Mechanised operational semantics of Rowhammer

MARTIN BERGER, University of Sussex, UK and Montanarius Ltd, UK
AMIR NASEREDINI, Huawei R&D UK Ltd, UK

Rowhammer is a hardware vulnerability in dynamic random-access memory (DRAM) in which repeated accesses to one or more aggressor rows can induce bit-flips in nearby victim rows. This phenomenon violates a core assumption of conventional programming language semantics: that reading from or writing to one memory location does not modify others. Despite the security importance of this phenomenon, there is no widely established formal framework connecting Rowhammer faults with program behaviour. This makes it difficult to reason rigorously about the efficacy of proposed Rowhammer defences and the program-level guarantees they provide. To address this gap, we present a probabilistic small-step operational semantics for an idealised imperative language subject to Rowhammer-style faults. The semantics is high-level in that it abstracts from DRAM internals and semiconductor physics. A general probabilistic fault model parameterises the semantics, representing Rowhammer-style faults by assigning probabilities to bit-flips during read or write operations within a specified victim region. The resulting distributions are propagated through programs using the standard monadic structure of probabilistic computation. As a case study, we formalise physical separation, a well-known defence that places program variables sufficiently far apart in physical memory that an access to one variable cannot disturb another. We prove a distribution-independent semantic collapse theorem: for every finite execution, including prefixes of terminating and non-terminating executions, the protected projection of the probabilistic Rowhammer semantics is the Dirac distribution of the corresponding Rowhammer-free execution. Furthermore, we develop an observation-parametric account of secure information flow. Non-interference is expressed as a hyperproperty comparing the distributions of low observations from low-equivalent initial memories. Under physical separation, ordinary non-interference and probability-sensitive Rowhammer non-interference coincide for every observer of protected behaviour. Consequently, physical separation preserves non-interference for every admissible fault model, while every Rowhammer non-interference violation reflects a violation already present in the Rowhammer-free semantics. The development is fully mechanised in Lean using mathlib, relying on no unfinished proofs or problem-specific axioms.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Formal methods**; • **Security and privacy →** *Information flow control*; *Systems security*; **Logic and verification**.

Additional Key Words and Phrases: Probability theory, Probabilistic fault models, Rowhammer, Programming language semantics, Operational semantics, Relational safety, Mechanised verification, Lean 4, Secure information flow, Non-interference.

1 Introduction

Most reasoning about software relies on a locality assumption about memory access effects:

Reading or writing one memory location does not modify other locations.

Rowhammer violates this assumption. Repeated activation of one or more aggressor rows in dynamic random-access memory (DRAM) can induce bit-flips in physically nearby victim rows [16, 35]. The resulting mismatch between the memory model assumed by software semantics and the behaviour of its semiconductor implementation creates a semantic gap: hardware research describes disturbance mechanisms, address mappings, and mitigations, whereas software verification reasons about commands, stores, traces, observations, and security properties. This gap is particularly important for security. Rowhammer is usually presented as an integrity failure, but corruption of a high-security value can alter control flow or a later low-security output, while corruption of a low-security location can directly change an attacker-visible result. Existing theories of secure

Authors' Contact Information: Martin Berger, contact@martinfriedrichberger.net, University of Sussex, Brighton, UK and Montanarius Ltd, London, UK; Amir Naseredini, sahnaseredini@gmail.com, Huawei R&D UK Ltd, Cambridge, UK.

information flow and non-interference [10, 34] therefore cannot simply be applied unchanged: they require a semantics that makes the probabilistic, non-local effect of memory accesses explicit. This paper asks closely connected research questions.

- **RQ1.** Can Rowhammer be modelled compositionally in the familiar language of operational semantics, without modelling device physics?
- **RQ2.** Can the Rowhammer fault mechanism be captured by a small, conceptually non-intrusive semantic interface?
- **RQ3.** Can such an interface be made substantially independent of the target programming language?
- **RQ4.** Can such a model express and justify a concrete Rowhammer defence?
- **RQ5.** Can the abstraction make Rowhammer amenable to existing semantic security frameworks, such as information flow and non-interference reasoning?

We answer all questions affirmatively.

1.1 The semantic view: abstracting memory access

A standard way to give semantics to an imperative language is to treat programs as state transformers. In its most concrete form, assignment and variable lookup are explained by clauses such as

$$[[x := e]](m) = m[x \mapsto v] \qquad \text{where} \qquad [[e]](m) = v,$$

and

$$[[x]](m) = m(x).$$

where m ranges over memories and e over expressions. These equations present memory as a concrete map and use two primitive operations on that map: read the value stored at a location, and update the value stored at a location. A recurring lesson of programming language semantics is that these two operations should be made explicit. Once lookup and update are abstracted from the concrete function space $\mathsf{Memory} = \mathsf{Loc} \to \mathsf{Val}$, where Loc are locations in memory, and Val the values being stored, the syntax of the language no longer has to know how memory is represented. The same assignment rule can then be interpreted over different models of store, aliasing, heaps, concurrency, failure, or other effects.

This idea has appeared repeatedly, under different names. McCarthy's select/store axioms for arrays isolate reading and updating as primitive operations on memories [23]. Monadic and algebraic-effect semantics make the separation systematic: computations with state are interpreted as effectful maps, and state itself may be presented by operations such as lookup and update together with their laws [24, 32]. Modular operational semantics pursues a related goal from the small-step side, factoring stores and other auxiliary components out of individual transition rules so that language constructs can be specified more independently [25]. The common pattern is that memory access becomes an interface rather than an implementation detail.

Our Rowhammer semantics uses this idea. Ordinary memory operations are deterministic:

$$\mathsf{read} : (\mathsf{Loc} \times \mathsf{Memory}) \to \mathsf{Val} \qquad \mathsf{write} : (\mathsf{Loc} \times \mathsf{Val} \times \mathsf{Memory}) \to \mathsf{Memory}$$

Rowhammer-affected read and write are instead probabilistic operations[1]:

$$\mathsf{read} : (\mathsf{Loc} \times \mathsf{Memory}) \to \mathsf{Dist}(\mathsf{Val} \times \mathsf{Memory}) \quad \mathsf{write} : (\mathsf{Loc} \times \mathsf{Val} \times \mathsf{Memory}) \to \mathsf{Dist}(\mathsf{Memory})$$

The operations still have the shape of a local memory access, but their results are distributions that include the post-access memory. The distributions model Rowhammer faults.

[1] We discuss in Section 5 why the codomain of read needs to be $\mathsf{Dist}(\mathsf{Val} \times \mathsf{Memory})$ and not $\mathsf{Val} \times \mathsf{Dist}(\mathsf{Memory})$.

The rest of the semantics is then obtained monadically: deterministic steps are embedded as Dirac distributions, and compound computations are sequenced by monadic bind for the probability monad. Thus the Rowhammer model is not a separate hardware semantics bolted onto the language. It is a small probabilistic reinterpretation of the read/write interface already present in ordinary state-transformer semantics. This is the key abstraction boundary used throughout the paper: Rowhammer enters only at memory accesses, the surrounding operational semantics lifts compositionally around that local probabilistic effect.

This move from deterministic to probabilistic state transformers is not unique to Rowhammer: deterministic Turing machines and programming languages naturally become probabilistic by allowing programs to branch according to a probability distribution. In those settings, probabilistic choice is an intentional feature of the syntax or model. Rowhammer is different. Its probabilistic behaviour is an unwanted, observable consequence of the physical substrate, making it conceptually closer to Shannon's noisy-channel model from information theory: an intended memory operation is transmitted through an unreliable medium and may be received as corrupted state. In contrast, this noise is not merely ambient: it is actively shaped by an attacker's chosen access patterns and strictly constrained by physical locality in DRAM. Our semantic task is therefore to expose this hardware-level unreliability without modifying the language syntax, placing a probabilistic fault model directly behind the ordinary read/write interface. Ultimately, the novelty lies not in using probabilistic state transformers per se, but in the observation that Rowhammer can be elegantly isolated as a probabilistic reinterpretation of memory access.

1.2 Contributions

To address the research questions, we make the following contributions:

- A fine-grained labelled operational semantics in which Rowhammer has a small, explicit interface at reads and writes.
- A general probabilistic fault model separating spatial confinement from numerical assumptions about bit-flips.
- A distribution-independent soundness theorem for physical separation, a well-known Rowhammer defence, covering terminating and non-terminating executions.
- An observation-parametric theorem that physical separation preserves and reflects probability-sensitive non-interference.

The complete development is mechanised in Lean [6] using mathlib [37]. The development contains no unfinished proofs or problem-specific axioms. An axiom audit (`scripts/axiomAudit.sh` in the repo) confirms that all results rest only on Lean's three foundational axioms (`propext`, `Classical.choice`, `Quot.sound`): no probability axioms are added. The Lean excerpts shown are automatically extracted verbatim from the sources so they cannot drift from the development. All Lean code is open-sourced and anonymised at [1]. Full operational rules, auxiliary metatheory, and extended mechanisation details are provided in the supplementary material. The appendices cited in the text are provided as anonymised supplementary material.

2 Introduction to Rowhammer

To appreciate what the present work does and does not promise, it helps to understand the physical mechanism it abstracts. The rest of the paper relies on only a few concepts from DRAM architecture, rows, activation, refresh, and the locality of disturbance, and we introduce them here from first principles. Readers already familiar with Rowhammer may skip to the examples at the end of this section. For the underlying semiconductor physics we refer to [39]. For a broad survey of the attack landscape see [26].

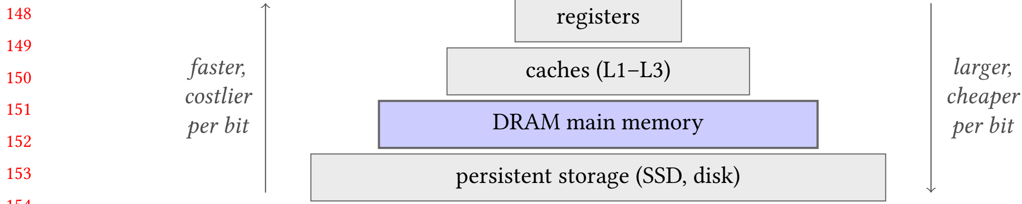


Fig. 1. The memory hierarchy. Each level trades speed for capacity. DRAM (highlighted) wins its place through cell minimalism, one transistor and one capacitor per bit, and the resulting drive towards ever-denser packing is the root cause of the disturbance effects this paper models.

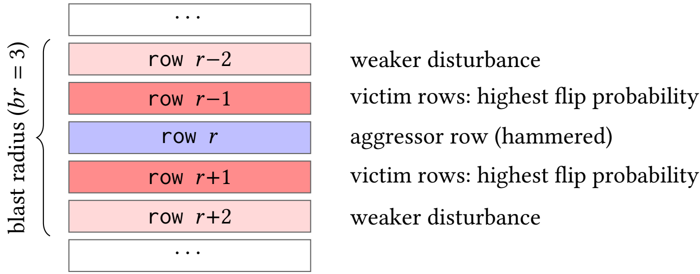
The memory hierarchy. DRAM occupies an important point in a trade-off (Figure 1). Above it sit registers and caches: fast, but expensive. Below sits persistent storage: vast, but orders of magnitude slower. DRAM's role is to be as large and as cheap as possible while remaining just fast enough. It achieves this with only one transistor and one capacitor per bit, against the six transistors of a cache cell. This economic pressure towards density is Rowhammer's root cause. The more tightly cells are packed, the stronger the electrical coupling between them, which is why the problem has worsened with every DRAM generation [16].

How DRAM stores data. At the hardware level, each DRAM bit is stored as electrical charge in a capacitor, guarded by an access transistor. Cells are organised into large two-dimensional grids called *banks*, which operate independently of one another. Within a bank, a horizontal line of cells, a *row*, typically a few kilobytes wide, shares a common control wire known as the *wordline*, while cells in the same column share a *bitline*. Individual cells cannot be addressed directly. To access any cell, the memory controller must *activate* the row containing it: raising the row's wordline voltage connects every cell of the row to its bitline and copies the row's contents into the bank's *row buffer*, where reads and writes then take place. Accessing a different row of the same bank requires closing the currently open row, meaning lowering its wordline voltage before activating the new one. Capacitors leak charge. Left alone, a DRAM cell loses its stored charge within a fraction of a second, so the memory controller periodically rewrites every row—a *refresh*. In commodity DRAM, every row is typically guaranteed a refresh once every 64 ms [16]. The reliability of DRAM therefore rests on a quantitative margin: between two refreshes, a cell's charge must stay on the correct side of a threshold, despite whatever electrical disturbance its neighbourhood produces.

The Rowhammer effect. Kim et al. [16] showed in 2014 that this margin can be exhausted *deliberately*. Every activation of a row causes slight voltage fluctuations on its wordline, and these fluctuations accelerate charge leakage in the cells of *physically adjacent* rows, an electromagnetic coupling effect that grows stronger as cells are packed more densely. A single activation is harmless. But a program that activates the same row hundreds of thousands of times within one refresh interval, *hammering* it, can drain a neighbouring cell past its threshold before the next refresh arrives. The value read thereafter is not the value last written: a bit has flipped in a row that was never written to, or even addressed, by the program. The repeatedly activated rows are called *aggressor* rows and the corrupted ones *victim* rows (Figure 2). The minimum number of activations needed is called the *Rowhammer threshold*: it has dropped sharply as DRAM density has increased, from over a hundred thousand activations on DDR3 to tens of thousands on modern chips [7, 16], which is why the problem is getting worse, not better, with each DRAM generation.

Rowhammer is not an exotic laboratory effect. It is exploitable from ordinary unprivileged code, including JavaScript in a web browser [11], and flipping a single page-table or capability bit can

197
198
199
200
201
202
203
204
205



206
207
208
209

Fig. 2. Disturbance is local. Repeatedly activating (*hammering*) row r accelerates charge leakage in nearby rows. The flip probability is highest for the immediate neighbours $r \pm 1$ and decays with distance. Beyond the blast radius br the effect is negligible. The victim rows are corrupted although the program never writes to or even addresses them.

210
211
212
213
214
215
216
217

escalate one hardware fault into arbitrary memory access [35]. The effect has also grown richer since its discovery. Half-Double [17] induces flips beyond immediate neighbours by combining accesses at different distances, TRRespass [7] and Blacksmith [15] bypass in-DRAM mitigations with many-sided and non-uniform hammering patterns, and RAMBleed [20] turns the data-dependence of flip probabilities into a *read* primitive, making Rowhammer a confidentiality problem as well as an integrity problem.

218
219
220
221
222
223
224
225

Locality and the blast radius. The one saving grace of DRAM physics is its strong spatial locality: the disturbance an activation causes decays rapidly with physical distance, and beyond a few rows it is negligible. In the rest of the text we call the maximum distance at which an access can disturb a cell the *blast radius*, abbreviated br. The blast radius of a given DRAM device can be approximated empirically, and it is the key intuition behind an entire class of defences: if everything worth protecting is kept at least br rows away from everything an attacker can hammer, the disturbance never reaches it.

226
227
228
229
230
231
232
233
234
235
236
237

Physical separation defences. Defences based on *placement* exploit exactly this locality. CATT [2] partitions kernel from user memory with unused *guard rows*, so that user-space hammering cannot reach kernel rows. ZebRAM [18] interleaves guard rows between all data rows (Figure 3). The correctness claim of such defences is physical: every row an attacker can reach is separated from every row worth protecting by more than the disturbance radius. In this paper the discipline is captured by a predicate $\mathrm{safe}(br, locs)$: the victim region generated by an access to any protected location is disjoint from the protected set $locs$. In the distance-based instance this means that distinct protected rows lie at least br apart. But separation is required *within* the protected set too, because the program's own accesses hammer their own neighbourhoods. The guard-row layouts above are the same discipline applied to the whole address space, with the attacker-reachable region kept at distance br from everything protected.

238
239
240
241
242
243
244
245

The semantic gap. Program verification speaks a different language: commands, stores, traces, observables. A proof about a program is a proof about its semantics, and standard semantics assume that memory changes only when written. Rowhammer breaks exactly that assumption, and placement defences restore it, but no theorem connects the two levels. What is missing is a semantics in which the physical claim ("flips are confined to a blast radius") and the placement claim ("protected rows are outside every blast radius") can both be stated, and a theorem that together they restore the ordinary meaning of programs. This paper supplies both. The examples below,

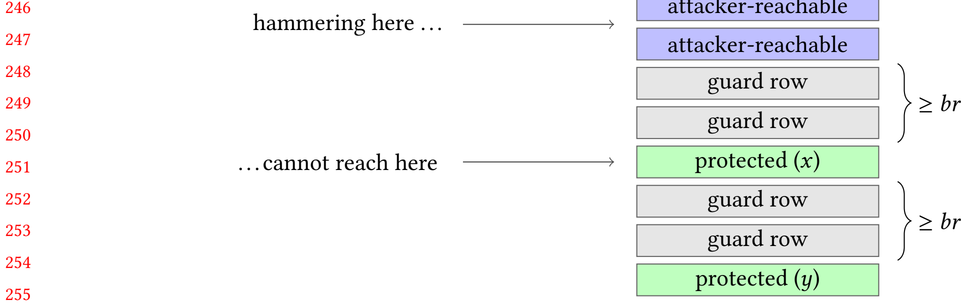


Fig. 3. Physical separation with a blast radius of $br = 3$. Guard rows keep every protected row at least br rows away from every row the program (or an attacker) accesses, including the *other* protected rows, since accesses to x hammer x's own neighbourhood. This is the layout discipline that the predicate $\mathsf{safe}(br, locs)$ of Section 5.1 abstracts.

adapted from [27], illustrate what goes wrong without separation, and, read against Theorem 6.1, identify exactly which hypothesis each failure violates.

Example 2.1 (Corruption without a write). Let x and y occupy adjacent rows, and let the program hammer y while never assigning to x:

$$x := 1;\ \mathsf{while}\ (0 = 0)\ \mathsf{do}\ y := y + 1.$$

Every access to y disturbs the blast radius of y's row, which contains x. In every ordinary semantics x holds 1 forever. Under Rowhammer, the probability that x still holds 1 decays with every iteration. No analysis that equates "unwritten" with "unchanged" can see this.

Example 2.2 (A dead branch awakens). Corruption is not limited to data, it can also redirect control. Let x and y be adjacent and consider

$$x := y;\ x := x;\ \mathsf{if}\ x = y\ \mathsf{then\ skip\ else}\ P.$$

Ordinarily the guard is always true and P is dead code. But the reads and writes of x hammer x's neighbourhood, which contains y: a flip in y between the first assignment and the guard makes the guard false, and the machine executes P. Which code runs is now probabilistic. This is why the protected view of Section 6.1 tracks the *residual program* and the *access trace*, not just memory: a defence that preserved memory but not control flow would not restore the meaning of programs. Note also that both x and y here are the program's *own* variables: if the protected set $locs = \{x, y\}$ places them in adjacent rows, $\mathsf{safe}(br, locs)$ fails, and the theorem correctly does not apply: separation is required *within* the protected set, not only between the program and the outside world.

Example 2.3 (An unowned access). Separation of the program's own rows is not enough either. Suppose $locs = \{x\}$ is safely laid out, but the program also touches a row $z \notin locs$ that happens to be adjacent to x:

$$z := z + 1;\ \mathsf{if}\ x = 0\ \mathsf{then\ skip\ else}\ P.$$

The access to z triggers a fault in z's blast radius, about which $\mathsf{safe}(br, locs)$ says nothing, and that radius contains x. This is what the well-formedness hypothesis $\mathsf{wf}_{locs}(P)$ of Theorem 6.1 rules out: every location the program touches must be part of the protected layout, so that every fault the program can trigger is one the separation condition constrains.

295 These examples give us the shape of the theorem we need: quantified over fault behaviour (flip
296 probabilities are the attacker's, not ours), conclusion covering memory, control, and accesses alike,
297 and hypotheses (frame-locality, separation, well-formedness) each of which is violated by some
298 concrete program above when dropped.
299

3 Mathematical preliminaries

We write $\mathcal{P}(S)$ for the *powerset* of a set S and $\mathcal{P}_{fin}(S)$ for the set of all finite subsets of S. The Rowhammer semantics assigns to each transition a *distribution* over successor configurations. This section fixes the small amount of probability theory involved. Readers familiar with the probability monad (also known as Giry monad) [8, 21] or with monadic treatments of probabilistic programs [5, 19, 33] can skim it for notation. All probability spaces in this paper will be discrete, *i.e.*, finite or countably infinite, so all occurring measure spaces are trivial and we shall not mention them. For a type α, we write $\mathrm{Dist}(\alpha)$ for the type of discrete probability distributions on α, *i.e.*, functions $\mu : \alpha \to [0,1]$ with $\sum_x \mu(x) = 1$. The *support* of μ, written $\mathrm{supp}(\mu)$, is the set of x with $\mu(x) > 0$. The *Dirac distribution*, also called *point mass*, at $x \in \alpha$ is the distribution $\delta_x \in \mathrm{Dist}(\alpha)$ given by $\delta_x(y) = 1$ if $y = x$, and $\delta_x(y) = 0$ otherwise, for all $y \in \alpha$. A *probability kernel* from α to β is an input-indexed family of distributions, *i.e.*, a function $\kappa : \alpha \to \mathrm{Dist}(\beta)$. For each input $a \in \alpha$, the value $\kappa(a)$ is therefore a distribution on β. Any distribution $\mu \in \mathrm{Dist}(\beta)$ determines the constant kernel $a \mapsto \mu$. Likewise, a deterministic function $f : \alpha \to \beta$ embeds into the probabilistic setting as the Dirac kernel $a \mapsto \delta(f(a))$. In this paper, "kernel" always means a discrete probability kernel of the form $\alpha \to \mathrm{Dist}(\beta)$. In the Lean development this is represented as a function returning a `PMF` (probability mass function), rather than by mathlib's more general measure-theoretic type `ProbabilityTheory.Kernel`. Sequential composition of probabilistic computations is monadic. For our purposes, a monad consists of a type constructor M together with operations *unit* $\mathrm{return}_\alpha : \alpha \to M(\alpha)$ and *bind* $\gg\!= : M(\alpha) \to \big(\alpha \to M(\beta)\big) \to M(\beta)$, satisfying the left-identity, right-identity, and associativity laws [22, 24]. Intuitively, $M(\alpha)$ is a type of computations producing values of type α, return embeds a pure value, and bind sequences. The classical probability monad $\mathcal{G}$ instantiates this structure with probability measures on measurable spaces. Since all spaces in this paper are discrete, we work with its discrete analogue, spelled out next. On objects it is given by $\mathcal{G}(\alpha) \triangleq \mathrm{Dist}(\alpha)$. On morphisms, $\mathcal{G}$ acts by pushforward of distributions: for $f : \alpha \to \beta$, we have $\mathcal{G}(f) : \mathrm{Dist}(\alpha) \to \mathrm{Dist}(\beta)$, and for $\mu \in \mathrm{Dist}(\alpha)$, we set $\mathcal{G}(f)(\mu)(y) = \sum_{x \in \alpha, f(x)=y} \mu(x)$. The unit $\mathrm{return}_\alpha : \alpha \to \mathcal{G}(\alpha)$ is the Dirac distribution δ_x, in other words, the point mass concentrated at x. $\mathcal{G}$'s bind operation, given a distribution $\mu \in \mathrm{Dist}(\alpha)$ and a probability kernel $\kappa : \alpha \to \mathrm{Dist}(\beta)$, is written

$$\textbf{let } x \leftarrow \mu \textbf{ in } \kappa(x),$$

and defined pointwise by $(\textbf{let } x \leftarrow \mu \textbf{ in } \kappa(x))\,(y) = \sum_{x \in \alpha} \mu(x)\,\kappa(x)(y)$. In the Lean development, $\mathcal{G}$ is represented by `PMF`. The Lean development mirrors this notation in the file `DiscreteDist.lean`. There, `Dist` is an abbreviation for mathlib's `PMF`, the type of discrete probability mass functions. Thus the formalisation does not assume finite support. The file provides the small interface used in the proofs: Dirac distributions, bind, map, support lemmas, and injectivity of Dirac.

4 A deterministic small-step operational semantics for a WHILE language

The aim of this paper is to express Rowhammer in terms of programming language semantics, so as to be able to reason about program correctness and the efficacy of Rowhammer defences using the extensive tool-sets developed over the previous decades by the programming language and verification communities, but without having to go through semiconductor physics, or even knowledge of DRAM organisation. For this purpose we need a paradigmatic programming language that is simple

344 enough to allow us to explain the key ideas without being drowned out by language complexity,
345 but at the same time general enough to cover interesting Rowhammer-related phenomena. We
346 choose the well-known WHILE language [29, 38]. Its syntax is given by the following rules.

$$\begin{aligned} e &::= c \mid x \mid e + e \\ b &::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \\ P &::= \mathsf{skip} \mid x := e \mid P; P \mid \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ b\ \mathsf{do}\ P \end{aligned}$$

352 Here c ranges over integers, and x over memory locations, collectively denoted Loc. It is natural to
353 assume that Loc = $\mathbb{Z}$, but nothing in our development depends on that. Since the language lacks
354 higher-order functions and dynamic memory allocation, we can associate with each e, b and P the
355 set of locations syntactically mentioned. This gives rise to the `WellFormed` predicates in our Lean
356 formalisation: e.g., `p.WellFormed locs` means that the locations occurring syntactically in the
357 program `p` are contained in the set `locs`. Clearly well-formedness is preserved by reduction.

358 We give a conventional small-step operational semantics with two minor twists. The operational
359 semantics is a deliberately *fine-grained* small-step semantics: arithmetic expressions, Boolean
360 expressions, and programs all reduce one step at a time, so every memory read and every write is
361 an individual transition. Transitions carry a *label* recording the access they perform:

$$\ell ::= \mathsf{read}(x) \mid \mathsf{write}(x) \mid \tau.$$

364 A read of variable x is labelled $\mathsf{read}(x)$, the write performed by an assignment to x is labelled
365 $\mathsf{write}(x)$, and every other transition, operator reduction, branch selection, sequencing, loop un-
366 folding, is silent (τ). We write Acc for the set of all such labels. The labels play no role in the
367 deterministic semantics itself (only assignment writes modify memory) but help injecting the
368 Rowhammer faults (Section 5). Congruence rules propagate the label of their premise. We use three
369 labelled judgements on configurations

$$\langle e, m\rangle \xrightarrow{\ell}_a \langle e', m'\rangle \qquad \langle b, m\rangle \xrightarrow{\ell}_b \langle b', m'\rangle \qquad \langle P, m\rangle \xrightarrow{\ell}_p \langle P', m'\rangle$$

372 In all cases, m ranges over memories, *i.e.*, functions of type Loc $\to \mathbb{Z}$. We use Memory to refer to the
373 set of all memories. Evaluation is left-to-right, and conjunction is deliberately non-short-circuiting.
374 Values and skip have no outgoing transition. The relations are deterministic, and every non-terminal
375 configuration can progress. The definitions are standard, and we give the most relevant rules next.

$$\frac{-}{\langle x, m\rangle \xrightarrow{\mathsf{read}(x)}_a \langle m(x), m\rangle} \qquad \frac{\langle e, m\rangle \xrightarrow{\ell}_a \langle e', m'\rangle}{\langle x := e, m\rangle \xrightarrow{\ell}_p \langle x := e', m'\rangle} \qquad \frac{-}{\langle x := v, m\rangle \xrightarrow{\mathsf{write}(x)}_p \langle \mathsf{skip}, m[x \mapsto v]\rangle}$$

$$\frac{-}{\langle \mathsf{while}\ b\ \mathsf{do}\ P, m\rangle \xrightarrow{\tau}_p \big\langle \mathsf{if}\ b\ \mathsf{then}\ \big(P; \mathsf{while}\ b\ \mathsf{do}\ P\big)\ \mathsf{else}\ \mathsf{skip}, m\big\rangle}$$

383 The full rule sets and this basic metatheory are standard and given in Appendix A. We write $\overset{s}{\underset{p}{\Rightarrow}}$
385 for the reflexive and transitive closure of $\xrightarrow{l}_p$. The length $|s|$ counts every transition, expression
386 steps included; the *access trace* $\mathsf{acc}(s)$ is the subsequence of non-τ labels, whose length is the
387 number of memory accesses the run performs. A program *terminates* with final memory m', written
388 $\langle P, m\rangle \Downarrow m'$, when $\langle P, m\rangle \overset{s}{\underset{p}{\Rightarrow}} \langle \mathsf{skip}, m'\rangle$ for some s; it *diverges*, written $\langle P, m\rangle \Uparrow$, when for every n
390 some s with $|s| = n$ extends its run. The transitions give a partial function on configurations $\langle P, m\rangle$.
391 It will later be convenient to extend this to total functions. We do this by defining a deterministic

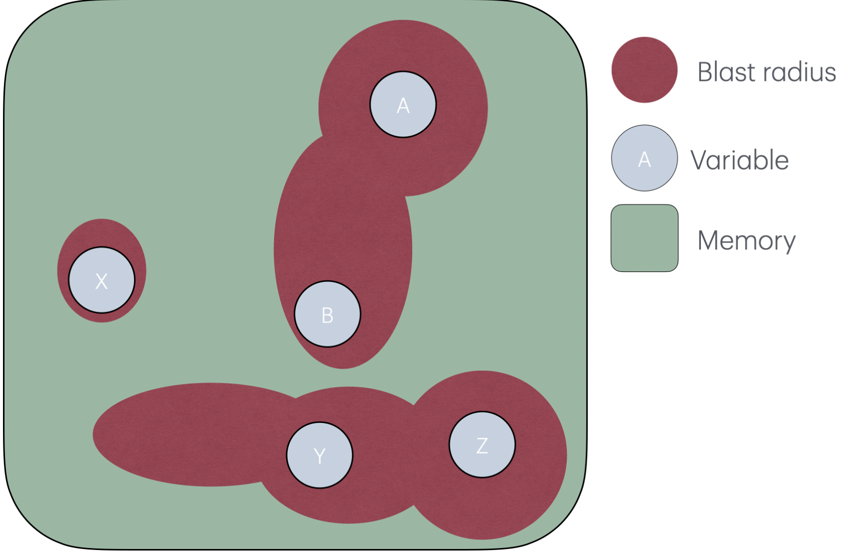


Fig. 4. The idea behind the fault model is that each variable x has a 'blast radius', a set of other memory locations that may get flipped when x is accessed. Locations outside the blast radius will not be flipped. The fault model formalises the shape of each blast radius, and the flip probabilities. Unlike Figures 2 and 3, this image does not show row shapes, because our formal model does not assume any specific geometry of blast radii.

absorbing run denoted $\mathrm{run}^n_{\mathrm{det,tr}}(C)$: iterate the step function n times, letting terminal configurations step to themselves with label τ, and accumulate the labels. Its result is the unique configuration-with-trace reached after n steps.

5 A probabilistic model of Rowhammer

This section addresses RQ1, RQ2 and RQ3: we now refine the ordinary semantics from the previous section with probabilistic Rowhammer fault models. We aim for the interface between the programming language semantics and the fault model to be minimal and canonical. The syntax, evaluation order, and reduction contexts are unchanged, and the refinement touches exactly one place: the $\mathrm{read}(x)$- and $\mathrm{write}(x)$-labelled transitions of Section 4 additionally sample from the Rowhammer fault model and produce a probability distribution (e.g., bit-flips in successor memories), while τ-labelled transitions remain deterministic.

With the changes to the read and write functions already explained in the Introduction, this amounts to changing the state transformer semantics of programs from type

$$\mathsf{Memory} \longrightarrow \mathsf{Memory}$$

from Section 4 into probabilistic state-transformers

$$\mathsf{Memory} \longrightarrow \mathsf{Dist}(\mathsf{Memory})$$

(and likewise for expressions). The fault probabilities are lifted in a canonical way to general programs using the probability monad from Section 3.

442 *Why reads return a joint distribution.* There is a small but important typing choice in the Rowham-
443 mer semantics. The ordinary read operation has type
444
445
$$\text{read} : (\text{Loc} \times \text{Memory}) \longrightarrow \text{Val}$$

446 Since a Rowhammer access to a location x is intended to disturb only other locations, one might
447 expect the probabilistic read operation to have type
448
449
$$(\text{Loc} \times \text{Memory}) \longrightarrow (\text{Val} \times \text{Dist}(\text{Memory}))$$

450
451 returning the value stored at x together with a distribution over successor memories. In the model
452 used in this paper, this would indeed be sufficient for a *single* read step: the value returned by
453 the read is determined by the input memory, and the probabilistic fault only affects the successor
454 memory. We instead use the slightly more general type
455
$$\text{read} : (\text{Loc} \times \text{Memory}) \longrightarrow \text{Dist}(\text{Val} \times \text{Memory})$$

456
457 This is the standard form of a probabilistic state transformer with a return value: it produces
458 a joint distribution over the observed result and the successor state. The more specialised type
459 $\text{Val} \times \text{Dist}(\text{Memory})$ embeds into this one by pairing every successor memory with the same
460 returned value. Using the joint form $\text{Dist}(\text{Val} \times \text{Memory})$ avoids baking this special case into the
461 semantic interface and ensures that read operations share the same monadic shape as the rest of
462 the semantics. This leaves the abstraction general enough to model bit-flip errors, where a physical
463 fault correlates the specific value returned by a noisy read with the resulting successor memory,
464 without changing the semantic type of reads.
465
466
5.1 The fault-kernel contract

467 Fault kernels is our name for the probability kernels that model Rowhammer. The idea behind fault
468 kernels is simple: they describe *what* memory cells may be affected by Rowhammer effects when a
469 location x is accessed and *how*. Figure 4 visualises this intuition. We formalise this intuition with
470 the following predicates.
471
- $\text{adj}(br, locs, x)$ gives the set of all memory locations within the blast radius br of location x.
472
- $\text{pflip}(S, m) \in \text{Dist}(\text{Memory})$ is a probability distribution that is allowed to flip bits in the
473 finite set S of locations in memory m. Locations outside of S must remain unchanged.

474
475 We can now state the modelling assumptions we will use in Section 6 for proving the soundness of
476 physical separation as a Rowhammer defence.

477
- **(F1) Access-triggered.** Faults are sampled exactly at the $\text{read}(x)$- and $\text{write}(x)$-labelled
478 transitions of Section 4, τ-transitions are fault-free, modelled as the Dirac distribution. (F1)
479 is enforced by construction of the rules below, not assumed.
480
- **(F2) Spatial confinement (frame-locality).** Every memory in the support of $\text{pflip}(S, m)$
481 agrees with m outside the victim set S. Within S the corruption may assign arbitrary
482 probabilities to arbitrary, correlated corruptions: the kernel may behave adversarially,
483 subject only to this clause.
484
- **(F3) History-freeness.** The kernel depends only on the current victim set and the current
485 memory: pflip is a function $\mathcal{P}_{fin}(\text{Loc}) \longrightarrow \text{Memory} \longrightarrow \text{Dist}(\text{Memory})$, so a fault cannot
486 depend on the access history, on time, or on hidden hardware state. This is a genuine mod-
487 elling restriction. It excludes, for example, charge accumulation across repeated activations
488 and refresh-cycle effects, generalising to stateful kernels is future work, as discussed in
489 Section 9.

490

- **(F4) Stationarity.** The same kernel governs every access of a run, so fault behaviour does not drift over time. Like (F3), this is encoded by pflip being a single function rather than assumed as a separate axiom.

Clauses (F1-F4), considered in isolation, do not require the location whose access triggers a fault to lie outside its own victim set: the abstract contract permits $x \in \mathrm{adj}(br, locs, x)$. This generality is not needed for Rowhammer which does not allow "self-hammering": the aggressor location cannot itself be a victim. Nevertheless, no additional axiom is required for the executions covered by our main theorem. Program well-formedness gives $x \in locs$ for every program access, and safe physical separation then gives $x \notin \mathrm{adj}(br, locs, x)$ by instantiating protected–victim disjointness with the protected location x itself. Clause (F2) consequently ensures that the fault preserves the accessed location. Thus, the abstract contract is slightly more general than the intended DRAM interpretation, while the non-self-disturbance property required for protected executions is a derived consequence of the theorem hypotheses rather than a separate kernel axiom.

5.2 Physical adequacy and scope of our fault model

Our semantics is not a physically faithful device-level model of the electrical mechanisms underlying Rowhammer, nor is it intended to predict the probability of a bit-flip on a particular DRAM module. Our abstraction can be read as a conservative attacker model for Rowhammer-style disturbance: the bit-flips admissible by our model can be more extreme than those observed in physical DRAM. Therefore, a universal theorem proved for every admissible abstract kernel applies a fortiori to any more constrained concrete device model that refines it.

5.3 The fault model, mechanised

The Lean development renders the contract in three declarations. Separation is an abstract structure: a blast-radius function, a safety predicate, and the single fact the soundness proof uses about them: with physical separation, the victim set of an access to a protected location is disjoint from the protected set.

```
structure SeparationModel where
  adj : Nat → LocSet → Loc → LocSet
  isSafe : Nat → LocSet → Prop
  /-- Safe layouts: the victim set of an access to a protected row is
      disjoint from the protected set. -/
  safe_disjoint : ∀ {br : Nat} {locs : LocSet} {x : Loc},
    isSafe br locs → x ∈ locs → ∀ y ∈ locs, y ∉ adj br locs x
```

Our abstract fault contract does not build the row structure of DRAM directly into the programming language semantics. Instead, a separation model supplies an abstract victim-set function $\mathrm{adj}(br, locs, x)$, identifying the locations that may be corrupted by an access to x. The theory does not require this function to be derived from a metric, or even to be symmetric. This is both simpler and more general than fixing a particular DRAM organisation.

The mechanisation includes a concrete distance-based instance in which locations are natural numbers and the victim set is determined by numerical distance. This instance is primarily a simple witness that the abstract contract is non-vacuous: numerical address distance should not itself be identified with physical DRAM distance unless locations have already been interpreted as physical row identifiers. A row-based model is a special case of the abstract interface: locations may be taken to denote rows directly, or the victim-set function may be defined through a mapping from program-visible locations to DRAM banks and rows.

540 A fault kernel is a function from victim sets and memories to distributions over memories. The
541 *type* already encodes (F3) and (F4): the kernel sees neither history nor time.
542
543
544

```
abbrev ProbRHFlip (_sep : SeparationModel) :=
  LocSet → Memory → Dist Memory
```

545 Clause (F2) is the sole behavioural assumption, stated as a typeclass:
546
547
548
549

```
class HasProbFlipFrame (sep : SeparationModel) (f : ProbRHFlip sep) : Prop where
  outside_unchanged : ∀ (S : LocSet) (m m' : Memory) (x : Loc),
    m' ∈ (f S m).support → x ∉ S → m' x = m x
```

550 Clause (F1) is enforced by construction of the transition rules below.
551 The contract is not vacuous, and not satisfied only by fault-free kernels. The identity kernel
552 `noFlip` (every fault is a no-op) satisfies it and embeds the ordinary semantics (Section 6.2). At the
553 other extreme, so does the worst-case kernel that deterministically corrupts the *entire* victim set at
554 every access:
555
556
557

```
noncomputable def corruptAll (sep : SeparationModel) : ProbRHFlip sep :=
  fun S m => dirac (fun x => if x ∈ S then m x + 1 else m x)
```

558 The mechanisation proves `corruptAll` frame-local and genuinely faulting (every outcome differs
559 from the pre-fault memory at every victim location), and exhibits concrete physical separation of
560 the distance model together with a well-formed program over it, so every hypothesis of the goal
561 theorem is satisfiable simultaneously with a kernel that really flips bits.
562

563 ### 5.4 Probabilistic transitions

564 The transition judgements
565

566 $$\langle e, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu, \qquad \langle b, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu, \qquad \langle P, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mu$$

567 return a *distribution* μ over successor configurations rather than a single successor, and are otherwise
568 the rules of Section 4 with the same left-to-right, non-short-circuit evaluation order. Exactly two
569 rules sample the kernel, matching (F1). A variable read first obtains the value $m(x)$ and then samples
570 the fault triggered by that access. The sampled fault therefore cannot retroactively change the
571 value returned by the current read, but it may change the memory seen by later reads:
572
573

574 $$\frac{}{\langle x, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \big(\mathbf{let}\ m' \leftarrow \mathsf{pflip}\big(\mathsf{adj}(br, locs, x), m\big)\ \mathbf{in}\ \delta\big(\langle m(x), m'\rangle\big)\big)} \quad \text{(RH-A-READ)}$$

575
576 Dually, once the right-hand side of an assignment is a value, the assignment writes it and samples
577 the fault triggered by the write, on the *updated* memory:
578

579 $$\frac{}{\langle x := v, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \big(\mathbf{let}\ m' \leftarrow \mathsf{pflip}\big(\mathsf{adj}(br, locs, x), m[x \mapsto v]\big)\ \mathbf{in}\ \delta\big(\langle \mathsf{skip}, m'\rangle\big)\big)} \quad \text{(RH-P-ASSIGN)}$$

580
581 Under the well-formedness and physical separation hypotheses of the main theorem, the accessed
582 location lies outside its victim set. Hence frame-locality preserves the stored value at a read location
583 and preserves the value just installed by an assignment. This fact is derived from physical separation.
584 It is not an additional assumption on the fault kernel. Every other rule is the Dirac or functorial
585 lifting of its deterministic counterpart. The full rule sets for all three judgements are in Section B.
586 There is no transition from $\langle \mathsf{skip}, m\rangle$. The rules are syntax-directed, so every non-terminal program
587 configuration determines a unique one-step distribution: the fault randomises the memory contents,
588

never which access happens next. The semantics is therefore probabilistic but not nondeterministic. We write $\mathsf{step}^{\mathsf{RH}}_{br,locs}(\langle P, m\rangle)$ for the program-level distribution.

5.5 Finite probabilistic executions

For iteration, define an absorbing extension of the one-step kernel by

- $\widehat{\mathsf{step}}^{\mathsf{RH}}_{br,locs}\big(\langle \mathsf{skip}, m\rangle\big) = \delta\big(\langle \mathsf{skip}, m\rangle\big)$,
- $\widehat{\mathsf{step}}^{\mathsf{RH}}_{br,locs}\big(\langle P, m\rangle\big) = \mathsf{step}^{\mathsf{RH}}_{br,locs}\big(\langle P, m\rangle\big)$ assuming that $P \neq \mathsf{skip}$,

and the distribution after exactly n applications of the absorbing kernel by

- $\mathsf{run}^{0}_{br,locs}(C) = \delta(C)$,
- $\mathsf{run}^{n+1}_{br,locs}(C) = \textbf{let } C' \leftarrow \widehat{\mathsf{step}}^{\mathsf{RH}}_{br,locs}(C) \textbf{ in } \mathsf{run}^{n}_{br,locs}(C')$.

The absorbing extension exists only to define finite runs. The operational semantics still has no transition from a terminal configuration. Because expression reductions are lifted through assignments and conditionals one step at a time, the index n counts genuine small-step transitions. A read or write together with its immediately triggered fault sample counts as one probabilistic transition.

In the mechanisation the absorbing kernel is a total *function* (`stepRHHat`), and the displayed judgement rules are inductive relations, that the two agree on non-terminal configurations (the kernel computes exactly the judgement's distribution, and the judgement's distribution is unique) is proved (`pRHStep_iff_stepRHHat`), so results stated over the runs below are statements about the displayed rules.

Because the rules are syntax-directed, *which* access a configuration performs next is a function of the configuration alone. The n-step run therefore extends canonically to a trace-carrying run $\mathsf{run}^{n}_{br,locs,\mathrm{tr}}$ that accumulates the label sequence of Section 4 alongside the configuration, assigning to every outcome its access trace. Absorbing steps at terminal configurations contribute τ, which the access trace erases. The goal theorem's trace agreement (Section 6.2) is stated over this run.

5.6 Mixed termination

Without mitigations, Rowhammer faults may change not only final values but also whether a program terminates. Indeed, a single initial configuration may induce a probabilistic mixture of terminating and non-terminating executions. Consider the following program:

$$\mathsf{while}\ x > 17\ \mathsf{do}\big(y := z + 1; x := x - 1\big).$$

In the ordinary semantics, every iteration decreases x by one, so the program terminates. Now suppose a Rowhammer bit-flip triggered by writing y may corrupt x. As a concrete illustration, consider a kernel that leaves x unchanged with probability $1 - p$, changes x to $x + 2$ with probability $0 < p < 1$ after the write to y, and otherwise leaves the locations relevant to the example unchanged. Hence the program terminates with positive probability and runs forever with positive probability. This example is deliberately schematic: it demonstrates behaviour permitted by the abstract fault contract, not a quantitative claim about a particular DRAM device.

For sufficiently many execution steps, some probability mass may already be on configurations $\langle \mathsf{skip}, m'\rangle$, while the remaining mass is on non-terminal configurations. The exact probabilities of non-termination are available only as limits. We do not mechanise these limiting probabilities in the present development. The present finite step probability-mass-function approach is sufficient for the theorem we establish in later sections. An explicit account of divergence is possible, but would require additional semantic infrastructure.

638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686

5.7 Language independence

The language-parametricity of the model comes from the small semantic signature required by fault kernels. To instantiate the interface for a language, one must identify memories, locations, and the semantic clauses that perform memory actions. The ordinary clauses for those actions are then factored into an intended logical access followed by a call to the fault kernel determined by the accessed location. All other language constructs interact with Rowhammer only through the memory actions they eventually perform. Consequently, the interface can be applied directly to explicitly imperative languages and indirectly to non-imperative languages by applying it to an intermediate language, runtime semantics, or machine semantics in which memory actions are explicit.

The language-independence claim is also supported by recent work on reusable architecture semantics. ArchSem [31], for example, factors instruction set architecture semantics through a small algebraic-effect interface between ISA descriptions and memory models. In that interface, instruction semantics does not directly manipulate a monolithic global machine state. Instead, it emits outcomes such as memory reads and writes, barriers, cache operations, TLB operations, and exceptions. The memory-relevant part of the interface is particularly close to what our model requires: a memory request records an access kind, address, address space, size, and tag information, and the outcome interface contains distinguished memory-read and memory-write effects. Our Rowhammer interface can be understood as an analogous factorisation, but at a different semantic boundary. ArchSem factors ISA behaviour from the memory and concurrency model. We factor the physical fault mechanism from the programming language or machine semantics. To instantiate our model for a new language or architecture, one need not rebuild the Rowhammer theory around that language's syntax. It suffices to identify the semantic events that correspond to memory accesses, compute the set of physical locations exposed by each such access, and interpret the access through the probabilistic fault kernel. In a WHILE language these events are source-level variable reads and writes, but in an architecture semantics such as ArchSem they would correspond instead to memory outcomes such as `MemRead` and `MemWrite`. This is the precise sense in which the interface is substantially independent of the target programming language: the language-specific part is the production of memory-access events, while the Rowhammer-specific part is a reusable interpretation of those events as possible probabilistic disturbances of physical memory.

6 Soundness of physical separation

This section addresses our RQ4 and proves the paper's first main theorem: under the fault kernel contract of Section 5.1, physical separation eliminates Rowhammer faults from being observable in the results of program execution. We first state the goal informally, then define the projection being compared, give the formal statement together with its mechanised form, walk through every hypothesis, and sketch the proof.

The idea is simple: given a program P such that all variables *locs* occurring in P are physically separated, meaning that they are all outside each other's blast radius. Then the probabilistic semantics of $\langle P, m \rangle$ when restricted to *locs* is the Dirac distribution of the deterministic run of $\langle P, m \rangle$. All formal ingredients are already defined, except comparing a deterministic with a probabilistic run of $\langle P, m \rangle$.

6.1 The protected view

We call any element of $(\mathrm{Prog} \times \mathrm{Memory}) \times \mathrm{Acc}^*$ a *configuration with labels*. Write $\mathrm{run}^n_{\mathrm{det,tr}}(C) \in (\mathrm{Prog} \times \mathrm{Memory}) \times \mathrm{Acc}^*$ for the deterministic n-step absorbing run with its accumulated label sequence (terminal configurations step to themselves, contributing τ, Section 4), and $\mathrm{run}^n_{br,locs,\mathrm{tr}}(C) \in$

687 $\mathrm{Dist}\big((\mathrm{Prog} \times \mathrm{Memory}) \times \mathrm{Acc}^*\big)$ for the trace-carrying Rowhammer run of Section 5.5. The *protected*
688 *view* of a configuration with labels is
689
$$\pi_{locs}(\langle P, m\rangle, s) \;\triangleq\; \big(P,\ m{\restriction}locs,\ \mathrm{acc}(s)\big),$$
690
691 the residual program, the restriction of the memory to the protected locations, and the access trace.
692 In Lean, the restriction is a total map on the subtype of protected locations, and the access trace
693 erases τ:
694
```
def protView (locs : LocSet) :
    (Prog × Memory) × List Access →
      Prog × ({x : Loc // x ∈ locs} → Val) × List Access :=
  fun r => (r.1.1, fun x => r.1.2 x.1, accessTrace r.2)
```
695
696
697
698
699 These three components are exactly what the program's execution can depend on. What the
700 projection discards, unprotected DRAM subject to Rowhammer outside *locs*, is exactly what a
701 physical separation defence sacrifices to the fault, and a well-formed program never reads it.
702
6.2 Formal statement
703
704 We are now in a position to state the soundness theorem, both informally and with extracted Lean.
705
706 Theorem 6.1 (Soundness of physical separation, formal). *Let* pflip *satisfy the fault kernel*
707 *contract (Section 5.1), let* $\mathrm{safe}(br, locs)$ *hold, and let* $\mathrm{wf}_{locs}(P)$*. Then for every initial memory m*
708 *and every* $n \in \mathbb{N}$*,*
709
$$\mathrm{Dist}.\mathit{map}\ \pi_{locs}\ \big(\mathrm{run}^n_{br,locs,\mathrm{tr}}(\langle P, m\rangle)\big) \;=\; \delta\Big(\pi_{locs}\big(\mathrm{run}^n_{\mathrm{det},\mathrm{tr}}(\langle P, m\rangle)\big)\Big).$$
710
711
712
713 Equivalently, by the support characterisation of the point mass (Section 3): every outcome in the
714 support of the n-step Rowhammer run has the residual program, the protected memory, and the
715 access trace of the n-step ordinary run. The mechanised statement quantifies in addition over every
716 separation geometry:
717
```
def PhysicalSeparationSound : Prop :=
  ∀ (sep : SeparationModel) (f : ProbRHFlip sep),
    HasProbFlipFrame sep f →
    ∀ (br : Nat) (locs : LocSet),
      sep.isSafe br locs →
      ∀ (p : Prog), p.WellFormed locs →
        ∀ (m : Memory) (n : Nat),
          Dist.mapD (protView locs)
              (runRHTrace sep f br locs n (p, m))
            = Dist.dirac (protView locs (runDTrace n (p, m)))
```
718
719
720
721
722
723
724
725
726
727 It might be of interest to see that all hypotheses in the mechanised statement are needed, and we
728 walk through them in order next.
729
- **`sep : SeparationModel`.** The theorem is universal over abstract separation geometries
730 (Section 5.3): a blast-radius function adj, a safety predicate safe, and the disjointness fact
731 `safe_disjoint` connecting them. The proof uses nothing else about the geometry, no
732 row arithmetic, no linear layout, so the result applies to any placement scheme whose
733 safety condition implies the disjointness fact, with the distance model of Section 5.3 as the
734 motivating instance.
735

736 • **f : ProbRHFlip sep.** The fault kernel. Its type, LocSet → Memory → Dist(Memory), encodes contract clauses (F3) and (F4): the kernel can consult only the current victim set and memory, and the same kernel governs every access. Nothing about the distribution itself is fixed.
740 • **HasProbFlipFrame sep f.** Contract clause (F2), spatial confinement is *the sole behavioural hypothesis*. Every memory in the support of $\mathrm{pflip}(S, m)$ agrees with m outside S. Dropping it makes the theorem false immediately: a kernel that flips a protected location on some access satisfies everything else, and its Rowhammer run visibly diverges from the ordinary one on protected memory.
745 • **br and locs.** The blast radius and the protected set. Both are abstract parameters threaded through adj where *locs* is a *finite* set of locations (a Finset), the memory locations the program uses for computation.
748 • **sep.isSafe br locs.** Physical separation, the condition the defence establishes, e.g., by guard rows. Through `safe_disjoint` it yields the only fact the proof needs: the victim set of an access to a protected location is disjoint from the protected set. Without it the theorem fails even for kernels obeying the whole contract: accessing one protected location may corrupt another protected location inside its blast radius (Theorem 2.2).
753 • **p.WellFormed locs.** Every location the program reads or writes is in *locs* (Section 4). This clause makes separation compose with execution: it guarantees that every access the run performs is an access to a *protected* location, so `safe_disjoint` applies to every victim set the run generates. Without it, an access to some $z \notin locs$ triggers a fault on $\mathrm{adj}(br, locs, z)$, about which $\mathrm{safe}(br, locs)$ says nothing, the blast radius of an unprotected location may well intersect *locs* (Theorem 2.3).
759 • **m and n.** The initial memory and the number of steps, both universally quantified. Quantifying over every finite n is the relational-safety reading of Theorem 6.1: it covers terminating and diverging programs alike, with no limit construction.

762 It may also be interesting to see what is *not* assumed: no bound on flip probabilities (faults may be certain), no independence between locations or between fault events, no unbiasedness, no particular distribution. Within the blast radius the kernel may be adversarial. The guarantee is distribution-free: it follows from spatial confinement and separation alone.
766
767
6.3 Corollaries

768 Two clauses of Theorem 6.1 are worth recording as standalone consequences of the point mass equation.
770
771 Corollary 6.2 (Exact termination correspondence). *Under the hypotheses of Theorem 6.1, if* $\langle P, m\rangle \overset{s}{\underset{p}{\Rightarrow}} \langle \mathsf{skip}, m'\rangle$*, then every configuration in the support of* $\mathsf{run}^{|s|}_{br,locs}(\langle P, m\rangle)$ *is terminal and agrees with* m' *on locs: the Rowhammer execution terminates after exactly the same number of abstract-machine steps, with the same protected final memory (and, by Theorem 6.1, the same access trace).*
776

```
def TerminationCorrespondence : Prop :=
  ∀ (sep : SeparationModel) (f : ProbRHFlip sep),
    HasProbFlipFrame sep f →
    ∀ (br : Nat) (locs : LocSet),
      sep.isSafe br locs →
      ∀ (p : Prog), p.WellFormed locs →
        ∀ (m m' : Memory) (s : List Access),
```
784

785
786
787

```
        PTrace (p, m) s (Prog.skip, m') →
        ∀ c ∈ (runRH sep f br locs s.length (p, m)).support,
          c.1 = Prog.skip ∧ ∀ x ∈ locs, c.2 x = m' x
```

788
789
790
791
792

Corollary 6.3 (Divergence preservation). *Under the hypotheses of Theorem 6.1, if $\langle P, m\rangle \Uparrow$, then for every n every configuration in the support of* $\mathsf{run}^{n}_{br,locs}(\langle P, m\rangle)$ *is non-terminal (and agrees with the ordinary run on the protected state).*

793
794
795
796
797
798
799
800
801
802
803

```
def DivergencePreservation : Prop :=
  ∀ (sep : SeparationModel) (f : ProbRHFlip sep),
    HasProbFlipFrame sep f →
    ∀ (br : Nat) (locs : LocSet),
      sep.isSafe br locs →
      ∀ (p : Prog), p.WellFormed locs →
        ∀ (m : Memory), Diverges p m →
          ∀ (n : Nat),
            ∀ c ∈ (runRH sep f br locs n (p, m)).support,
              c.1 ≠ Prog.skip
```

804
805
806

6.4 Proof sketch

807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833

The proof is a probabilistic simulation whose invariant is *parametric in the preserved region*: a predicate keep on locations, instantiated only at the very end with membership in *locs*. It proceeds in three main steps.

One-step invariant (`stepRHHat_agree`). Suppose the Rowhammer and ordinary configurations share the same residual program and their memories agree on keep, and suppose (i) every location the program uses satisfies keep, and (ii) every fault sample triggered by an access the program can perform preserves keep-locations. Then every outcome of one application of the Rowhammer kernel again shares the residual program of the ordinary step and agrees with it on keep. The residual program is preserved exactly, not just up to keep, because reads return the same values on keep-agreeing memories, so both machines make the same control decisions.

Iteration (`runRHTrace_protected`). The one-step invariant is iterated over the step count n. Because the rules are syntax-directed, the label of the next transition is a function of the residual program alone. Preserving the residual program exactly therefore forces both runs to emit the *same label sequence*, which gives the access-trace clause for free. Well-formedness is preserved along the run, so clauses (i) and (ii) remain available at every step.

Instantiation and conversion. Take $\mathsf{keep}(\ell) \triangleq \ell \in locs$. Clause (i) is program well-formedness. Clause (ii) is exactly frame-locality (F2) composed with `safe_disjoint`: an access to $x \in locs$ triggers a fault confined to $\mathsf{adj}(br, locs, x)$, which is disjoint from *locs*, so every sample fixes every protected location. The invariant then says that every outcome in the support of $\mathsf{run}^{n}_{br,locs,\mathrm{tr}}$ has the protected view of $\mathsf{run}^{n}_{\mathrm{det,tr}}$. The support characterisation of Section 3 converts this into the point mass equation. The corollaries follow from the central equation together with determinism, progress, and the trace realisation lemma of the ordinary semantics (Section A). The no-fault embedding is by computation.

We can now show that the deterministic semantics is a special case of the probabilistic semantics. This gives us confidence in the correctness of the probabilistic semantics.

834 835 836 837 838 839 840

Theorem 6.4 (No-fault embedding). *Let sep be any separation model, let br be any blast-radius parameter, let locs be any protected set, and let c be any initial configuration. If the Rowhammer fault kernel is the identity kernel* noFlip(sep)*, then n steps with the Rowhammer semantics is exactly the Dirac distribution concentrated on the n-step deterministic trace run: This requires neither safe physical separation nor program well-formedness.*

841 This theorem is mechanised in Lean as `runRHTrace_noFlip`.

842 843

7 Information flow security under Rowhammer

844 We now address our RQ5 whether the well-developed and widely used theory of information flow
845 [10, 34] can be lifted to our Rowhammer model. We do this by defining *relative non-interference*
846 and prove that physical separation guarantees relative non-interference. Relative non-interference
847 means that any information leak that happens with Rowhammer also happens without Rowham-
848 mer[2]. This is of interest, as Rowhammer is usually presented as an integrity failure: an access to
849 one physical row may corrupt data stored in another. But the same mechanism can also create a
850 confidentiality failure: a Rowhammer induced flip in a high location may alter a branch condition,
851 loop condition, or value that is later copied to low memory. Moreover, a fault in a low location
852 may directly change an attacker-visible result. The theory of information flow is tailor-made for
853 studying such phenomena, and provides a natural test of whether physical separation removes the
854 semantic consequences of Rowhammer rather than merely making faults less likely.

855

856 *Low equivalence and non-interference.* In order to explain relative non-interference in detail,
857 we need to sketch the key ideas behind information flow security, see [10, 34] for details. Fix a
858 policy that classifies every memory location as either L or H. Two memories are *low equivalent*
859 when they agree at every L location (they may differ arbitrarily at H locations). A program is
860 *non-interfering* when executions from low-equivalent initial memories end with memories that are
861 also low-equivalent (we discuss termination later). Non-interference is therefore a *hyperproperty*
862 [3]: it relates pairs of executions rather than inspecting one execution in isolation. The ordinary
863 semantics is deterministic, so indistinguishability means equality of the two observations. The
864 Rowhammer semantics maps an initial memory to a probability distribution over executions.
865 Probability-sensitive non-interference consequently requires equality of the induced distributions
866 over low observations.

867 *What physical separation can and cannot guarantee.* The physical separation discussed in previous
868 sections must protect every location used by the program, not only its low locations. Protecting
869 only low variables is insufficient: corrupting a high variable may change control flow and thereby
870 alter a later low write, low access, or termination behaviour. The separation condition used in this
871 paper therefore protects the complete set of locations over which the program is well formed. That
872 means we can only give a *no-new-leaks* guarantee: physical separation does not make a program
873 secure that is insecure in the deterministic semantics! For example,

874

$$L := H$$

875

876 explicitly leaks a high value into a low variable and is therefore interfering in both semantics.
877 Physical separation only ensures that the Rowhammer semantics has exactly the same protected
878 observations as the ordinary semantics. Hence, with physical separation, a Rowhammer information
879 flow counterexample is only a counterexample that was already present without faults. Without

880 [2]Cf. relative consistency results like Gödel's proof that classical logic does not introduce unsoundness not already present
881 in intuitionistic logic [12], or the Gödel / Cohen results that ZFC set theory is relatively consistent over ZF [4, 9].
882

883 separation this reasoning fails. If an access to a program's location can disturb another location, a
884 frame-local fault kernel may change a low variable or a high value controlling future low behaviour.
885 The fault kernel may assign arbitrary probabilities to such corruptions inside the victim set. Safe
886 separation rules out this channel structurally: the victim set generated by an access to a protected
887 location is disjoint from every protected location.
888
889
8 Mechanised observation-parametric non-interference

890 This section formalises and proves our relative non-interference results. The Rowhammer semantics
891 of previous sections remains unchanged. Our central design choice is that every observer consumes
892 the protected view already covered by the physical separation theorem. Thus the security results
893 are consequences of semantic collapse proven already, rather than a second operational simulation
894 argument. This gives us additional confidence in our mathematical rendering of Rowhammer as a
895 programming language feature.
896
897
8.1 Formal development

898 *Observation-parametric security.* There is no single universally appropriate notion of low obser-
899 vation (e.g., is termination low observable). The mechanised development therefore parameterises
900 non-interference by a function on the existing protected view, which contains the residual program,
901 protected memory, and access trace. We define four concrete observers:
902
- Protected low memory only
903
- The address trace of reads and writes to low locations
904
- Terminal status, protected low memory, and the low access trace
905
- Protected low memory and the low access trace, but not terminal status

906
907 The third observer is progress-sensitive because observations are compared at every finite number
908 of steps: it can reveal the exact abstract-machine step at which the residual program first becomes
909 `skip`. Omitting the terminal-status bit does not by itself yield classical termination-insensitive
910 non-interference, since low memory or low accesses observed at successive steps may still reveal
911 progress. The mechanisation therefore makes no claim that its fourth observer is a complete
912 formalisation of classical termination-insensitive non-interference.

913 *Security policies and low equivalence, protected views and observations.* Let $\mathsf{Sec} \triangleq \{\mathsf{L}, \mathsf{H}\}$ and let
914 a security policy be a function $\gamma : \mathsf{Loc} \to \mathsf{Sec}$. The low and high location subtypes are $L_\gamma \triangleq \{x :$
915 $\mathsf{Loc} \mid \gamma(x) = \mathsf{L}\}$, and $H_\gamma \triangleq \{x : \mathsf{Loc} \mid \gamma(x) = \mathsf{H}\}$. For $m \in \mathsf{Memory}$, the two restricted views are
916
- $\mathsf{lowView}_\gamma(m)(x) \triangleq m(x)$ assuming $(x \in L_\gamma)$
917
- $\mathsf{highView}_\gamma(m)(x) \triangleq m(x)$ assuming $x \in H_\gamma$.

918
919 Two memories are low equivalent when

920
$$m_1 \approx_\gamma m_2 \quad \Longleftrightarrow \quad \forall x : \mathsf{Loc}.\ \gamma(x) = \mathsf{L} \Longrightarrow m_1(x) = m_2(x).$$

921
922 The mechanisation proves that this pointwise definition is equivalent to

923
$$\mathsf{lowView}_\gamma(m_1) = \mathsf{lowView}_\gamma(m_2),$$

924
925 and proves reflexivity, symmetry, and transitivity. Let Access contain labels $\mathsf{read}(x)$, $\mathsf{write}(x)$, and
926 τ. The function accessTrace filters out τ. For the finite protected set *locs*, define

927
$$\mathsf{PView}(locs) \triangleq \mathsf{Prog} \times \big(\{x : \mathsf{Loc} \mid x \in locs\} \to \mathsf{Val}\big) \times \mathsf{List}(\mathsf{Access}).$$

928
929 The protected projection is

930
$$\mathsf{protView}_{locs}((Q, m), s) \triangleq \big(Q,\ m|_{locs},\ \mathsf{accessTrace}(s)\big).$$

931

It retains exactly the three components covered by physical semantic collapse: residual syntax, protected memory, and the access trace. The deterministic protected view at step count n is

$$\mathsf{DView}^{n}_{locs}(P, m) \triangleq \mathsf{protView}_{locs}\big(\mathsf{runDTrace}^{n}(P, m)\big).$$

The Rowhammer protected-view distribution is

$$\mathsf{RHView}^{n}_{sep,f,br,locs}(P, m) \triangleq \mathsf{Dist.map}\ \mathsf{protView}_{locs}\big(\mathsf{runRHTrace}^{n}_{sep,f,br,locs}(P, m)\big).$$

Non-interference definitions. Recall that we do not hard-code just one specific notion of how the program execution is observed. Instead, let Ω be an observation type. A protected observer is a function

$$O : \mathsf{PView}(locs) \to \Omega.$$

The two observations used in the definitions below are

$$\mathsf{ObsOrd}^{n}_{locs,O}(P, m) \triangleq O\big(\mathsf{DView}^{n}_{locs}(P, m)\big)$$

and

$$\mathsf{ObsRH}^{n}_{sep,f,br,locs,O}(P, m) \triangleq \mathsf{Dist.map}\ O\big(\mathsf{RHView}^{n}_{sep,f,br,locs}(P, m)\big).$$

See the Lean definitions `rowhammerProtectedView` and `rowhammerObservation` that match this two-stage definition. We can now define our notion of observer parametric non-interference.

Definition 8.1 (Ordinary step bound non-interference). The predicate $\mathsf{OrdinaryNI}(\gamma, locs, O, P)$ holds when

$$\forall n \in \mathbb{N}.\ \forall m_1, m_2 \in \mathsf{Memory}. \quad m_1 \approx_\gamma m_2 \implies \mathsf{ObsOrd}^{n}_{locs,O}(P, m_1) = \mathsf{ObsOrd}^{n}_{locs,O}(P, m_2).$$

This is the Lean definition `OrdinaryNI`.

Definition 8.2 (Rowhammer step bound non-interference). For a fixed separation model sep, fault kernel f, blast radius br, and protected set $locs$, the predicate $\mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P)$ holds when

$$\forall n \in \mathbb{N}.\ \forall m_1, m_2 \in \mathsf{Memory}. \quad m_1 \approx_\gamma m_2 \implies \mathsf{ObsRH}^{n}_{sep,f,br,locs,O}(P, m_1) = \mathsf{ObsRH}^{n}_{sep,f,br,locs,O}(P, m_2).$$

The equality is equality of complete finite distributions. The same fixed kernel f is used in both runs. This is the Lean definition `RowhammerNI`.

Definition 8.3 (Robust Rowhammer non-interference). The predicate $\mathsf{RobustRowhammerNI}(\gamma, sep, br, locs, O, P$ holds when

$$\forall f.\ \mathsf{HasProbFlipFrame}(sep, f) \implies \mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P).$$

Safety of the layout and well-formedness of P are deliberately not fields of this definition, they are hypotheses of the preservation theorem. This matches the Lean definition `RobustRowhammerNI`.

For convenience, the mechanisation also defines

$$\begin{aligned}\mathsf{Admissible}(sep, f, br, locs, P) \quad \iff \quad & \mathsf{HasProbFlipFrame}(sep, f) \\ & \wedge\ sep.\mathsf{isSafe}(br, locs) \\ & \wedge\ P.\mathsf{WellFormed}(locs).\end{aligned}$$

The checked theorem statements below take these three assumptions as separate arguments, exactly as their Lean counterparts do. We can now define the four concrete checked observers, promised above. Towards that aim, let

$$L_{\gamma,locs} \triangleq \{x : \mathsf{Loc} \mid x \in locs \wedge \gamma(x) = \mathsf{L}\}.$$

981
The development defines the following protected observers.

- **Protected-low-memory observer.** It hides residual syntax, terminal status, and the access trace:
$$O^{\mathsf{mem}}_{\gamma,locs}(Q, \bar{m}, s) \triangleq \bar{m}|_{L_{\gamma,locs}}.$$
- **Low-access observer.** Since labels contain addresses but not transferred values, this observer captures address-trace leakage rather than value-labelled I/O leakage.
$$O^{\mathsf{access}}_{\gamma}(Q, \bar{m}, s) \triangleq \mathsf{lowAccessTrace}_{\gamma}(s),$$
where the filter retains $\mathsf{read}(x)$ and $\mathsf{write}(x)$ exactly when $\gamma(x) = \mathsf{L}$.
- **Progress-sensitive observer.** Because the definition compares every finite run, this observer reveals exact abstract-machine progress, including the first step-count at which termination is visible.
$$O^{\mathsf{progress}}_{\gamma,locs}(Q, \bar{m}, s) \triangleq \big(\mathsf{terminal}(Q), \bar{m}|_{L_{\gamma,locs}}, \mathsf{lowAccessTrace}_{\gamma}(s)\big),$$
where $\mathsf{terminal}(Q)$ is the Boolean that is true exactly for $Q = \mathsf{skip}$.
- **Terminal-status-hidden observer.** This omits the terminal-status bit but is not claimed to formalise classical termination-insensitive non-interference: low memory or low accesses at fixed step count may still reveal progress.
$$O^{\mathsf{hidden}}_{\gamma,locs}(Q, \bar{m}, s) \triangleq \big(\bar{m}|_{L_{\gamma,locs}}, \mathsf{lowAccessTrace}_{\gamma}(s)\big).$$

Checked theorem statements. The following statements are proved in Lean without `sorry` and without new problem-specific axioms.

Theorem 8.4 (Protected semantic collapse). *Let f be frame-local, let $sep.\mathsf{isSafe}(br, locs)$ hold, and let $P.\mathsf{WellFormed}(locs)$ hold. Then, for every memory m and n,*
$$\mathsf{RHView}^{n}_{sep,f,br,locs}(P, m) = \delta\big(\mathsf{DView}^{n}_{locs}(P, m)\big).$$
This is theorem `protected_semantic_collapse`*, a direct restatement of the existing theorem* `physical_separation_sound`.

Theorem 8.5 (Observation collapse). *Under the same frame-locality, safety, and well-formedness assumptions, for every protected observer O, memory m, and step count n,*
$$\mathsf{ObsRH}^{n}_{sep,f,br,locs,O}(P, m) = \delta\big(\mathsf{ObsOrd}^{n}_{locs,O}(P, m)\big).$$
This is theorem `observation_collapse`.

Theorem 8.6 (Non-interference equivalence). *Let f be frame-local, let $sep.\mathsf{isSafe}(br, locs)$ hold, and let $P.\mathsf{WellFormed}(locs)$ hold. Then, for every security policy γ and protected observer O,*
$$\mathsf{OrdinaryNI}(\gamma, locs, O, P) \iff \mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P).$$
This is theorem `noninterference_equivalence`.

Corollary 8.7 (Robust preservation). *If* sep.isSafe$(br, locs)$, P.WellFormed$(locs)$, *and* OrdinaryNI$(\gamma, locs, O, P)$ *hold, then*

$$\mathsf{RobustRowhammerNI}(\gamma, sep, br, locs, O, P).$$

Equivalently, every frame-local kernel f *satisfies*

$$\mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P).$$

This is theorem `robust_noninterference_preservation`.

Corollary 8.8 (Reflection for a fixed kernel). *Under frame-locality, safety, and well-formedness,*

$$\neg\mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P) \implies \neg\mathsf{OrdinaryNI}(\gamma, locs, O, P).$$

This is theorem `rowhammer_interference_reflection`.

Corollary 8.9 (Existential reflection). *Assume* sep.isSafe$(br, locs)$ *and* P.WellFormed$(locs)$. *Then*

$$\big(\exists f.\ \mathsf{HasProbFlipFrame}(sep, f) \land \neg\mathsf{RowhammerNI}(\gamma, sep, f, br, locs, O, P)\big)$$
$$\implies \neg\mathsf{OrdinaryNI}(\gamma, locs, O, P).$$

In the Lean code, this is theorem `exists_rowhammer_interference_reflection`.

8.2 Proof sketches

Structural preservation. The existing lemma `stepDHat_wf` proves that one absorbing deterministic step preserves well-formedness over *locs*. Its proof follows the syntax of the residual program. Expression reduction preserves expression well-formedness. Sequencing preserves the well-formedness of the untouched continuation. A conditional either reduces its guard or selects an already well-formed branch. Unfolding a loop constructs a conditional and sequence from already well-formed components. This lemma is what allows the step bound simulation invariant to be applied recursively to the successor program.

One-step agreement. The generic relation Agree$(keep, m_f, m_d)$ states that the faulty and deterministic memories agree at every location satisfying *keep*. Lemma `stepRHHat_agree` assumes that every location used by the program is kept and that every supported fault outcome preserves all kept locations. If the input memories agree on *keep*, then every supported Rowhammer successor has exactly the same residual program as the deterministic successor and its memory still agrees with the deterministic memory on *keep*. The proof is by structural induction on the program, using corresponding agreement lemmas for arithmetic and Boolean expressions. In a read or write case, frame preservation ensures that the sampled fault cannot alter a kept location. Since values read from kept locations agree, both semantics reduce the same expression, write the same value, select the same branch, and unfold the same control construct.

Finite step bound invariant. Lemma `runRHTrace_protected` iterates one-step agreement by induction on n. For every outcome r in the support of the n-step Rowhammer trace run, it proves three facts:

- The residual program of r equals the deterministic residual program.
- The memories agree on every kept location.
- The complete accumulated label lists are equal.

In the successor case, `stepRHHat_agree` supplies the next related configuration, `stepDHat_wf` supplies well-formedness of its residual program, and the induction hypothesis supplies the remaining step bound.

Instantiation by physical separation. For protected semantic collapse, the proof instantiates $keep(x)$ with $x \in locs$. The premise that faults preserve kept locations is derived by combining frame-locality with safe separation: for an access to $x \in locs$, `safe_disjoint` shows that every $\ell \in locs$ lies outside the victim set, and `outside_unchanged` therefore preserves ℓ in every supported fault outcome. The finite step bound invariant then shows that every supported Rowhammer outcome has the same protected view as the deterministic run. The finite-distribution lemma `mapD_eq_dirac_of_support` turns this support-wise fact into the point mass equation of Theorem 8.4.

Observation collapse. Unfolding the two observation definitions exposes a map of O over the Rowhammer protected-view distribution. Rewriting that distribution with Theorem 8.4 leaves O mapped over a Dirac distribution. The finite-distribution law

$$\mathsf{Dist.map}\ O\ (\delta(v)) = \delta(O(v))$$

gives Theorem 8.5.

Non-interference equivalence and its corollaries. For low-equivalent m_1 and m_2, observation collapse rewrites the two Rowhammer observation distributions to

$$\delta\big(\mathsf{ObsOrd}^n_{locs,O}(P, m_1)\big) \quad \text{and} \quad \delta\big(\mathsf{ObsOrd}^n_{locs,O}(P, m_2)\big).$$

Ordinary non-interference implies equality of the two points and hence congruence of their Dirac distributions. Conversely, equality of the two Dirac distributions implies equality of their points by injectivity of δ. This proves Theorem 8.6. Robust preservation introduces an arbitrary frame-local kernel and applies the forward direction. Fixed-kernel and existential reflection are direct contrapositives of that same direction.

8.3 No-fault sanity results

The checked development also proves, without any safety or well-formedness assumption, that the no-fault kernel embeds deterministic execution exactly:

$$\mathsf{runRHTrace}^n_{sep,\mathsf{noFlip}(sep),br,locs}(c) = \delta\big(\mathsf{runDTrace}^n(c)\big).$$

The proof is an induction on n using the existing one-step no-fault theorem, Dirac bind, and mapping over a Dirac distribution. It yields

$$\mathsf{ObsRH}^n_{sep,\mathsf{noFlip}(sep),br,locs,O}(P, m) = \delta\big(\mathsf{ObsOrd}^n_{locs,O}(P, m)\big)$$

and therefore

$$\mathsf{OrdinaryNI}(\gamma, locs, O, P) \quad \Longleftrightarrow \quad \mathsf{RowhammerNI}(\gamma, sep, \mathsf{noFlip}(sep), br, locs, O, P).$$

In Lean they are `runRHTrace_noFlip`, `noFault_observation`, and `ordinaryNI_iff_noFaultNI`.

9 Conclusion

This paper has given an abstract, compositional operational semantics for a representative programming language in the presence of Rowhammer. The semantics treats Rowhammer as a probabilistic effect on memory accesses, rather than as a direct model of DRAM organisation or semiconductor physics. The no-fault embedding theorem, Theorem 6.4, recovers the ordinary deterministic semantics as the point mass special case of the Rowhammer semantics, answering RQ1. The model isolates the Rowhammer effect as a small generalisation from state transformers to probabilistic state transformers: memory reads and writes are intercepted by a fault kernel, while the rest of the operational semantics is lifted compositionally using the probability monad. This gives the small, conceptually non-intrusive interface sought in RQ2. The interface is also largely independent of

the ambient programming language. The development uses only that programs perform memory reads and writes, and that the ordinary operational semantics can expose those accesses. This is not by itself a formal semantics for every language or machine model, but it gives a clear route for instantiating the same Rowhammer interface in other languages, intermediate representations, or instruction-set semantics. In this sense, the paper affirms RQ3. The semantic-collapse theorem, Theorem 6.1, proves the soundness of physical separation: if every protected program access has a victim set disjoint from the protected locations, then every admissible Rowhammer kernel induces exactly the deterministic protected behaviour. This justifies physical separation as a Rowhammer defence within the abstract fault contract, answering RQ4. Finally, the information flow results show that the abstraction supports standard semantic security reasoning. In particular, Theorems 8.4, 8.5 and 8.6 transport non-interference reasoning from ordinary executions to Rowhammer-affected executions, and back again, under the same physical separation hypotheses. Thus the model does not merely describe faulty executions, it makes Rowhammer amenable to established semantic security frameworks, addressing RQ5.

10 Future work

Refining the Rowhammer model. The fault kernel contract history-freeness (F3) and stationarity (F4) exclude genuinely history-dependent faults such as charge accumulation across repeated activations and refresh-cycle effects. Generalising requires extending the Rowhammer state with an activation history, refresh state, or other hardware state. Our theorems would then quantify over arbitrary stateful kernels satisfying the same protected-region preservation condition, and we expect the parametric-invariant proof architecture to carry over. Moreover, DRAM vendors typically deploy proprietary, undocumented mitigations, most notably Target Row Refresh (TRR), whose opaque nature complicates formal security guarantees. Bridging the gap between our formal models and these black-box defences requires rigorous experimental reverse engineering. Pioneering methodologies such as the U-TRR framework [13], uncover the internal mechanics of TRR by taking data-retention failures as a side channel. Alongside this, complementary characterisation of DRAM's disturbance sensitivities [30] establishes clear empirical boundaries on when such defences succeed or fail, and in turn informs principled mitigations such as BlockHammer [40]. By experimentally exposing the true behaviour and limitations of in-DRAM mitigations, such efforts provide the precise physical parameters and fault models that a formal framework like ours needs in order to deliver end-to-end software security guarantees.

Deploying physical separation. Our soundness theorem is conditional on the data placement satisfying physical separation. Establishing that on modern hardware is nontrivial: DRAM vendors do not publish row geometry, in-DRAM address remapping obscures physical adjacency, and the effective blast radius must be measured per device. Empirical work on learning the Rowhammer parameters of a given device [28] is complementary to our results: it supplies the physical facts that instantiate *br* and safe, while the theorem turns those facts into a semantic guarantee. The separation between the abstract geometry (`SeparationModel`) and the semantics in our mechanisation is designed exactly for this division of labour.

Probabilistic non-termination. A natural extension is to complement the finite step bound semantics with an explicit account of eventual termination and divergence. This could be done either by taking limits of the finite step bound termination probabilities, by developing a sub-probability semantics over final memories, or by constructing a probability measure on infinite execution paths. A fully event-based treatment, in which divergence is identified with a measurable set of infinite executions, may additionally require the construction of an appropriate probability measure on

1177 infinite paths and an adequacy theorem relating that measure to the n-step operational semantics.
1178 Each route introduces nontrivial order-theoretic or measure-theoretic mechanisation.

Connections with Shannon-Scott information. A further direction is to relate the present probabilistic account of Rowhammer to semantic accounts of information based on both Shannon entropy and Scott domains. Shannon's theory of communication provides a natural language for noisy channels and probabilistic leakage [36], while domain-theoretic and topological methods are commonly used in programming language semantics to model partial information and non-terminating computation. Recent work has connected these two perspectives in the setting of information flow and non-interference [14]. Our results compare the distributions of low observations after a fixed number of execution steps. It would be interesting to investigate whether this account can be extended to an infinite-horizon semantics in which Rowhammer-induced probabilistic leakage, partial observations, and non-termination are treated in a single Shannon-Scott framework.

Acknowledgements

Chunyan Mu drew our attention to Rowhammer being an open problem from a non-interference angle. We thank David Clark for numerous discussions about information flow and non-interference, Davide Bartolini and Xubin Tan for helping us to understand memory controllers, and Lev Mukhanov for alerting us to the difficulty of reproducing the Rowhammer attacks. Fredrik Dahlqvist helped us better to understand Kozen-style probabilistic semantics of probabilistic programming languages. We thank Yusuke Izawa and Mohammad M. Ahmadpanah for helpful comments, which greatly improved the presentation of this paper.

1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225

1226
References

1227 [1] Anonymous. 2026. Lean proofs for "Mechanised operational semantics of Rowhammer". https://anonymous.4open.science/r/rh_separation_paper_anon_lean-48C7/.
1228
1229 [2] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks Targeting Kernel Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, Canada, 117–130. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser
1230
1231
1232 [3] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. doi:10.3233/JCS-2009-0393
1233
1234 [4] Paul J. Cohen. 1963. The Independence of the Continuum Hypothesis. *Proceedings of the National Academy of Sciences* 50, 6 (1963), 1143–1148. doi:10.1073/pnas.50.6.1143
1235
1236 [5] Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. 2020. *Semantics of Probabilistic Programming: A Gentle Introduction*. Cambridge University Press, 1–42. doi:10.1017/9781108770750.002
1237 [6] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*. Springer, 625–635. doi:10.1007/978-3-030-79876-5_37
1238
1239 [7] Pietro Frigo, Emanuele Vannacci, 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 *2020 IEEE Symposium on Security and Privacy (SP)*. 747–762. doi:10.1109/SP40000.2020.00090
1240
1241 [8] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*. Lecture Notes in Mathematics, Vol. 915. Springer, 68–85. doi:10.1007/BFb0092872
1242
1243 [9] Kurt Gödel. 1940. *The Consistency of the Continuum-Hypothesis*. Number 3 in Annals of Mathematics Studies. Princeton University Press, Princeton, NJ. doi:10.1515/9781400881635
1244
1245 [10] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*. 11–20. doi:10.1109/SP.1982.10014
1246
1247 [11] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 300–321. doi:10.1007/978-3-319-40667-1_15
1248
1249 [12] Kurt Gödel. 1933. On Intuitionistic Arithmetic and Number Theory. *Ergebnisse eines mathematischen Kolloquiums* 4 (1933), 34–38. doi:10.1093/oso/9780195147209.003.0063 English translation by Stefan Bauer-Mengelberg and Jean van Heijenoort. Reprinted in: *Kurt Gödel: Collected Works, Volume I*, Oxford University Press (1986).
1250
1251 [13] Hasan Hassan, Yahya Can Tuğrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*. doi:10.1145/3466752.3480110
1252
1253
1254 [14] Sebastian Hunt, David Sands, and Sandro Stucki. 2023. Reconciling Shannon and Scott with a Lattice of Computable Information. *Proc. ACM Program. Lang.* 7, POPL, Article 68 (jan 2023), 30 pages. doi:10.1145/3571740
1255
1256 [15] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *2022 IEEE Symposium on Security and Privacy (SP)*. 716–734. doi:10.1109/SP46214.2022.9833772
1257
1258 [16] 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 *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. 361–372. doi:10.1109/ISCA.2014.6853210
1259
1260
1261 [17] Andreas Kogler, Jonas Juffinger, Sulaiman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Matt Nissler, and Daniel Gruss. 2022. Half-Double: Hammering From the Next Row Over. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 3807–3824. https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double
1262
1263
1264
1265 [18] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association. https://www.usenix.org/conference/osdi18/presentation/konoth
1266
1267
1268 [19] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350. doi:10.1016/0022-0000(81)90036-2
1269
1270 [20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 695–711. doi:10.1109/SP40000.2020.00020
1271 [21] Francis William Lawvere. 1962. The Category of Probabilistic Mappings: With Applications to Stochastic Processes, Statistics, and Pattern Recognition. (1962). https://lawverearchives.com/wp-content/uploads/2025/07/1962.probmap.pdf Manuscript, 12 pages.
1272
1273
1274

1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323

[22] Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2 ed.). Graduate Texts in Mathematics, Vol. 5. Springer, New York. doi:10.1007/978-1-4757-4721-8

[23] John McCarthy. 1962. Towards a Mathematical Science of Computation. In *Information Processing 1962: Proceedings of IFIP Congress 62*, Cicely M. Popplewell (Ed.). North-Holland, Amsterdam, 21–28. https://www-formal.stanford.edu/jmc/towards.pdf

[24] Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4

[25] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logical and Algebraic Methods in Programming* 60–61 (2004), 195–228. doi:10.1016/j.jlap.2004.03.008

[26] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2020), 1555–1571. doi:10.1109/TCAD.2019.2915318

[27] Amir Naseredini. 2024. *Towards Automatic Analysis of Microarchitectural Attacks.* Ph. D. Dissertation. University of Sussex. https://sussex.figshare.com/articles/thesis/Towards_automatic_analysis_of_microarchitectural_attacks/24610929

[28] Amir Naseredini, Martin Berger, Matteo Sammartino, and Shale Xiong. 2023. ALARM: Active LeArning of Rowhammer Mitigations. In *Proceedings of the 11th International Workshop on Hardware and Architectural Support for Security and Privacy* (Chicago, IL, USA) *(HASP '22)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/3569562.3569563

[29] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-1-84628-692-6

[30] Lois Orosa, A. Giray Yağlıkçı, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. 2021. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*. doi:10.1145/3466752.3480069

[31] Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell. 2026. ArchSem: Reusable Rigorous Semantics of Relaxed Architectures. *Proc. ACM Program. Lang.* 10, POPL (2026), 204–234. doi:10.1145/3776650

[32] Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science, Vol. 2303)*. Springer, 373–393. doi:10.1007/3-540-45931-6_24

[33] Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 154–165. doi:10.1145/503272.503288

[34] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[35] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html Code at https://github.com/google/rowhammer-test.

[36] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x

[37] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. doi:10.1145/3372885.3373824

[38] Cláudio Vasconcelos and António Ravara. 2016. The While language. arXiv:1603.08949 [cs.PL] doi:10.48550/arXiv.1603.08949

[39] Andrew Walker, Sungkwon Lee, and Dafna Beery. 2021. On DRAM Rowhammer and the Physics of Insecurity. *IEEE Transactions on Electron Devices* PP (03 2021), 1–11. doi:10.1109/TED.2021.3060362

[40] A. Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. doi:10.1109/HPCA51647.2021.00037

1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372

Supplementary Appendix: Mechanised high-level operational semantics of Rowhammer

A Material for Section 4: A deterministic small-step operational semantics for a WHILE language

This appendix presents the full small-step operational semantics that were given only partially in Section 4. For more on small-step operational semantics see any textbook on programming language semantics like e.g., [4–6].

Recall that the syntax (arithmetic expressions, Boolean expressions, programs) are given by

$$\begin{aligned} e &::= c \mid x \mid e + e \\ b &::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \\ P &::= \mathsf{skip} \mid x := e \mid P; P \mid \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ b\ \mathsf{do}\ P \end{aligned}$$

We identify arithmetic constants with their denoted values. Thus $v, v_1, v_2 \in \mathsf{Val}$ range over arithmetic values, represented by constants in the expression syntax. Boolean values are $t, t_1, t_2 \in \{\mathsf{true}, \mathsf{false}\}$.

Evaluation is left-to-right. Conjunction is deliberately non-short-circuiting, matching the expression semantics used elsewhere in the paper. Although deterministic expression reduction does not modify memory, memories are included in expression configurations so that the ordinary and faulty transition systems have the same shape.

Transitions carry a *label* recording the memory access they perform:

$$\ell ::= \mathsf{read}(x) \mid \mathsf{write}(x) \mid \tau.$$

A read of variable x is labelled $\mathsf{read}(x)$, the write performed by an assignment to x is labelled $\mathsf{write}(x)$, and every other transition is silent (τ); congruence rules propagate the label of their premise. The labels play no role in the deterministic semantics itself, but they localise the fault points of the probabilistic Rowhammer semantics of Section 5 and make access counting definitional. Single steps are always written with their label; every multi-step notion below is indexed by the *sequence* of labels performed (written with a double arrow), so the labelled judgement is the only transition relation in play.

A.0.1 Arithmetic expressions. An arithmetic-expression configuration has the form $\langle e, m\rangle$. The labelled relation

$$\langle e, m\rangle \xrightarrow{\ell}_a \langle e', m'\rangle$$

is the least relation generated by the following rules. Arithmetic values have no outgoing transition.

$$\frac{}{\langle x, m\rangle \xrightarrow{\mathsf{read}(x)}_a \langle m(x), m\rangle} \quad \text{(A-Read)}$$

$$\frac{\langle e_1, m\rangle \xrightarrow{\ell}_a \langle e_1', m'\rangle}{\langle e_1 + e_2, m\rangle \xrightarrow{\ell}_a \langle e_1' + e_2, m'\rangle} \quad \text{(A-Plus-L)}$$

$$\frac{\langle e_2, m\rangle \xrightarrow{\ell}_a \langle e_2', m'\rangle}{\langle v_1 + e_2, m\rangle \xrightarrow{\ell}_a \langle v_1 + e_2', m'\rangle} \quad \text{(A-Plus-R)}$$

$$\frac{v = v_1 + v_2}{\langle v_1 + v_2, m\rangle \xrightarrow{\tau}_a \langle v, m\rangle} \quad \text{(A-Plus)}$$

A.0.2 Boolean expressions. A Boolean-expression configuration has the form $\langle b, m\rangle$. The labelled relation

$$\langle b, m\rangle \xrightarrow{\ell}_b \langle b', m'\rangle$$

is generated by the following rules. The Boolean values true and false have no outgoing transition.

$$\frac{\langle b, m\rangle \xrightarrow{\ell}_b \langle b', m'\rangle}{\langle \neg b, m\rangle \xrightarrow{\ell}_b \langle \neg b', m'\rangle} \quad \text{(B-NOT-STEP)}$$

$$\frac{}{\langle \neg \mathsf{true}, m\rangle \xrightarrow{\tau}_b \langle \mathsf{false}, m\rangle} \quad \text{(B-NOT-TRUE)}$$

$$\frac{}{\langle \neg \mathsf{false}, m\rangle \xrightarrow{\tau}_b \langle \mathsf{true}, m\rangle} \quad \text{(B-NOT-FALSE)}$$

$$\frac{\langle b_1, m\rangle \xrightarrow{\ell}_b \langle b_1', m'\rangle}{\langle b_1 \wedge b_2, m\rangle \xrightarrow{\ell}_b \langle b_1' \wedge b_2, m'\rangle} \quad \text{(B-AND-L)}$$

$$\frac{\langle b_2, m\rangle \xrightarrow{\ell}_b \langle b_2', m'\rangle}{\langle t_1 \wedge b_2, m\rangle \xrightarrow{\ell}_b \langle t_1 \wedge b_2', m'\rangle} \quad \text{(B-AND-R)}$$

$$\frac{t = t_1 \wedge t_2}{\langle t_1 \wedge t_2, m\rangle \xrightarrow{\tau}_b \langle t, m\rangle} \quad \text{(B-AND)}$$

$$\frac{\langle e_1, m\rangle \xrightarrow{\ell}_a \langle e_1', m'\rangle}{\langle e_1 = e_2, m\rangle \xrightarrow{\ell}_b \langle e_1' = e_2, m'\rangle} \quad \text{(B-EQ-L)}$$

$$\frac{\langle e_2, m\rangle \xrightarrow{\ell}_a \langle e_2', m'\rangle}{\langle v_1 = e_2, m\rangle \xrightarrow{\ell}_b \langle v_1 = e_2', m'\rangle} \quad \text{(B-EQ-R)}$$

$$\frac{t = (v_1 = v_2)}{\langle v_1 = v_2, m\rangle \xrightarrow{\tau}_b \langle t, m\rangle} \quad \text{(B-EQ)}$$

A.0.3 Programs. A program configuration has the form $\langle P, m\rangle$. The labelled program transition relation

$$\langle P, m\rangle \xrightarrow{\ell}_p \langle P', m'\rangle$$

is the least relation generated by the following rules.

An assignment first reduces its right-hand side one expression step at a time. Once the expression is a value, the assignment performs the write.

$$\frac{\langle e, m\rangle \xrightarrow{\ell}_a \langle e', m'\rangle}{\langle x := e, m\rangle \xrightarrow{\ell}_p \langle x := e', m'\rangle} \quad \text{(P-ASSIGN-STEP)}$$

$$\frac{}{\langle x := v, m\rangle \xrightarrow{\mathsf{write}(x)}_p \langle \mathsf{skip}, m[x \mapsto v]\rangle} \quad \text{(P-ASSIGN)}$$

Sequential composition reduces its left-hand program until that program has terminated.

1422
1423
1424
1425

$$\frac{\langle P, m\rangle \xrightarrow{\ell}_p \langle P', m'\rangle}{\langle P; Q, m\rangle \xrightarrow{\ell}_p \langle P'; Q, m'\rangle} \quad \text{(P-SEQ-STEP)}$$

1426
1427
1428

$$\frac{}{\langle \mathsf{skip}; Q, m\rangle \xrightarrow{\tau}_p \langle Q, m\rangle} \quad \text{(P-SEQ-SKIP)}$$

1429 A conditional reduces its guard one Boolean-expression step at a time and then selects the
1430 corresponding branch.

1431
1432
1433
1434

$$\frac{\langle b, m\rangle \xrightarrow{\ell}_b \langle b', m'\rangle}{\langle \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q, m\rangle \xrightarrow{\ell}_p \langle \mathsf{if}\ b'\ \mathsf{then}\ P\ \mathsf{else}\ Q, m'\rangle} \quad \text{(P-IF-STEP)}$$

1435
1436
1437

$$\frac{}{\langle \mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q, m\rangle \xrightarrow{\tau}_p \langle P, m\rangle} \quad \text{(P-IF-TRUE)}$$

1438
1439

$$\frac{}{\langle \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q, m\rangle \xrightarrow{\tau}_p \langle Q, m\rangle} \quad \text{(P-IF-FALSE)}$$

1440 A while loop unfolds by one standard small step. Its guard is subsequently reduced by the
1441 conditional rules above.

1442
1443
1444
1445

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ P, m\rangle \xrightarrow{\tau}_p \big\langle \mathsf{if}\ b\ \mathsf{then}\ \big(P; \mathsf{while}\ b\ \mathsf{do}\ P\big)\ \mathsf{else\ skip}, m\big\rangle} \quad \text{(P-WHILE)}$$

1446 There is no transition from $\langle \mathsf{skip}, m\rangle$; these are the terminal program configurations.

1447 *A.0.4 Finite and terminating executions.* A finite execution is indexed by the sequence of labels it
1448 performs. Write
1449

$$\langle P, m\rangle \overset{s}{\underset{p}{\Rightarrow}} \langle P', m'\rangle$$

1450
1451 for $s = \ell_1 \cdots \ell_k$ when there are configurations
1452
1453

$$\langle P, m\rangle = C_0 \xrightarrow{\ell_1}_p C_1 \xrightarrow{\ell_2}_p \cdots \xrightarrow{\ell_k}_p C_k = \langle P', m'\rangle.$$

1454 The length $|s|$ is the number of transitions performed. Because expression reductions are propagated
1455 through assignments and conditionals one step at a time, it counts both expression reductions and
1456 command reductions. The *access trace* $\mathrm{acc}(s)$ is the subsequence of non-τ labels of s, in order; its
1457 length is the number of memory accesses (reads and writes) the run performs. Both are definitions
1458 over the label sequence, not counts of rule occurrences inside derivation trees. The analogous
1459 trace-indexed relations $\overset{s}{\underset{a}{\Longrightarrow}}$ and $\overset{s}{\underset{b}{\Longrightarrow}}$ at the expression levels are defined in the same way.
1460
1461 A program terminates with final memory m', written
1462

$$\langle P, m\rangle \Downarrow m',$$

1463
1464 when $\langle P, m\rangle \overset{s}{\underset{p}{\Rightarrow}} \langle \mathsf{skip}, m'\rangle$ for some label sequence s. It terminates within at most n small steps,
1465 written
1466

$$\langle P, m\rangle \Downarrow_{\leq n} m',$$

1467
1468 when moreover some such s has $|s| \leq n$. A program diverges from m, written
1469

$$\langle P, m\rangle \Uparrow,$$

1470

1471 when for every $n \in \mathbb{N}$ there are a label sequence s with $|s| = n$ and a configuration $\langle P_n, m_n \rangle$ such
1472 that $\langle P, m \rangle \overset{s}{\underset{p}{\Longrightarrow}} \langle P_n, m_n \rangle$. The index n here is a literal bound on the number of small-step transitions.
1473
1474 Sequential composition cannot duplicate or reuse a budget already consumed by its left-hand
1475 program.

1476 *A.0.5 Basic metatheory.* The lemmas of this subsection are standard and proved on paper. The
1477 mechanisation contains those the soundness theorem rests on (determinism and progress in func-
1478 tional form, trace realisation, divergence, and well-formedness preservation). The remaining lemmas
1479 below are stated for completeness of the presentation and are not part of the Lean development.
1480

1481 Lemma A.1 (Values are terminal). *There are no ℓ and C with $\langle v, m \rangle \xrightarrow{\ell}_a C$; none with $\langle t, m \rangle \xrightarrow{\ell}_b$*
1482 *C; and none with $\langle \mathsf{skip}, m \rangle \xrightarrow{\ell}_p C$.*
1483

1484 Lemma A.2 (Determinism). *Each of the labelled relations is a partial function on configurations,*
1485 *including its label: if $C \xrightarrow{\ell_1}_p C_1$ and $C \xrightarrow{\ell_2}_p C_2$ then $\ell_1 = \ell_2$ and $C_1 = C_2$, and likewise for $\xrightarrow{\cdot}_a$ and $\xrightarrow{\cdot}_b$.*

1486 Proof. Rule induction. For each syntactic form at most one rule applies: the congruence rules
1487 require a non-value in the position being reduced (values are terminal by Theorem A.1, so their
1488 premises are otherwise unsatisfiable), and the redex rules require values in all positions. □
1489

1490 Lemma A.3 (Progress). *If e is not a value then $\langle e, m \rangle \xrightarrow{\ell}_a \langle e', m' \rangle$ for some ℓ, e', m'; if b is not a*
1491 *value then $\langle b, m \rangle \xrightarrow{\ell}_b \langle b', m' \rangle$ for some ℓ, b', m'; and if $P \neq \mathsf{skip}$ then $\langle P, m \rangle \xrightarrow{\ell}_p \langle P', m' \rangle$ for some ℓ,*
1492 *P', m'.*
1493

1494 Proof. Induction on the syntax, using totality of memories for A-Read. □

1495 Corollary A.4 (Termination–divergence dichotomy). *For every configuration $\langle P, m \rangle$, exactly*
1496 *one of the following holds: $\langle P, m \rangle \Downarrow m'$ for a unique m', or $\langle P, m \rangle \Uparrow$.*

1497 Proof. By Theorem A.2 the maximal transition sequence from $\langle P, m \rangle$ is unique, and by Theo-
1498 rem A.3 it can stop only at a terminal configuration $\langle \mathsf{skip}, m' \rangle$. If that sequence is finite it yields
1499 $\Downarrow m'$, with m' unique by determinism iterated along the run; if it is infinite, every n is realised and
1500 $\Uparrow$ holds. The two cases exclude each other because a terminated run has no continuation. □
1501

1502 Lemma A.5 (Memory invariance of deterministic expression steps). *If $\langle e, m \rangle \xrightarrow{\ell}_a \langle e', m' \rangle$ or*
1503 *$\langle b, m \rangle \xrightarrow{\ell}_b \langle b', m' \rangle$, then $m' = m$.*
1504

1505 Deterministically, expression-level small steps are thus redundant: only P-Assign modifies the
1506 memory. This lemma is the precise sense in which the fine-grained presentation is overkill for the
1507 ordinary semantics; the Rowhammer refinement of Section B breaks exactly this lemma, a read
1508 may disturb the blast radius of the accessed location, and nothing else in the design.

1509 Lemma A.6 (Expression evaluation terminates, with explicit cost). *Let $r(e)$ be the number*
1510 *of variable occurrences plus the number of operators in e. Then $\langle e, m \rangle \overset{s}{\underset{a}{\Longrightarrow}} \langle v, m \rangle$ for a unique value v*
1511
1512 *and some s with $|s| = r(e)$, and similarly for Boolean expressions. Each transition fires exactly one*
1513 *A-Read or one redex rule under congruences, so r strictly decreases; $\mathsf{acc}(s)$ is the sequence of reads of*
1514 *e, in left-to-right order.*

1515 Lemma A.7 (Adequacy of explicit reads). *$\langle e, m \rangle \overset{s}{\underset{a}{\Longrightarrow}} \langle v, m \rangle$ for some s iff $[\![e]\!]^{\mathsf{ord}}(m) = v$, and*
1516
1517 *$\langle b, m \rangle \overset{s}{\underset{b}{\Longrightarrow}} \langle t, m \rangle$ for some s iff $[\![b]\!]^{\mathsf{ord}}(m) = t$, where $[\![\cdot]\!]^{\mathsf{ord}}$ is the atomic expression evaluation defined*
1518 *above.*
1519

1520 Each dynamic variable read is exactly one read(x)-labelled transition and each dynamic assign-
1521 ment write exactly one write(x)-labelled transition, so the number of potential access-triggered
1522 Rowhammer events in a run is the length of its access trace. The total transition count is an exact
1523 cost for this abstract machine, but it is *not* the number of memory accesses: it also counts operator
1524 reductions, branch selection, sequencing elimination, and loop unfolding, silent (τ) steps that trigger
1525 no fault in the semantics of Section 5.

1526

1527 ## B Material for Section 5: A probabilistic model of Rowhammer

1528 This appendix presents the full small-step probabilistic operational semantics that was given only
1529 partially in Section 5. For more on probabilistic operational semantics see e.g., [1–3].
1530 The syntax and evaluation order are those of Section A. Throughout, the blast radius *br* and the
1531 protected set *locs* are ambient parameters, and pflip is a fault kernel subject to the contract (F1–F4)
1532 of Section 5.1. The judgements

1533

$$\langle e, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu, \qquad \langle b, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu, \qquad \langle P, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mu$$

1534

1535 relate a configuration to a *distribution* over successor configurations, with $\mu \in \mathsf{Dist}(\mathsf{AExp} \times$
1536 $\mathsf{Memory})$, $\mu \in \mathsf{Dist}(\mathsf{BExp} \times \mathsf{Memory})$, and $\mu \in \mathsf{Dist}(\mathsf{Prog} \times \mathsf{Memory})$ respectively. Exactly two
1537 rules sample the kernel, RH-A-Read and RH-P-Assign: every redex rule is the Dirac lifting of its
1538 deterministic counterpart, and every congruence rule pushes the reduction context through the
1539 premise distribution with Dist.*map*.

1540 *B.0.1 Arithmetic expressions.* Arithmetic values have no outgoing transition. A variable read first
1541 obtains the value $m(x)$ and then samples the fault triggered by that access.

1542

1543

1544 $$\frac{}{\langle x, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \big(\mathbf{let}\ m' \leftarrow \mathsf{pflip}(\mathsf{adj}(br, locs, x), m)\ \mathbf{in}\ \delta(\langle m(x), m'\rangle)\big)} \quad \text{(RH-A-Read)}$$

1545–1547 $$\frac{\langle e_1, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu}{\langle e_1 + e_2, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle e_1', m'\rangle.\, \langle e_1' + e_2, m'\rangle\big)(\mu)} \quad \text{(RH-A-Plus-L)}$$

1548–1549 $$\frac{\langle e_2, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu}{\langle v_1 + e_2, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle e_2', m'\rangle.\, \langle v_1 + e_2', m'\rangle\big)(\mu)} \quad \text{(RH-A-Plus-R)}$$

1550–1552 $$\frac{v = v_1 + v_2}{\langle v_1 + v_2, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \delta(\langle v, m\rangle)} \quad \text{(RH-A-Plus)}$$

1553 *B.0.2 Boolean expressions.* The Boolean values true and false have no outgoing transition.

1554

1555–1556 $$\frac{\langle b, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu}{\langle \neg b, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle b', m'\rangle.\, \langle \neg b', m'\rangle\big)(\mu)} \quad \text{(RH-B-Not-Step)}$$

1557

1558–1559 $$\frac{}{\langle \neg\mathsf{true}, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \delta(\langle \mathsf{false}, m\rangle)} \quad \text{(RH-B-Not-True)}$$

1560

1561–1562 $$\frac{}{\langle \neg\mathsf{false}, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \delta(\langle \mathsf{true}, m\rangle)} \quad \text{(RH-B-Not-False)}$$

1563–1564 $$\frac{\langle b_1, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu}{\langle b_1 \wedge b_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle b_1', m'\rangle.\, \langle b_1' \wedge b_2, m'\rangle\big)(\mu)} \quad \text{(RH-B-And-L)}$$

1565–1567 $$\frac{\langle b_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu}{\langle t_1 \wedge b_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle b_2', m'\rangle.\, \langle t_1 \wedge b_2', m'\rangle\big)(\mu)} \quad \text{(RH-B-And-R)}$$

1568

1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617

$$\frac{t = t_1 \wedge t_2}{\langle t_1 \wedge t_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \delta\big(\langle t, m\rangle\big)} \tag{RH-B-And}$$

$$\frac{\langle e_1, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu}{\langle e_1 = e_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle e_1', m'\rangle.\, \langle e_1' = e_2, m'\rangle\big)(\mu)} \tag{RH-B-Eq-L}$$

$$\frac{\langle e_2, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu}{\langle v_1 = e_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle e_2', m'\rangle.\, \langle v_1 = e_2', m'\rangle\big)(\mu)} \tag{RH-B-Eq-R}$$

$$\frac{t = (v_1 = v_2)}{\langle v_1 = v_2, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \delta\big(\langle t, m\rangle\big)} \tag{RH-B-Eq}$$

B.0.3 Programs. An assignment first reduces its right-hand side one expression step at a time. Once the expression is a value, the assignment writes that value and samples the fault triggered by the write, on the *updated* memory.

$$\frac{\langle e, m\rangle \rightsquigarrow_a^{\mathsf{RH}} \mu}{\langle x := e, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle e', m'\rangle.\, \langle x := e', m'\rangle\big)(\mu)} \tag{RH-P-Assign-Step}$$

$$\frac{}{\langle x := v, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \big(\textbf{let } m' \leftarrow \mathsf{pflip}\big(\mathsf{adj}(br, locs, x), m[x \mapsto v]\big) \textbf{ in } \delta\big(\langle \mathsf{skip}, m'\rangle\big)\big)} \tag{RH-P-Assign}$$

Sequential composition reduces its left-hand program until that program has terminated.

$$\frac{\langle P, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mu}{\langle P; Q, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mathsf{Dist}.map\big(\lambda\langle P', m'\rangle.\, \langle P'; Q, m'\rangle\big)(\mu)} \tag{RH-P-Seq-Step}$$

$$\frac{}{\langle \mathsf{skip}; Q, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \delta\big(\langle Q, m\rangle\big)} \tag{RH-P-Seq-Skip}$$

A conditional reduces its guard one Boolean-expression step at a time and then selects the corresponding branch.

$$\frac{\langle b, m\rangle \rightsquigarrow_b^{\mathsf{RH}} \mu}{\langle \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \mathsf{Dist}.map\ (\lambda\langle b', m'\rangle.\ \langle \mathsf{if}\ b'\ \mathsf{then}\ P\ \mathsf{else}\ Q, m'\rangle)\ (\mu)} \tag{RH-P-If-Step}$$

$$\frac{}{\langle \mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \delta\big(\langle P, m\rangle\big)} \tag{RH-P-If-True}$$

$$\frac{}{\langle \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \delta\big(\langle Q, m\rangle\big)} \tag{RH-P-If-False}$$

A while loop unfolds by one standard small step. Its guard is subsequently reduced by the conditional rules above.

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ P, m\rangle \rightsquigarrow_p^{\mathsf{RH}} \delta\left(\big\langle \mathsf{if}\ b\ \mathsf{then}\ \big(P; \mathsf{while}\ b\ \mathsf{do}\ P\big)\ \mathsf{else\ skip}, m\big\rangle\right)} \tag{RH-P-While}$$

There is no transition from $\langle \mathsf{skip}, m\rangle$. These are the terminal program configurations. The rules are syntax-directed, so every non-terminal configuration determines a unique one-step distribution,

1618 written $\text{step}^{\text{RH}}_{br,locs}(\langle P, m\rangle)$ at the program level. The absorbing kernel $\widehat{\text{step}}^{\text{RH}}_{br,locs}$ and the finite runs
1619 $\text{run}^{n}_{br,locs}$ and $\text{run}^{n}_{br,locs,\text{tr}}$ built from it are defined in Section 5.5.
1620
1621 ## B.1 The mechanised proof of the central theorem

1622 For reference we reproduce the Lean proof of Theorem 6.1 from Soundness.lean. The prose sketch
1623 is in Section 6.4.
1624

```
1625 theorem physical_separation_sound : PhysicalSeparationSound := by
1626   intro sep f hframe br locs hsafe p hwf m n
1627   apply Dist.mapD_eq_dirac_of_support
1628   intro r hr
1629   have hsub : ∀ x ∈ locs, x ∈ locs := fun x hx => hx
1630   have hpres : ∀ x ∈ locs, ∀ (m₀ m' : Memory),
1631       m' ∈ (f (sep.adj br locs x) m₀).support → ∀ ℓ, ℓ ∈ locs → m' ℓ = m₀ ℓ :=
1632     fun x hx m₀ m' hm' ℓ hℓ =>
1633       hframe.outside_unchanged _ m₀ m' ℓ hm' (sep.safe_disjoint hsafe hx ℓ hℓ)
1634   obtain ⟨h1, h2, h3⟩ :=
1635     runRHTrace_protected (keep := (· ∈ locs)) hsub hpres n hwf
1636       (AgreeP.refl _ m) r hr
1637   simp only [protView, Prod.mk.injEq]
1638   refine ⟨h1, ?_, ?_⟩
1639   · funext x
1640     exact h2 x.1 x.2
1641   · rw [h3]
```

1642 The script follows the three moves of the sketch. The intro line introduces the separation geometry,
1643 the kernel with its frame-locality hypothesis (hframe, clause F2), the blast radius and protected set,
1644 physical separation (hsafe), the well-formed program (hwf), the initial memory, and the step bound.
1645 Dist.mapD_eq_dirac_of_support is the support characterisation of the point mass (Section 3): it
1646 reduces the distributional equation to showing that every outcome r in the support of the trace-
1647 carrying Rowhammer run has the protected view of the deterministic run. The two haves discharge
1648 the two clauses of the parametric invariant at keep $\triangleq$ $(\cdot \in locs)$: hsub is clause (i), used locations
1649 are kept, trivial at this instantiation, where it is exactly well-formedness, hpres is clause (ii), fault
1650 samples preserve kept locations, composed from safe_disjoint (the victim set of a protected
1651 access avoids *locs*) and hframe.outside_unchanged (samples fix everything outside the victim
1652 set). The iterated invariant runRHTrace_protected, started from the reflexive agreement on the
1653 initial memory, then yields the three components of the protected view: h1, equal residual programs,
1654 h2, memories agreeing on *locs*, h3, equal label sequences. The remainder repackages them as an
1655 equality of protected views: funext turns pointwise agreement on protected locations into equality
1656 of the restricted memories as functions on the subtype, and h3 rewrites the access traces.
1657
Appendix References

1658 [1] Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. 2020. *Semantics of Probabilistic Programming: A Gentle Introduction.*
1659 Cambridge University Press, 1–42. doi:10.1017/9781108770750.002
1660 [2] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2010. Probabilistic Semantics and Program Analysis. In
1661 *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1–42. doi:10.1007/978-3-642-13678-8_1
1662 [3] Ugo Dal Lago and Margherita Zorzi. 2012. Probabilistic operational semantics for the lambda calculus. *RAIRO -*
1663 *Theoretical Informatics and Applications* 46 (2012), 413–450. doi:10.1051/ita/2012012
1664 [4] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer (Undergraduate Topics in*
1665 *Computer Science)*. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-1-84628-692-6
1666 [5] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics: With Isabelle/HOL*. Springer. doi:10.1007/978-3-319-10542-0

[6] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages.* The MIT Press. doi:10.7551/mitpress/3054.001.0001

1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715