VeriPRAC: Verifiable Formal Contracts for Secure Per Row Activation Counting

Ali Hajiabadi
ETH Zurich
Zurich, Switzerland
ahajiabadi@ethz.ch

Silvan Niederer
ETH Zurich
Zurich, Switzerland
nsilvan@ethz.ch

Kaveh Razavi
ETH Zurich
Zurich, Switzerland
kaveh@ethz.ch

Abstract

Rowhammer remains a major security concern for modern DRAM, yet no deterministic solution exists in commercial devices. The latest JEDEC specification introduces Per Row Activation Counting (PRAC) to provide a principled framework for secure in-DRAM mitigation. Since the introduction of PRAC, a number of mitigations have been proposed based on PRAC. However, these proposals only provide ad hoc security analyses without necessarily considering the worst-case scenarios. In this work, we define PRAC contracts that provide formal bounds for a generic set of mechanisms in a PRAC mitigation. Our key insight is that these generic mechanisms introduce gaps in tracking accesses to DRAM, making it possible for an attacker to craft a variety of strategies that silently survive the mitigation actions performed by PRAC. Using these silent survival strategies, we then construct EchoTrail, a provably optimal attack structure against PRAC contracts. Leveraging EchoTrail, we then develop VeriPRAC, an automatic contract verification tool that checks a given PRAC-based mitigation against its intended PRAC contract. VeriPRAC finds that all state-of-the-art mitigations violate their intended contract, and in one case, an aggressor can reach up to 7.5k activations despite the intended bound of the contract being only 99.

CCS Concepts

• **Security and privacy → Hardware attacks and countermeasures**; **Formal security models**.

Keywords

Rowhammer, DRAM, Reliability, Per Row Activation Counting

ACM Reference Format:
Ali Hajiabadi, Silvan Niederer, and Kaveh Razavi. 2026. VeriPRAC: Verifiable Formal Contracts for Secure Per Row Activation Counting. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3830454.3832626

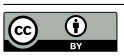
1 Introduction

Rowhammer attacks continue to compromise system security in a variety of scenarios [4–7, 10, 11, 18, 28, 31]. To address Rowhammer, JEDEC has recently introduced Per Row Activation Counting (PRAC) as a basis for designing secure mitigations [12]. Without a formal foundation to reason about the security of PRAC, however, existing mitigation proposals [3, 25, 34] resort to ad hoc security analyses that do not necessarily consider worst-case scenarios. This paper addresses this gap by constructing formal contracts for PRAC-based mitigations and provably-optimal attack patterns against these contracts.

PRAC contracts. The PRAC specification defines a framework that enables communication between the DRAM device and the CPU for reactive mitigations, and assume that the DRAM tracks activations on a per row basis [12]. In principle, this combination *can* comprehensively mitigate Rowhammer. However, there are many different ways to build PRAC-based mitigations since the JEDEC specification intentionally leaves out important details which ultimately determine the actual security guarantees of such mitigations. For instance, the mitigation needs to determine what the per row counters should actually count, when counters should reset, and how the mitigation should queue critical rows when the demand exceeds the mitigation rate allowed by the communication protocol. Prior work has proposed specific implementations [3, 25, 34], and while they all do some ad hoc security analysis, it remains unclear whether they truly consider the worst-case scenarios for the particular set of design choices that they make. This demonstrates the urgent need for a formal foundation and rigorous security analysis of PRAC to prevent devastating flaws in a mitigation that is supposed to end the Rowhammer battle once and for all. In this work, we introduce *PRAC contracts* that formalize the fundamental mechanisms behind PRAC-based mitigations and the security bounds that they provide. To calculate the security bounds, however, we need to know the optimal attacks against these contracts.

EchoTrail. Existing PRAC mitigation proposals, regardless of their contract, often rely on assumptions and strategies from prior work on proactive mitigations [19] to analyze their security. However, we show that these assumptions do not directly translate to reactive mitigations such as PRAC. Reactive mitigations are typically inactive during normal operation and trigger mitigations only when necessary. This fundamental difference from prior proactive mitigations introduces new, *silent strategies* that allow an attacker to evade PRAC mechanisms and prevent victim hammering from being observed altogether. Our security analysis of these differences leads to the EchoTrail attack structure, which constructs optimal attack patterns and determines the optimal placement of different silent and active strategies, each with configurable durations. We show that different configurations of EchoTrail cover all PRAC contracts considered in this work, allowing us to compute optimal security bounds for each of these contracts.

VeriPRAC. Building on the EchoTrail attack, we develop a contract verification tool that *automates* the security analysis of a given



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832626

PRAC implementation which is accompanied by its intended contract (i.e., expected security bounds). VeriPRAC is designed in a modular fashion: the implementation of a PRAC-based mitigation is integrated by modifying only the specific functions that implement design choices such as counting, resetting, and queuing mechanisms. The EchoTrail execution unit within VeriPRAC then tests a mitigation's implementation against a comprehensive set of PRAC contracts, ranging from the most secure ($PRAC_{ideal}$) to the most insecure ($PRAC_{disabled}$), and reports the strongest attack for that implementation. An implementation satisfies its contract if the reported strongest attack is equal to or weaker than the intended one. VeriPRAC thus enables early detection of implementation flaws before a mitigation is deployed in final products.

Applying VeriPRAC. We integrate and evaluate two state-of-the-art PRAC mitigations in VeriPRAC: MOAT [25] and QPRAC [34]. The results show that both mitigations *violate* their contracts. Our analysis of MOAT demonstrates that its security is equivalent to a PRAC contract that only counts explicit activation commands (i.e., ignoring the activations induced by mitigative refreshes), while the intended contract counts both explicit activations and implicit activations. This means that MOAT's effective security is equivalent to a naive PRAC configuration that only counts explicit activations; reaching up to $7.5k$ activations on an aggressor, while the intended contract's bound is only 99. This vulnerability arises from an overlooked implementation choice in the queuing mechanism, which can be easily fixed using the VeriPRAC analysis, enabling the deployment of a secure version of MOAT on a device. Our analysis of QPRAC shows that it also violates its contract, where the VeriPRAC's reported contract has a longer attack duration (i.e., more opportunities for attack), compared to the one claimed in the original security analysis of the work. These results directly show the benefits of relying on VeriPRAC for security analysis of secure PRAC implementations in the future.

Contributions. The following summarizes our contributions:

- We provide a formal foundation for the rigorous security analysis of PRAC. Our analysis formalizes PRAC contracts and introduces the EchoTrail attack, which identifies and expresses the optimal attack strategies, sequences, and durations for a given PRAC contract by making a selective use of silent strategies.
- We develop VeriPRAC, a contract verification tool that automatically tests the contract satisfaction of a PRAC implementation. VeriPRAC enables early detection of design flaws before deployment.
- We integrate two state-of-the-art PRAC implementation proposals in VeriPRAC and uncover that neither one of them satisfies their intended contract, highlighting the importance of formal and automated security analysis during the early design stages of PRAC mitigations.

You can find more information about VeriPRAC, including its artifacts and verification tool, using the following link: https://comsec.ethz.ch/veriprac.

2 Background

In this section, we provide the necessary background on DRAM (Section 2.1), Rowhammer attacks (Section 2.2), and Rowhammer mitigations (Section 2.3). We then discuss prior formal security analyses of proactive mitigations (Section 2.4), followed by the new mechanisms introduced by PRAC and its reactive mitigations (Section 2.5).

Table 1: DDR5 timing with the PRAC enabled or disabled.

Parameter	Description	PRAC disabled	PRAC enabled
tRC	the time between two consecutive ACTs	46 ns	52 ns
tRP	the time to precharge an open row	14 ns	36 ns
tRAS	minimum time for a row to be open	32 ns	16 ns
tREFI	refresh interval between two REFs	3.9 µs	
tREFW	refresh period	32 ms	
tRFC	the execution time of the REF command	410 ns	
tRFM	the execution time of the RFM command	350 ns	

2.1 DRAM organization and operation

DRAM is the primary form of main memory in modern systems. On a last-level cache (LLC) miss, the memory controller (MC) issues a sequence of DRAM commands to read or write data. Internally, DRAM consists of multiple ranks, each with several banks (e.g., 32 in DDR5), and each bank is an array of rows. Rows hold memory cells, where each cell stores a single bit using a capacitor. To operate reliably, the MC follows JEDEC DRAM standards. Since rows lose charge over time, the MC periodically issues refresh (REF) commands to restore charge in row groups, pausing other operations for *tRFC* (410 ns in DDR5).

To access data, the MC uses a (rank, bank, row) address and issues an activate (ACT) command to load the target row into a row buffer for reading or writing. A precharge (PRE) command deactivates the bank and prepares the buffer for the next ACT. DDR5 introduces a Refresh Management (RFM) command, giving DRAM extra time to resolve data integrity issues such as Rowhammer, as we will discuss. Table 1 summarizes DDR5 timing parameters. *tREFI* defines the interval between REF commands, and *tREFW* is the window in which all row groups must be refreshed at least once.

2.2 Rowhammer

Rowhammer is a phenomenon in DRAM first revealed by Kim et al. [15], where repeatedly activating an aggressor row can compromise the data integrity of adjacent victim rows. These repeated activations accelerate charge leakage, potentially causing bit flips before the victim rows are refreshed within a *tREFW*. Rowhammer vulnerabilities have been exploited in numerous attacks to compromise system security [2, 4, 13, 17, 18, 23, 28, 29, 31–33], and more recently demonstrating the vulnerability of DDR5 devices [11, 20].

2.3 Rowhammer mitigations

Many solutions have been proposed to mitigate Rowhammer, falling into two categories: (1) in-CPU and (2) in-DRAM solutions. Many proposals focused on in-CPU defenses, integrating mitigations into the MC [8, 15, 21, 22, 27, 30]. However, since CPU vendors have little incentive to address a DRAM-specific issue, later efforts moved mitigations into DRAM itself; the natural location to solve Rowhammer [1, 9, 14, 19, 26].

Designing in-DRAM mitigations is challenging because they often lack sufficient SRAM for precise tracking, and additionally,

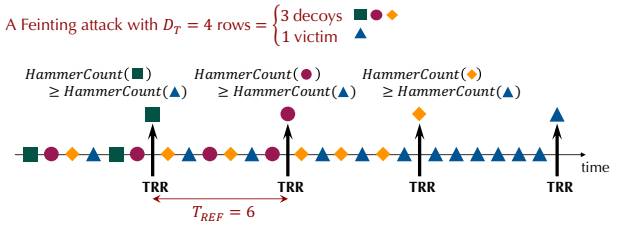


Figure 1: Activation pattern in Feinting with three decoys and one victim. Each TRR removes one decoy, leaving the victim (blue triangle) as the final TRR target.

must act within strict timing constraints. Early in-DRAM solutions borrow time from regular REF operations to refresh potential victim rows, a method known as Target Row Refresh (TRR). That is, they *proactively* refresh victim rows before they experience a bit flip [9, 14, 19, 26]. In Section 2.5, we discuss Per Row Activation Counting (PRAC), recently introduced in JEDEC standards, which addresses both space and timing challenges to enable *reactive* mitigations.

2.4 Feinting: security of proactive mitigations

Marazzi et al. [19] formally analyze the security of proactive in-DRAM mitigations and provide an optimal attack against ideal TRR. In this model, TRR refreshes a limited number of rows (V) at each periodic REF, always refreshing the V most hammered rows. Given T_{REF} possible activation between two consecutive TRRs, Marazzi et al. introduce the optimal attack, called Feinting.

The Feinting attack lasts *tREFW*, the natural refresh window of the victim row. More importantly, Feinting does not spend its entire activation budget on hammering the victim row, since this would immediately make the victim the most hammered row and refreshed by the first TRR mitigation. Instead, to maximize the hammer count, Feinting selects a set of decoy rows and hammers them alongside the victim. This strategy allows the victim row to survive TRR mitigations until at least V other decoy rows have been hammered as frequently as the victim. The activation budget of Feinting is distributed *symmetrically* across all active decoy rows (i.e., those not yet refreshed by TRR events) and the victim row. Figure 1 illustrates a simplified example with $V = 1$, $T_{REF} = 6$, and four TRR events before the victim is refreshed.

2.5 PRAC: enabling reactive mitigations

To address the space and timing challenges of proactive in-DRAM mitigations, the latest JEDEC DDR5 specification introduces Per Row Activation Counting (PRAC). PRAC integrates per row activation counters within the DRAM arrays to alleviate the space limitations of precise activation tracking. It also requires updates to DDR5 timing parameters to account for the added latency of counter updates. Table 1 lists the revised timings; for instance, *tRC* increases from 46 ns to 52 ns, and *tRP* from 14 ns to 36 ns.

To address timing challenges, the new specifications introduce ALERT Back-off (ABO) protocol, which allows DRAM request extra time for mitigation when needed, enabling *reactive* mitigations. Figure 2 illustrates the ABO protocol. When DRAM triggers an ALERT, normal activity continues for 180 ns (equivalent to three

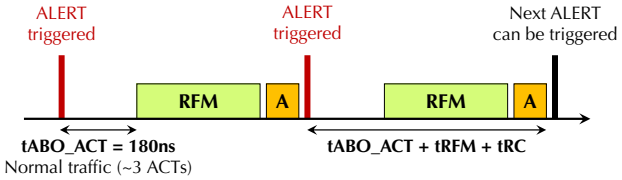


Figure 2: ABO protocol with mitigation level of 1.

ACTs to the same bank), after which the MC issues an RFM to be used for mitigation. ABO also enforces a minimum number of ACTs before the next ALERT can occur. The number of RFMs and post-RFM ACTs corresponds to the ABO level, which can be 1, 2, or 4.

3 Requirements of a PRAC Mitigation

Since the introduction of PRAC specification [12], the architecture community has proposed defenses leveraging per row counters and PRAC's reactive mitigations [3, 25, 34]. What remains missing is a rigorous security analysis of PRAC that identifies potential pitfalls and enables systematic, secure configuration of PRAC. This section outlines the key questions for secure configuration of PRAC mitigations.

Counting mechanism in PRAC. One of the notable ambiguities in the JEDEC specification is the lack of clarity regarding what per row counters should actually count (i.e., the counting mechanism). Here are some quotes from the JEDEC manual regarding PRAC (Section 16 of JESD79-5C_v1.30 [12]):

> *JESD79-5C_v1.30 (Section 16.1):* **Activations to a row include the ACT, REF and RFM commands**, and the MPC Manual ECS function.

> *JESD79-5C_v1.30 (Section 16.3):* Per Row Activation Counting (PRAC) requires additional counter cells per row in the DRAM to store **Activation (ACT) command counts**.

> *JESD79-5C_v1.30 (Section 16.4):* Since the DRAM is **counting ACT commands on a per row basis**, action may be required upon one or more rows reaching a counter threshold level or levels, or when a queue holding row addresses for targeted refreshing reaches a critical level.

A naive PRAC implementation increments per-row counters only on ACT commands (*basic counting*). However, REF and RFM also induce implicit activations that can cause bit flips in adjacent rows of the refreshed rows [16], leaving such activations untracked under basic counting. Although the JEDEC specification does not specify the counting mechanism, a secure PRAC design should account for both explicit and implicit activations (*complete counting*). Nevertheless, analyzing basic counting remains important: we show that some implementation flaws, as we observed in the state of the art [25], can effectively reduce a mitigation with complete counting to the security level of basic counting.

Question 1: What are the security implications of different ***counting*** mechanisms in a PRAC mitigation?

Queuing mechanism in PRAC. Ideally, a reactive mitigation should immediately mitigate an aggressor row once it reaches a critical activation level. In practice, however, PRAC's back-off protocol allows additional activations between consecutive ALERTs. When the number of rows reaching the threshold exceeds the mitigation volume of a single ALERT, a service queue becomes necessary to track these critical rows. Consequently, the design of this queuing mechanism becomes a key security component, as it directly determines the security guarantees of a PRAC mitigation.

Question 2: How should a PRAC mitigation ***queue*** critical aggressor rows to ensure timely and secure mitigations?

Counter resetting mechanism in PRAC. Since activations only increase per-row counters, a PRAC mitigation must include a reset mechanism to prevent counter saturation. The choice of resetting policy is critical for security, as it can potentially create tracking gaps and lose information about certain aggressor rows (e.g., resetting an unmitigated row). Existing designs employ different resetting mechanisms like periodically during refreshes or only upon ALERTs, yet their security analyses often ignore how these choices affect the optimal attack.

Question 3: What are the security implications of ***counter resetting*** mechanisms in a PRAC mitigation?

To answer these questions, we formalize the core elements of PRAC mitigations in Section 4 and conduct a rigorous security analysis in Section 5. The outcome of this analysis is EchoTrail, which provides a unified way to express the security of PRAC mitigations. Building on EchoTrail, we develop VeriPRAC in Section 6 which enables an automatic security analysis.

4 PRAC Contracts

A PRAC *contract* represents a generic mitigation that specifies the key elements of a PRAC mitigation, as discussed in Section 3, while a PRAC *implementation* is the concrete implementation of such a contract. The goal of formalizing and analyzing a PRAC contract is to specify (1) the expected behavior of an implementation, and (2) the security guarantees of the contract by identifying the optimal attacks against it. This contract can then serve as a reference when evaluating an implementation, allowing us to verify whether the implementation remains faithful to its intended contract. In this section, we aim to first define a PRAC contract (Section 4.1), and then provide the necessary definitions to reason about the security of such contracts (Section 4.2).

4.1 Elements of a PRAC contract

We define a PRAC contract as $PRAC_{\mathbf{c}}$ where it has four elements expressed as $\mathbf{c} = \langle \mathcal{Q} | C | \mathcal{R} | \mathcal{P} \rangle$: (1) queuing element $\mathcal{Q}$, (2) counting element C, (3) counter resetting element $\mathcal{R}$, and (4) back-off protocol $\mathcal{P}$. While JEDEC standards have a clear specification for the back-off protocol $\mathcal{P}$, it is up to the mitigation to specify the other elements.

Whenever a MC command is received by the DRAM, these elements are queried to update the PRAC states and queues. At any given moment, the PRAC contract has a *pracState* and a *queueState*; the former indicates the per row activation counter values stored by PRAC, and the latter indicates the rows that are queued for mitigation. The *pracState* is updated only by the $\mathcal{Q}$, C, and $\mathcal{R}$ elements of the PRAC contract. Table 2 describes the interface of each element.

Table 2: Signatures of a PRAC contract $\mathbf{c}$.

Element	Functions
C	$update : cmds \times pracStates \rightarrow pracStates$
$\mathcal{R}$	$update : cmds \times pracStates \rightarrow pracStates$
$\mathcal{Q}$	$AlertTrigger : cmds \times queueStates \rightarrow \{0, 1\}$ $AlertQ.update : cmds \times queueStates \rightarrow queueStates$ $AlertQ.head : cmds \times queueStates \rightarrow \{d_1, \ldots, d_V\} \vee \emptyset$

4.1.1 Queuing element $\mathcal{Q}$. A queuing element decides *when* to trigger an ALERT and *which* row(s) to mitigate. We define a queuing element as $\mathcal{Q} = \langle A_{thresh}, AlertTrigger, AlertQ \rangle$ where it has three components: (1) ALERT threshold A_{thresh}, (2) ALERT trigger method $AlertTrigger$, and (3) the ALERT queue structure $AlertQ$.

A_{thresh} indicates the critical activation count level based on the per row counters, and $AlertTrigger$ indicates when to trigger an ALERT. For example, one possible $AlertTrigger$ method is **Greater-or-Equal** (ge) that signals a row as critical whenever its per row counter becomes greater than or equal to A_{thresh}. Note that A_{thresh} is distinct from the Rowhammer threshold (i.e., R_{thresh}). The former is a mitigation parameter, whereas the latter is a property of the DRAM device; we will define Rowhammer threshold in Definitions 3 and 4.

The rows for which the $AlertTrigger$ asserts a signal must be forwarded to the $AlertQ$ for queuing. More precisely, the $AlertQ$ serves as a service queue that buffers rows signaled as critical by the $AlertTrigger$, and upon each mitigation event (i.e., when an RFM command is issued by the MC), it selects a subset of these buffered rows for mitigation. The $AlertQ$ can be implemented in different ways, e.g., prior work [34] uses a priority queue that always keeps the rows with the highest PRAC counter values.

4.1.2 Counting element C. A counting element decides *when* to increment a per row counter. As discussed in Section 3, we consider two counting mechanisms C:

- **Basic** (*basic*): this mechanism increments the $pracState(a)$ counter for row a only upon explicit ACT(a) commands.
- **Complete** (*complete*): this mechanism increments $pracState(a)$ on each ACT(a), REF on row a, and RFM refreshes on row a.

4.1.3 Counter resetting element $\mathcal{R}$. A PRAC contract also needs to configure the mechanism to reset per row counters. For example, once row a is mitigated, counter $pracState(a)$ can be reset (i.e., when an RFM is received and $AlertQ$ selects row a for mitigation) [34]. Some mitigations also reset the counters in a more fine-grained fashion on periodic REF commands [25].

4.1.4 Back-off protocol $\mathcal{P}$. Protocol $\mathcal{P} = \langle t\_preM, V, postM \rangle$, as illustrated in Figure 3, captures the sequence of actions during an ALERT, where V is the reactive mitigation volume (i.e., the number of RFMs and rows that are mitigated per ALERT). In addition, t_preM denotes the time between the ALERT trigger and the reception of the RFM. $preM$ represents the maximum number of possible ACTs during t_preM, defined as $preM = \lfloor t\_preM / tRC \rfloor$. Similarly, $postM$ refers to the number of ACTs required after the mitigation(s) before

the next ALERT can be triggered. JEDEC's ABO protocols can be defined as $\mathcal{P}_{ABO} = \langle 180\,\text{ns}, L, L\rangle$, where L denotes the mitigation level and can take values from the set $L = \{1, 2, 4\}$.

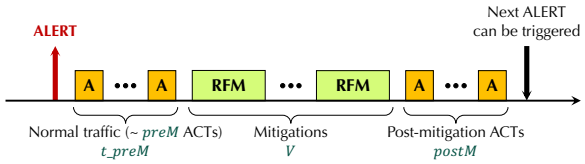


Figure 3: Back-off protocol $\mathcal{P} = \langle t\_preM, V, postM\rangle$.

4.2 Rowhammer and security definitions

First, we distinguish between the activation count tracked by a PRAC contract (i.e., *pracState*) and the actual number of activations performed during an attack. This distinction is crucial for understanding the gap between the PRAC contract states and the true extent to which an aggressor has been activated.

Definition 1: Activation count

An aggressor row $\tilde{r}$ is activated either when explicitly activated with the ACT command or when activations are induced by REF and RFM commands. *Activation count* of an aggressor row refers to the number of times it has been activated since all its victim rows within the blast radius have been refreshed. We denote the activation count of row $\tilde{r}$ as $a(\tilde{r})$.

Definition 2: PRAC activation count

$p(\tilde{r})$ denotes the current value of the PRAC counter (*pracState*) for aggressor row $\tilde{r}$.

In addition, we define the victim hammer count in Definition 10 in Appendix A. Informally, a victim row is hammered whenever any of its aggressor rows within the blast radius is activated. We denote the hammer count of row $\tilde{v}$ as $h(\tilde{v})$.

Next, we provide definitions for the Rowhammer threshold and mitigation security. While prior formalization efforts [19] base their analysis on *victim-tracking* mitigations (i.e., tracking the victim hammer counts $h(\tilde{v})$), we derive our definitions and analysis for *aggressor-tracking* mitigations in order to align with the JEDEC specification for PRAC, as well as our $PRAC_c$ contract.

Rowhammer threshold. We carefully derive safe threshold bounds that are specifically applicable to aggressor-tracking mitigations.

Definition 3: Rowhammer threshold R_{thresh}

Rowhammer threshold R_{thresh} is the minimum $h(\tilde{v})$ required to flip a bit in a victim row $\tilde{v}$.

In other words, R_{thresh} is the cumulative required number of activations to the adjacent aggressor(s) of the victim row $\tilde{v}$, within the blast radius, for causing a bit flip in row $\tilde{v}$.

Definition 4: Critical aggressor activation threshold

$\mathcal{F}(R_{thresh})$ provides a critical threshold for aggressor activation counts in a way that:

$$\forall \tilde{r} \in Rows : a(\tilde{r}) < \mathcal{F}(R_{thresh})$$
$$\implies \forall \tilde{v} \in Rows : \sum_{i=1}^{B} \Big(a(\tilde{v}+i) + a(\tilde{v}-i)\Big) < R_{thresh}$$

We assume a worst-case $\mathcal{F}(R_{thresh}) = \lfloor R_{thresh}/(B \times 2)\rfloor$, where the attacker distributes its activations evenly across all aggressor rows within the blast radius of the victim. This represents the worst case because it allows *all* aggressors to reach their maximum limit (i.e., $\mathcal{F}(R_{thresh})$) while still satisfying Definition 4.

Mitigation security. We provide definitions to asses the security of aggressor tracking mitigations, and more specifically, PRAC contracts. To achieve this, we provide the definitions for optimal Rowhammer attacks that follow prior formalization efforts [19], and then define the contract satisfaction of PRAC contracts.

Definition 5: Rowhammer attack

A Rowhammer attack $\mathcal{S}$ is a finite sequence of $\mathcal{L}_{attack}$ activations, and $A(\mathcal{S}, \mathcal{M})$ is the highest aggressor activation count $a(\tilde{r})$ for $\tilde{r} \in Rows$ during the attack when mitigation $\mathcal{M}$ is enabled. $\mathcal{U}$ denotes the set of all possible $\mathcal{S}$ attack sequences.

Given the Rowhammer attack definition, we define the optimal attack against a given mitigation $\mathcal{M}$.

Definition 6: $A_{max}(.)$ and optimal Rowhammer attack

We define $A_{max}(\mathcal{M})$ as:

$$A_{max}(\mathcal{M}) = \max_{s \in \mathcal{U}} A(s, \mathcal{M})$$

Equivalently, sequence $\mathcal{S}^{\star}(\mathcal{M})$ is the optimal attack against mitigation $\mathcal{M}$ iff

$$\mathcal{S}^{\star}(\mathcal{M}) \in \arg\max_{s \in \mathcal{U}} A(s, \mathcal{M})$$

Given the optimal attack definitions, we define *contract satisfaction* of a given PRAC implementation against its contract.

Definition 7: PRAC contract satisfaction

A mitigation implementation $\mathcal{M}$ satisfies a contract $PRAC_c$ iff: $A_{max}(\mathcal{M}) \leq A_{max}(PRAC_c)$

Table 3 in Appendix A provides a summary of notations and symbols used in our analysis.

5 EchoTrail

In this section, we aim to identify the optimal attacks against different PRAC contracts. In Section 5.1, we first show how gaps in a PRAC contract enable attackers to evade mitigations or accelerate their attacks. Based on these observations, we define a baseline contract, referred to as $PRAC_{base}$, in Section 5.2 and analyze its security, which leads to the EchoTrail attack structure. We then study how the main elements of a PRAC contract (counting, counter resetting, and queuing) affect security in Sections 5.3, 5.4, and 5.5, respectively, and show that they all conform to the EchoTrail structure. In other

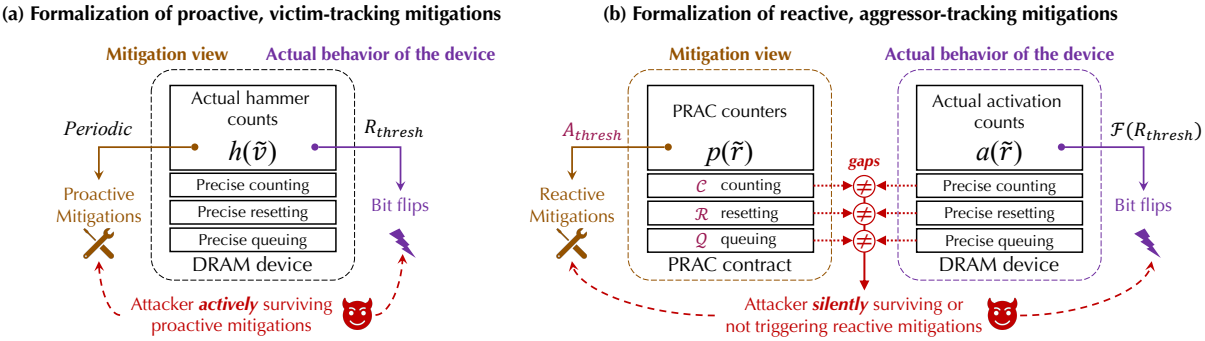


Figure 4: (a) Prior formalization of proactive mitigations [19], with a precise view of the DRAM device behavior. (b) Gaps between what PRAC contracts consider and the actual impact of the aggressor activations, resulting in silent survival strategies.

words, we prove that the optimal attacks against the common PRAC contracts analyzed in this work all follow the EchoTrail structure, differing only in their configuration parameters.

5.1 New attack strategies with PRAC contracts

To achieve an effective, and ultimately optimal, attack, the attacker must use actions that survive mitigations under a fixed activation budget (*tREFW*). We call these actions *survival strategies*. Prior work on proactive mitigations, such as Feinting [19] (Section 2.4), considers only an *active* survival strategy, where decoy aggressors keep the mitigation busy while still hammering the victim. This is because they model proactive mitigations with precise counting, resetting, and queuing; in addition, these mitigations are triggered periodically, as shown in Figure 4a. Since the mitigation is always active, attacks must also actively survive it. However, reactive PRAC mitigations are fundamentally different: they trigger ALERTs only when necessary. This shift enables a new class of *silent* survival strategies. Silent strategies exploit the reactive, and potentially imprecise, nature of PRAC to avoid detection entirely, allowing the attacker to continue hammering the victim for as long as possible without being noticed.

Figure 4b illustrates the gap between how a PRAC contract triggers mitigations (via ALERTs) and what actually causes bit flips in DRAM. Attackers can exploit this gap to increase the hammering effectiveness. Any mismatch between (1) the PRAC contract state, such as the per-row counter $p(\tilde{r})$ (Definition 2), and (2) the actual activation impact on rows, captured by $a(\tilde{r})$ (Definition 1), enables silent survival strategies that allow the attacker to hammer the victim more frequently. Concretely, PRAC issues mitigations based on A_{thresh}, $p(\tilde{r})$, and its queuing mechanism, whereas bit flips depend only on the true hammer count: when $a(\tilde{r})$ reaches $\mathcal{F}(R_{thresh})$ (Definitions 3 and 4). We next analyze how these imprecisions and reactive nature of PRAC contracts enable silent survival strategies.

Silent survival strategy #1: indirect activations. As discussed in Section 4.1.2, a *basic* counting element tracks only explicit ACT(a) commands. However, RFM and REF also activate rows [16]. For example, when a PRAC contract raises an ALERT for row a, the resulting RFM refreshes its victims (rows $a \pm 1$ for $B = 1$). These refreshes activate rows $a \pm 1$, which in turn hammer their victims (rows $a \pm 2$), yet none of these activations are tracked by *basic* counting. This gap enables silent *survival strategy #1: indirect activations*. Here, the attacker can rely only on indirect activations to hammer a target aggressor $\tilde{r}$. It repeatedly activates row $\tilde{r}+1$, forcing PRAC to trigger ALERTs and mitigate $\tilde{r}+1$. Each resulting RFM implicitly activates $\tilde{r}$, which is invisible to *basic*, as shown in Figure 5.

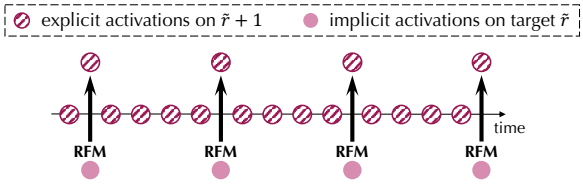


Figure 5: Silent *survival strategy #1: indirect activations* in *basic* counting.

However, this survival strategy is not effective once a PRAC contract employs a *complete* counting mechanism.

Silent survival strategy #2: stealthy priming. A precise resetting mechanism (Definition 1) resets a row's counter whenever all its victims are refreshed. In contrast, some PRAC contracts [34] reset a row's counter only when the row is selected for mitigation, i.e., its victims are refreshed only due to mitigative actions. We denote this as *onMitig* resetting. This mismatch allows the attacker prime all per-row counters to $A_{thresh}-1$ (Definition 8) with essentially unbounded time, and then spend the entire activation budget within a single *tREFW* on hammering the target victim. This enables silent *survival strategy #2: stealthy priming*. It is particularly powerful when combined with active survival strategies, since a decoy aggressor is useful only if it is primed, i.e., one activation away from being selected for mitigation. We later show that, even with more precise resetting elements, this strategy can still be useful for stealthy priming of a limited number of decoy aggressors, depending on A_{thresh}.

Definition 8: Primed aggressor row

Row $\tilde{r}$ is *primed* if any additional increase of its PRAC activation count makes the row eligible for mitigation, i.e.:

$$p(\tilde{r}) = A_{thresh} - 1$$

Figure 6 illustrates the stealthy priming strategy.

Crucially, prior work ignores this strategy and assumes a suboptimal attack in which aggressor priming must occur within the

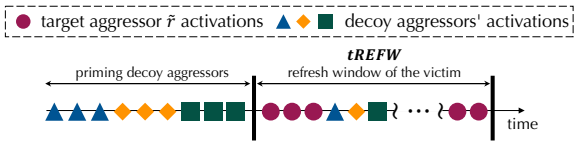


Figure 6: Silent *survival strategy #2: stealthy priming*.

victim's refresh window (the fixed budget within a *tREFW*). This assumption prevents reaching the true worst-case attack bounds.

Silent survival strategy #3: direct evasion. Another avenue for enabling a silent survival strategy is through mismatches that allow even direct activations on the target aggressor $\tilde{r}$ to go untracked. For example, an ideal queuing element (*idealQ*) maintains an unbounded queue in which rows are sorted by their activation counts, with the most critical row at the top. In practice, however, queues are bounded and often simplified for efficiency, creating discrepancies between the PRAC contract's view and the actual activation behavior. This gap enables silent *survival strategy #3: direct evasion*. For instance, prior work [1] proposes a FIFO queue that triggers an ALERT only when the queue becomes full. This design provides no security: an attacker can repeatedly activate the target aggressor, using the pattern $\texttt{ACT}(\tilde{r}) \cdot \texttt{ACT}(\tilde{r}) \cdots \texttt{ACT}(\tilde{r})$, and spend the entire activation budget on $\tilde{r}$. Since the FIFO queue always contains only this single row, it never becomes full and no ALERT is triggered. Figure 7 illustrates this direct evasion strategy.

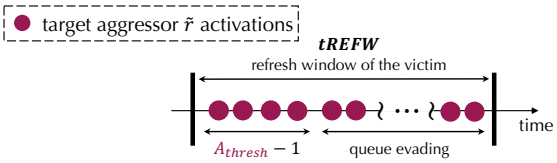


Figure 7: Silent *survival strategy #3: direct evasion* in *fifoQ*.

Active survival strategy #4: decoy activations. In addition to the silent survival strategies introduced by PRAC contracts, the attacker can still employ the active survival strategy discussed in Section 2.4 (see Figure 1), where activating decoy aggressors helps surviving mitigations.

Definition 9: Active aggressor survival from mitigation

Upon an ALERT, V aggressor rows are selected for mitigation: $\tilde{d}_1, ..., \tilde{d}_V$. A *target* aggressor row $\tilde{r}$ (the row within the blast radius of the victim row) survives the mitigation if:

$$\tilde{r} \notin \{\tilde{d}_1, ..., \tilde{d}_V\}$$

We refer to $\tilde{d}_i$ rows as *decoys*.

As discussed earlier, all decoy rows must be primed (Definition 8) before they can participate in the attack.

Problem formalization. Given Definition 5 and Definition 6 and our assumptions on survival strategies, an attacker must find an attack sequence that maximizes $a(\tilde{r})$ and minimizes $\mathcal{L}_{attack} - a(\tilde{r})$. Since all the rows are naturally refreshed within a refresh window (*tREFW*), $\mathcal{L}_{attack}$ for each attack is limited and we need to answer the following questions to achieve an optimal attack:

- What is the optimal distribution of the $\mathcal{L}_{attack}$ activations given the active and silent survival strategies?
- What is the optimal sequence and duration of different survival strategies?

Answering these questions leads us to the EchoTrail attack structure. A key novel aspect of EchoTrail is its integration of silent survival strategies, introduced by PRAC contracts, alongside active strategies. EchoTrail identifies the effective survival strategies against a given PRAC contract and provides the optimal scheduling and configuration of each strategy to construct a globally optimal attack. In the remainder of this section, we define and analyze different PRAC contracts and present the optimal attacks against each of them, and further show that all of these attacks conform to the EchoTrail structure.

Optimality scope. The optimality of EchoTrail and its structure relies on two key sets of assumptions:

- **PRAC contracts**: our attacks only target PRAC-based mitigations, following the contract formalization in Section 4.1. We assume the mitigation only selects queuing, counting, and counter resetting elements, and that inconsistencies in these elements create opportunities for attacks to survive mitigations. As a result, attack patterns identified by EchoTrail may not be optimal against mitigations that deviate from PRAC contract definitions.
- **Rowhammer understanding**: we also assume that an aggressor row is activated only via (1) direct ACT commands or (2) indirect activations through REF and RFM, which affect only rows within its blast radius, and (3) a limited activation budget (i.e., *tREFW*). Our definitions (Section 4.2) follow our current understanding of Rowhammer; thus, EchoTrail's bounds may become sub-optimal if these assumptions change in the future.

5.2 $PRAC_{base}$ security analysis

We define the baseline contract $PRAC_{base}$ as follows:

- *Counting element*: we consider *basic* counting;
- *Resetting element*: $PRAC_{base}$ resets the per row counters only if the corresponding row is subject to an RFM mitigation (i.e., *onMitig*);
- *Queuing element*: we assume an ideal queuing where we have an unlimited *AlertQ* and *AlertQ.head* always returns the V rows that have the highest counter values. *AlertTrigger* asserts an ALERT whenever one of the per row counters becomes greater than or equal to the A_{thresh} (denoted as *ge*).

Hence, the $PRAC_{base}$ contract is defined as:

$$base = \langle A_{thresh}, ge, idealQ|basic|onMitig|\mathcal{P}_{ABO}\rangle$$

where $\mathcal{P}_{ABO}$ is the JEDEC's standard ABO protocol.

Without any PRAC mitigation, the trivial optimal attack is spending all the possible activation budget within a *tREFW* to explicitly activate a target aggressor $\tilde{r}$. However, $PRAC_{base}$ easily prevents this attack by incrementing $p(\tilde{r})$ upon each ACT $(\tilde{r})$ and triggering an ALERT and mitigating $\tilde{r}$ once $p(\tilde{r})$ reaches A_{thresh}. Hence, an attack against $PRAC_{base}$ can explicitly activate $\tilde{r}$ at most $A_{thresh} - 1$ times without being mitigated, after which it must use some *survival strategies* to evade mitigations. The attacker can exploit all active and silent survival strategies discussed in Section 5.1 to

survive mitigation, except for *survival strategy #3: direct evasion*, since $PRAC_{base}$ employs an ideal queuing mechanism. The following strategies remain effective: (1) Silent *indirect activations* because of *basic* counting in $PRAC_{base}$; (2) Silent *stealthy priming* because of *onMitig* resetting in $PRAC_{base}$; (3) Active *decoy activations* because of being effective against all contracts, as the attacker can trigger back-to-back ALERTs.

Informal overview of EchoTrail. The goal of our optimal attack, EchoTrail, is to maximize the number of activations on the target aggressor $\tilde{r}$. In the following lemmas and theorem, we prove the optimality of the EchoTrail attack structure, summarized in Figure 8, based on the survival strategies from Section 5.1. To provide an informal roadmap, Lemma 1 proves that the attack begins by priming all decoy aggressors without budget constraints (the *priming* phase). This is enabled by the stealthy priming strategy under the *onMitig* mechanism of $PRAC_{base}$: without an ALERT, no counters reset and all rows remain primed. Lemma 2 then requires the attacker to activate the target aggressor $\tilde{r}$ exactly $A_{thresh}-1$ times (the *direct* phase). Afterward, the attack proceeds with the post-*direct* survival strategies. Lemma 3 enforces that *survival strategy #4: decoy activations* must be applied symmetrically across all decoys and the target aggressor $\tilde{r}$, without interruption; this defines the *decoying* phase. Lemma 4 further shows that this phase must occur at the end of the attack. The interval between the *direct* and the *decoying* is therefore devoted to *survival strategy #1: indirect activations*, forming the *indirect* phase. We collectively refer to the *indirect* and *decoying* as the post-direct *survival* phases. Finally, Theorem 1 derives the optimal duration of each phase and thus yields the concrete optimal attack against $PRAC_{base}$.

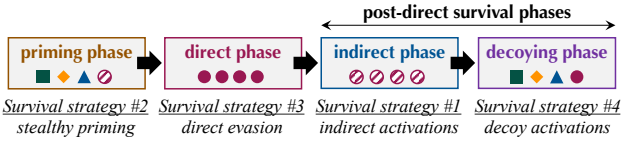


Figure 8: The EchoTrail attack structure.

Formal analysis. A target aggressor $\tilde{r}$ is naturally mitigated every *tREFW*. We assume the attacker can precisely synchronize its attack with the target aggressor's natural mitigation (i.e., the $\mathcal{L}_{attack}$ budget can effectively use all the activations within the *tREFW*). Despite the common assumption in prior PRAC mitigations that the entire attack must occur within the target *tREFW* [3, 25, 34], Lemma 1 shows that, for $PRAC_{base}$, the decoy priming phase does not need to spend any of the $\mathcal{L}_{attack}$ activation budget of the refresh window (*tREFW*) of the target aggressor $\tilde{r}$. This is a direct result of the *onMitig* resetting mechanism of $PRAC_{base}$, enabling the *survival strategy #2: stealthy priming*. We provide the proofs for all the lemmas in Appendix A.

Lemma 1: Priming phase for $PRAC_{base}$

Priming (see Definition 8) of all rows except the target aggressor row $\tilde{r}$ is optimally performed outside a *tREFW*.

▶ **Intuition.** Given the *onMitig* resetting, $p(a)$ for row a does not reset across *tREFW*. (1) The upcoming *decoying* phase of the attack requires all decoy aggressors to be primed in order to help the target aggressor $\tilde{r}$ survive mitigations, and (2) for decoy rows outside the blast radius of the target aggressor $\tilde{r}$, there is no benefit to activating and priming them within the refresh window of the victim, since their activations have no effect on the victim and waste the $\mathcal{L}_{attack}$ budget.

Once the target aggressor is naturally mitigated (i.e., we enter a new refresh window), the most optimal approach for the attacker is to explicitly activate the target aggressor $\tilde{r}$ until its PRAC counter $p(\tilde{r})$ reaches $A_{thresh}-1$. This *direct* phase of the attack does not trigger any ALERTs yet.

Lemma 2: Direct activations

Direct activations on a target aggressor row $\tilde{r}$ are optimal to increase $a(\tilde{r})$, but bounded by $p(\tilde{r})$ reaching $A_{thresh}-1$.

▶ **Intuition.** Direct activations on the target aggressor $\tilde{r}$ are the most useful activations to spend from the $\mathcal{L}_{attack}$ budget as each activation increases $a(\tilde{r})$, however, any additional activation of row $\tilde{r}$ beyond $A_{thresh}-1$ activations, causes $p(\tilde{r})$ reaching A_{thresh} and raise an ALERT and select $\tilde{r}$ for mitigation.

Any additional activations on the target aggressor $\tilde{r}$ following the *direct* phase trigger an ALERT; from this point onward, the attacker *must* use a survival strategy to evade mitigation selections. The subsequent lemmas establish that the *decoying* phase of the attack must (1) distribute activations symmetrically across all decoys and the target aggressor $\tilde{r}$, (2) proceed continuously with back-to-back ALERTs, and (3) serve as the final phase, concluding the attack.

Lemma 3: Symmetry and contiguity of the decoying phase

Once a *decoying* attack phase has started with a set of decoy aggressors $\{d_1, d_2, ..., d_{R_0-1}\}$ and a target aggressor $\tilde{r}$, it must always activate the participating (unmitigated) rows with the lowest activation count in every step without interruption in order to be optimal.

▶ **Intuition.** To maximize $a(\tilde{r})$ and minimize $\mathcal{L}_{attack}-a(\tilde{r})$, all rows in $\{d_1, d_2, \ldots, d_{R_0-1}, \tilde{r}\}$ must have identical $p(.)$ counts. Activating any row outside this set, or activating the decoy rows more frequently than the target aggressor $\tilde{r}$, wastes the $\mathcal{L}_{attack}$ budget without contributing to the attack. Additionally, activating the target aggressor $\tilde{r}$ more often than the decoys causes it to be selected by the mitigation, preventing it to survive.

Lemma 4: *decoying* attack phase end

In the final round of the *decoying* phase, the target aggressor row $\tilde{r}$ is selected for mitigation by the last ALERT, since all decoys have already been mitigated. Therefore, the final action within a *tREFW* is an activation (and mitigation) of the target aggressor $\tilde{r}$, which concludes the attack.

▶ **Intuition.** Once the target aggressor $\tilde{r}$ is selected for mitigation at the end of the *decoying* phase, both $a(\tilde{r})$ and $p(\tilde{r})$ are reset to zero. Therefore, if this does not occur at the very end of the attack, i.e., if the last activation of the $\mathcal{L}_{attack}$ budget is not used to complete the *decoying* phase, the remaining activations are wasted and cannot contribute to maximizing $a(\tilde{r})$.

With the *priming* and the *direct* phases at the beginning, and the *decoying* phase at the end, the attacker can only use the window between the *direct* and the *decoying* phase to perform the *indirect* phase (Figure 8). Lemma 5 establishes that both the *direct* and post-*direct survival* phases (i.e., *direct* +*indirect* +*decoying* phases) must take place within the *tREFW* before the target aggressor would otherwise be naturally mitigated.

Lemma 5: Optimal duration for *direct* + post-*direct survival*

The sequence of *direct*, *indirect*, and *decoying* phases must occur in one *tREFW* to maximize $a(\tilde{r})$ and minimize $\mathcal{L}_{attack} - a(\tilde{r})$ activations.

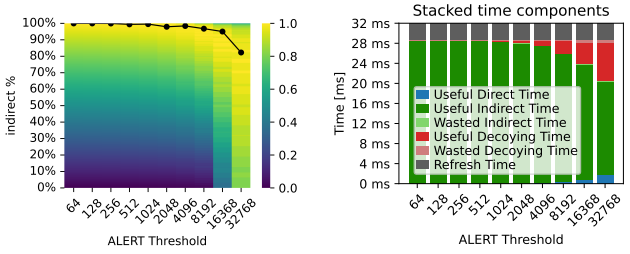
▶ **Intuition.** Since $a(\tilde{r})$ resets at the end of the refresh window *tREFW* (Definition 1), activations beyond *tREFW* no longer increase, or help maximize, $a(\tilde{r})$. Conversely, using fewer than the maximum possible activations within *tREFW* results in a sub-optimal attack, as the unused activations could otherwise help survival strategies and further increase $a(\tilde{r})$.

While the prior lemmas establish the structure of the EchoTrail attack, Theorem 1 provides the optimal configuration of this structure against $PRAC_{base}$. The functions $F_{dir}(A_{thresh})$, $F_{indir}(A_{thresh}, t)$, and $F_{decoy}(A_{thresh}, t)$ denote the effective number of activations to the target aggressor during the *direct*, *indirect*, and *decoying* phases, respectively, for a given ALERT threshold A_{thresh} and duration t. Their sum yields the maximum total number of activations on the target aggressor, $A_{max}(PRAC_{base})$. The core of the theorem is the function $F_{surv}(.)$, defined as $F_{surv}(A_{thresh}, t_s) = F_{indir}(A_{thresh}, t) + F_{decoy}(A_{thresh}, t)$, which optimally allocates the post-*direct* survival time t_s between the *indirect* and *decoying* phases. It therefore determines both the maximum achievable activations on $\tilde{r}$ in these phases and their optimal time split.

Prior work [3, 19, 34] defines the *decoying* behavior recursively. In contrast, $F_{decoy}(A_{thresh}, t)$ is a closed-form expression derived from the recursive Feinting formulation, directly giving the number of activations $a(\tilde{r})$ achievable within a time budget t. Its structure captures two effects: (1) $\frac{\ln(1/R_0)}{\ln(1-c)}$ determines the number of rounds until only one aggressor remains, and (2) $1 - c$ is the decay factor describing how mitigated rows are removed proportionally to the remaining aggressors. Even in the final round, when only one row remains, the attacker can still issue $(L - 1) + \lfloor tABO\_ACT/tRC \rfloor$ activations before mitigation, incorporated into the closed form. Table 3 provides a summary of the notations used in this section.

Theorem 1: Optimal attack against $PRAC_{base}$

An attack is optimal if every activation is chosen to maximize the contribution towards increasing $a(\tilde{r})$.



(a) (b)

Figure 9: Optimizing $PRAC_{base}$ shows nonlinear trends. The heatmap (a) shows the distribution of possible solutions, with one best solution marked with a black line. (b) shows the components of a *tREFW* for this optimal solution. Note that the optimal solution always includes a combination of a *decoying* and *indirect* phase.

$$
\begin{aligned}
t_s &= tREFW - (A_{thresh} - 1) \cdot tRC - tRFC \cdot 8K \\
c &= \frac{V}{\lfloor \frac{tABO\_ACT}{tRC} \rfloor + L} \\
tA2A &= L \cdot tRC + tABO\_ACT + tRFM \cdot V \\
R_0 &= \frac{t_s - t - tABO\_ACT}{tA2A} \cdot V \\
F_{dir}(A_{thresh}) &= A_{thresh} - 1 \\
F_{indir}(A_{thresh}, t) &= 1 + \frac{t - (tRC + tABO\_ACT + tRFM)}{tRC \cdot A_{thresh} + tABO\_ACT + tRFM} \\
F_{decoy}(A_{thresh}, t) &= \frac{\ln \frac{1}{R_0}}{\ln(1-c)} + (L-1) + \lfloor \frac{tABO\_ACT}{tRC} \rfloor \\
F_{surv}(A_{thresh}, t_s) &= \max_{t=0}^{t_s} \Big(F_{indir}(A_{thresh}, t) \\
&\qquad + F_{decoy}(A_{thresh}, t_s - t) \Big) \\
A_{max}(PRAC_{base}) &= F_{dir}(A_{thresh}) + F_{surv}(A_{thresh}, t_s)
\end{aligned}
$$

Proof. The proof follows from Lemmas 1, 2, 3, 4, 5. Priming decoys to $p(\tilde{d}_i) = A_{thresh} - 1$ persists and must be performed in a prior *tREFW* (Lemma 1). Because *direct* activations are optimal and increment $a(\tilde{r})$ with each activation (Lemma 2), they must always be part of the solution, but are limited to $A_{thresh} - 1$ activations. Lemmas 3 and 4 show that the *decoying* phase must be uninterrupted and end at the end of the *tREFW*. Recall that an *idealQ* queue results in repeated mitigations to rows with $a(\tilde{r}) \geq A_{thresh}$, regardless of activations. Because the number of required activations grows geometrically with the timeframe allocated to F_{decoy} (as longer preparations cause more decoy rows to be mitigated), there is a point at which F_{indir} yields more activations per unit time, despite the high cost of priming to A_{thresh} during the attack phase. Because there is no other activation strategy than *direct*, *indirect*, and *decoying*, and *direct* is bound by $A_{thresh} - 1$, the optimization problem reduces to the split between F_{indir} and F_{decoy}. □

In Figure 9a, we show the effectiveness of different splits of the *indirect* and *decoying* phases for different A_{thresh}: `indirect%` indicates the percentage of post-*direct survival* phase that has been spent on the *indirect* phase. As the activation costs using implicit activations during the *indirect* phase increase with A_{thresh}, the *decoying* phase becomes more useful for larger A_{thresh}. Figure 9b shows the time

allocations for the optimal solution. Because the time might not evenly distribute, a small amount is wasted on working towards an activation that does not fit. Interestingly, our results indicate that the optimal attack against the $PRAC_{base}$ contract is always a combination of the two survival strategies, *survival strategy #1: indirect activations* and *survival strategy #4: decoy activations*, i.e., in all A_{thresh} levels: 0% < `indirect%` < 100%.

EchoTrail notation. We use the notation $\text{EchoTrail}_{\mathcal{N}}^{tAttack}$ to express the attack sequence: $tAttack$ refers to the entire duration of the attack (*priming* + *direct* + *indirect* + *decoying*), and $\mathcal{N}$ refers to the set of required decoys for the attack (i.e., the decoys needed for the *decoying* phase). These parameters enable the attack reconstruction.

5.3 Security impact of the counting

The next contract to analyze is $PRAC_{complete}$ that is a variant of $PRAC_{base}$, differing only in the counting element: $C = complete$. In other words, a per row counter $p(x)$ for row x is incremented on all commands that either explicitly or implicitly activate the row, such as `ACT`, `RFM`, and `REF`. This means that the *survival strategy #1: indirect activations* will not be effective against $PRAC_{complete}$.

Theorem 2: Optimal attack for $PRAC_{complete}$

$$\mathcal{S}^{\star}(PRAC_{complete}) = \text{EchoTrail}_{\mathcal{N}}^{(|\mathcal{N}|+1)\cdot(A_{thresh}-1)\cdot tRC+t_s}$$

Where $t_s = tREFW - 8K \cdot tRFC - (A_{thresh}-1)\cdot tRC$
and $|\mathcal{N}| = \dfrac{t_s - tABO\_ACT}{tA2A} \cdot V$.

► **Intuition.** For $PRAC_{complete}$, the implicit component $F_{indir}(.)$ degrades to 0 because it can no longer hide activations unless it is part of the *decoying* phase. The optimal attack is therefore limited to the *direct* and *decoying* approaches, while *survival strategy #2: stealthy priming* remains effective and the *priming* phase is placed in the prior $tREFW$.

We provide the proofs for theorems in Appendix A.

5.4 Security impact of the counter resetting

We analyze the security impact of two new contracts that deploy different resetting mechanisms:

- **Fine-grained resetting** (*fine*): besides the on-mitigation resets, counters reset in a spatially continuous fashion every $tREFI$ on `REF`s, for a row-group that gets refreshed [25];
- **Coarse-grained resetting** (*coarse*): besides the on-mitigation resets, *all* counters reset every $tREFW$ (similar to many prior Rowhammer mitigations [19, 22]);

$PRAC_{fine}$ and $PRAC_{coarse}$ are two contracts that are variations of the $PRAC_{complete}$ contract where they only differ in their resetting element. They use *fine* and *coarse*, respectively, instead of *onMitig* resetting in $PRAC_{complete}$.

Despite prior work assuming only one $tREFW$ window for the entire attack with the *fine* resetting [25], the attacker can potentially have some extra time for priming, still allowing the *survival strategy #2: stealthy priming*. As shown in Figure 10, the first decoy that gets selected for mitigation during the *decoying* phase has a different $tREFW$ window for counter resetting ($tREFW_1$ in Figure 10), and can be primed before the target's counter reset ($tREFW_2$). Interestingly, the rate of back-to-back ALERTs during the *decoying* phase is higher than the counter resets: 6 ALERTs per $tREFI$ vs. 4 counter resets per $tREFI$ (assuming $B = 1$ and row-group size of 8). This further helps *decoying* phase to consume the decoys faster than being reset.

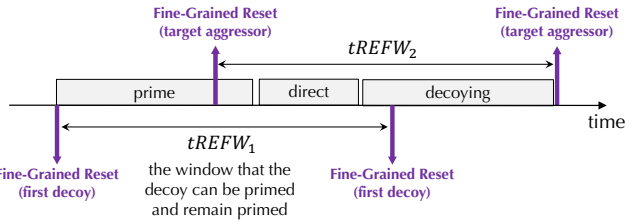


Figure 10: With a *fine* resetting, the attacker is not strictly bound to one $tREFW$ (i.e., $tREFW_1$) for the entire attack, still allowing the *survival strategy #2: stealthy priming*.

Theorem 3: Optimal attack for $PRAC_{fine}$

$$\mathcal{S}^{\star}(PRAC_{fine}) = \text{EchoTrail}_{\mathcal{N}}^{t_{prime}+t_{direct}+t_{decoy}}$$

Where $t_{direct} = (A_{thresh}-1)\cdot tRC$,
$|\mathcal{N}| = \min\left(\left\lfloor \frac{(t_s - tREFI - tABO\_ACT)\cdot V}{tA2A} \right\rfloor, \left\lfloor \frac{\max(t_{prime})}{(A_{thresh}-1)\cdot tRC} \right\rfloor\right)$,
$t_{prime} = |\mathcal{N}| \cdot (A_{thresh}-1)\cdot tRC$,
$t_{decoy} = \frac{|\mathcal{N}|\cdot tA2A}{V} + tABO\_ACT$

► **Intuition.** In *fine* resetting with *complete* counting, we have two phases with constraints. The *priming* cannot be longer than $tREFW - tRC\cdot 8192 - tREFI - t_{direct}$ (otherwise, the counters will reset before we make use of the primed rows), and the *decoying* cannot be longer than $tREFW - t_{direct} - 8K\cdot tRFC - tREFI$. Because the *decoying* phase requires primed decoys, the priming phase is the bounding factor for typical A_{thresh}.

Figure 11 shows that the *decoying* phase of the optimal EchoTrail attack against $PRAC_{fine}$ achieves higher activations (the y-axis) for small thresholds compared to the assumption in prior work, assuming only one $tREFW$ for the entire attack.

In the $PRAC_{coarse}$ contract, all counters reset at the same time every $tREFW$, allowing an EchoTrail attack to be executed twice: once before and once after the counters' reset. As shown in Figure 12, these *coarse* resets can be unaligned with the natural refreshes and mitigations of the target aggressor. Prior work typically assumes the counters reset is exactly aligned at the middle of the victim's refresh window ($tREFW_1$ in Figure 12) [22]. However, our optimal EchoTrail attack reveals that resetting earlier within $tREFW_1$ is more effective, as it leaves more time for the second attack, which has more limited time for (re-)priming of decoys.

Theorem 4: Optimal attack for $PRAC_{coarse}$

$$\mathcal{S}^{\star}(PRAC_{coarse}) = \text{EchoTrail}_{\mathcal{N}_1}^{tREFW-r_1} \cdot \text{EchoTrail}_{\mathcal{N}_2}^{tREFW-r_2}$$

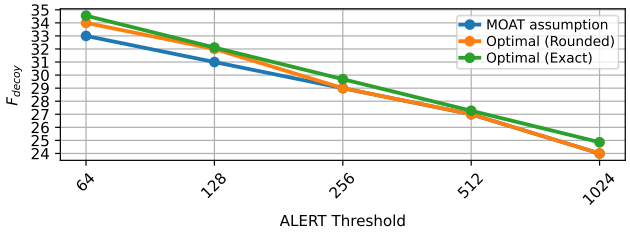


Figure 11: EchoTrail against $PRAC_{fine}$ (*fine* resetting). By starting to prime in a prior *tREFW*, we can allocate more time to the *decoying* phase, breaking prior assumptions [25].

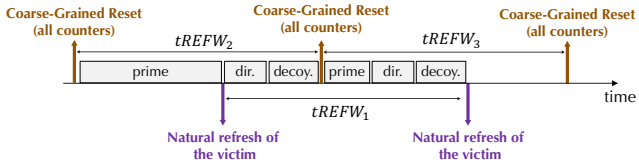


Figure 12: With a *coarse* resetting, the attacker needs to find the optimal alignment with the all-counters resets.

where $r_1 = t_{direct1} + t_{decoy1}$, $r_2 = t_{direct2} + t_{decoy2}$,
$r_1 + r_2 = tREFW - 8K \cdot tRFC$.

▶ **Intuition.** We first note that *coarse* resetting is not bound to the *tREFW* in which the EchoTrail attack takes place. Therefore, we can split the phases into two concatenated EchoTrail sequences to maximize the outputs. While the first attack can make use of priming outside of *tREFW*, the second attack needs to prime immediately before doing the second *direct* activations. The shift of the *coarse* reset window is an optimization problem (i.e., finding r_1 and r_2).

5.5 Security impact of the queuing

We analyze two different queuing mechanisms, other than *idealQ*:

- **Priority queue** includes the activation count for each entry and it always keeps it sorted. Upon each activation, the row is inserted if its per row counter is higher than any of the existing rows. The mitigation always selects the rows with the maximum count in the queue [25, 34]:

$$Q_1 = \langle A_{thresh}, ge, priorityQ \rangle$$

- **Full-trigger FIFO queue** uses a FIFO queue where aggressors are inserted whenever their per row counter reaches the A_{thresh}. An ALERT is triggered only if the queue is full (*fullTrigger*) [1]:

$$Q_2 = \langle A_{thresh}, fullTrigger, fifoQ \rangle$$

$PRAC_{priorityQ}$ and $PRAC_{fullFifo}$ are two variants of $PRAC_{complete}$ that differ only in their queuing: they use Q_1 and Q_2, respectively.

Theorem 5: Optimal attack for $PRAC_{priorityQ}$

$PRAC_{priorityQ}$ has the same optimal attack guarantees of the $PRAC_{complete}$:

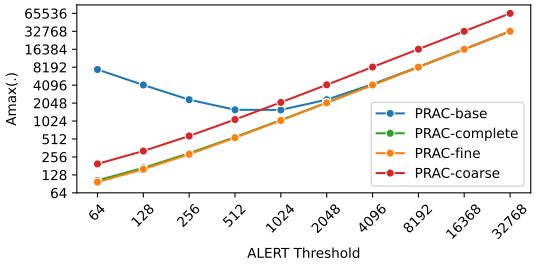


Figure 13: $A_{max}(.)$ comparison of four discussed contracts: $PRAC_{base}$, $PRAC_{complete}$, $PRAC_{fine}$, and $PRAC_{coarse}$. $PRAC_{priorityQ}$ has similar bounds as the $PRAC_{complete}$ contract.

$$\mathcal{S}^\star(PRAC_{priorityQ}) = \mathcal{S}^\star(PRAC_{complete}) = \text{EchoTrail}_{\mathcal{N}}^{(|\mathcal{N}|+1)\cdot(A_{thresh}-1)\cdot tRC+t_s},$$

where $t_s = tREFW - 8K \cdot tRFC - (A_{thresh} - 1) \cdot tRC$,
and $|\mathcal{N}| = \dfrac{t_s - tABO\_ACT}{tA2A} \cdot V$.

▶ **Intuition.** The *priorityQ* queue's behavior is not identical to the *idealQ* queue, as assumed by prior work [34]. In particular, the *priorityQ* queue maintains the rows that have been activated in a recent sequence, while the *idealQ* queue maintains a sorted list of activated rows based on the global activation history. The following counterexample illustrates this difference. Assume a state where $p(\tilde{r}_1) = A_{thresh} + 2$ and $p(\tilde{r}_2) = A_{thresh} + 1$. Consider a *priorityQ* queue of size one that currently contains only $\tilde{r}_1$. When an ALERT is issued, $\tilde{r}_1$ is selected for mitigation and removed from the queue. Although $\tilde{r}_2$ already exceeds A_{thresh}, it is inserted into the *priorityQ* queue only if it is activated again; in particular, it is not selected for mitigation if the next ALERT is issued without a new activation to $\tilde{r}_2$. In contrast, with an *idealQ* queue, $\tilde{r}_2$ moves to the top of the queue after $\tilde{r}_1$ is mitigated and is selected for mitigation by the next ALERT, regardless of whether $\tilde{r}_2$ was activated recently. That said, because an optimal attack must activate the decoys again for survival (Lemma 3), *priorityQ* becomes equivalent to *idealQ*, resulting in the same total activation count.

Theorem 6: Optimal attack for $PRAC_{fullFifo}$

$$\mathcal{S}^\star(PRAC_{fullFifo}) = \text{EchoTrail}_0^{tREFW-8K\cdot tRFC}$$

▶ **Intuition.** The *fullFifo* queue is unsafe for $|AlertQ| \geq 1$ because an attacker would only activate a single row, where the *AlertQ* never gets full and *fullTrigger* never signals the row for mitigation. This single-row activation attack is optimal because the cost per activation is minimal (see Lemma 2) and the mitigation never triggers.

Summary of $A_{max}(.)$ contracts. Figure 13 shows the $A_{max}(.)$ of the optimal EchoTrail attacks across various PRAC contracts (excluding the $PRAC_{fifoQ}$ contract as it has similar bounds of disabling any PRAC mitigation; cf. Theorem 6). For low A_{thresh}, $A_{max}(PRAC_{base})$ experiences the highest activation counts. However, as the threshold increases, its higher vulnerability compared to other contracts

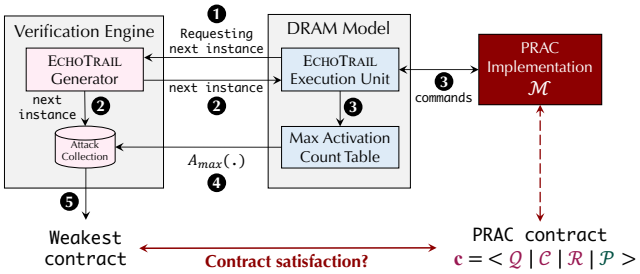


Figure 14: VeriPRAC design to verify contract satisfaction.

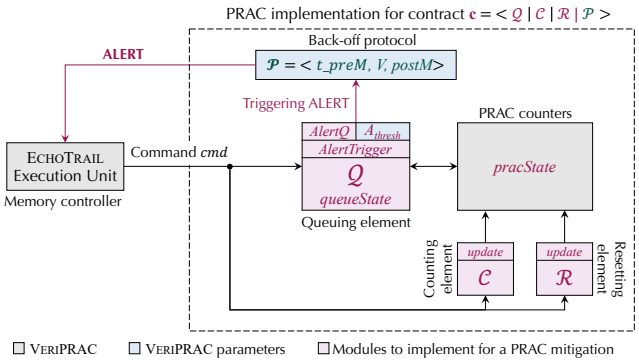


Figure 15: Modular implementation of a PRAC mitigation in VeriPRAC.

decreases due to the growing cost of the *indirect* phase, eventually converging with the other contracts (e.g., $PRAC_{complete}$ and $PRAC_{fine}$). Additionally, $PRAC_{coarse}$ consistently results in higher activation counts than other contracts that use *complete* resetting or *onMitig* resetting, primarily because it allows two separate EchoTrail attacks.

6 VeriPRAC

Building on PRAC contract analysis, we present VeriPRAC, a verification tool that checks whether a given PRAC implementation satisfies its intended contract (Definition 7).

6.1 VeriPRAC design

Figure 14 provides an overview of VeriPRAC, which consists of three main components: (1) a DRAM model, (2) a verification engine, and (3) a PRAC implementation that is accompanied by its contract.

(1) DRAM model. This component emulates all the functionalities of a DRAM device based on DDR5 PRAC timings. The EchoTrail execution unit of the model requests and then receives an EchoTrail attack instance as an input (steps ❶ and ❷) and executes the given EchoTrail attack (step ❸). In other words, this execution engine is similar to a memory controller that performs the attack on the plugged-in PRAC implementation. Additionally, we store a Max Activation Count Table (MACT) inside the DRAM model, which accurately counts and maintains the maximum number of aggressor activations for all rows during the execution of an EchoTrail instance (including both implicit and explicit activations). The MACT table reports $A_{max}(.)$ of the tested contract at the end of the attack (step ❹).

(2) Verification engine. This component systematically iterates over a list of EchoTrail instances that exercise all possible survival strategies and sends these instances to the DRAM execution unit upon each request (steps ❶ and ❷). In addition, the verification engine tracks the reported maximum aggressor activation count from the DRAM model (step ❹), which are eventually used to determine the weakest contract that the plugged PRAC implementation satisfies (step ❺). Note that VeriPRAC verifies whether an implementation satisfies its given contract (Definition 7), which is provided as input. Based on our systematic analysis in Section 5 for {*idealQ*, *priorityQ*, *fifoQ*} queuing elements, {*basic*, *complete*} counting elements, and {*onMitig*, *fine*, *coarse*} resetting elements, along with all possible survival strategies that introduce discrepancies, VeriPRAC guarantees detection of contract violations by testing all EchoTrail attack configurations incorporating such strategies. For example, it tests an EchoTrail attack using the *survival strategy #1: indirect activations* strategy to determine whether an implementation of *complete* counting resists the attack or allows it to survive and violate the intended contract. We will discuss the considerations to extend VeriPRAC to analyze new contracts and PRAC mitigations in Section 6.2.

(3) PRAC implementation. This component corresponds to the concrete implementation $\mathcal{M}$ of a PRAC mitigation, with the intended contract **c**. VeriPRAC provides a modular framework for implementing PRAC mitigations, as illustrated in Figure 15. To integrate a new PRAC-based mitigation, users instantiate a `Mitigation` class and implement abstract functions such as `queuingMechanism`, `countingMechanism`, and `resettingMechanism`, where these functions get an aggressor as input (i.e., direct and indirect activation commands); the mitigation's implementation for the abstract functions decides how to queue, count, or reset its per row counters (i.e., *pracState*). For example, implementing the core mechanisms of the state of the art, MOAT [25], required only 129 lines of C++ code. The rest of the testing tool then operates automatically and evaluates the mitigation implementation. The implementation receives commands from the EchoTrail execution unit (step ❸), which are used to perform mitigations; for example, it can trigger ALERTs and specify which rows to mitigate upon `RFM` commands.

Contract satisfaction. In the final step ❺, where all EchoTrail instances are tested, the weakest contract identified by the verification unit is compared to the intended contract of the mitigation. If the two contracts match, or if the identified contract is stronger than the intended one, the PRAC implementation is considered to satisfy its contract (Definition 7).

6.2 Integration of existing PRAC mitigations

We integrate two state-of-the-art PRAC mitigations in VeriPRAC: MOAT [25] and QPRAC [34].

Queuing. Both MOAT and QPRAC use a priority queue, modeled as $Q = \langle A_{thresh}, ge, priorityQ \rangle$. MOAT employs a single-entry queue and supports only ABO level 1, whereas QPRAC uses a multi-entry queue and supports all ABO levels.

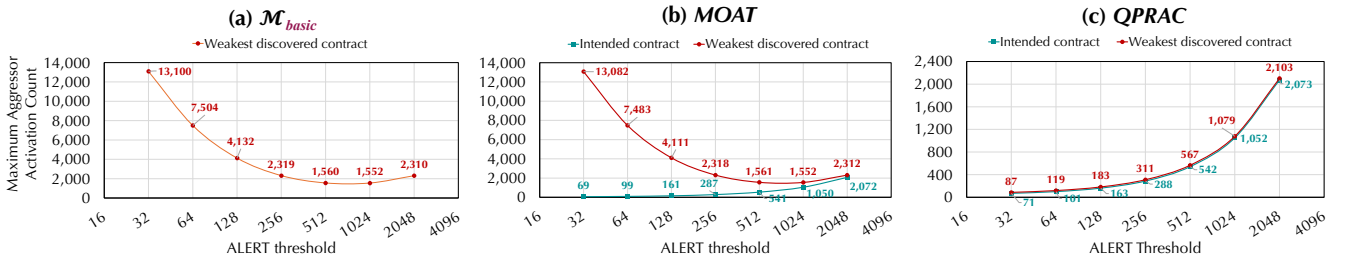


Figure 16: VeriPRAC results for $\mathcal{M}_{basic}$, MOAT and QPRAC with different A_{thresh} values, assuming ABO level of 1.

Resetting. The two designs differ in their resetting mechanisms: MOAT uses *fine*, while QPRAC uses *onMitig*.

Counting. QPRAC specifies a *complete* mechanism. While MOAT does not explicitly specify its counting policy, we use its available artifacts as a reference [24], which show that per row counters are incremented upon RFM commands. We therefore implement MOAT with *complete* counting as well.

We additionally evaluate a baseline PRAC implementation, denoted as $\mathcal{M}_{basic}$, which intends to implement the following contract:

$$\langle A_{thresh}, ge, idealQ|basic|fine|\mathcal{P}_{ABO}\rangle$$

As Figure 13 shows, a contract with *basic* counting provides the weakest security bounds, i.e., the highest $A_{max}(.)$, for lower thresholds. We use this mitigation to assess how MOAT and QPRAC compare to this design point and, more importantly, how subtle implementation flaws can cause an implementation assumed to be secure to degrade to this basic baseline.

6.3 Evaluation

Figure 16 presents the VeriPRAC results for $\mathcal{M}_{basic}$, *MOAT*, and *QPRAC* implementations. We report the EchoTrail attack bounds of both (1) the intended contract of each implementation, and (2) the weakest contract identified by VeriPRAC. The y-axis is the $A_{max}(.)$ for different A_{thresh} values, assuming an ABO level of 1. Figure 16a shows that $\mathcal{M}_{basic}$ behaves as expected, following the same trend as the $PRAC_{base}$ contract in Figure 13.

MOAT. Surprisingly, *MOAT* violates its intended contract: the weakest contract VeriPRAC identifies is almost identical to that of $\mathcal{M}_{basic}$, which uses *basic* counting, even though *MOAT* implements *complete* counting. Consequently, as shown in Figure 16b, for $A_{thresh} = 64$, the original paper reports $A_{max}(MOAT) = 99$, while VeriPRAC identifies $A_{max}(MOAT) = 7{,}483$ and this optimal bound increases for lower ALERT thresholds.

A closer examination reveals that the main root cause of this violation is that MOAT invalidates its queuing registers after mitigative RFMs during an ALERT. These invalidations cause the *AlertQ* to lose track of the implicit activations induced by RFMs, effectively downgrading the design to a weaker contract with *basic* counting. In addition, the original MOAT artifacts [24] reveal that per-row counter increments caused by RFMs do not update the queuing registers: a row whose per-row counter exceeds A_{thresh}, and is higher than the row currently residing in the queue, does not replace the tracked row when it is activated through RFM refreshes. Based on VeriPRAC's analysis, it is straightforward to fix MOAT by avoiding queue invalidation upon mitigation and updating the queuing registers upon all direct and indirect activations.

QPRAC. The VeriPRAC results for *QPRAC* (Figure 16c) have similar trend as reported in the original paper, but still violating its contract. For example, the original paper reports $A_{max}(QPRAC) = 542$ for $A_{thresh} = 512$, whereas VeriPRAC identifies $A_{max}(QPRAC) = 567$. The higher bounds in VeriPRAC 's identified contract arise from the fact that *QPRAC* uses an *onMitig* counter resetting, which allows the attacker to prime decoy rows outside the *tREFW* and then use the entire *tREFW* for the *direct* and *decoying* phases of EchoTrail. However, the original paper assumes that decoy priming must also occur within the *tREFW*, leading to sub-optimal bounds.

7 Discussion and Related Work

Extension of VeriPRAC. In case future PRAC implementations intend to define new contracts (e.g., using other queuing mechanisms than *idealQ*, *priorityQ*, or *fifoQ* that we analyzed in Section 5), they must first reason about the applicable survival strategies for their new queuing, counting, and resetting elements and identify the optimal configuration of EchoTrail against them, similar to our analysis in Theorems 1–6. The corresponding EchoTrail instances can then be added to VeriPRAC as JSON inputs. Additionally, note that VeriPRAC only supports PRAC-based mitigations that can be expressed under our contract formalization in Section 4.1, i.e., consisting of queuing, counting, and resetting elements; arbitrary mitigations that deviate from this structure are not supported by EchoTrail and VeriPRAC, as their vulnerability space (i.e., possible set of silent and active survival strategies) may differ depending on added or omitted elements.

Multi-banks analysis. ALERT is channel-wide: when bank B_0 triggers an ALERT, all banks (B_0 to B_{31}) receive RFMs. Non-triggering banks can use this opportunity to mitigate the row at the head of their own *AlertQ*. Prior work treats these opportunistic mitigations purely as a performance optimization [34]. Crucially, we show they are *necessary* for security. Without them, an attacker can trigger ALERTs from non-target banks, preventing the target aggressor from ever being selected for mitigation and creating a powerful survival strategy (see Appendix B for more details).

Chronus. Chronus [3] is a recent in-DRAM reactive mitigation that relies on per row activation counters but introduces a JEDEC-incompatible back-off protocol which keeps the ALERT signal asserted until all rows with counters more than or equal to A_{thresh}

are mitigated. Since VeriPRAC aims to evaluate PRAC-based mitigations that follow JEDEC specifications, analyzing out-of-specification designs such as Chronus falls outside of its scope.

Panopticon. Panopticon [1] is the first work that introduced per row activation counters, and the JEDEC PRAC specifications are inspired by this work. However, Panopticon does not specify a clear back-off protocol for ALERTs. Panopticon uses a *fifoQ* queuing mechanism that we prove its insecurity in Theorem 6. Prior work has also demonstrated the vulnerabilities of the Panopticon's queuing and counting mechanisms [25, 34].

Proactive mitigations in PRAC. Some prior PRAC-based designs also incorporate proactive TRR mitigations, where every f REFs a row with a high PRAC counter is mitigated. For example, MOAT uses a TRR eligibility threshold of $A_{thresh}/2$. These proactive mechanisms are mainly introduced for performance, to reduce the probability that benign applications reach A_{thresh}, while security is primarily attributed to the reactive mitigations. We exclude proactive mitigations in order to analyze the security of PRAC contracts in isolation and to highlight the impact of their fundamental properties. This choice is also motivated by energy efficiency: a key goal of PRAC is to avoid the overhead of proactive mitigations, so that applications that never reach A_{thresh} incur no additional energy, reliability, or performance cost. Finally, prior work already provides a formal analysis of proactive mitigations [19].

8 Conclusion

In this work, we introduced formal contracts for PRAC mitigations and their implementation, providing mathematically-proven security bounds for the first time. We achieved this by formalizing PRAC contracts that capture the key elements of a generic PRAC-based mitigation. To calculate the security bounds for these contracts, we constructed EchoTrail, a theoretical attack structure that we prove to be optimal in the reactive setting defined by PRAC. Using PRAC contracts and their security bounds, we developed VeriPRAC, an automatic contract verification tool that checks whether a given implementation satisfies its intended PRAC contract. Applying VeriPRAC to state-of-the-art PRAC mitigations revealed that they all violate their intended contracts. In one extreme case, the identified attack reaches up to $7.5k$ activations on the aggressor, while the contract's intended bound is only 99.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful feedback, which improved the final version of this paper. We also thank Michele Marazzi for his discussions and feedback on early drafts of the paper. This work has been supported by the Zurich Information Security and Privacy Center (ZISC) and the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

Ethical Considerations

Stakeholders. This paper analyzes the security of PRAC mitigations proposed for future DRAM devices to defend against Rowhammer attacks. Such attacks affect DRAM vendors, system manufacturers, end users, and security researchers who design and evaluate Rowhammer defenses.

Ethical analysis. The security analysis enabled by EchoTrail and VeriPRAC provides a formal tool to evaluate reactive, PRAC-based mitigations before deployment. This benefits all stakeholders: DRAM vendors can identify and eliminate flaws prior to production, system manufacturers and end users gain strong guarantees in the security of deployed DRAM devices, and security researchers obtain a principled tool to reason about and verify their designs.

In principle, an attacker could also use EchoTrail and VeriPRAC to iteratively refine an attack and derive an optimal Rowhammer strategy against a PRAC-enabled device. However, PRAC is not currently deployed in any commodity hardware, and all our analyses are purely theoretical or simulation-based. Therefore, this work poses no risk to existing systems or stakeholders.

Decision. Although VeriPRAC could assist attackers against future PRAC deployments, we believe its benefits clearly outweigh its risks. VeriPRAC enables vendors to rigorously analyze and validate their mitigation implementations before deployment, helping to identify weaknesses early and prevent flawed implementations from reaching production hardware.

Open Science Policy

We have open-sourced our EchoTrail and VeriPRAC tools through Zenodo and our website to support the community and future Rowhammer research by providing accessible and reusable tools for formal, automated security analysis.

Generative AI usage

We used ChatGPT and Codex for lightweight writing and coding assistance. All text and code were originally written by the authors, and changes were manually reviewed, verified, and applied.

References

[1] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. 2021. Panopticon: A Complete In-DRAM Rowhammer Mitigation. In *Workshop on DRAM Security (DRAMSec)*.

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Symposium on Security and Privacy (S&P)*.

[3] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Nisa Bostancı, İsmail Emir Yüksel, Haocong Luo, Oğuz Ergin, and Onur Mutlu. 2025. Chronus: Understanding and Securing the Cutting-Edge Industry Solutions to DRAM Read Disturbance. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[4] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security Symposium*.

[5] Finn De Ridder, Patrick Jattke, and Kaveh Razavi. 2025. Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands. In *USENIX Security Symposium*.

[6] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *Symposium on Security and Privacy (S&P)*.

[7] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[8] Ali Hajiabadi, Michele Marazzi, and Kaveh Razavi. 2025. CHaRM: Checkpointed and Hashed Counters for Flexible and Efficient Rowhammer Mitigation. In *Conference on Computer and Communications Security (CCS)*.

[9] Aamer Jaleel, Gururaj Saileshwar, Stephen W Keckler, and Moinuddin Qureshi. 2024. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In *International Symposium on Computer Architecture (ISCA)*.

[10] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *Symposium on Security and Privacy (S&P)*.

[11] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *USENIX Security Symposium*.
[12] JEDEC. 2024. JESD79-5C: DDR5 SDRAM Specifications. (2024).
[13] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. 2024. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. In *USENIX Security Symposium*.
[14] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2022. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In *International Symposium on High-Performance Computer Architecture (HPCA)*.
[15] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture (ISCA)*.
[16] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. 2022. Half-Double: Hammering from the Next Row Over. In *USENIX Security Symposium*.
[17] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading Bits in Memory Without Accessing Them. In *Symposium on Security and Privacy (S&P)*.
[18] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. 2020. Nethammer: Inducing Rowhammer Faults through Network Requests. In *European Symposium on Security and Privacy Workshops (EuroSPW)*.
[19] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. 2022. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *Symposium on Security and Privacy (S&P)*.
[20] Diego Meyer, Patrick Jattke, Michele Marazzi, Salman Qazi, Daniel Moghimi, and Kaveh Razavi. 2026. Phoenix: Rowhammer Attacks on DDR5 with Self-Correcting Synchronization. In *Symposium on Security and Privacy (S&P)*.
[21] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, A. Giray Yaglikci, Geraldo F. Oliveira, and Onur Mutlu. 2024. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In *USENIX Security Symposium*.
[22] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *International Symposium on Microarchitecture (MICRO)*.
[23] Rui Qiao and Mark Seaborn. 2016. A New Approach for Rowhammer Attacks. In *International symposium on hardware oriented security and trust (HOST)*.
[24] Moinuddin Qureshi and Salman Qazi. 2024. *(Artifacts) MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters.* doi:10.5281/zenodo.14061375
[25] Moinuddin Qureshi and Salman Qazi. 2025. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[26] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. 2024. MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker. In *International Symposium on Microarchitecture (MICRO)*.
[27] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. 2022. Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking. In *International Symposium on Computer Architecture (ISCA)*.
[28] Kaveh Razavi, Ben Gras, Cristiano Giuffrida, Erik Bosman, Bart Preneel, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.
[29] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat* (2015).
[30] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigating Wordline Crosstalk Using Adaptive Trees of Counters. In *International Symposium on Computer Architecture (ISCA)*.
[31] Andrei Tatar, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, Elias Athanasopoulos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*.
[32] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Conference on Computer and Communications Security (CCS)*.
[33] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
[34] Jeonghyun Woo, Prashant J Nair, Aamer Jaleel, Gururaj Saileshwar, et al. 2025. QPRAC: Towards Secure and Practical PRAC-based Rowhammer Mitigation Using Priority Queues. In *Symposium on High Performance Computer Architecture (HPCA)*.

Table 3: Summary of notations and symbols.

Notation	Description
Contract notations and symbols (Section 4.1)	
$PRAC_{\mathbf{c}}$	A PRAC contract specified by $\mathbf{c}$ elements
$\mathcal{Q}$	The queuing element of a PRAC contract
$\mathcal{C}$	The counting element of a PRAC contract
$\mathcal{R}$	The counter resetting element of a PRAC contract
A_{thresh}	ALERT threshold of a PRAC contract
AlertTrigger	The ALERT trigger mechanism of a queuing element
AlertQ	The queue type of a queuing mechanism
basic	A basic counting element only counting ACT commands
complete	A complete activation counting
onMitig	A counter resetting on mitigations
coarse	All counters resetting every *tREFW* + *onMitig*
fine	Counter resetting every *tREFI* + *onMitig*
idealQ	An ideal queuing type for *AlertQ*
priorityQ	A priority queue with limited size for *AlertQ*
fifoQ	A FIFO queue with limited size for *AlertQ*
fullTrigger	An *AlertTrigger* when *AlertQ* is full
ge	An *AlertTrigger* for counters $\geq A_{thresh}$
$\mathcal{P}$	A back-off protocol configuration
t_preM	The time between the ALERT trigger and an RFM reception
preM	The maximum number of possible ACTs during *t_preM*
V	The number of rows mitigated per ALERT
postM	The number of required ACTs after the ALERT's RFMs
Rowhammer and security definitions (Section 4.2)	
$a(\tilde{r})$	The precise, actual per-row activation count
$h(\tilde{v})$	The precise, actual per-row hammer count
$p(\tilde{r})$	The per-row counter tracked by a PRAC contract
R_{thresh}	Rowhammer threshold required for a bit flip
$\mathcal{F}(R_{thresh})$	The critical activation count level for a bit flip
$\mathcal{S}$	A Rowhammer attack sequence
$\mathcal{L}_{attack}$	The activation budget for a Rowhammer attack
$\mathcal{U}$	The universal set of all possible $\mathcal{S}$ attack sequences
$A(\mathcal{S}, \mathcal{M})$	The highest $a(.)$ achieved by $\mathcal{S}$ against mitigation $\mathcal{M}$
$\mathcal{S}^{\star}(\mathcal{M})$	The optimal attack sequence $\mathcal{S}$ against mitigation $\mathcal{M}$
$A_{max}(\mathcal{M})$	The highest $a(.)$ achieved by the optimal attack against $\mathcal{M}$
EchoTrail and DRAM parameters (Section 5)	
$F_{dir}(.)$	The effective #activations during the *direct* phase
$F_{surv}(.)$	The effective #activations for post-*direct* phases
$F_{indir}(.)$	The effective #activations during the *indirect* phase
$F_{decoy}(.)$	The effective #activations during the *decoying* phase
R_0	The total number of rows participating in the attack
tREFI	Refresh interval between two REFs
tREFW	Refresh window
tRFM	The execution time of the RFM command
tRC	The time between two consecutive ACT commands
tABO_ACT	The time between the ALERT trigger and the RFM reception
tA2A	The time between two consecutive ALERTs
$\text{EchoTrail}^{t}_{\mathcal{N}}$	An EchoTrail attack with t duration and $\mathcal{N}$ decoy rows

A Additional Definitions and Proofs

Definition 10: Victim hammer count

A victim row $\tilde{v}$ is considered hammered when one of its neighboring aggressor rows, within the blast radius B, is activated. These aggressor rows may be activated either explicitly via the ACT command or implicitly through activations induced by REF or RFM commands. We denote the hammer count of a victim row as $h(\tilde{v})$. This count is reset whenever $\tilde{v}$ is refreshed via REF, mitigated via RFM, or explicitly activated via ACT.

Lemma 1: Priming phase for $PRAC_{base}$

Priming (see Definition 8) of all rows except the target aggressor row $\tilde{r}$ is optimally performed outside a *tREFW*.

Proof. Note that $p(\tilde{r})$ does not reset across a *tREFW* in the specification of $PRAC_{base}$. We differentiate two different types of rows: rows within the blast radius of the target aggressor $\tilde{r}$ and rows outside. Rows within the blast radius will not increase $a(\tilde{r})$ because we prime them to $A_{thresh} - 1$ which does not trigger a mitigation. Therefore, there is no activation benefit performing the activations after the start of the *tREFW* of the target aggressor row. Rows outside the blast radius have no effect on $\tilde{r}$ and therefore also do not have an activation benefit when activated inside the *tREFW* of the target aggressor. However, the optimal *decoying* attack phase requires that all activations raise ALERTs (see Lemma 3), so all decoy rows must be primed. □

Lemma 2: Direct activations

Direct activations on a target aggressor row $\tilde{r}$ are optimal to increase $a(\tilde{r})$, but bounded by $p(\tilde{r})$ reaching $A_{thresh} - 1$.

Proof. $a(\tilde{r})$ increases whenever the row is activated, either directly or through a triggered mitigation of another row within the blast radius. For an implicit activation through the mitigation of an adjacent row $\tilde{r}'$, A_{thresh} activations of $\tilde{r}'$ are required to trigger an increase in $a(\tilde{r})$, resulting in a cost of $A_{thresh} \cdot tRC + tRFM$ for each increase in $a(\tilde{r})$.

For an explicit activation when $p(\tilde{r}) \geq A_{thresh} - 1$, a mitigation is triggered, resulting in $a(\tilde{r}) = 0$ after the mitigation; thus, this activation does not increase $a(\tilde{r})$. In contrast, for a direct activation when $p(\tilde{r}) < A_{thresh} - 1$, the cost is only *tRC*, which minimizes the activation cost. □

Lemma 3: Symmetry and contiguity of the decoying phase

Once a *decoying* attack phase has started with a set of decoy aggressors $\{d_1, d_2, ..., d_{R_0-1}\}$ and a target aggressor $\tilde{r}$, it must always activate the participating (unmitigated) rows with the lowest activation count in every step without interruption in order to be optimal.

Proof. The mitigation selects the rows with the highest activation counts at every mitigation step. Consider a single round of activations

$$\{d_1, d_2, \ldots, d_n, \tilde{r}\}.$$

By the definition of the *decoying* attack (Definition 9), the invariant

$$p(d_1) \geq p(d_2) \geq \cdots \geq p(d_n) \geq p(\tilde{r})$$

holds, with the initial state $p(d_i) = p(\tilde{r}) = A_{thresh} - 1$, i.e., all rows are primed. Assume that we insert an additional activation to an aggressor $\tilde{z}$ at an arbitrary position i:

$$(d_1, d_2, \ldots, d_i, \tilde{z}, d_{i+1}, \ldots, d_n, \tilde{r}).$$

Per mitigation, the *decoying* attack preserves $\frac{3+L-1}{V}$ decoys. When an activation is used for a non-decoy, one fewer decoy is preserved. This increases the decay rate of the decoys, which must then be regenerated using $A_{thresh} - 1 + N$ activations for N target-aggressor activations. The cost of one additional activation to $\tilde{z}$ is therefore $A_{thresh} + N$. Even if the priming occurs outside the *tREFW*, the cost of the additional activation in round N is still N.

When $\tilde{z} = \tilde{r}$, we have

$$p(\tilde{r}) \geq \max_{d_k \in \{d_1, d_2, \ldots, d_N\}} (p(d_k)) + 1$$

at the end of the round, and $\tilde{r}$ will be mitigated. This breaks Lemma 4 and concludes the attack prematurely.

When $\tilde{z} = \tilde{d_i}$ for some decoy d_i, we have

$$p(\tilde{d_i}) \geq \max_{d_k \in \{d_1, \ldots, d_N\} \setminus \{d_i\}} (p(d_k)) + 1,$$

and $\tilde{d_i}$ will be mitigated immediately after its second appearance in the same round. Since this decoy could also be mitigated without the additional activation, the activation does not help the attack progress and is wasted. □

Lemma 4: *decoying* attack phase end

In the final round of the *decoying* phase, the target aggressor row $\tilde{r}$ is selected for mitigation by the last ALERT, since all decoys have already been mitigated. Therefore, the final action within a *tREFW* is an activation (and mitigation) of the target aggressor $\tilde{r}$, which concludes the attack.

Proof. Assume, for contradiction, that an optimal *decoying* schedule mitigates the target aggressor $\tilde{r}$ at time T_1, but T_1 is not the last useful timepoint within the *tREFW* of $\tilde{r}$. Then there exists a later timepoint T_2, with $T_1 < T_2$ and T_2 still within the same *tREFW*, at which the attack can increase the maximum actual activation count of $\tilde{r}$ beyond the value achieved at T_1.

However, the last ALERT of the *decoying* phase occurs only after all decoys have been consumed. Hence, the row selected by this ALERT is $\tilde{r}$. The corresponding mitigation refreshes $\tilde{r}$ and reset $a(\tilde{r})$ in the current refresh window. Let $a(\tilde{r})_{T1}$ denote the actual activation count reached by $\tilde{r}$ immediately before its mitigation. After the mitigation at T_1, the current activation count of $\tilde{r}$ is reset to zero. Therefore, for any later timepoint T_2 in the same *tREFW*,

$$a(\tilde{r})(T_2) \leq A_{\max}(T_1),$$

because activations after T_1 start from a reset state and cannot add to the pre-mitigation count accumulated before T_1.

Thus no operation after T_1 can increase the maximum actual activation count for $\tilde{r}$ beyond the value already achieved at the final *decoying* activation. This contradicts the assumption that delaying work past T_1 yields a better attack. Therefore, in an optimal schedule, the final target-aggressor activation and its mitigation occur as the last useful action within the *tREFW* of $\tilde{r}$. □

Lemma 5: Optimal duration for *direct* + post-*direct survival*

The sequence of *direct*, *indirect*, and *decoying* phases must occur in one *tREFW* to maximize $a(\tilde{r})$ and minimize $\mathcal{L}_{attack} - a(\tilde{r})$ activations.

Proof. Assume that an activation to $\tilde{r}$ occurs outside the selected *tREFW*. Then, by Definition 1, $a(\tilde{r})$ is reset to 0 once all neighboring rows are refreshed, regardless of any prior activations. Since our goal is to maximize $a(\tilde{r})$, and all activations before the refresh

are rendered ineffective, activation counts cannot be accumulated across refresh-window boundaries. In other words, for two activation windows W_1 and W_2 separated by the end of a refresh window, there is no activation sequence for which $a(\tilde{r}) = a(\tilde{r})_{W_1} + a(\tilde{r})_{W_2} \neq a(\tilde{r})_{W_2}$. □

Theorem 2: Optimal attack for $PRAC_{complete}$

$\mathcal{S}^\star(PRAC_{complete}) = \textsc{EchoTrail}_{\mathcal{N}}^{(|\mathcal{N}|+1)\cdot(A_{thresh}-1)\cdot tRC+t_s}$

Where $t_s = tREFW - 8K \cdot tRFC - (A_{thresh}-1)\cdot tRC$
and $|\mathcal{N}| = \dfrac{t_s - tABO\_ACT}{tA2A} \cdot V.$

Proof. Given the *complete* counting mechanism in $PRAC_{complete}$, we cannot use any *indirect* attack, because we cannot increase $a(\tilde{r})$ for the target aggressor row $\tilde{r}$ without also increasing $p(\tilde{r})$. Therefore, after the initial *direct* phase that sets $p(\tilde{r})$ to $A_{thresh}-1$, our t_s must be spent only on *decoying* attacks. □

Theorem 3: Optimal attack for $PRAC_{fine}$

$\mathcal{S}^\star(PRAC_{fine}) = \textsc{EchoTrail}_{\mathcal{N}}^{t_{prime}+t_{direct}+t_{decoy}}$

Where $t_{direct} = (A_{thresh}-1)\cdot tRC$,
$|\mathcal{N}| = \min\left(\left\lfloor \frac{(t_s - tREFI - tABO\_ACT)\cdot V}{tA2A} \right\rfloor, \left\lfloor \frac{\max(t_{prime})}{(A_{thresh}-1)\cdot tRC} \right\rfloor\right)$,
$t_{prime} = |\mathcal{N}| \cdot (A_{thresh}-1)\cdot tRC$,
$t_{decoy} = \frac{|\mathcal{N}|\cdot tA2A}{V} + tABO\_ACT$

Proof. We first note that there is one refresh interval *tREFI* for a row group of 8. For $B = 1$, this refresh mitigates 4 aggressors. However, we can perform $\frac{tREFI}{tA2A} \approx 6.7$ mitigations within the same time interval. As a result, *fine* resetting does not interfere with any attack that starts at the same time. In other words, because the *fine* resets and refreshes are staggered, and we are faster at consuming the decoys, this is not a limiting side-condition. However, within the *tREFW* of any given decoy aggressor, which includes both priming and its use in the *decoying* phase, we must fit both the *direct* activations and the activations performed during *decoying*. Because the *direct* phase provides the most useful activations per unit time, up to a maximum of $A_{thresh}-1$ activations, the maximum available time ($C2$) for the *decoying* phase is $tREFW - t_{direct} - 8K\cdot tRFC - tREFI$. For the *priming* phase ($C1$), we have $\max(t_{prime}) = tREFW - 8K\cdot tRFC - tREFI$. For the EchoTrail instance, this results in three conditions:

$$|\mathcal{N}| \leq \frac{\max(t_{prime})}{(A_{thresh}-1)\cdot tRC} \tag{C1}$$

$$t_{decoy} \leq tREFW - t_{direct} - 8K\cdot tRFC - tREFI \tag{C2}$$

$$t_{decoy} \geq \frac{|\mathcal{N}|\cdot tA2A}{V} + tABO\_ACT \tag{C3}$$

$C1$ follows from Lemma 3 and provides the upper bound on $|\mathcal{N}|$ for the *priming* phase. We therefore must choose an optimal $|\mathcal{N}|$ that maximizes t_{decoy} under the aforementioned conditions. $C3$ is the main condition that we aim to maximize within our bounds. $C1$ provides the first bound; from $C2 \cap C3$, we obtain the other one:

$$t_{decoy} = \frac{tA2A}{V}\min\{N_1, N_2\} + tABO\_ACT,$$
$$N_1 = \left\lfloor \frac{(t_s - tREFI - tABO\_ACT)\cdot V}{tA2A} \right\rfloor,$$
$$N_2 = \left\lfloor \frac{\max(t_{prime})}{(A_{thresh}-1)\cdot tRC} \right\rfloor.$$

Equivalently, $|\mathcal{N}| = \min\{N_1, N_2\}$, where $t_{prime} = |\mathcal{N}|\cdot(A_{thresh}-1)\cdot tRC$. □

This leads to the limiting element being either the *priming* phase or the *decoying* phase, depending on A_{thresh}. For example, $A_{thresh} \geq 64$ results in the *priming* phase providing the upper bound, while lower A_{thresh} are limited by the *decoying* itself.

Theorem 4: Optimal attack for $PRAC_{coarse}$

$\mathcal{S}^\star(PRAC_{coarse}) = \textsc{EchoTrail}_{\mathcal{N}_1}^{tREFW-r_1} \cdot \textsc{EchoTrail}_{\mathcal{N}_2}^{tREFW-r_2}$

where $r_1 = t_{direct1} + t_{decoy1}$, $r_2 = t_{direct2} + t_{decoy2}$,
$r_1 + r_2 = tREFW - 8K\cdot tRFC$.

Proof. Intuitively, the problem is the same as $PRAC_{complete}$, but with an additional all-counters reset within the window and limited priming. The problem therefore reduces to optimizing the placement of this additional reset within the window.

Trivially, aligning the counters reset with the refresh of the target row yields

$$\textsc{EchoTrail}_{\frac{tREFW-8K\cdot tRFC}{tRC\cdot A_{thresh}-1}}^{(n+1)\cdot(A_{thresh}-1)\cdot tRC+t_s},$$

i.e., $\mathcal{S}^\star(PRAC_{complete})$ with limited priming. However, because *direct* activations are the most efficient, an unaligned reset allows the attacker to perform $2\times(A_{thresh}-1)$ activations if the counters reset occurs between two direct phases. The optimal solution therefore maximizes activations by $F_{decoy}(.)$ by constructing two EchoTrail sequences (i.e., parameters r_1 and r_2). □

Theorem 5: Optimal attack for $PRAC_{priorityQ}$

$PRAC_{priorityQ}$ has the same optimal attack guarantees of the $PRAC_{complete}$: $\mathcal{S}^\star(PRAC_{priorityQ}) = \mathcal{S}^\star(PRAC_{complete})$
$= \textsc{EchoTrail}_{\mathcal{N}}^{(|\mathcal{N}|+1)\cdot(A_{thresh}-1)\cdot tRC+t_s}$,
where $t_s = tREFW - 8K\cdot tRFC - (A_{thresh}-1)\cdot tRC$,
and $|\mathcal{N}| = \dfrac{t_s - tABO\_ACT}{tA2A}\cdot V.$

Proof. We show that an *idealQ* queue and a *priorityQ* queue provide the same maximum activation count for any activation history between two mitigations, $\{\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_n\}$. We note that a *priorityQ* queue allows an attack to be *paused*: unlike an *idealQ* queue, it does not automatically mitigate all rows with $p(\tilde{r}) \geq A_{thresh}$, but only mitigates such a row when it is activated again and observed by the queuing element.

However, for any sequence $(r_1, r_2, \ldots, r_n)$ such that there exists a row $r \in \{r_1, \ldots, r_n\}$ with $p(r) \geq A_{thresh}$, the row with the maximum $p(r)$ in the sequence will be mitigated. As a result, a decoy $\tilde{d}$ with $a(\tilde{d}) \geq a(\tilde{r})$ is still required to protect the target aggressor row $\tilde{r}$, following the *decoying* attack.

While a *decoying* attack can be paused without losing any decoy, no activation exists that increases $a(\tilde{r})$ more efficiently. If such an

indirect attack existed and were more efficient than contiguous *decoying* activations, it would already have been used during the *indirect* phase. Moreover, because $p(\tilde{r}) \geq A_{thresh} - 1$ after the *direct* phase, any access that increases $p(\tilde{r})$ requires continuing the *decoying* attack. This proves the equivalence to the *idealQ* queue. □

Theorem 6: Optimal attack for $PRAC_{fullFifo}$

$\mathcal{S}^{\star}(PRAC_{fullFifo}) = \textsc{EchoTrail}_0^{tREFW-8K\cdot tRFC}$

Proof. From Lemma 2, we know that direct activations in the *direct* phase are the most efficient. For a *fullFifo* queue of length $|Q| > 1$, activations to a single target aggressor row $\tilde{r}$ do not trigger a mitigation even when $p(\tilde{r}) > A_{thresh}$. This removes the bound in Lemma 2 and makes activating the same row for the entire *tREFW* optimal. □

B Additional Discussions on Multi-banks setup

When bank B_0 triggers an ALERT, all banks (B_0 to B_{31}) receive RFMs for mitigation. There are three possible ways for non-triggering banks (banks B_1 to B_{31}) to handle these RFMs:

- *Lazy* handling: the non-triggering banks do not perform any mitigations;
- *Passive opportunistic* handling: all the non-triggering banks mitigate the same row ID selected by the triggering bank, i.e., $r = B_0.AlertQ.head$;
- *Active opportunistic* handling: each non-triggering bank B_i individually chooses its own row for mitigation based on its queuing mechanism, i.e., $r_i = B_i.AlertQ.head$, even if $p(r_i) < A_{thresh}$.

Our analysis shows that *active opportunistic* mitigations are essential for security. Without them, both lazy and passive opportunistic mechanisms provide attackers with a powerful survival strategy, allowing them to raise $A_{max}(\tilde{r})$ up to $196k$, independent of the A_{thresh} level. For instance, an attacker can trigger back-to-back ALERTs in banks other than the target bank and use the ACTs between these ALERTs to repeatedly access the target aggressor, which is never selected for mitigation because the selections are made by other banks.