor\_masks$ **do**
6: **if** apply_xor_mask_to_pile($mask$, $pile$) **then**
7: $func\_set$.insert($mask$)
8: **end if**
9: **end for**
10: $bank\_funcs$ = $bank\_funcs \cap func\_set$
11: **end for**
12: prioritize($bank\_funcs$)
13: remove_redundant($bank\_funcs$)
14: check_numbering($bank\_funcs$, $piles$)
15: **return** $bank\_funcs$

According to *Empirical Observation* in Section IV-A, bank address functions take some bank bits as input and output `XORed` values from those bank bits. Since we have grouped #bank number of piles and the addresses in each pile map to the same bank, we look into each pile and try all combinations of bank bits and apply each of them to the addresses in the pile. We look into the combination starting from one bit to the number of bank bits (line 1). If a combination of bank bits has the same XORed result for all the addresses in the pile, we consider it as a possible bank address function. After investigating all the piles, we can have all possible bank address functions (line 3-11).

However, some address functions are just linear combinations of the others so they are not the actual address functions and need to be removed. We consider the functions that have fewer bits have higher priority and remove the lower one if it is the linear combinations of higher priority functions. For instance, if (14, 18), (15, 19) and (14, 15, 18, 19) are 3 bank address functions. We consider the previous two have higher priority than the third, because the third is the linear combination of the previous two and we consider it as redundant (line 12-13).

Apart from that, the number of bank address functions should be $log_2(\#bank)$. There may be more than that number of functions after removing the redundant, and they are not actual address functions. So we test every combination of $log_2(\#bank)$ number of functions by considering them as bank address functions and using them to number the address

piles. The actual bank address functions can count those piles from 0 to $\#bank - 1$ (line 14).

E. Row & Column Bit Detection (Fine-Grained)

As discussed in section IV-C, we need to determine the row bits and column bits that are also bank bits. From the *Specifications* in Section IV-A, we know the exact number of row and column bits for a specific DRAM chip. As we have detected some row bits and column bits, we can determine how many row bits and column bits that are left to be uncovered.

For the remaining covered row bits, we start to investigate the bank address functions that consist of two bits. We select two physical address with only those two bits different and measure their latency. The two addresses actually map to the same bank. If the latency is actually high, it means either one bit is a row bit. We consider the higher one as the row bits as discussed in [15], [5]. If there are also row bits still covered after investigating two-bit address functions, we proceed to investigate address the bank functions that have more bits. In practice, we have not seen any case that needs to investigate the address functions that have three or more bits.

For the remaining covered column bits, we first check the number of remaining column bits that need uncovered, and then identify those bits that have not been identified as column bits in coarse-grained detection, denoted as C. Next, according to the *Empirical Observation* in Section IV-A, we know the lowest bit (denoted as l) of the function occupying the most number of bits is not a column bit. So we investigate the bits $\{C - l\}$, by the order from low to high, and consider the first requested number of bits as column bits.

V. Key Configurable Parameters

In this section, we describe 13 hammer methods, 4 hammer patterns, and 16 data patterns in detail.

A. Hammer Methods

Based on the Intel manual [31], `clflush` is applicable in all our test microarchitectures ranging from Intel Sandy Bridge to Coffee Lake shown in Table III. `clflushopt` only works since Skylake but costs less CPU cycles than `clflush`. As shown in Listings 1 and 2, `X` and `Y` represent two distinct virtual addresses. The first two instructions are for flushing relevant cache lines and thus the two `mov` instructions enable two DRAM read or write accesses to X and Y, respectively.

Unlike the cache-flush instructions above, non-temporal store instructions must be combined with either cached read or write memory accesses to flush the WC buffers [28]. For example, Listings 3 and 4 show the `movnti` and `movntdq`-based hammer methods, respectively.

```
1  mov (X), %eax     |     mov %eax, (X)
2  clflush (X)       |     clflush (X)
3  mov (Y), %eax     |     mov %eax, (Y)
4  clflush (Y)       |     clflush (Y)
5  mfence            |     mfence
```

Listing 1: clflush + read / write

```
1  mov (X), %eax     |     mov %eax, (X)
2  clflushopt (X)    |     clflushopt (X)
3  mov (Y), %eax     |     mov %eax, (Y)
4  clflushopt (Y)    |     clflushopt (Y)
5  mfence            |     mfence
```

Listing 2: clflushopt + read / write

```
1  movnti %eax, (X)    |     movnti %eax, (X)
2  mov (X), %eax       |     mov %eax, (X)
3  movnti %eax, (Y)    |     movnti %eax, (Y)
4  mov (Y), %eax       |     mov %eax, (Y)
5  mfence              |     mfence
```

Listing 3: movnti + read / write

```
1  movntdq %xmm0, (X)  |   movntdq %xmm0, (X)
2  mov (X), %eax       |   mov %eax, (X)
3  movntdq %xmm0, (Y)  |   movntdq %xmm0, (Y)
4  mov (Y), %eax       |   mov %eax, (Y)
5  mfence              |   mfence
```

Listing 4: movntdq + read / write

B. Hammer Pattern

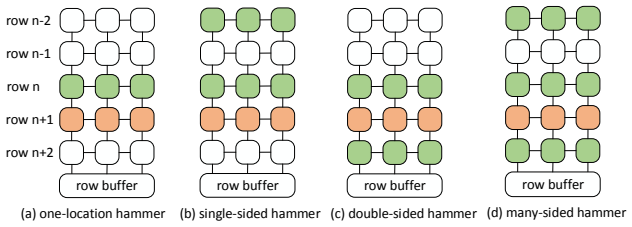


Fig. 3: Four efficient hammer patterns that bypass row buffer and hammer one or multiple rows in the same DRAM bank. (A row of green ovals is the aggressor row and a row of orange ovals is the victim row.)

Based on previous works [32], [14], BitMine has four hammer patterns shown in Figure 3, which are summarized as follows.

One-location Hammer: This pattern [11] randomly selects one single row for hammer without knowledge about virtual-to-DRAM mappings, as illustrated in Figure 3(a). It only applies to certain computing systems where advanced memory controllers employ a more sophisticated policy (e.g., closed row or adaptive page) to optimize performance, that is, pre-emptively closing accessed rows earlier than necessary and thus flushing the row buffer.

Single-sided Hammer: Intuitively, performing access to no less than two DRAM rows within the same bank causes row-buffer conflicts and thus clears the row buffer. As such, single-sided hammer randomly picks two aggressor rows for hammer, with the hope that one aggressor row is adjacent to a targeted victim row, as illustrated in Figure 3(b). The probability of the hope is decided by the total number of rows and it can be significantly improved if many rows are hammered at the same time.

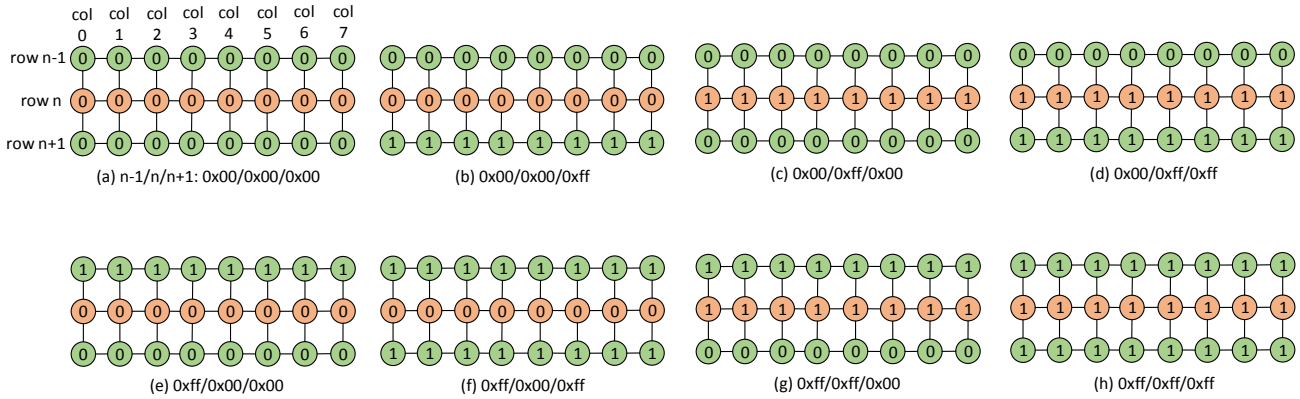


Fig. 4: 8 data patterns in row transformation for double-sided hammer. Every single bit in each row is the same and the value of each row is represented in hexadecimal.

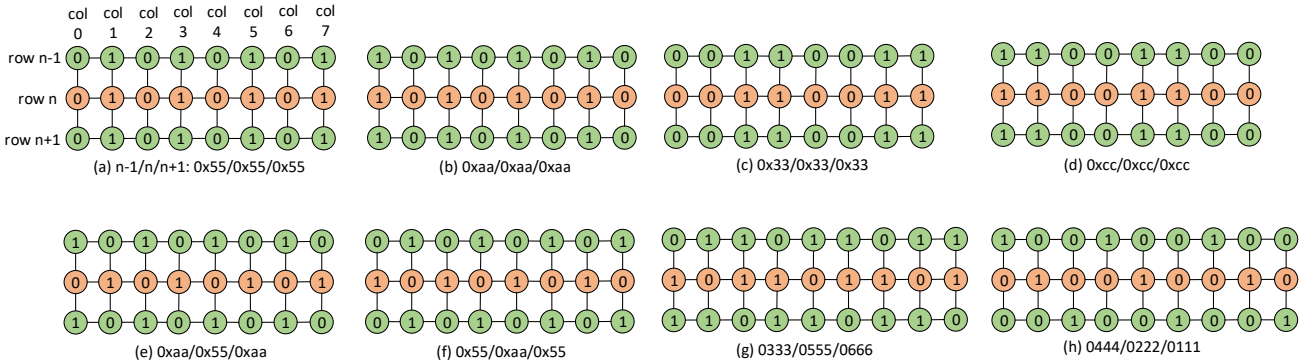


Fig. 5: 8 data patterns in column transformation for double-sided hammer. For (a)-(c), every single bit in each column for the three rows is the same. For (e)-(f), bit values in the aggressor rows are the same while the victim row holds the opposite bit values. For (g)-(h), their pattern in each row changes for each 3 consecutive bits and thus they are denoted using octal.

Double-sided Hammer: If two adjacent rows of a victim row are hammered alternately, as illustrated in Figure 3(c), the victim row is mostly likely to be bit-flipped compared to the above two hammer patterns. This hammer pattern needs a complete mapping from virtual addresses to DRAM addresses to position aggressor rows.

Many-sided Hammer: In this pattern, more than two aggressor rows within the same bank are hammered one after another, as illustrated in Figure 3(d). In present LpDDR4 and DDR4 chips, HardTRR was believed to eliminate the rowhammer vulnerability and none of the above hammer patterns can counteract it. TRRespass [14] proposed this new hammer pattern to defeat HardTRR and trigger bit flips again even in recent DRAM chips.

C. Data Pattern

For each hammer pattern, we have 16 data patterns to store corresponding data values in the aggressor and victim rows. For instance, Figures 4 and 5 show the data patterns for double-sided hammer. For data that transform in rows, we have 8 patterns as shown in Figure 4 based on previous works [1], [33]. For data that transform in columns, we also have 8 patterns as shown in Figure 5, inspired by [1], [34].

The reasons why we need both data transformation in rows and columns are two-fold. First, DRAM cells have an intrinsic property. If a cell uses a charged state to represent a value of "1", it is called *true-cell*. For a cell that represent a value of "1" with a discharged state, we call it *anti-cell*. As every cell in a victim row might lose charge due to rowhammer effect, we are not sure whether the cell is true-cell or anti-cell. To uncover bit flips of both true-cell and anti-cell, we store bit "1" or "0" in the cell in different data patterns. Second, we would like to figure out whether a bit flip of a victim cell is correlated with data stored in some other cells. These cells can reside in the neighboring rows of the victim row or other columns of the victim row.

VI. DRAMDig Evaluation

In this section, we use DRAMDig to uncover DRAM address mappings in 9 different machines, and compare DRAMDig against the other generic reverse-engineering algorithm (i.e., DRAMA [8]) in terms of performance and induced bit-flip number.

Machine No.	Microarch.	DRAM		Bank Address Functions	Row Bits	Column Bits
		Type, Size	Config.			
M1	Sandy Bridge i5-2400	DDR3, 8GiB	2, 1, 1, 8	(6), (14, 17), (15, 18), (16, 19)	17∼32	0∼5, 7∼13
M2	Ivy Bridge i5-3230M	DDR3, 8GiB	2, 1, 2, 8	(14, 18), (15, 19), (16, 20), (17, 21), (7, 8, 9, 12, 13, 18, 19)	18∼32	0∼6, 8∼13
M3	Ivy Bridge i5-3230M	DDR3, 4GiB	1, 1, 2, 8	(13, 17), (14, 18), (15, 19), (16, 20)	17∼31	0∼12
M4	Haswell i5-4210U	DDR3, 4GiB	1, 1, 1, 8	(13, 16), (14, 17), (15, 18)	16∼31	0∼12
M5	Haswell i7-4790	DDR3, 16GiB	2, 1, 2, 8	(14, 18), (15, 19), (16, 20), (17, 21), (7, 8, 9, 12, 13, 18, 19)	18∼32	0∼6, 8∼13
M6	Skylake i5-6600	DDR4, 16GiB	2, 1, 2, 16	(7, 14), (15, 19), (16, 20), (17, 21), (18, 22), (8, 9, 12, 13, 18, 19)	19∼33	0∼7, 9∼13
M7	Skylake i5-6200U	DDR4, 4GiB	1, 1, 1, 8	(6, 13), (14, 16), (15, 17)	16∼31	0∼12
M8	Coffee Lake i5-9400	DDR4, 8GiB	1, 1, 1, 16	(6 13), (14 17), (15 18), (16, 19)	17∼32	0∼12
M9	Coffe Lake i5-9400	DDR4, 16GiB	2, 1, 2, 16	(7, 14), (15, 19), (16, 20), (17, 21), (18, 22), (8, 9, 12, 13, 18, 19)	19∼33	0∼7, 9∼13

TABLE III: Uncovered DRAM Mappings on 9 different machine settings. (The **Config.** column presents a specific DRAM configuration in a quadruple: channel (#), DIMMs (#) per channel, ranks (#) per DIMM, banks (#) per rank.)

A. Uncovered DRAM Address Mappings

DRAMDig has successfully uncovered the DRAM mappings including row and column bits and bank address functions for all 9 different machine settings as shown in Table III. Machines in the table are all running Linux systems with different combinations of Intel microarchitectures and DRAM chips including DDR3 and DDR4. From the table, we see that DRAMDig uncovers bank address functions not only for common CPU microarchitectures such as Haswell, Sandy and Ivy Bridge, but also a much newer CPU architecture, i.e., Coffee Lake, which has never been discussed in previous works [5], [8]. Besides the bank address functions, row and column bits are also uncovered, including the shared bits between banks and rows or columns, which indicates that DRAMDig can uncover interleaved-mode DRAM address mappings. On top of that, we note that the column bits are not consecutive in some machine settings and they are uncovered based on our *Empirical Observation* in Section IV-A. Take the M2 as an example, there are five uncovered bank address functions while one that has the most number of physical-address bits is (7, 8, 9, 12, 13, 18, 19). As such, this function's first bit (i.e., 7) is not a column bit.

Distinguishing DRAMDig: When executing the code that Xiao el al. [5] shared with us, we found that it could not work on more than half of the microarchitectures shown in Table III, which makes itself only applicable to limited microarchitectures. Take M6 machine setting as an instance, DRAMDig uncovered 6 bank address functions. However, the code from Xiao el al. was stuck after (16, 20), (17, 21), (18, 22) had been uncovered.

For DRAMA [8], we ran their code for multiple times and found that it generated different DRAM mappings most of the time. As DRAMA is the first generic reverse-engineering algorithm while both Seaborn's [2] and Xiao's [5] algorithms are not generic and limited to one or multiple machine settings, we further compare our work against DRAMA in terms of performance overhead and induced bit-flip number in the following sections.

B. A comparison of DRAMDig and DRAMA

Performance Overhead: We present the respective performance costs of DRAMDig and DRAMA [8] shown in Fig. 6. DRAMDig can be completed from 69 seconds to 17 minutes in all machines settings (only 7.8 minutes on average). In comparison, DRAMA spent from almost 500 seconds to 2 hours, indicating a much higher time cost than DRAMDig. Particularly in M3 and M7 settings, it cost roughly two hours without producing any results before we killed it.

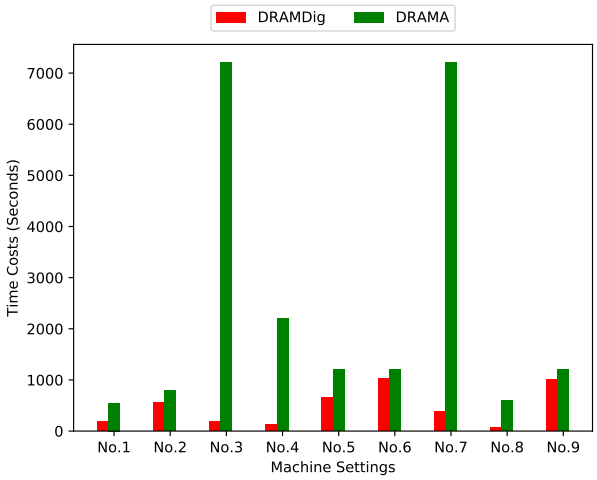


Fig. 6: Time costs for DRAMDig and DRAMA to uncover DRAM mappings on all machine settings in Table III. Overall, DRAMDig requires much less time than DRAMA.

For DRAMDig, most of its time cost comes from the physical address partition, which is heavily affected by the number of selected physical addresses. A higher number of selected addresses requires more measurements for access latency, thus making the partition procedure cost more time. Specifically, DRAMDig in M6 and M9 settings selected the highest number of physical addresses (almost 16,000), making itself timing-consuming. In contrast, it selected the least number (about 4000) in M8 setting, thus making itself time-saving.

Test Machines	Model	Microarch.	DRAM				
			Type	Manufacturer	Size	# Banks	Part Number
T1	Lenovo ThinkPad X230	i5-3230M Ivy Bridge	DDR3	Samsung	8GB	32	M471B5273DH0-CH9
T2	HP EliteBook 820	i5-4200U Haswell	DDR3	Samsung	4GB	16	M471B5273DH0-CK0
T3	Lenovo ThinkStation P300	i3-4130 Haswell	DDR3	Hynix	4GB	16	HMT351U6CFR8C-H9
T4	Lenovo ThinkStation P300	i3-4130 Haswell	DDR3	G.Skill	4GB	8	F3-14900CL9-4GBSR
T5	Gigabyte Z170G Baseboard	i7-6700K Skylake	DDR4	Kingston	8GiB	32	99P5701-005.A00G
T6	Asus B250M-K Baseboard	i7-7700K Kaby Lake	DDR4	Kingston	8GiB	32	99P5701-005.A00G

TABLE IV: Machine configurations for BitMine evaluation.

Induced Bit-flip Number: We select three machine settings from Table III, i.e., M1, M2 and M5 and use the DRAM address mappings uncovered by DRAMDig and DRAMA respectively to induce rowhammer bit flips. In this experiment, BitMine is configured to specify a hammer method of `clflush+read`, a hammer pattern of double-sided hammer and a data pattern of 0xff/0x00/0xff in Figure 4. For each machine setting, we run BitMine for 5 times and each run lasts 5 minutes. The results are shown in Table V. R1-R5 represent 5 runs and the bit-flip numbers of each run are displayed as DRAMDig/DRAMA. In particular, we can see from the row of **Total** that DRAMDig has induced significantly more bit flips than DRAMA while DRAMA even failed to induce any bit flips during some runs in M2 and M5 settings. As a result, the experiment results justify the correctness of the DRAM address mappings uncovered by DRAMDig.

Run No.	Machine No.		
	M1	M2	M5
	#DRAMDig/#DRAMA		
R1	296/197	959/240	12/7
R2	418/179	934/0	12/0
R3	226/177	976/18	12/0
R4	627/259	1039/947	10/0
R5	484/286	955/670	11/0
Total	2051/1098	4863/1875	57/7

TABLE V: The number of bit flips induced by DRAMDig and DRAMA. Both perform 5 runs on three selected machine settings in Table III where each run lasts 5 minutes. DRAMDig induced significantly more bit flips than DRAMA.

VII. BitMine Evaluation

In this section, we evaluate BitMine on the effectiveness of inducing bit flips by combining DRAMDig and the aforementioned configurable parameters. Our experiment settings are shown in Table IV. In particular, we have 2 DDR3-based laptops (i.e., T1 and T2), 1 DDR3-based desktop and 2 DDR4-based desktops. For each machine, we run BitMine for 2 days.

Specifically, BitMine applies a hammer pattern of double-sided with 11 hammer methods and 16 data patterns against all DDR3-based machines since `clflushopt` only supports Skylake or newer microachitectures. For comparison, single-sided and one-location hammer are used respectively in T1 test machine with 11 hammer methods but 12 data patterns (4 data patterns, i.e., Figures 4(b)(d)(e)(g) are removed as both hammer patterns have only one aggressor row.). Regarding DDR4-based machines, many-sided hammer with 13 hammer methods and 12 data patterns is fed into BitMine as other three hammer patterns cannot induce bit flips in the presence of Hardware Target Row Refresh (HardTRR). We leverage TRRespass [14] to figure out the best hammer pattern, i.e., triple-sided hammer that shows the best effectiveness in defeating HardTRR. As aggressor rows in data patterns of Figures 4(b)(d)(e)(g) have different data values, they are also removed in the triple-sided hammer. In the following sections, we show the experimental results in terms of hammer pattern, hammer method and data pattern, respectively.

A. Hammer Method

To evaluate the hammer efficiency of different hammer methods, we measure the CPU cycles that each method takes to complete one hammer against one aggressor row in the same data pattern (i.e., Figure 5(a)) and thus produce Figure 7 and Figure 8. As shown in the figures, different hammer methods have different costs in the same machine while the same method behaves differently in different machines. Among the evaluated hammer methods, `clflush+read`, `clflushopt+read`, `movnti+write`, `movntdq+write` and all the newly identified 5 hammer methods exhibit lower costs.

On top of that, we conduct double-sided hammer on T1–T4 test machines and triple-sided hammer on T5–T6, respectively, using every hammer method and one same data pattern (i.e., Fig.4(c)), the experimental results of which are shown in Figure 9 and Figure 10. It can be seen from the figures that T1 is the most vulnerable among all test machines. T5 and T6, two DDR4-based machines, are vulnerable to almost only two hammer methods, i.e., `clflush+read` and `clflushopt+read`. We note that the `movntpd+write` and `maskmovdqu+write`-based hammer methods trigger 1 and 2 bit flips on T6, respectively, which are negligible.

B. Hammer Pattern

Figure 11 shows the bit flips of T1 and T6 machines caused by different hammer patterns under different data patterns. We choose `clflush+read` as the hammer method, which is one of the most effective ones discussed in Section VII-A. Triple-sided hammer and the other three hammer patterns are evaluated on T6 and T1, respectively.

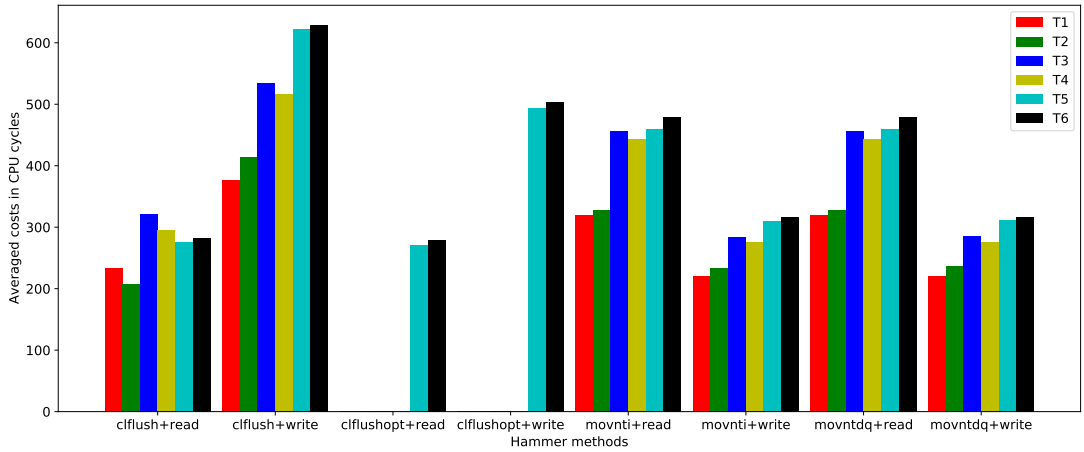


Fig. 7: Averaged costs in CPU cycles per hammer for different hammer methods with the same data pattern. We note that `clflushopt`-based hammer methods are only available in T5 and T6 test machines.

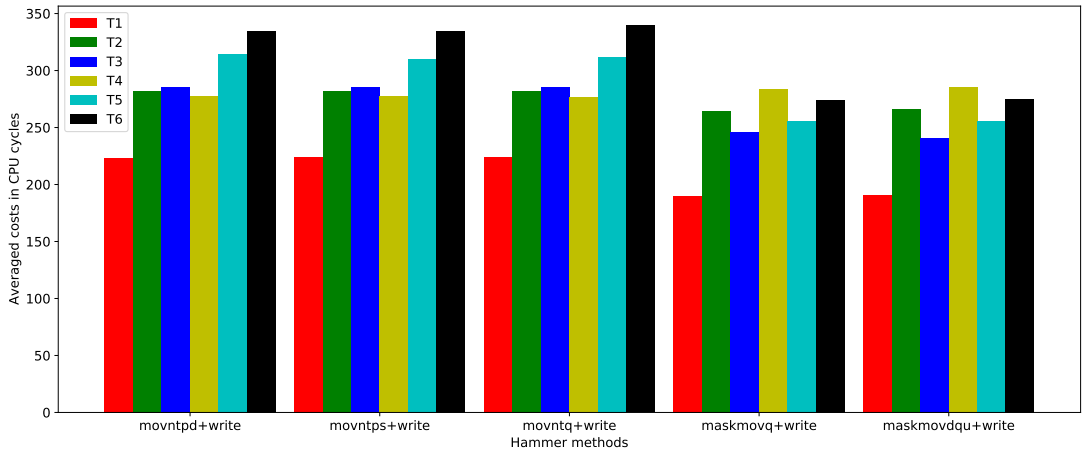


Fig. 8: Averaged costs in CPU cycles per hammer for newly identified non-temporal-store based hammer methods with the same data pattern.

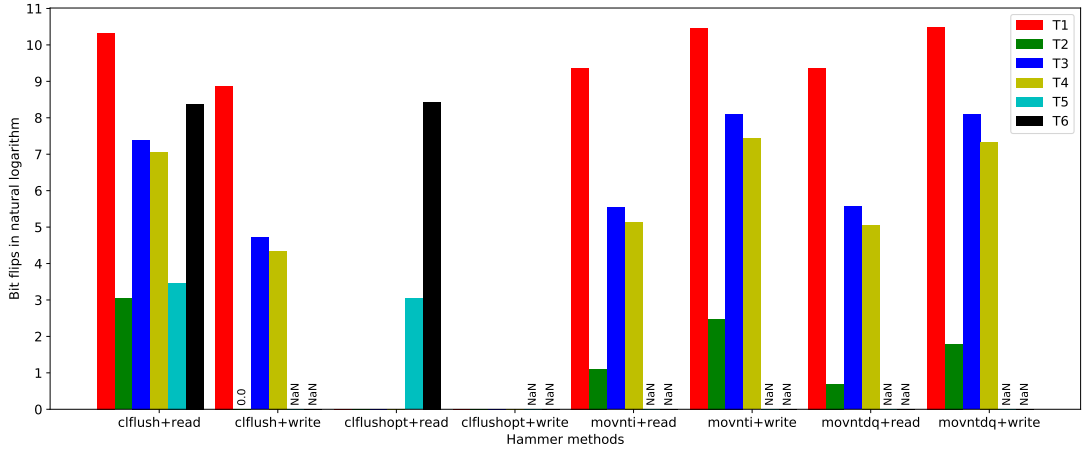


Fig. 9: Bit flips in natural logarithm for different hammer methods using the same data pattern (Double-sided hammer and triple-sided hammer are used in T1–T4 and T5–T6 machines, respectively.). We note that values of 0.0 and NaN in the bar chart respectively represent a single bit flip and no bit flip at all.

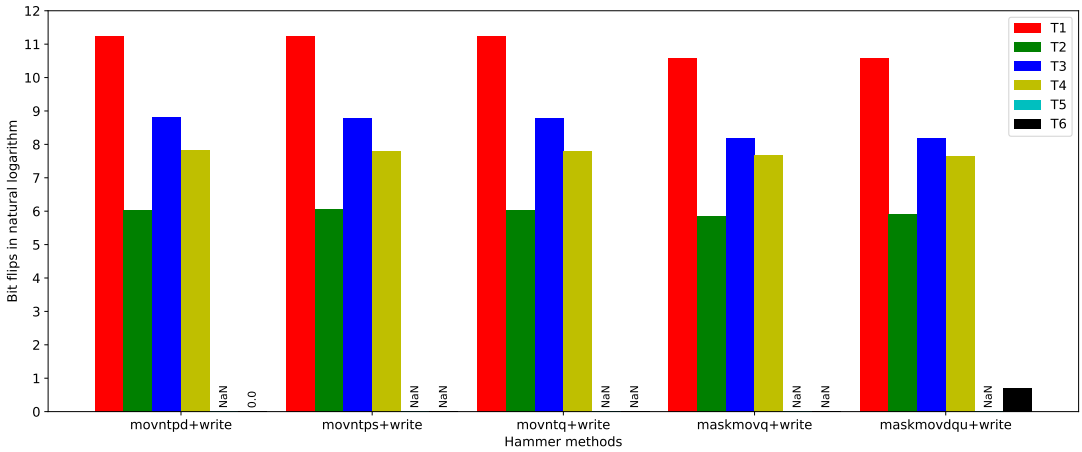


Fig. 10: Bit flips in natural logarithm for newly identified non-temporal-store based hammer methods using the same data pattern (Double-sided hammer and triple-sided hammer are used in T1–T4 and T5–T6 machines, respectively.).

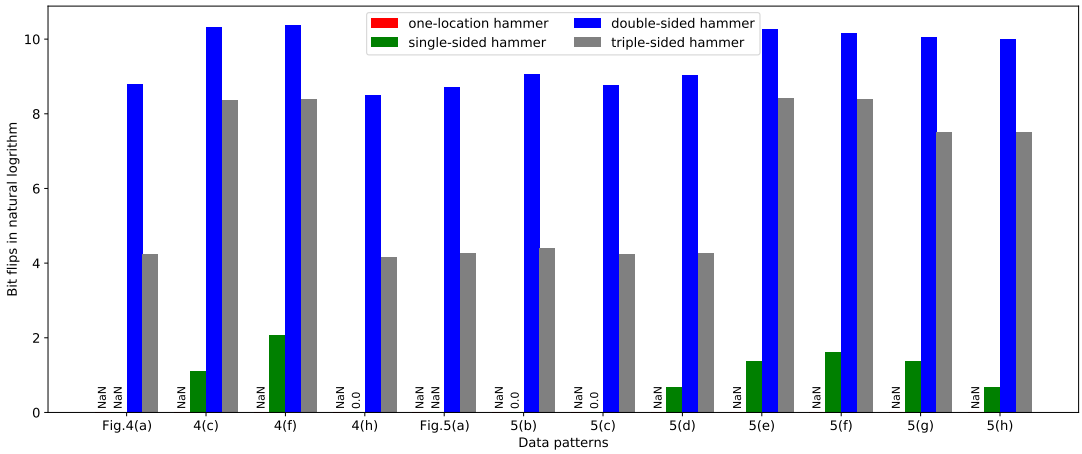


Fig. 11: Bit flips in natural logarithm for a hammer pattern in different data patterns using the same hammer method of `clflush+read`. Triple-sided hammer is evaluated on T6 while the other three hammer patterns are on T1, respectively.

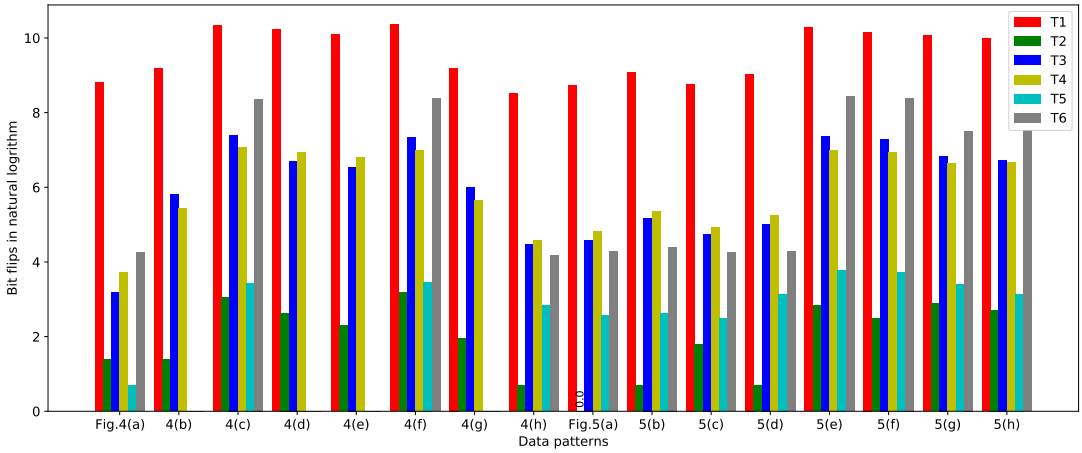


Fig. 12: Bit flips in natural logarithm for different patterns using the same hammer method of `clflush+read`. Double-sided hammer and triple-sided hammer are used on T1–T4 and T5–T6, respectively.

As shown in the figure, there is no single bit flip for one-location hammer in all test data patterns, indicating that the memory controller residing T1 does not support a closed-row or adaptive page policy [11], [35]. Besides, double-sided ham-

Rowhammer Detection	DRAM Address Mapping	Hammer Method	Hammer Pattern	Data Pattern
Google Project Zero [36]	blind	`clflush`	single-sided & double-sided	Figure 4(c)(f)
Rowhammer.js [3]	DRAMA [8]	`clflush` & `clflushopt`	double-sided	Figure 4(c)(f)
Memtest86 [37]	N/A	N/A	double-sided	N/A
BitMine	**DRAMDig**	13	**one-location/single-sided/double-sided/many-sided**	**16**

TABLE VI: A comparison of existing Rowhammer detection tools.

mer has induced much more bit flips than single-sided hammer in every data pattern on T1 machine. For the same hammer pattern, a machine is more vulnerable to a subset of the test data patterns. For instance, data patterns of Figure 4(c)(f) and Figure 5(e)(f) are more effective in T6 compared to other data patterns. For the same data pattern, different hammer patterns have different effectiveness in inducing bit flips. In particular, no bit flip is even observed using single-sided hammer for the data pattern of Figure 4(a).

C. Data Pattern

Figure 12 shows that each test machine has a distinct set of effective and ineffective data patterns in inducing bit flips. Specifically, for all DDR3-based machines T1–T4, the effective set of data patterns are: {Figure 4(c)(d)(e)(f), Figure 5(e)(f)(g)(h)}. The respective ineffective sets of T1, T2, T3–T4 are : {Figure 4(a)(h), Figure 5(a)(c)}, {Figure 4(a)(b)(h), Figure 5(a)(b)(d)} and {Figure 4(a)}.

Regarding DDR4-based machines, the respective effective sets of T5 and T6 are: {Figure 4(c)(f), Figure 5(d)(e)(f)(g)(h)} and {Figure 4(c)(f), Figure 5(e)(f)(g)(h)}. T5's ineffective set is same as T3–T4 but T6's ineffective set is: {Figure 4(a)(h), Figure 5(a)(b)(c)(d)}.

VIII. Related Work

In this section, we compare BitMine to prior related works, summarized in Table VI.

Google Project Zero published the first rowhammer detection tool [36] that leveraged a `clflush`-based hammer method. As this tool does not have knowledge of the DRAM address mapping, it is inefficient and ineffective in detecting rowhammer. Rowhammer.js [3] relied on DRAMA [8] to uncover the DRAM address mapping and further extended the tool of Google Project Zero with two hammer methods (i.e. `clflush` and `clflushopt`). As discussed in Section I and evaluated in Section VI-B, DRAMA is inefficient and fails in generating a deterministic DRAM address mapping. Besides, the rowhammer effectiveness of Rowhammer.js is further limited to a couple of hammer methods and data patterns. Memtest86 [37] is a commercial memory diagnostic tool that incorporates a test for detecting rowhammer bit flips [38]. Memtest86 uses only double-sided hammer for rowhammer detection, rendering itself ineffective on present DDR4 modules where HardTRR is enabled. In contrast, BitMine implements 4 hammer patterns, making itself effective on both DDR3 and DDR4 modules.

IX. Conclusion

In this paper, we presented DRAMDig to reverse-engineer the DRAM address mappings for different Intel-based computing systems that are equipped with either DDR3 or DDR4 chips. We further extended DRAMDig to BitMine, the first end-to-end system tool for detecting rowhammer vulnerability effectively. BitMine incorporated three configurable parameters and each has an observable effect on inducing rowhammer bit flips. By default BitMine implemented 13 hammer methods, 4 hammer patterns and 16 data patterns. Our experimental results showed that DRAMDig successfully reverse-engineered the DRAM address mappings and each key parameter in BitMine has distinct effectiveness in inducing bit flips for different machine models.

References

[1] 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*, 2014, pp. 361–372.

[2] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," in *Black Hat*, 2015.

[3] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 300–321.

[4] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1675–1689.

[5] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *USENIX Security Symposium*, 2016, pp. 19–35.

[6] Y. Cheng, Z. Zhang, S. Nepal, and Z. Wang, "CATTmew: Defeating software-only physical kernel isolation," in *IEEE Transactions on Dependable and Secure Computing*, 2019.

[7] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *International Symposium on Microarchitecture*, 2020.

[8] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016, pp. 565–581.

[9] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security Symposium*, 2007.

[10] M. Wang, Z. Zhang, Y. Cheng, and S. Nepal, "Dramdig: A knowledge-assisted tool to uncover dram address mapping," in *Design Automation Conference*, 2020.

[11] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," *arXiv preprint arXiv:1710.00551*, 2017.

[12] Micron, Inc., "DDR4 SDRAM MT40A2G4, MT40A1G8, MT40A512M16 data sheet," https://www.micron.com/products/dram/ddr4-sdram/, 2015.

[13] JEDEC Solid State Technology Association., "Low power double data rate 4 (LpDDR4)," https://www.jedec.org/standards-documents/docs/jesd209-4b, 2015.

[14] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *IEEE Symposium on Security and Privacy*, 2020.

[15] M. Seaborn, "How physical addresses map to rows and banks in dram," http://lackingrhoticity.blogspot.com.au/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

[16] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, *DRAM circuit design: fundamental and high-speed topics*. John Wiley & Sons, 2007, vol. 13.

[17] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[18] Z. Zhang, Z. Zhan, D. Balasubramanian, B. Li, P. Volgyesi, and X. Koutsoukos, "Leveraging EM side-channel information to detect rowhammer attacks," in *IEEE Symposium on Security and Privacy*, 2019.

[19] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *IEEE Symposium on Security and Privacy*, 2016, pp. 987–1004.

[20] Google, Inc., "Glitch vulnerability status," http://www.chromium.org/chromium-os/glitch-vulnerability-status, May 2018.

[21] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *USENIX Security Symposium*, 2020.

[22] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *USENIX Security Symposium*, 2019, pp. 497–514.

[23] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *USENIX Annual Technical Conference*, 2018.

[24] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: flipping secret exponent bits using timing analysis," in *Cryptographic Hardware and Embedded Systems*, 2016, pp. 602–624.

[25] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *IEEE Symposium on Security and Privacy*, 2020.

[26] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Security Symposium*, 2016, pp. 1–18.

[27] K. A. Shutemov, "pagemap: do not leak physical addresses to non-privileged userspace," http://goo.gl/Zvd0qf, 2015.

[28] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2016, pp. 161–166.

[29] Micron, Inc., "DDR3 SDRAM MT41K2G4RKB, MT41K1G8TRF, MT41K512M16VRN data sheet," https://www.micron.com/products/dram/ddr3-sdram/, 2015.

[30] Intel, Inc., "The role of ecc memory," https://www.intel.com/content/www/us/en/workstations/workstation-ecc-memory-brief.html, 2015.

[31] ——, "Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c," Oct. 2011.

[32] Y. Cheng, Z. Zhang, S. Nepal, and Z. Wang, "Cattmew: Defeating software-only physical kernel isolation," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[33] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint rowhammer: Suppressing unwanted bit flips on rowhammer attacks," in *Asia Conference on Computer and Communications Security*, 2019, pp. 549–560.

[34] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *International Symposium on Computer Architecture*, 2020.

[35] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *High Performance Computer Architecture*, 2010, pp. 1–12.

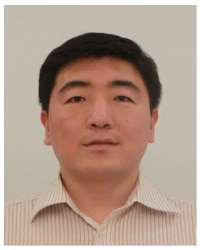
[36] "Test dram for bit flips caused by the rowhammer problem," https://github.com/google/rowhammer-test, 2014.

[37] "Memtest86," https://www.memtest86.com.

[38] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Rowhammer memtest," https://github.com/CMU-SAFARI/rowhammer, Sep. 2014.

Zhi Zhang is a postdoctoral fellow at CSIRO Data61. He received his Ph.D. in Computer Science from the University of New South Wales. His research interests are in the areas of system security, rowhammer and adversarial artificial intelligence.

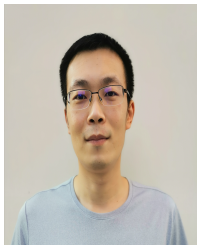
Wei He received the B.S. degree from Tongji University in 2019, and now studies for a master's degree at University of Chinese Academy of Sciences and SKLOIS, Institute of Information Engineering, CAS. His research interests include system security.



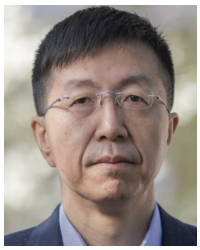
Yueqiang Cheng is head of security research at NIO. He earned his Ph.D. degree in School of Information Systems from Singapore Management University under the guidance of Professor Robert H. Deng and Associate Professor Xuhua Ding. His research interests are system security, trustworthy computing, software-only root of trust and software security.

Wenhao Wang received the B.S. degree from Ocean University of China in 2009, and the Ph.D. degree from University of Chinese Academy of Sciences in 2015. He is an Associate Professor in Institute of Information Engineering, Chinese Academy of Sciences. His research interests include Cryptography, System Security, Cloud Security and Trusted Execution Technologies.

Yansong Gao received his M.Sc degree from University of Electronic Science and Technology of China in 2013 and Ph.D degree from the School of Electrical and Electronic Engineering in the University of Adelaide, Australia, in 2017. He is now with School of Computer Science and Engineering, Nan-Jing University of Science and Technology, China. His current research interests are AI security and privacy, hardware security and system security.



Minghua Wang is a Staff Software Engineer of Baidu X-Lab. He obtained his Ph.D degree from University of Chinese Academy of Science. His research interests lie in program analysis, software security and side-channel attacks and mitigations.



Kang Li is the director of Baidu X-Lab. He received his B.S degree from Tsinghua University, Master degree from Yale Law School, and Ph.D degree from Oregon Graduate Institute at OHSU. His research interests are system security and privacy.

Surya Nepal is a Senior Principal Research Scientist at CSIRO Data61 and leads the distributed system security research group. His main research focus has been in the area of distributed systems, with a specific focus on security, privacy and trust. He has more than 200 peer-reviewed publications to his credit. He currently serves as an associate editor in IEEE Transactions on Service Computing and IEEE Transactions on Dependable and Secure Computing.

Yang Xiang received his PhD in Computer Science from Deakin University, Australia. He is currently a full professor and the Dean of Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is also leading the Blockchain initiatives at Swinburne. In the past 20 years, he has published more than 300 research papers in many international journals and conferences. He is the Editor-in-Chief of the SpringerBriefs on Cyber Security Systems and Networks. He serves as the Associate Editor of IEEE Transactions on Dependable and Secure Computing, IEEE Internet of Things Journal, and ACM Computing Surveys. He served as the Associate Editor of IEEE Transactions on Computers and IEEE Transactions on Parallel and Distributed Systems. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a Fellow of the IEEE.