

Knock-Knock: Black-Box, Platform-Agnostic DRAM Address-Mapping Reverse Engineering

Antoine Plin

GeorgiaTech, ESILV, CentraleSupélec, Inria, CNRS, IRISA

Thomas Rokicki

CentraleSupélec, Inria, CNRS, IRISA

Lorenzo Casalino

CentraleSupélec, Inria, CNRS, IRISA

Ruben Salvador

CentraleSupélec, Inria, CNRS, IRISA

Abstract

Modern Systems-on-Chip (SoCs) employ undocumented linear address-scrambling functions to obfuscate DRAM addressing, which complicates DRAM-aware performance optimizations and hinders proactive security analysis of DRAM-based attacks; most notably, Rowhammer. Although previous work tackled the issue of reversing physical-to-DRAM mapping, existing heuristic-based reverse-engineering approaches are partial, costly, and impractical for comprehensive recovery. This paper establishes a rigorous theoretical foundation and provides efficient practical algorithms for *black-box, complete physical-to-DRAM address-mapping recovery*.

We first formulate the reverse-engineering problem within a linear algebraic model over the finite field $\text{GF}(2)$. We characterize the timing fingerprints of row-buffer conflicts, proving a relationship between a bank addressing matrix and an empirically constructed matrix of physical addresses. Based on this characterization, we develop an efficient, noise-robust, and fully platform-agnostic algorithm to recover the full bank-mask basis in polynomial time, a significant improvement over the exponential search from previous works. We further generalize our model to complex row mappings, introducing new hardware-based hypotheses that enable the automatic recovery of a row basis instead of previous human-guided contributions.

Evaluations across embedded and server-class architectures confirm our method’s effectiveness, successfully reconstructing known mappings and uncovering previously unknown scrambling functions. Our method provides a 99% recall and accuracy on all tested platforms. Most notably, Knock-Knock runs in under a few minutes, even on systems with more than 500GB of DRAM, showcasing the scalability of our method. Our approach provides an automated, principled pathway to accurate DRAM reverse engineering.

1 Introduction

Modern computing systems rely on *Dynamic Random Access Memory* (DRAM) as a high-throughput, low-latency

memory subsystem. However, manufacturers often obscure the physical-to-DRAM addressing schemes using undocumented linear scrambling functions [26, 32]. Consequently, reverse engineering becomes a prerequisite for conducting precise side-channel and fault-injection analyses [2, 7, 22], as well as for developing effective mitigations [21]. Without knowledge of the DRAM addressing function, an attacker cannot reliably target specific DRAM rows or banks, limiting the feasibility and repeatability of DRAM-based attacks such as Rowhammer [19] or memory-aware cache attacks [2]. Similarly, defenders cannot deploy targeted mitigations, e.g., memory fencing, error detection, or access-pattern randomization [33], without understanding which physical addresses map to vulnerable DRAM structures [21]. Beyond security, this mapping is also critical in systems research: memory allocation [1, 25], or DRAM-aware data placement strategies [31, 34] all benefit from accurate knowledge of the underlying physical-to-DRAM mapping.

Logically, researchers have already tried to reverse engineer this physical-to-DRAM mapping. Early proposed solutions to reverse DRAM addressing functions either rely on invasive physical probing [16, 26] or exhaustive brute-force approaches [13, 26, 30]. This exhaustive search phase has exponential complexity in terms of the number of bits of the DRAM address, which means methods relying on brute force scale poorly with larger DRAM. As proved by previous work [30], exhaustive-search solutions [26] can take hours or even longer to execute successfully on current yet standardly-sized DRAM systems, severely limiting their applicability in practical scenarios. While some works [10, 32] tried addressing this complexity issue to reduce computation time by crafting customized solutions tailored to specific platforms, this came at the expense of generalizability and automation. Another limitation of existing approaches is noise: The first phase of the reverse engineering process relies on a timing side channel, where misclassification can happen, resulting in broken masks. While previous works [26, 30] repeated the measurements to reduce noise impact, a misclassification in the experimental phase will introduce errors in the reversing

phase.

To address these limitations, we present *Knock-Knock*, a principled, efficient, and entirely automated approach to reverse engineer physical-to-DRAM address translation. Our approach treats the physical-to-DRAM translation as a black box, reversing the entire pipeline without separating the memory controller and DRAM-internal mappings individually. Knock-Knock leverages the row-buffer conflict side channel [26] to build sets of addresses mapping to the same bank or row, then introduces a reduction of the search space using nullspace analysis. We present, to the best of our knowledge, the first **analytical framework for physical-to-DRAM address translation**, which we leverage to achieve a more efficient, broadly applicable, and precise analysis. Our solution eliminates the exponential search phase altogether and introduces four **contributions**:

Reduced search space through algebra Thanks to an algebraic formalization of the problem, we reduce the search space by proving that the bank/channel masks form a Nullspace(D) basis, where D is a sparse difference matrix. Computing that null space is at most of complexity $O(n^3)$ in the number of rows, removing the exponential term that limited prior tools.

Provable, noise-aware sample bound. We derived a closed-form expression that theoretically guarantees full mask recovery even with mislabeled samples, turning the heuristic "collect-until-it-works" from previous works into a one-line calculation.

Automatic, low-weight row masks From this reduced search space, we retrieve the row addressing function with an algorithm using minor hardware-based hypotheses.

DRAM-agnostic approach Because the method needs only the existence of row-buffer conflicts, Knock-Knock does *not* assume prior knowledge of the DRAM hierarchy (module, rank, bank group, *etc.*) to reverse-engineer the bank masks. Our method only uses empirically verified assumptions about the hardware to reverse row masks. This approach extends applicability beyond systems with well-known or documented memory configurations, e.g., desktops or HPC-class servers, to more constrained environments like undocumented mobile, IoT SoCs, and cloud platforms. In these cases, components may be soldered, obfuscated in the PCB, or remote with limited access to system information. Existing tools often fail under such conditions, or require manual guidance to function effectively [10, 26].

Validated on 10 platforms, server, consumer, and embedded SoCs, Knock-Knock turns DRAM address reverse engineering from a labor-intensive, sometimes platform-specific effort

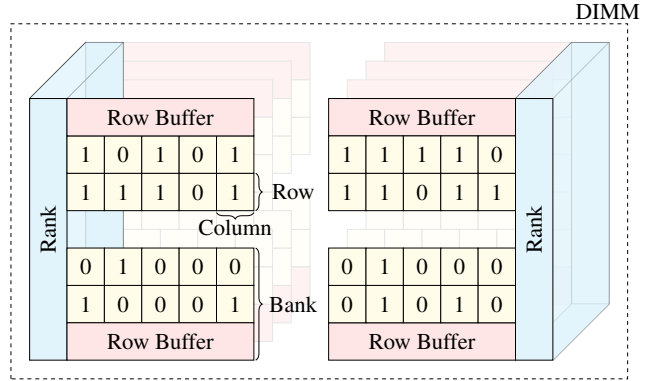


Figure 1: Organization of an example DRAM module. It is composed of two ranks, each handling 8 banks. Each bank is composed of 5 columns and 2 rows. Memory systems may contain several channels, each containing multiple modules.

into a generic, noise-resistant push-button procedure that completes in minutes with 99% accuracy. Most notably, Knock-Knock recovered the physical-to-DRAM mapping functions of a server-class SoC with 512 GB of RAM in a few minutes. To support reproducible and maintainable research, we publish our code and data in an open repository¹.

The remainder of this paper is organized as follows: Section 2 presents relevant background and Section 3 surveys the state-of-the-art. Section 4 introduces the reverse-engineering methodology proposed by Knock-Knock, along with the mathematical formalization used in subsequent sections, which are at the core of our contributions. Then, Section 5 shows how Knock-Knock reverse engineers the parity masks of the addressing function. In Section 6, we present our method to retrieve the full row bits of the addresses and generalize our approach to more complex addressing functions. Section 7 describes the experimental setup used to validate our methodology and the results obtained on a range of different types of platforms. We finally discuss the results obtained in Section 8 and conclude the paper in Section 9.

2 Background

2.1 DRAM Organization

As illustrated in Figure 1, DRAM modules are organized in a hierarchy of channels, modules, ranks, banks, rows, and columns, down to individual cells, each containing a single bit [14, 15]. Each channel can be used independently, allowing access distribution. These channels can contain multiple modules, which can be further divided into ranks, often one on the front and one on the back of the module. Those ranks are

¹<https://github.com/antpln/Knock-Knock>

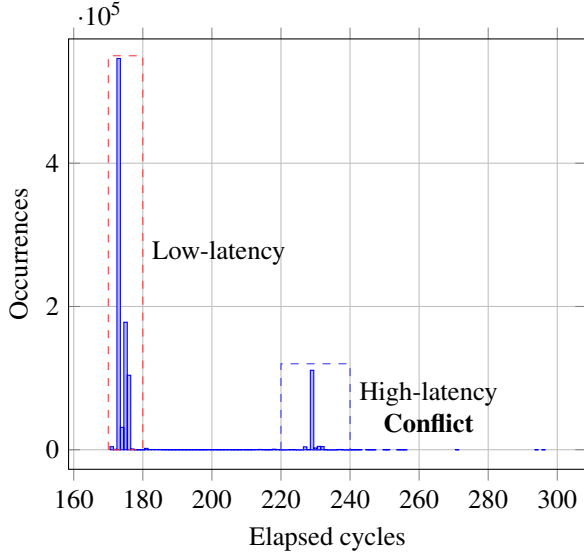


Figure 2: Distribution of elapsed cycles between loads with and without row conflicts

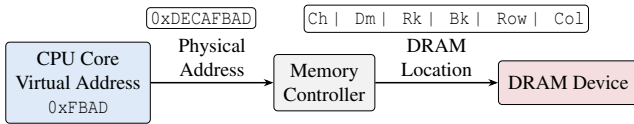


Figure 3: Simplified view of translations from Virtual Address to DRAM Location containing the channel, module, rank, bank, row, and column of the accessed byte

composed of different row/column arrays, called banks. Besides the memory array, each bank contains circuits to serve the individual cells, notably the *row buffer*. From a logical perspective, the row buffer can be seen as a cache for the most recently accessed row. Subsequent accesses to the same row are directly served from this buffer, reducing access latency. However, if a *different row is accessed in the same bank*, the DRAM controller must first close the current row and open the new one, which incurs a higher latency. Figure 2 shows that the access latency follows a mixture of two distributions: one with lower latency when accessing the same row (on the left), and one with higher latency when accessing a different row (on the right). This event is called a *row conflict* and constitutes a fundamental side channel to enable improved Rowhammer [7, 18, 20, 22, 26] and cache [2, 28] attacks.

2.2 DRAM Addressing

The *memory controller* is responsible for translating *physical addresses* used by the CPU to coordinate system mapping to the location of the data in the DRAM hierarchy. Figure 3 summarizes the translations made from virtual addresses to

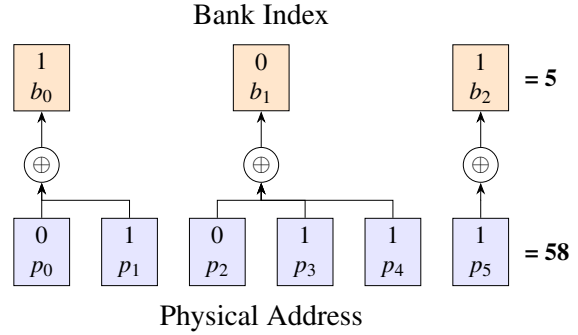


Figure 4: Example of a linear function to address banks. It uses 3 parity masks. b_i and p_i represent, respectively, the bank and physical address bits.

DRAM. This translation is opaque to any process running on the CPU. The result of the physical-to-DRAM-location address translation is composed of a 6-tuple $[Ch, Dm, Rk, Bk, Row, Col]$, which indicates, respectively, the channel, module, rank, bank, row, and column of the target cell. Previous works showed that this translation is done *linearly* [10, 19, 26] by XORing certain bits of the physical address between them to form the DRAM address. An example of such mapping is shown in Figure 4, where physical address bits P_i are mapped to bank index bits B_i through XORs. Which bits are used as inputs for each XOR operator are defined by the so-called *parity masks*. In some cases [23], the addressing can directly map bits without using XOR functions. This represents a sub-case of linearity, as a direct bit-to-bit assignment is just a parity mask with only one bit set.

This XOR scrambling mechanism allows for optimizations such as bank and channel interleaving [10], reducing both stress on the DRAM cells and access latency. While AMD publicly documented the functions for older generations of processors in their *BIOS and Kernel Developer's Guide* [26], manufacturers often do not document them publicly.

Therefore, the relation between a physical address and the location of its related byte in the DRAM — described as the tuple $[Ch, Dm, Rk, Bk, Row, Col]$ — is often unknown.

In this work, we address the problem of unknown physical-to-DRAM address mappings and provide a new, provably efficient method to reverse-engineer DRAM address mapping functions that is platform- and DRAM-geometry-agnostic.

3 Related Works

In this section, we survey previous related work. To better motivate the need to know the physical-to-DRAM mapping, we first (Section 3.1) present different attacks that were only

possible after unlocking these addressing functions. Then (Section 3.2) we introduce important works from the literature, describe their strategies to reverse-engineer address mappings, and provide a qualitative comparison among them and our Knock-Knock proposal. We finish the review of related works (Section 3.3) with a deep focus on works that use a linear algebra approach to discover unknown mapping functions.

3.1 Attacks Through Known Addressing Functions

Various works have already shown how to leverage knowledge of the physical-to-DRAM mapping to craft more powerful attacks. We can distinguish two main categories:

Cache attacks While generally not related to DRAM addressing, cache attacks can be enabled or improved by the knowledge of DRAM addressing. In some cache-based side channels, e.g., Flush+Reload, the row hits can be confused for cache hits, potentially adding noise for the attack [26]. Thus, taking into account physical-to-DRAM addressing reduces the risk of this confusion. Bechtel et al. [2] built a *DRAM-aware* Denial-of-Service attack on the shared cache of two multi-core embedded platforms. Knowing the translation from physical addresses, used for cache addressing, to DRAM addressing, the authors force the system’s cache misses to target the same DRAM bank. This greatly increases load latency, as cache misses will also cause a row-buffer conflict and prevent bank-level parallelism. By using this method, they achieve a slowdown of two orders of magnitude compared to state-of-the-art Denial-of-Service attacks.

Schwarz et al. [28] demonstrated an attack targeting RSA implementations executed within Intel’s SGX enclaves by constructing cache eviction sets that leverage DRAM address mapping. The authors start by identifying clusters of addresses in a contiguous memory region (e.g., a bank) by using the row-buffer conflict side channel. Then, the authors use the reversed physical-to-DRAM mapping to recover bits of information about the physical addresses of their set. Using these recovered eviction sets, they build an eviction set able to attack RSA implementations running inside Intel’s SGX security enclave.

Rowhammer attacks Rowhammer is a micro-architectural fault attack allowing attackers to flip bits outside of the memory allocated to their malicious unprivileged program. It is an attack on *process memory isolation*. It takes advantage of a previously known reliability issue of DRAM chips: Repeated high-frequency accesses to the same position can leak charge to neighboring cells, possibly causing their values to flip [19]. Manufacturers have tried to mitigate such attacks, most notably with *Target Row Refresh*. It uses counters of accesses to DRAM rows in order to refresh victim rows in case of a suspicious number of accesses. However, subsequent works have leveraged a known DRAM mapping to find contiguous memory and hence build more complex *hammering patterns*

that circumvent TRR either by overloading the counter [7] or by using the TRR-induced refreshes to do the hammering [20]. Moreover, *RAMBleed* [22] showed that precise hammering, requiring in-depth knowledge of the physical-to-DRAM mapping, and the same knowledge of contiguous memory could be used as a read side channel, thus leaking secrets. This side channel has been developed to broaden its threat model to attacks against DNNs [27] or cryptographic implementations [6]

3.2 Empirical DRAM Mapping Reverse-Engineering Strategies

DRAMA [26] introduced the first generic methodology to recover the addressing through the row-buffer-based side channel described in Section 2.1. In particular, by identifying which address pairs create a row-buffer conflict, the authors can create a cluster of addresses belonging to the same bank. As a result, it is now possible to target the same bank, which becomes useful for different attack strategies. For instance, in the rowhammer case, this enables hammering several lines in the same bank, circumventing existing countermeasures like TRR [7]. In the case of a Denial-of-Service attack on a cache, this allows a potential attacker to force each memory access to be a row conflict, thus increasing latency on top of cache misses [2]. DRAMA requires sampling the latencies between pairs of addresses to build as many conflicting sets as the expected total number of banks to find. Once those sets are built, XOR masks are exhaustively tried until one is found that explains the whole set. This brute-force approach exhibits exponential complexity with respect to the number of DRAM address bits, resulting in poor scalability on systems with larger memory capacities. For example, DRAMA requires several hours to compute address masks on systems equipped with 16 GB of DRAM [30].

Building upon DRAMA, DRAMDig [30] uses knowledge about the geometry of specific DRAM chips and the processor micro-architecture to reduce the search space of the exponential-time search. While this made DRAMDig methodology results more *precise* than DRAMA, it severely reduces its portability by making it platform-specific.

Helm et al. [10] improved DRAMA’s methodology by using Intel CPUs’ performance counters; specifically, the authors used counters tracking the number of accesses to each channel, rank, and bank. Similarly to DRAMDig, its gains in performance and reliability come at the cost of generalizability because those counters are only available on a few Intel CPUs, again negatively affecting portability.

Heckel and Adamsky [9] introduced AMDRE, a novel framework tackling DRAM addressing reverse-engineering on AMD platforms that effectively adapts the methodology originally developed in DRAMA. Jatke et al. [13] proposed DARE, an AMD-specific technique that exploits enhanced timing-based synchronization and platform-dependent

Table 1: Qualitative comparison of Knock-Knock with representative software-only approaches for DRAM address-mapping recovery.

| Tool | Core idea | Worst-case search complexity | Provable sample bound | Noise tolerance | Platform Agnostic | Support for complex row addressing |
|---------------------------|--|--|-----------------------|-----------------|-------------------|------------------------------------|
| DRAMA [26] | <ul style="list-style-type: none"> Cluster conflicts. Enumerate masks. | Exponential in candidate address bits | ○ | ○ | ● | ○ |
| DRAMDig [30] | <ul style="list-style-type: none"> Detect row/column bits. Guided search. | Exponential in reduced candidate address bits | ○ | ○ | ○ | ○ |
| Helm et al. [10] | <ul style="list-style-type: none"> Cluster conflicts. Uses Intel counters. | Exponential in reduced candidate address bits | ○ | ● | ○ | ○ |
| Sudoku [32] | <ul style="list-style-type: none"> Cluster conflicts. 2 timing channels. | Exponential in reduced candidate address bits | ○ | ○ | ○ | ○ |
| AMDRE [9] | <ul style="list-style-type: none"> Cluster conflicts. Enumerate masks. | Exponential in candidate address bits | ○ | ○ | ○ | ○ |
| DARE [13] | <ul style="list-style-type: none"> Cluster conflicts. Enumerate masks. | Exponential in candidate address bits | ○ | ● | ○ | ○ |
| RISC-H [23] | <ul style="list-style-type: none"> Cluster conflicts. Enumerate masks. | Exponential in candidate address bits | ○ | ○ | ○ | ○ |
| Knock-Knock (ours) | <ul style="list-style-type: none"> Detect conflicts. Null space analysis. | Polynomial through gaussian elimination on $N \times n$ matrix | ● | ● | ● | ● |

insights. Notably, they observed that the DRAM address mapping on these systems exhibits non-linear behavior due to physical address remapping. This means, as a result, that the addressing functions found by previous methods only work on small DRAM clusters but fail on larger, non-consecutive areas. By introducing a constant offset prior to applying XOR operations, they demonstrated that the mapping can, in some cases, be effectively reduced to a linear form. Nonetheless, DARE fundamentally relies on an exhaustive brute-force exploration of the XOR mask space. Marazzi and Razavi [23] introduced the first Rowhammer attack on a RISC-V architecture, basing their approach on AMDRE [9].

The Sudoku tool [32] revisits DRAMA’s clustering approach and augments it with two additional timing channels using refresh latencies and amplified consecutive-access latencies to *label* each discovered mask with its hierarchy level: channel, rank, bank-group, bank. While this labeling improves human interpretability, Sudoku does not improve DRAMA and still inherits its exponentially complex mask-enumeration step, requires timing parameters from the memory controller registers, and leaves XOR-scrambled rows unresolved.

Table 1 compiles the existing works that approach the problem of non-algebraic reverse-engineering DRAM address mapping, and provides a comparison among them. Based on the provided survey of the state of the art, we can see how the field has tackled the problem to improve the performance from the seminal work in DRAMA [26], by crafting solutions highly adapted to specific platforms. These approaches have considered specific Intel processor performance counters [10], precise knowledge about DRAM geometry [30], and adaptations to other platforms like AMD [9] or RISC-V [23], as well as improved methodologies [13] based on DRAMA/AMDRE

but only for AMD platforms. As a result:

We observe how (1) portability has been greatly sacrificed at the cost of performance, and (2) no work has challenged the fundamental problem of reducing the complexity of the brute-force-search approach.

3.3 Linear Algebra For Unknown Mapping Functions Discovery

An algebraic approach to discovering an unknown mapping function consists of formulating the identification of the function as a mathematical problem, and collecting pairs of conflicting addresses to identify such a function while respecting the problem’s constraints. The use of linear algebra finds use also in the construction of mapping functions [29]. Such employment may provide improvements or new reverse-engineering strategies; yet, its scope is different from ours and, for such, we will not discuss the related body of work.

Hofmann et al. [11], from a minimal set of conflicting addresses, recover linear indexing functions through the iterative computation of the basis of the function’s nullspace. The iterative step verifies whether any of the addresses in the conflicting set is also part of the function’s kernel. Simulation-driven analyses, tested on both cache indexing and DRAM bank-indexing functions, show that the approach finds the indexing function with fewer conflict checks with respect to a brute-force approach.

Gerlach et al. [8] propose an automated and generic approach to reconstruct linear and non-linear mapping functions. Their approach splits the mapping function f as the compo-

sition of several mapping functions f_i , where $0 \leq i < n$, and n is the bitwidth of the mapping function’s output. Their approach recovers each function f_i independently, converting each function into a logic formula in *Disjunction Normal Form* (DNF), transforming it to a system of polynomial equations, and retrieving a compact form of this system through the computation of a Gröbner basis for the original system. This process is then repeated for the subterms of the new system of equations to further reduce the complexity of the final mapping function formula. Finally, they build the unknown mapping function from the minimized system of equations.

Compared to these previous works, although Knock-Knock resembles the work of Hofmann et al. [11], it substantially differs in both the approach, the focus, and the results. Our approach targets the physical-to-DRAM mapping function instead of generic (linear) mapping functions, allowing for the recovery of bank and row mappings. We base our identification strategy on the computation of the conflicting set’s nullspace; as such, we avoid the iterative check and computation of the function’s nullspace. An extensive validation campaign, encompassing several architectures, supports our methodology, compared to their simulation-only approach that neglects the impact of noisy measurements. Regarding the work from Gerlach et al. [8], the authors recover non-linear functions, whereas we target linear ones. Nonetheless, Gerlach et al.’s generality implies a higher execution time to recover linear functions. Also, Gerlach et al. need to probe each input bit (e.g., the address bits) to know which has an impact on the mapping function’s output. Our algebraic approach inherently identifies such bits (i.e., the mask bits) Furthermore, the Gröbner base method is quite time-consuming, requiring more than 10 hours to reverse a DRAM function.

In summary, state-of-the-art algebraic approaches lack empirical evaluation, DRAM-specific metrics, and row-mapping reversing. Consequently, a fully generic, black-box (i.e., platform- and DRAM-geometry-agnostic) and faster methodology is needed for more scalable, precise, and efficient discovery of physical-to-DRAM mappings. This is the open problem that we address with Knock-Knock, where we target an analytical, complexity-bounded, and provable reverse-engineering methodology. Specifically, the research question (RQ) that we try to answer with this work is:

RQ: Can we generalize the observations from the row-buffer conflict side-channel and derive an analytical approach to design a reverse-engineering methodology portable to the maximum number of targets?

4 High-Level Overview of Knock-Knock

In the rest of this paper, we describe Knock-Knock, our solution for fast, black-box, and platform-agnostic physical-to-

DRAM address-mapping reverse engineering. Figure 5 illustrates Knock-Knock’s reverse-engineering pipeline, which is divided into two phases: *data generation* (executed on the target system) and *reversing* (performed offline). Within the figure, for the reader’s convenience, we add references to sections where important parts of the methodology are described.

In the first data generation phase, the system generates random address pairs (A, B) and measures the access latency between them. Pairs with a high-latency access are clustered as conflicting, i.e., belonging to different rows of the same bank, whereas low-latency pairs are labeled as non-conflicting, i.e., belonging to different banks or the same row of the same bank. This phase is identical to the clustering phase of DRAMA [26].

Knock-Knock then enters its first reversing phase, described in Section 5. Then, we use the conflicting pairs to build a difference matrix, enabling the recovery of the *bank addressing function* by solving a system of linear equations.

The bank mappings are then used in the second data generation phase to leverage a variant of the row-buffer side channel to build pairs of addresses belonging to different rows in the same bank. The system generates address pairs (A, B) belonging to the same bank and measures the access latency between them. Pairs with a high-latency access denote a row-buffer conflict, i.e., belonging to different rows. On the contrary, pairs with a low access latency indicate a row-buffer hit, i.e., both addresses belong to the same row of the same bank.

The second reversing phase, described in Section 6, uses the cluster of addresses belonging to the same row of the same bank to construct another difference matrix to derive the *row addressing function*. The inferred bank and row functions form the *complete physical-to-DRAM addressing function*.

4.1 Using Linear Algebra for Reverse-Engineering

The fundamental idea behind Knock-Knock is to formulate the problem of identifying the parity masks as a linear algebra problem. We first mathematically define when two addresses cause a row-buffer conflict. Second, from this definition and from a set of randomly generated address pairs, we show how to build a system of linear equations that puts conflicting addresses in relation. Third, we prove that the solution to this system is a valid set of parity masks. By describing the search problem as the computation of a solution to a system of linear equations, we effectively bound the time complexity of the worst-case scenario to the time complexity of the particular algorithm used to solve the linear system. While this reduction is sufficient for the channel and bank masks, for which we only want to check equality between addresses, we later introduce some different hypotheses to determine functions for the rows.

An important assumption supporting our methodology is the *linearity* of the address mapping function, which many previous works verified in practice in different in-

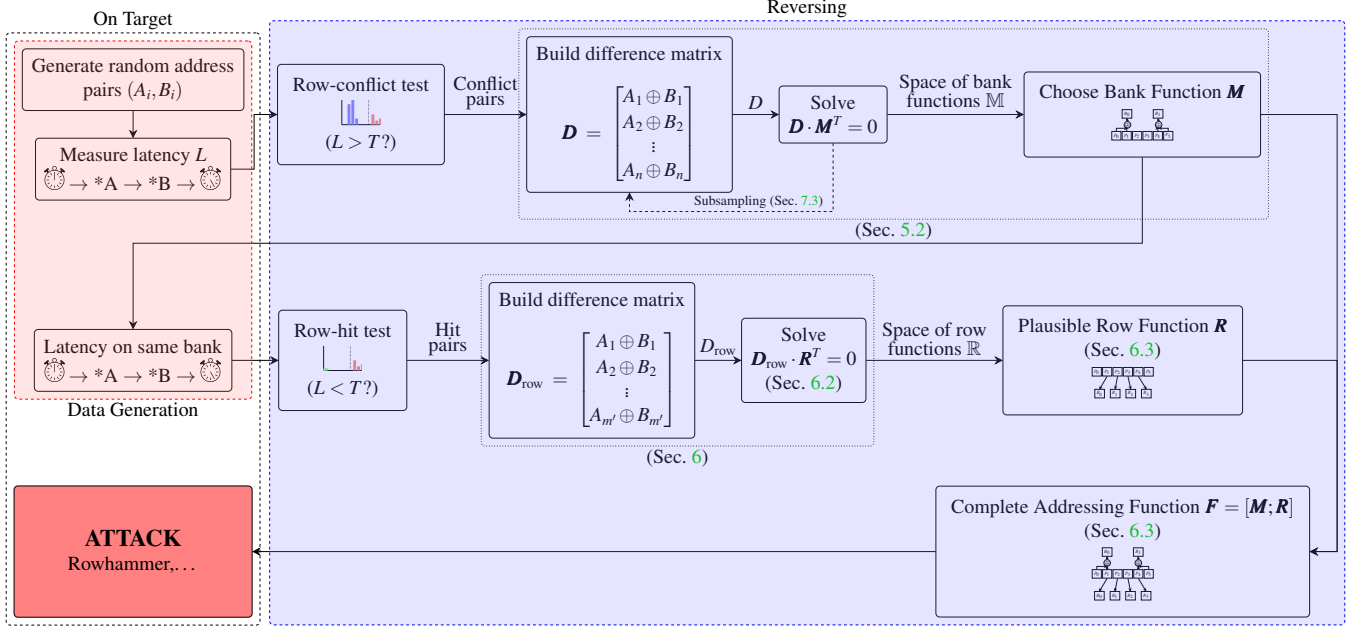


Figure 5: Pipeline of the reverse-engineering methodology

stances [10, 20, 26, 30]. To the best of our knowledge, only ZenHammer [13] showed a case of non-linear mapping, implied by an offset added to physical addresses. We believe that our methodology is fully applicable even in such cases: As ZenHammer’s authors showed, it is possible to remove this offset, making the mapping linear.

4.2 Mathematical Definitions and Notation

We denote scalar variables with small italic letters (e.g., n number of bits). Capital italic letters denote n -bit vectors (e.g., a physical address A); we also use the notation A_i to define several vectors, where the scalar index i is bound by the context. A capital bold italic letter denotes a matrix (e.g., \mathbf{M}) defined on the set $\{0, 1\}$. We use a blackboard bold capital letter to denote a set (e.g., a set \mathbb{S}). The operator $|\cdot|$ defines the cardinality of a set (e.g., $|\mathbb{S}|$).

We consider a DRAM addressed with n -bit addresses, and describe its locations with 6-tuples $\langle \text{Channel}, \text{module}, \text{Rank}, \text{Bank}, \text{Row}, \text{Column} \rangle$. For a given physical address A , we use the notation $\text{Row}(A)$, $\text{Bank}(A)$, $\text{Channel}(A)$ to extract the respective components of A ’s DRAM location. Given Φ and Δ , the DRAM’s address space and location space, respectively, we define

$$f : \Phi \rightarrow \Delta \quad (1)$$

as the DRAM address mapping function.

We denote the access latency for two physical addresses A and B with $L(A, B)$. As explained in Section 2.1, for two randomly chosen addresses A and B , the access latency follows

a mixture of two distributions. We define T as the threshold latency that separates the two distributions. We denote with \wedge , according to the context, either the bit-wise or the logical AND. Then, we can describe the relation between row-buffer conflict and access latency with the following proposition:

$$L(A, B) > T \Leftrightarrow \text{Row}(A) \neq \text{Row}(B) \wedge \text{Bank}(A) = \text{Bank}(B) \\ \wedge \text{Chan}(A) = \text{Chan}(B)$$

Denoting with \oplus the bit-wise XOR and with $(A)_l$ the access to the l -th bit of the n -bit address A , we define

$$p(A) = \bigoplus_{l=0}^{n-1} (A)_l \quad (2)$$

the parity of A .

For any matrix \mathbf{M} , we define with $\text{rk}(\mathbf{M})$ the number of linearly independent rows (equivalently, columns) of \mathbf{M} . With \mathbf{M}^T we denote the transpose of \mathbf{M} .

We use $\text{HW}(V)$ as the Hamming weight of the bit vector V . We denote with $P(\cdot)$ the probability of a certain event.

5 Finding Bank and Channel Parity Masks via Nullspace Analysis

This section tackles the first reverse engineering phase of Knock-Knock, illustrated in Figure 6. The goal is to retrieve bank and channel parity masks in a purely platform-agnostic approach. While our approach does not separate which bits are

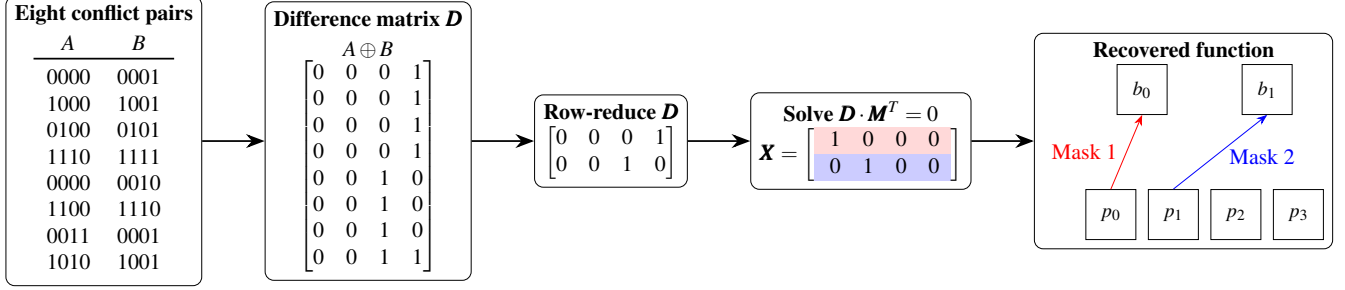


Figure 6: The eight conflict pairs (A, B) only differ in their two low-order bits, so every XOR difference $A \oplus B$ contains at least 0001 or 0010. Stacking those differences produces the matrix \mathbf{D} . We eliminate rows to only keep independent rows. We solve $\mathbf{D} \cdot \mathbf{M}^T = 0$ and get a basis, giving us the functions that associate the physical address bits p_i to the bank index bits b_i .

used to address the channel and which to address the banks, this mask allows the translation from a physical address to a unique bank. By using the clusters of pairs of addresses belonging to the same bank, we build a difference matrix, *i.e.*, a matrix containing the results of the XOR of each pair of addresses. This matrix is then used to produce a system of linear equations. The solution to this system gives a basis for the bank bits of the physical address.

5.1 Problem statement

The problem of reverse-engineering the bank and channel parity masks is to find a set of $k \geq 1$ masks $M_j \in \{0, 1\}^n$ such that, for any two physical addresses $A_i, B_i \in \{0, 1\}^n$ in the same memory module², the following condition holds $\forall j \in [1, k]$:

$$p(A_i \wedge M_j) = p(B_i \wedge M_j) \Leftrightarrow \text{Bank}(A_i) = \text{Bank}(B_i) \wedge \text{Chan}(A_i) = \text{Chan}(B_i) \quad (3)$$

That is, the parity distinguishes which addresses are in the same bank and channel.

5.2 Reduction to Nullspace Analysis

Let us consider a set \mathbb{S} of $m \geq 1$ randomly generated address pairs A_i, B_i :

$$\mathbb{S} = \{(A_i, B_i) \in \Phi^2 \mid \text{Bank}(A_i) = \text{Bank}(B_i) \wedge \text{Chan}(A_i) = \text{Chan}(B_i)\} \quad (4)$$

Using the definition of parity (Equation (2)), we refactor the *if* term of Equation (3) as:

$$\bigoplus_{l=0}^{n-1} (A_i \wedge M_j \oplus B_i \wedge M_j)_l = 0,$$

²In order to simplify notations, we do not include memory module in the equation.

and since \wedge is distributive over \oplus , we have that:

$$\bigoplus_{l=0}^{n-1} ((A_i \oplus B_i) \wedge M_j)_l = 0. \quad (5)$$

Given $D_i = A_i \oplus B_i$ the *i*-th difference word, we rewrite Equation (5) as:

$$\bigoplus_{l=0}^{n-1} (D_i)_l \wedge (M_j)_l = D_i \cdot M_j^T = 0. \quad (6)$$

Then, we can define the following difference matrix:

$$\mathbf{D} = \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{pmatrix} \in \{0, 1\}^{m \times n}.$$

For k parity masks, we define the mask matrix:

$$\mathbf{M} = \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_k \end{pmatrix} \in \{0, 1\}^{k \times n}.$$

Given \mathbf{D} and \mathbf{M} , we can rewrite the *if* part of Equation (3) in matrix form:

$$\mathbf{D} \cdot \mathbf{M}^T = 0.$$

Under the condition of sufficient m number of address pairs (see Section 7.2), the Rank-Nullity theorem [17] guarantees that the nullspace of the difference matrix \mathbf{D} coincides with the set \mathbb{M} of possible parity masks:

$$\text{nullspace}(\mathbf{D}) = \{M_j \in \{0, 1\}^n \mid \mathbf{D} \cdot M_j^T = 0\}. \quad (7)$$

Thus, we have :

$$\forall \mathbf{D} \in \{0, 1\}^{m \times n}, \mathbb{M} = \text{nullspace}(\mathbf{D}).$$

Any base of \mathbb{M} describes, in matrix form, a bank addressing function. We remark that for a given difference matrix \mathbf{D} , the nullspace basis is not unique in general. However, any two bases $\mathbf{B}, \mathbf{B}' \in \{0, 1\}^{k \times n}$ of $\text{nullspace}(\mathbf{D})$ classify any address pair in the same manner (*i.e.*, as conflicting or non-conflicting addresses) under the condition that there exists an invertible matrix $\mathbf{P} \in \{0, 1\}^{k \times k}$ such that $\mathbf{B} = \mathbf{P} \cdot \mathbf{B}'$. Indeed, given two randomly generated addresses A and B , we can expand Equation (6) as:

$$\mathbf{D} \cdot \mathbf{B}^T = (\mathbf{A} \oplus \mathbf{B}) \cdot \mathbf{B}^T = (\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{P} \cdot \mathbf{B}')^T = (\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{B}')^T \cdot \mathbf{P}^T.$$

Being \mathbf{P} invertible and $\mathbf{D} \cdot \mathbf{B}^T = 0$, we have that:

$$(\mathbf{A} \oplus \mathbf{B}) \cdot \mathbf{B}'^T = (\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{B}')^T = 0.$$

We have shown how the computation of the nullspace of a matrix built from conflicting addresses provides us with a set of parity masks indexing the bank and channel of an address. We now show how to recover the masks used to index the row of an address.

Summary 1: Using nullspace analysis, we retrieve a set of parity masks that describe the mapping from physical addresses to banks and channels.

6 Finding Row Parity Masks via Nullspace Analysis

Now that we have a set of masks M_j determining if two addresses belong to the same bank and channel, we need to identify the masks indexing the row bits in the addresses. To this end, we apply the same nullspace approach described in Section 5, but we consider the subset of only-conflicting addresses: Addresses belonging to the same bank have a lower access latency when mapped to the same row (*i.e.*, row hits) than when mapped to different rows (*i.e.*, row conflicts). This different use of the row-buffer-conflict side channel corresponds to the second Data Generation phase of Figure 5. The objective of this section is to build a difference matrix from this data cluster, which enables us to build a row addressing function with minimal hardware hypotheses, corresponding to the second reversing phase of Figure 5.

6.1 A Sufficient Condition to Reduce the Row Masks Search

Using our clusters, we begin by searching a vector of bits M_{row} that never vary between two addresses in the same row. In fact, if a bit varies in the same row address pair, it cannot be used for row calculation. Let us consider the set \mathbb{S} of randomly generated addresses A and B (Equation (4)).

Thanks to Equation (3) and the bank masks M_j we have found, we have that:

$$\mathbb{S} = \{(A_i, B_i) \in \Phi^2 \mid \forall j \in [1, k], p(A_i \wedge M_j) = p(B_i \wedge M_j)\}$$

Being T the threshold separating low-latency access addresses from the high-latency ones (Section 4.2), we define \mathbb{S}_{low} the set of address pairs in \mathbb{S} mapping to the same row (*i.e.*, they have a lower latency access):

$$\mathbb{S}_{\text{low}} = \{(A_i, B_i) \in \mathbb{S} \mid L(A_i, B_i) < T\} \subseteq \mathbb{S}.$$

As stated above, we look for parity masks $M_{\text{row,low}}$ selecting bits that are always the same in both $A_i, B_i \in \mathbb{S}_{\text{low}}$:

$$\forall l \in [0, n-1] : (M_{\text{row,low}})_l = \bigwedge_{(A_i, B_i) \in \mathbb{S}_{\text{low}}} [(A_i)_l = (B_i)_l]. \quad (8)$$

From Equation (8), we derive the following **sufficient condition** $\forall A_i, B_i \in \mathbb{S}_{\text{low}}$:

$$M_{\text{row,low}} \wedge A_i = M_{\text{row,low}} \wedge B_i \implies \text{Row}(A_i) = \text{Row}(B_i). \quad (9)$$

This condition restricts the search of row parity masks $M_{\text{row,low}}$ to only those that satisfy Equation (8).

A simple row mapping (*i.e.*, all the row bits set sequentially in the physical address) will give the complete row mapping.

Summary 2: By comparing same-bank addresses belonging to the same row, we build a first coarse-grained row addressing function.

6.2 Building a Basis for the Row Parity Masks

From the set \mathbb{S}_{low} of address pairs mapped to the same bank and row, we create the following difference matrix:

$$\mathbf{D}_{\text{row}} = \begin{pmatrix} A_1 \oplus B_1 \\ A_2 \oplus B_2 \\ \vdots \\ A_{m'} \oplus B_{m'} \end{pmatrix} \in \{0, 1\}^{m' \times n},$$

where $A_i, B_i \in \mathbb{S}_{\text{low}}$, and $m' = |\mathbb{S}_{\text{low}}|$.

Given $k' \geq 1$ row masks, we define the row mask matrix:

$$\mathbf{R} = \begin{pmatrix} R_1 \\ R_2 \\ \vdots \\ R_{k'} \end{pmatrix} \in \{0, 1\}^{k' \times n}.$$

We rewrite the *if* condition of Equation (9) as:

$$\mathbf{D}_{\text{row}} \cdot \mathbf{R}^T = 0.$$

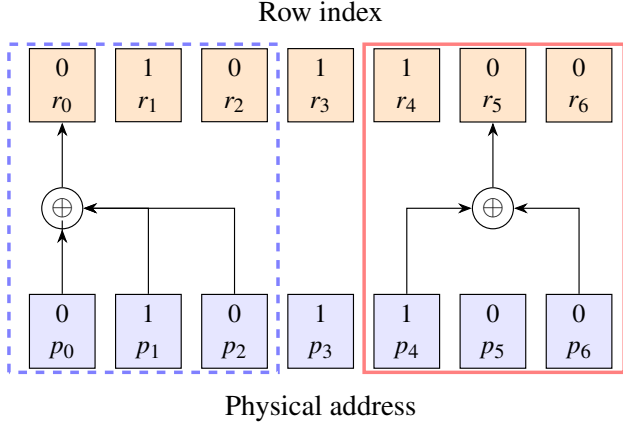


Figure 7: Two row addressing functions. The one highlighted in blue verifies item **C.1** while the one in red does not. r_i and p_i represent, respectively, the row and physical address bits.

Under the condition that \mathbb{S}_{low} contains a sufficiently large number m' of address pairs (see Section 7.2), the Rank-Nullity theorem [17] guarantees that:

$$\forall \mathbf{D}_{\text{row}} \in \{0, 1\}^{m' \times n}, \mathbb{R} = \text{nullspace}(\mathbf{D}_{\text{row}}).$$

We reduce \mathbb{R} by removing from it the parity masks that selects bits not causing a row conflict when flipped (*i.e.*, not contributing to the row-index computation). Thus, we remove the masks that select column bits in the address.

A basis \mathbf{R} of \mathbb{R} is equivalent to the \mathbf{R}_{true} basis underlying the real row address mapping, up to a change of basis (Section 5.2); that is, given any two addresses A and B , the row parity masks generated by \mathbf{R} provide the same classification for A and B (*i.e.*, they row-conflict or not) as the \mathbf{R}_{true} 's row parity masks.

The complete address mapping function f (Equation (1)) is defined, in matrix form, as:

$$\mathbf{F} = \begin{pmatrix} \mathbf{M} \\ \mathbf{R} \end{pmatrix}.$$

In the next section, we provide an algorithm to determine such a basis \mathbf{R} .

6.3 An Algorithm to Recover the Row-Mapping Function

We aim to recover a plausible basis $\mathbf{R} \in \{0, 1\}^{k' \times n}$ for the row parity verifying the following conditions:

- C.1** Each row \mathbf{R}_j includes bit position j , meaning $\mathbf{R}_j[j] = 1$, as illustrated in Figure 7. This pivot mirrors the common hardware implementation of a direct wire combined with a few XOR gates and gives us a canonical basis,

Algorithm 1 Rank-aware Row Basis Search

Require: $\mathbf{M} \in \{0, 1\}^{k \times n}$ — fixed bank-mask matrix,
 $\mathbb{R} \subseteq \{0, 1\}^n$ — candidate row-mask vectors,
 k' — target row-space dimension.

Ensure: $\mathbf{R} \in \{0, 1\}^{k' \times n}$ with $\text{rk}[\mathbf{M}; \mathbf{R}] = k + k'$ and minimal total Hamming weight.

```

1: function BACKTRACK( $j, \mathbf{B}, r, w$ )
2:   if  $r + (k' - j) < r_{\text{init}} + k'$  then  $\triangleright$  rank can no longer reach  $k + k'$ 
3:     return
4:   if  $j = k'$  then  $\triangleright$  full basis selected
5:     if  $w < w_{\text{min}}$  then
6:        $\mathbf{B}_{\text{best}} \leftarrow \mathbf{B}, w_{\text{min}} \leftarrow w$ 
7:     return
8:   for all  $V \in \mathbb{C}_j$  do
9:     if  $\text{rk}([\mathbf{M}; \mathbf{B}; V]) = r$  then  $\triangleright V$  linearly dependent
10:      continue
11:       $\mathbf{B}' \leftarrow [\mathbf{B}; V], r' \leftarrow r + 1, w' \leftarrow w + \text{HW}(V)$ 
12:      if  $w' < w_{\text{min}}$  then
13:        BACKTRACK( $j + 1, \mathbf{B}', r', w'$ )
14:
15:  $r_{\text{init}} \leftarrow \text{rk}(\mathbf{M})$   $\triangleright r_{\text{init}} = k$ 
16: for  $j \leftarrow 0$  to  $k' - 1$  do
17:    $\mathbb{C}_j \leftarrow \{V \in \mathbb{R} \mid V[j] = 1\}$   $\triangleright$  sort  $\mathbb{C}_j$  by increasing Hamming weight
18:
19:  $\mathbf{B}_{\text{best}} \leftarrow [], w_{\text{min}} \leftarrow \infty$ 
20: BACKTRACK( $0, [], r_{\text{init}}, 0$ )
21:  $\mathbf{R} \leftarrow \mathbf{B}_{\text{best}}$ 
22:
23: return  $\mathbf{R}$ 

```

- C.2** The basis has a Hamming weight as low as possible, implying a lower amount of logical gates in hardware. We hypothesize this as a reasonable assumption to minimize resources, which is sustained by previous findings [16, 20, 26],

- C.3** To guarantee that \mathbf{F} provides an equivalent addressing to the targeted physical-to-DRAM addressing function, the rank of \mathbf{R} must satisfy:

$$\text{rk}(\mathbf{F}) = \text{rk}(\mathbf{M}) + \text{rk}(\mathbf{R}).$$

We discuss the justification and implications of these hypotheses in Section 8. We remark that the linear independence of \mathbf{R} 's rows guarantees the surjectivity of the row addressing function.

Algorithm 1 describes a procedure that builds a basis \mathbf{R} satisfying the above-mentioned conditions. Firstly, the algorithm partitions the set of row masks \mathbb{R} into totally ordered sets $\mathbb{C}_j = \{V \in \mathbb{R} \mid V[j] = 1\}$, each sorted by Hamming weight (Line 16 – Line 17). Then, the algorithm starts the search for a basis \mathbf{R} by calling the recursive function `Backtrack` (Line 20). `Backtrack` searches in \mathbb{C}_j a row mask V that select the j -th row bit (Line 8 – Line 13). The function skips any row mask V that does not increase the current rank r of $[\mathbf{M}; \mathbf{B}]$ (*i.e.*, the composition of \mathbf{M} and \mathbf{B} along the row-axis) (Line 9 – Line 10); otherwise, `Backtrack` adds V to the current basis \mathbf{B} , updates the rank and the Hamming weight (Line 11), and calls itself to explore the set \mathbb{C}_{j+1} (Line 13).

The algorithm interrupts the recursive search (i.e., it backtracks) in two occasions: when the current basis \mathbf{B} cannot reach full rank (i.e., it cannot satisfy condition C.3) (Line 2 – Line 3); when the new identified basis has lower Hamming weight than the best basis found so far (Line 4 – Line 7). The in-order exploration of each set \mathbb{C}_j provides a row basis \mathbf{R} whose diagonal elements are set to 1 (i.e., the algorithm satisfies condition C.1). The algorithm provides a final basis \mathbf{B}_{best} such that $\text{rk}[\mathbf{M}; \mathbf{R}] = \text{rk}(\mathbf{M}) + k'$ (i.e., the algorithm satisfies condition C.3) while minimizing the basis' Hamming weight (i.e., the algorithm satisfies condition C.2).

Thus, from the previously retrieved bank masks and a new targeted latency analysis using these masks, we can build an addressing function for the rows.

Summary 3: Using the same methodology as for the banks in Section 5.2, our previously built coarse-grained row function, and hardware-based hypotheses, we build a plausible row addressing function that explains observed latencies.

7 Evaluation

7.1 Experimental setup

We evaluate our methodology in a range of platforms from embedded to server-class SoCs, and for three different Instruction Set Architectures (ISA). For x86 targets, we use the `clflush` instruction to evict the address pairs from the cache, allowing us to be sure of measuring DRAM latencies. We use the `rdtsc` instruction for timing measurements. On ARMv8, we use DC CIVAC for cache eviction, and the PMCCNTR cycle counter for measurement. On ppc64le we flush each cache line with the `dcbf` instruction and time the access using the 64-bit time-base counter read by `mftb`.

7.2 Number of Required Samples

To ensure the practical effectiveness of our methodology, we propose a theoretical upper bound to the cardinality $m = |\mathbb{S}|$ (i.e., the set of randomly generated address pairs) required to achieve bank mask recovery. With $t = n - k$ unknown mask dimensions and latency misclassification rate $\theta \in [0, 1)$, choosing m samples such that:

$$m \geq \frac{2^k}{1 - \theta} \cdot \log_2 \left(\frac{2^{n-k} - 1}{\epsilon} \right)$$

suffices to have $\text{rk}(\mathbf{D}) = t$ with a probability greater or equal than $1 - \epsilon$, with $\epsilon \in [0, 1)$ an arbitrarily defined accepted failure probability.

We provide a similar bound on the cardinality $m = |\mathbb{S}_{\text{low}}|$ (i.e., the set of address pairs in \mathbb{S} exhibiting low access latency): it is sufficient to replace k with k' (the number of row

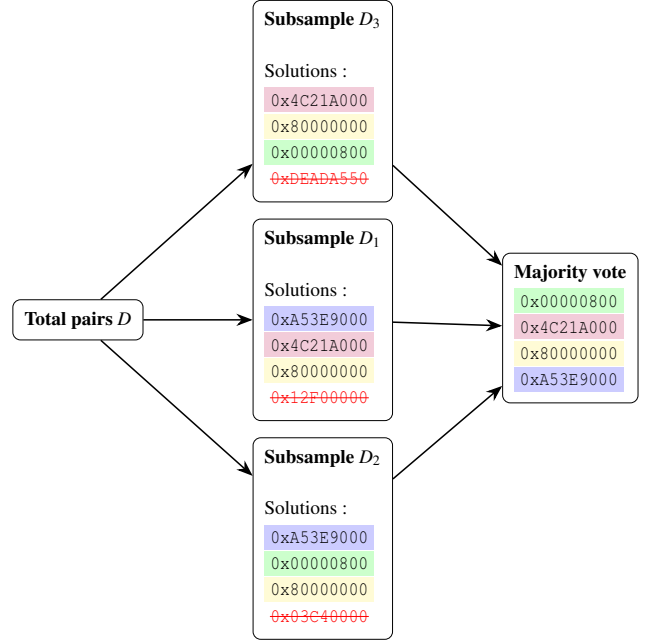


Figure 8: Subsampling with majority vote. Spurious masks (red strike-through) appear in only one subsample and are discarded; masks present in at least two subsamples survive in the final set.

parity masks), and n with $n - k$ (as we have already determined the identity of k out of n address bits).

The complete bound's derivation is provided in Appendix A.1.

7.3 Subsampling Technique

Empirical latency measurements inherently contain noise, causing occasional misclassifications of address pairs as conflicting or non-conflicting. Because such pairs do not verify the same relationships as their correctly-labeled counterparts, they increase the rank of the difference matrix \mathbf{D} , thus decreasing the dimension of the nullspace and reducing the number of found masks. To handle this noise, we introduce a subsampling technique illustrated in Figure 8:

- Instead of solving only once, we run our methodology on multiple smaller sets of pairs.
- Final masks are chosen by a majority vote, selecting masks that appear the most consistently.

7.4 Results and Validation

To validate our methodology and the implemented code, we first use synthetic data generated from functions retrieved by previous works [20] to check if Knock-Knock is able to give back \mathbf{D} the same mappings found by these works. In a

Table 2: Retrieved parity masks and address mappings for evaluated platforms.

| Platform | Architecture | SoC | DRAM | Comparison to Previous Work |
|---------------------------|--------------|-----------------------------|-------------|-----------------------------|
| Raspberry Pi 3B+ | ARMv8 | Broadcom BCM2837B0 | 1 GB LPDDR2 | Matches [18] |
| Google Pixel 3a | ARMv8 | Snapdragon 670 | 4 GB LPDDR4 | New |
| Switch P4 | x86 | Intel Pentium D1517 | 8 GB DDR4 | New |
| Dell Precision Tower 5810 | x86 | Intel E5-1650 v3 | 32 GB DDR4 | New |
| Dell Precision Tower 7875 | x86 | AMD Threadripper PRO 7955WX | 64 GB DDR5 | New |
| Dell PowerEdge R630 | x86 | Intel Xeon E5-2630 v3 | 128 GB DDR4 | Matches [26] |
| HPE Proliant DL360 Gen10+ | x86 | Intel Xeon Silver 4314 | 256 GB DDR4 | New |
| ThinkSystem SR630 V2 | x86 | Intel Xeon Gold 5318Y | 256 GB DDR4 | New |
| Nvidia DGX-1 | x86 | Intel Xeon E5-2698 v4 | 512 GB DDR4 | New |
| IBM PowerNV S822LC | ppc64le | IBM POWER8NVL 1.0 | 128 GB DDR4 | New |

second step, we use real-world datasets obtained by running Knock-Knock on target platforms as shown in Table 2.

In a first step, for targets with known mappings that we did not have access to, we predict whether a conflict would happen or not between two randomly generated addresses using the documented functions. Then, we input this artificial data into our algorithm to check its ability to retrieve the functions and compare them back with the mappings found in the original work [20]. We run experiments using a synthetic dataset built from up to 100,000 randomly generated addresses, and we find a 100% accuracy in the mapping functions found. With this method, we could also test for the tolerance to noise. We did so by modifying the dataset with up to 5% of misclassified timings. The results in this case, still giving a 100% accuracy of recovering, showed how the methodology can tolerate measurement noise.

Secondly, we evaluate the accuracy of our masks by measuring their ability to correctly predict high and low-latency address pairs. These are generated through an additional validation step using 10,000 new latency measurements. For this real-world data, we leverage the privileged `/proc/pagemap` interface to translate virtual addresses to physical addresses, and measure access latencies using high-resolution timers. We discuss how this requirement does not nullify the threat model in Section 8. When available, the recovered masks are validated against known hardware configurations or previous works. For the E5-2630 v3, while DRAMA [26] documents found masks on their 64GB setup, we could not reproduce the result using their source code [12] on our 128GB platform. This is probably due to the exponential search phase, reaching its timeout after a few hours, before fully achieving parity-mask recovery. Knock-Knock allowed us to retrieve similar masks to those documented in the original article in just a few minutes. The list of platforms and their respective results is summarized in Table 2 and Table 3. We find functions for previously undocumented targets, across different architectures and use cases, ranging from embedded to server-class hardware. We compare our functions with previous works [18, 26]

and find similar functions when the target matches. We use standard classification metrics like precision ($TP/TP+FP$) and recall ($TP/TP+FN$) to evaluate our results, where TP , FP , TN and FN are defined as follows:

TP: True positive - Predicted and real conflict;

FP: False positive - Predicted conflict, real non-conflict;

TN: True negative - Predicted and real non conflict;

FN: False negative - Predicted non-conflict, real conflict.

For each target, we evaluate the recall and the precision and find that they are both greater than 99%, meaning that the found masks correctly explain the observed conflicts.

Summary 4: Our method works with > 99% recall and precision, even in noisy environments. The results match previous reverse-engineering works, with a faster search phase and a black-box approach.

8 Discussion

Closed-Page Policy: We tested some platforms, notably the Raspberry Pi 4 4GB and the Nvidia Jetson Nano, for which we were unable to retrieve any part of the mapping. The latency measurements showed a single distribution, indicating a closed-page policy. A closed-page policy means that the memory controller closes the row after each access [3], which prevents us from differentiating between row hits and row conflicts. Because the row buffer is always closed after use, each memory access has to be served from the bank, eliminating the lower latency peak seen in Figure 2.

Privileged access to the pagemap: While we assumed privileged access to the `/proc/pagemap` interface, which is a common threat model [26, 30], we consider that this does not invalidate the attack vector that this work makes possible. Indeed, for tested devices, the mapping was consistent across different devices of the same model. Therefore, an attacker owning the same device as their victim could use the

knowledge obtained from characterising their device, and use it later on from a malicious process on the victim’s device, even if it only has access to the virtual address [2, 20]. The labeling methodology described in Sudoku [32] could be used as a post-processing step on top of Knock-Knock to improve human readability, although it does not bring any improvement in the considered use case of attacks building on known DRAM address mappings.

Ground truth: Our attack being purely based on software, we consider the translation from physical address to DRAM rows. This addressing relies on several different mappings, such as the memory controller’s mapping and the DRAM’s internal addressing. Our method cannot disentangle these mappings; instead, it treats the entire translation pipeline as a single black box. While this remains sufficient to mount attacks on the DRAM, such as Rowhammer, this approach may present limitations. For instance, modifying the CPU configuration may affect the memory controller map only, and not the DRAM internal mapping. This approach remains the state-of-the-art standard for software-based physical-to-DRAM addressing [9, 10, 13, 30, 32], where we improve previous methods with speed, reliability, and portability. Only a few works [5, 26] leveraged physical probing to get the ground truth necessary to reverse the memory controller mapping. We believe combining our method with physical probing could enable separating the memory controller and DRAM internal mappings, providing a more complete understanding of the translation process, potentially resulting in a finer-grained control of the DRAM. Such physical probing could also confirm the efficiency of our method by ruling out other sources of contention when using the row-buffer side channel.

Hypotheses: Another limitation of Knock-Knock lies in the hypotheses of Section 6.3. While we only need them to reverse physical-to-row mappings, which is one of the new contributions of Knock-Knock, we acknowledge that they still reduce the portability of our method. Without these, the indexing of rows might be false, resulting in an incorrect mapping. However, we consider these hypotheses realistic, as they were verified on all tested systems in the evaluation and compared with previous work when available [20, 26].

Applications on countermeasures Due to its automated and fast approach, Knock-Knock can be used to defeat mitigations based on DRAM-address randomization [24]. By just running Knock-Knock after each SoC boot, effectively reversing the randomization, an attacker could still implement attacks requiring physical-to-DRAM-addressing functions in a reasonable time. We leave the evaluation of the impact of Knock-Knock on existing countermeasures to future work.

9 Conclusion

In this work, we have presented *Knock-Knock*, a black-box and platform-agnostic methodology that formalizes physical-to-DRAM address-mapping reverse engineering. To achieve

so, we have developed an analytical and provable methodology that bounds the complexity of the search space of possible DRAM addressing functions. The key contribution of Knock-Knock is the formalization of physical-to-DRAM mapping reverse engineering, which enables retrieving functions with only timing measurements and elementary linear algebra. This formalization allows for improved noise resilience, which we tackled during the analytical phase of the methodology, compared to previous works that have tried to improve mislabeling during the data generation and reversing phase. We have validated our method on 10 machines, spanning from embedded SoCs to server-grade clusters, and achieved a 99% recall and precision rate on all targets, running in only a few minutes on systems with up to 512GB of DRAM. Knock-Knock paves the way to more extensive, precise, and generic studies on microarchitectural security of memory systems across different system classes and architectures. To that end, we publish our code and data in a public repository³.

Acknowledgments

Experiments presented in this paper were done on Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see <https://www.grid5000.fr>). This work is funded by the French Agence Nationale de la Recherche (ANR) Young Researchers (JCJC) program, under grant number ANR-21-CE39-0018 (project ATTILA).

References

- [1] Yang Bai, Yizhi Huang, Si Chen, and Renfa Li. PaLLOC: Pairwise-based low-latency online coordinated resource manager of last-level cache and memory bandwidth on multicore systems. *Journal of Systems Architecture*, 164:103427, 2025.
- [2] Michael Garrett Bechtel and Heechul Yun. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *IEEE Transactions on Computers*, 71(9):2351–2357, 2022.
- [3] Matthew Blackmore. A Quantitative Analysis of Memory Controller Page Policies. Master’s thesis, Portland State University, 2013.
- [4] Thomas John I’Anson Bromwich. *An introduction to the theory of infinite series*. AMS Chelsea Publishing, 1991.
- [5] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End

³<https://github.com/antpln/Knock-Knock>

- Methodology for Cloud Providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728. IEEE, 2020.
- [6] Kemal Derya, M. Caner Tol, and Berk Sunar. FAULT+PROBE: A Generic Rowhammer-based Bit Recovery Attack. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS '25*, page 1219–1234. ACM, 2025.
- [7] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.
- [8] Lukas Gerlach, Simon Schwarz, Nicolas Faröß, and Michael Schwarz. Efficient and Generic Microarchitectural Hash-Function Recovery. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3661–3678. IEEE, 2024.
- [9] Martin Heckel and Florian Adamsky. Reverse-Engineering Bank Addressing Functions on AMD CPUs. In *3rd Workshop on DRAM Security (DRAMSec)*, 2023.
- [10] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.
- [11] Jana Hofmann, Cédric Fournet, Boris Köpf, and Stavros Volos. Gaussian Elimination of Side-Channels: Linear Algebra for Memory Coloring. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24*, pages 2799–2813. ACM, 2024.
- [12] isec-tugraz. DRAMA - Source code repository. <https://github.com/isec-tugraz/drama>, June 2016. Commit used : c5c8347.
- [13] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölskei, and Kaveh Razavi. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1615–1633, Philadelphia, PA, 2024. USENIX Association.
- [14] JC-42. Low Power Double Data Rate 4 (LPDDR4) | JEDEC.
- [15] JC-42.3C. DDR4 SDRAM STANDARD | JEDEC.
- [16] Matthias Jung, Carl C. Rheinländer, Christian Weis, and Norbert Wehn. Reverse Engineering of DRAMs: Row Hammer with Crosshair. In *Proceedings of the Second International Symposium on Memory Systems, MEM-SYS '16*, pages 471–476. ACM, 2016.
- [17] Yitzhak Katznelson and Yonatan R. Katznelson. *A (Terse) Introduction to Linear Algebra*. American Mathematical Soc., 2008.
- [18] Anandpreet Kaur, Pravin Srivastav, and Bibhas Ghoshal. Flipping Bits Like a Pro: Precise Rowhammering on Embedded Devices. *IEEE Embedded Systems Letters*, 15(4):218–221, 2023.
- [19] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372. IEEE Computer Society, 2014.
- [20] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824. USENIX Association, 2022.
- [21] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriess, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710. USENIX Association, 2018.
- [22] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.
- [23] Michele Marazzi and Kaveh Razavi. RISC-H: Rowhammer Attacks on RISC-V. In *4th Workshop on DRAM Security (DRAMSec)*, 2024.
- [24] Brett Meadows, Nathan Edwards, and Sang-Yoon Chang. On-Chip Randomization for Memory Protection Against Hardware Supply Chain Attacks to DRAM. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 171–180. IEEE, 2020.
- [25] Xing Pan, Yayaswini Jyothi Gownivaripalli, and Frank Mueller. TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring. In *2016 IEEE*

- International Parallel and Distributed Processing Symposium (IPDPS)*, pages 363–372. IEEE Computer Society, 2016.
- [26] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581. USENIX Association, 2016.
- [27] Adnan Siraj Rakin, Md Hafizul Islam Chowdhury, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1157–1174. IEEE, 2022.
- [28] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017*, volume 10327 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2017.
- [29] Hans Vandierendonck and Koenraad De Bosschere. XOR-Based Hash Functions. *IEEE Transactions on Computers*, 54(7):800–812, 2005.
- [30] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
- [31] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S. Kim, Juan Gómez Luna, Mohammad Sadrosadati, Nika Mansouri Ghiasi, and Onur Mutlu. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 313–328, 2020.
- [32] Minbok Wi, Seungmin Baek, Seonyong Park, Mattan Erez, and Jung Ho Ahn. Sudoku: Decomposing DRAM Address Mapping into Component Functions. In *5th Workshop on DRAM Security (DRAMSec)*, 2025.
- [33] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 333–346. IEEE, 2023.
- [34] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row Buffer Locality-Aware Data Placement in Hybrid Memories. *SAFARI Technical Report*, 5, 2011.

Table 3: Retrieved parity masks and address mappings for evaluated platforms. [x,y] indicate that the masks have bits x and y set, while for masks with multiple bits we describe them using their hexadecimal representations.

| Platform | Retrieved Bank/Channel Masks |
|------------------------------|--|
| Raspberry Pi 3B+ | [13], [14], [15] |
| Google Pixel 3a Switch P4 | 0x274e9000, 0x69d3a000, 0x53a74000, 0x80000000 [6,20], [17, 21], [18,22], [19, 23], [32, 33] |
| Dell Precision Tower 5810 | 0x8000, 0x100000000, 0x200000000, 0x400000000, 0x8000040, 0x1100000, 0x2200000, 0x4400000, 0x55080, 0x88a2100 |
| Dell Precision Tower 7875 | 0x84201000, 0x40214100, 0x188400200, 0x1421002000, 0x310800400, 0x1842100800, 0xff80000, 0xd6f700440 |
| Dell PowerEdge R630 | 0x800040, 0xa00000000, 0xc00000000, 0x3000000000, 0x4408000, 0x2820000000, 0x5500000, 0x6600000, 0x88a2100, 0x4455080 |
| HPE Proliant DL360 Gen10+ | [15], [35], [36], [37], [6,23], [20,24], [21,25], [22, 26], 0x4004100, 0x6024800 |
| Nvidia DGX-1 | [37], [38], [16], [15], [21,25], [6, 24], [7, 17], [23, 27], [22, 26], [8, 12, 14, 18, 20, 24] |
| ThinkSystem SR630 V2 | [16], [35], [36], [37], [6, 24], [21, 25], [22, 26], [23, 27], [8, 14, 26], [9, 15, 27], [11, 14, 17, 25, 26] |
| IBM PowerNV S822LC | [7], [8], [9], [10], [11], [12], [13], [14], [15], [32,34], [33,34] |

A Appendix

A.1 Sample Complexity for Recovering the Bank/Channel Masks

For our methodology, it is important to randomly generate a sufficient number m of n -bit address pairs to satisfy Equation (7) (Section 5.2).

In practice, we have to bound s such that:

$$P[\text{rk}(\mathbf{D}) = t] \geq 1 - \varepsilon, \quad (10)$$

where \mathbf{D} is the difference matrix of size $m \times n$, $t = n - k$ is the targeted rank for \mathbf{D} , k is the dimension of nullspace(\mathbf{D}), and $\varepsilon \in [0, 1)$ is the arbitrarily defined accepted failure probability. We define $\theta \in [0, 1)$ as the proportion of misclassified pairs, *i.e.*, the proportion of address pairs that are not conflicts but are classified as such. We do not consider $\varepsilon = 1$ and $\theta = 1$ as, in such cases, it is not possible to recover the bank and channel masks.

Then, we define m_c as the number of address pairs classified as conflicts. Therefore, for a required m_c conflicting pairs, we need to randomly generate m address pairs such that:

$$m = \frac{2^k}{1 - \theta} \cdot m_c. \quad (11)$$

To simplify the computation, we calculate a lower bound for this probability using $\mathbf{D}_t \in \{0, 1\}^{m_c \times t}$, which only contains the t first columns of \mathbf{D} .

By construction :

$$P[\text{rk}(\mathbf{D}) = t] \geq P[\text{rk}(\mathbf{D}_t) = t].$$

We denote d_j as the j -th column of \mathbf{D}_t . We remark that all \mathbf{D}_t 's columns are randomly drawn with replacement from a uniform distribution over $\{0, 1\}^t$.

Let us assume that we have already found j linearly independent columns. These columns span a set of vectors of size 2^j . Therefore, the probability that column d_{j+1} is linearly independent from the already chosen columns is:

$$P[d_{j+1} \text{ is linearly independent}] = 1 - \frac{2^j}{2^{m_c}} = 1 - 2^{j-m_c}.$$

Then, the probability of randomly drawing t linearly independent rows is:

$$P[\text{rk}(\mathbf{D}_t) = t] = \prod_{j=0}^{t-1} (1 - 2^{j-m_c}).$$

According to the Weierstrass' product inequality [4]:

$$P[\text{rk}(\mathbf{D}_t) = t] \geq 1 - 2^{-m_c} \cdot \sum_{j=0}^{t-1} (2^j - 1) = 1 - \frac{2^t - 1}{2^{m_c}} \quad (12)$$

Combining Equation (10) and Equation (12), we have that:

$$\varepsilon = \frac{2^t - 1}{2^{m_c}}. \quad (13)$$

By applying Equation (11) to Equation (13), we find that Equation (10) is satisfied for:

$$m \geq \frac{2^k}{1 - \theta} \cdot \log_2\left(\frac{2^{n-k} - 1}{\varepsilon}\right).$$

As an example, for $n = 32$ bit addresses (*i.e.*, 4 GB Memory), $k = 4$ (*i.e.*, 8 banks and 2 channels), $\theta = 0.05$ (*i.e.*, 5% misclassifications), and $\varepsilon = 0.01$ we would need $m \geq 584$ randomly generated address pairs to guarantee the identification of parity masks for banks and channels.

To guarantee the recovery of the row parity masks, we follow the same reasoning to provide a bound similar to the one we have derived for bank parity masks:

$$m' \geq \frac{2^{k'}}{1-\theta} \cdot \log_2\left(\frac{2^{n-k-k'}-1}{\epsilon}\right).$$

Using the same parameters defined in the previous example, and for $k' = 4$ row bits, we would need $m' \geq 517$ address pairs exhibiting low access latency.