

Mechanized Proofs of ORAM Correctness, Security, and Failure Bounds

Manuel Barbosa^{1,2}, Gilles Barthe^{3,4}, Gustavo Delerue², Benjamin Grégoire⁵,
Pierre-Yves Strub², and Xingyu Xie³

¹ University of Porto (FCUP) and INESC TEC, Porto, Portugal
`mbb@fc.up.pt`

² PQShield, Paris, France

`{gustavo.delerue,pierre-yves.strub}@pqshield.com`

³ Max Planck Institute for Security and Privacy, Bochum, Germany

`{gilles.barthe,xingyu.xie}@mpi-sp.org`

⁴ IMDEA Software Institute, Madrid, Spain

⁵ Inria Sophia-Antipolis, France

`benjamin.gregoire@inria.fr`

Abstract. ORAM is a fundamental cryptographic technique that permits outsourcing memory storage while probabilistically hiding memory access patterns of client programs. We use the `EasyCrypt` proof assistant to obtain fully mechanized proofs of correctness and security of Simple ORAM [Chung and Pass, 2013] and Path ORAM [Stefanov et al., CCS 2013]. To the best of our knowledge, these are the first machine-checked proofs that cover the correctness bound, i.e., the failure probability, of an ORAM construction. Our proofs refine, simplify, clarify and slightly improve the paper proofs, and many parts of our development can be reused to obtain similar results for other ORAM constructions. As a final contribution, we provide a general library that handles the recursive composition of ORAM constructions to reduce storage demands on the client, and apply this to Path ORAM to capture a practically-relevant instantiation.

1 Introduction

ORAM (Oblivious RAM) is a fundamental cryptographic primitive which, thanks to numerous recent efforts, has rapidly evolved from an object of theoretical interest to a family of practically deployed constructions that offer different performance and privacy tradeoffs [21,22,31,32,30,33,2,29,13,19,14,4,23,27,17,28,35]. In particular, a Path ORAM implementation is deployed in Signal’s contact discovery infrastructure as a critical component for privacy-preserving interaction with users [20].

The relevant security properties for an ORAM construction are correctness, of course, and obliviousness. The former guarantees that the ORAM construction can replace the functionality of a local RAM with operations outsourced to an untrusted server; the latter guarantees, informally, that the untrusted server

observes nothing that leaks information about which memory entries are being accessed and what are their contents. One peculiarity of ORAM is that the correctness arguments are quite complex and that, once correctness is established, obliviousness is often straightforward to prove.

Formal verification of cryptography has also progressed fast in the last decade and several works have looked at ORAM [12,16,34]. However, all of these prior works are limited in their treatment of the construction: they use various approaches to establish obliviousness, but they only consider non-recursive ORAM construction and assume that correctness is perfect—they sidestep the more intricate part of the analysis.

In this paper we give the first machine-checked treatment of ORAM that covers the correctness bounds for the constructions and builds an end-to-end security theorem that explicitly captures the impact of a correctness error in the security guarantees.

Main Results. We consider the following natural definition of oram security.

Definition 1 (Informal). *An ORAM algorithm is secure if it is*

1. almost-correct, *i.e.*, no distinguisher can tell apart the output values of the ORAM algorithm from those of an ideal RAM, except with small advantage; and
2. almost-oblivious, *i.e.*, for any two (adversarially-chosen) data access sequences of the same length, given the sequence of positions accessed on the remote storage, no distinguisher can tell which data access sequence was executed, except with small advantage.

We cover two ORAM constructions, Simple ORAM [15] and Path ORAM [32]. The former is a warm up example, whereas the latter is practically relevant. For both we prove in EasyCrypt that almost-correctness holds with advantage at most ϵ and almost-obliviousness holds with advantage at most 2ϵ , where ϵ is the probability of a failure/overflow event occurring in the construction. Our main contributions are the analyses of these failure probabilities, which to the best of our knowledge happens for the first time in a machine-checked setting. Our main theorems are as follows.

Theorem 1 (Simple ORAM Security, Informal). *Simple ORAM is secure with a failure bound $2^{-\lfloor Z/2 \rfloor + 2} \cdot nT$, where Z is a security parameter (bucket capacity), n is the size of the memory, and T is the number of data requests.*

The bound in the literature [15] is $2^{-(Z/2)+1} \cdot nT$. The difference comes from the fact that machine-checked environment forces the bridging and clarification of minor gaps in the original paper proof (see Section 3.5 for details).

Theorem 2 (Path ORAM Security, Informal). *Path ORAM is secure with a failure bound $0.37236 \cdot (2^Z / (2^{Z+1} - 8))^R \cdot T$, where $Z \geq 5$ and R are security parameters (bucket capacity and stash size), and T is the number of data requests.*

The bound in the literature [32] is $14 \cdot 0.6002^R \cdot T$ for $Z = 5$. Our statement is more general and slightly tighter: 1) the new bound applies to bucket sizes of at least 5 while the original bound only applies for the bucket size exactly 5; and 2) for a bucket size of 5, the new bound reduces the stash size needed by 8.78% asymptotically.

Our Contributions. More in detail, our proofs consolidate prior paper proofs in the following aspects. Our mechanized proofs are based on layered abstractions, connecting implementation-level descriptions to high-level analyses. They make explicit many low level details omitted in the paper proofs, both in the description of the algorithms and in the proofs themselves, where the machine-checked environment requires full justification of the abstraction steps. For the obliviousness results we adopt a constructive (leakage-simulation-based) approach to the proof: rather than proving indistinguishability directly—as is the case of prior machine-checked works—we prove the stronger result that leakage is simulatable as a partial permutation of the randomness tape of the client. For the failure bound proof of simple oram, we simplify the reduction to the core abstraction for overflow event analysis (called the dart game) by observing that this event is symmetric over time and time reversal can be avoided. For the failure bound proof of path oram, we simplify the proof and optimize the bound by observing that, instead of reducing to the involved “infinite eviction game”, with the help of a more sophisticated negative association usage, the reduction can naturally go to the abstractions for simple oram (a balls-and-bins game and the dart game), which furthermore enables a more precise bound analysis.

As a final contribution, we constructed in EasyCrypt a general framework for the recursive composition of ORAM. This fixes a uniform interface for ORAM that captures both our Simple and Path ORAM, and proves the security and correctness with client-side storage reduced by outsourcing position map information to the server using lower-level ORAM(s). This comprises a general composition result, which can be instantiated for an arbitrary number of recursive steps and with arbitrary constructions. We state our final results for a two-level Path ORAM construction, refined to clearly separate the code of client and server, which implements ORAM computations in a list-based low-level semantics that opens the way to connect our results to real-world implementations, for instance, the one deployed by Signal [20].

Reusable Artifacts. A significant part of our proof is general and can be used to obtain similar results for other oram variants in the following concrete aspects. We provide necessary reusable mathematical libraries in EasyCrypt, providing extended support for basic calculus, moment-generating functions, and negative association analysis. We define an oram tree as a map from an address to its location on the tree and associated value. This definition is general for all tree-based oram constructions and carefully tailored during the development. Intuitively, dealing with oram tree structures in formal verification of correctness implies proving the preservation of strong invariants through the algorithm execution, such as uniqueness of the address on the tree and the fact that each

block sits at a node along the assigned path. The definition we use in our analysis guarantees all these invariants by construction, and serves perfectly as a proof artifact. Then, a one-time proof handles the isomorphism to an implementation-level description of the concrete oram construction; this means that a lot of algorithm-specific reasoning is avoided by this abstraction step alone. Our proof of obliviousness provides a general constructive (simulation-based) approach: the real-world leakage can be simulated from full path leakages, and these paths in turn are a partial permutation of the random tape of the client. This implies leakage uniformity at the path level and leakage indistinguishability at the implementation level: the standard security notion for oram. Of all these steps, only the lifting from real-world leakage to path leakage is algorithm-specific. The proof of the path oram overflow bound—reducing to an infinite oram abstraction and then the steps to analyze the associated non-trivial Markov chain—can guide similar proofs for other PathORAM-like scheme such as Ring ORAM [30] and Circuit ORAM [33]. Finally, the part of the development focusing on recursive composition is fully general and thus can be reused without change in the future.

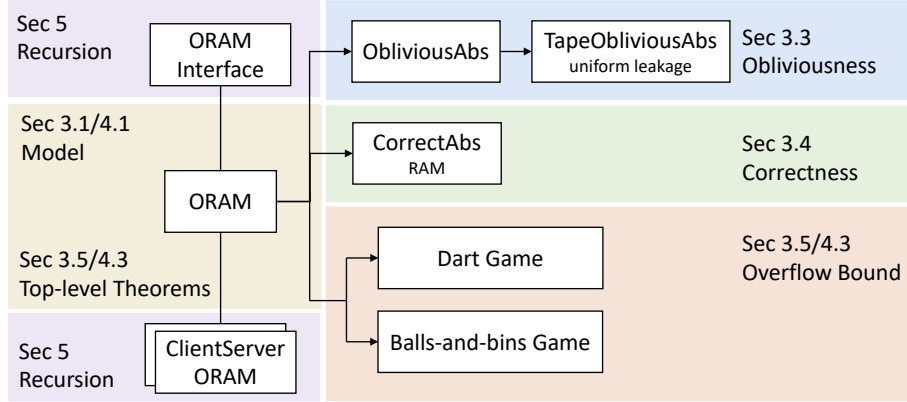


Fig. 1: Bird’s eye view of the mechanization. Nodes are models and arrows are abstractions/reductions.

Development Overview. Figure 1 shows a bird’s eye view of our mechanized proofs for Simple ORAM and Path ORAM. Both proofs follow the same structure and share lots of infrastructure. The developments are centered on a core representation, and structured in multiple levels of abstraction. The root of the proof tree, which we show on the left of the figure, contains the formal model for ORAM and the top-level obliviousness and correctness theorems. From the root, the proof goes into three sub-goal branches as shown in the figure. The two branches shown in blue and green at the top establish obliviousness and functional correctness assuming no overflow occurs, and the orange branch on the

bottom takes care of the overflow bounds. For obliviousness, we first introduce an abstraction that retains only the sampling operations that fully characterize the leakage, which is expressed in a canonical form: leakages of traversing a path are first compressed as the path itself and, second, the abstraction is reshaped to pre-sample a random tape and produce leakage as a partial permutation of this tape. Proving leakage uniformity at this level of abstraction can be achieved by a general result on applying a partial permutation to the uniform distribution over lists; low-level leakage indistinguishability then follows from the fact that low-level leakage is a deterministic function of the uniform distribution. For bounding overflow, both consider the block accumulation on nodes in two cases—internal nodes and leaves, and reduce the internal node case to the dart game and the leaf case to the balls-and-bins game. Note that, besides the reduction to and analysis of games, Path ORAM first needs a sophisticated normalization as in Section 4.3. Finally, in the purple region, we concretize the main representation into a list-based representation, and clearly separate the algorithm on the client side and the server side. We finally obtain a two-level recursive ORAM with the help of a general ORAM interface and composability result.

Related Works. Motivated by the verification of ORAM, there is a line of efforts of designing program logics, notably probabilistic separation logic [12] and its variants [5,7,24,25,26,6]. At the time of writing, failure bound and recursion are out of the reach of all existing efforts. In terms of the mechanization efforts, the obliviousness proof for ORAM is carried in [12,34], but only on pen-and-paper, whereas [34] mechanizes the soundness of their logic. [16] uses a type system with independence region to prove obliviousness for a variant of ORAM that discards stored values when nodes exceed capacity.

Experience with Using AI. This was a one year project during which we gradually adopted AI to boost the scalability of formal verification. We hope our experience can be useful for other works in the area. We converged to a workflow that aims to guarantee human-verifiable results: humans write definitions, specifications and statements, and maintain the skeleton of the mechanized proofs. Then, we use AI to help plan and generate the proof scripts and spot mistakes. To maintain control and simplify human intervention, we typically store one key lemma per file, and split the file into three sections: definitions, intermediate proofs, and final lemma statements, where the human carefully checks the definitions and the statements. We started with no AI involvement at all and ended with almost 100% AI-based proof script generation and a much compressed timeline in the final steps of our development.

Artifact. We provide our mechanized proofs in EasyCrypt in <https://doi.org/10.5281/zenodo.23114189>. This artifact contains approximately 41k lines of EasyCrypt code. Of these lines, 1k lines of code are definitions and statements, which belongs to the trusted base and was carefully reviewed by the authors; the remaining lines are mechanically checked by EasyCrypt proof assistant.

2 Preliminaries

2.1 EasyCrypt

EasyCrypt [10] is a proof assistant with dedicated support for mechanizing cryptographic proofs [9,8,1]. Its primary logical basis is probabilistic Relational Hoare Logic (pRHL [11]), which we use to establish game-hopping/model-refinement arguments, sometimes conditioned on non-occurrence of a failure event. pRHL is designed to align two programs and prove relations between their states step by step, without analyzing concrete distributions on each side. EasyCrypt provides powerful and easy-to-use proof tools for *up-to-bad* reasoning: a probabilistic statement of the form

$$|\Pr[G_{i+1} \Rightarrow 1] - \Pr[G_i \Rightarrow 1]| \leq \Pr[G_{i+1} \Rightarrow \text{bad}]$$

translates into a statement in pRHL in which G_i and G_{i+1} are run side by side, and where one needs to prove 1) that **bad** takes the same value in both games (i.e., that it occurs with the same probability); and 2) that, $\neg \text{bad}$ implies that both games produce the same result. Then EasyCrypt can prove such goals interactively and conveniently.

Besides pRHL, EasyCrypt also incorporates many other probabilistic program logics for various use cases, including non-relational logics such as probabilistic Hoare Logic (**phoare**) and expectation Hoare Logic (eHL [3]). We use these latter logics, which capture the probabilistic semantics of a single program, to bound concrete probabilities of failure events. On top of program logics, EasyCrypt offers a number of generic libraries for complex reasoning about game transformations, which can be instantiated for specific proofs. In particular, PROM (programmable random oracle model) is a library where different random oracles are defined and proved to be the same. This library establishes generically that all samplings can be deferred until the values are actually used. Symmetrically, we use PROM to eagerly extract all samplings to a random tape, which is pre-sampled and consumed sequentially.

2.2 Oblivious RAM

Defining ORAM Oblivious RAM enables outsourcing memory storage to an (honest-but-curious) adversary observing the memory accesses—but not the contents being read or written, which can be encrypted. The security goal is that the adversary only learns the number of memory accesses. ORAM constructions are traditionally presented as *compilers*, transforming the *logical* memory accesses of the original program running in a Random Access Machine (RAM) to *physical* memory accesses at the remote storage. The transformation preserves the functionality of the program, but the physical access pattern is rendered essentially independent of the logical access pattern.

A Random Access Machine (RAM) consists of a CPU with a small number of registers (e.g., constant or polylogarithmic) and a random access memory of

some bounded size. The CPU can read and write each register by name and each memory cell by logical address. For the purpose of this paper we can consider the RAM program simply as a sequence of memory access instructions, i.e., read or write operations. The syntax of RAM programs is defined as follows:

$$i ::= \text{read}(addr) \mid \text{write}(addr, value) \quad P ::= \epsilon \mid i; P$$

where $addr$ ranges over a logical address space of size n , and $value$ ranges over a value domain. The semantics of a program P , $\llbracket P \rrbracket$, is defined as a value list of the same length as P , where the i -th element of the list is the result of the i -th operation, namely, the latest value written to/read from the address $addr$ of the i -th instruction. When considering the functional correctness of ORAM we will consider these semantics partially defined over all syntactically well-formed programs: we rule out the cases where an address is read before being ever written. This definition is conceptually equivalent to returning a dummy value when reading a fresh address.

An ORAM compiler is a randomized algorithm $\mathcal{C}$ which transforms a RAM program P into another RAM program $\mathcal{C}(P)$ that is executed in a client. The compiled program emulates the same sequence of read/write operations by interacting with remote storage maintained by an adversarial server. We abstract the notion of remote storage as a dictionary mapping bit strings $p \in \{0, 1\}^*$ to buckets b .⁶ Access to remote storage is performed via two operations $\text{put}(p, b)$ and $b \leftarrow \text{get}(p)$. We define the leakage of $\mathcal{C}(P)$, denoted $\mathcal{L}(\mathcal{C}(P))$, as the sequence of remote positions $p_1, p_2, \dots$ it accesses in remote storage. The semantics of $\mathcal{C}(P)$ are defined similarly to those of P , and they correspond to the observed values at the read/write interface.

The transformation performed by the ORAM compiler needs to 1) preserve the semantics, while using remote storage, and 2) prevent the adversary monitoring that storage from learning sensitive information. For example, the adversary cannot learn 1) which data is being accessed, 2) how old the accessed data is, 3) whether the same data is accessed, 4) access pattern (e.g., whether there is a loop), or 5) whether the access is a read or write. Formally, the requirements on the ORAM compiler can be defined as follows:

Definition 2 (ORAM Compiler). *A randomized algorithm $\mathcal{C}$ is an ORAM compiler if it guarantees the following properties:*

- *(almost) obliviousness: for any two input programs P_1 and P_2 of the same length, the distributions of access histories, $\mathcal{L}(\mathcal{C}(P_1))$ and $\mathcal{L}(\mathcal{C}(P_2))$, are indistinguishable;*
- *(almost) correctness: for any input program P , the outputs of the original program and the compiled program, $\llbracket P \rrbracket$ and $\llbracket \mathcal{C}(P) \rrbracket$, are indistinguishable.*

In this definition, “indistinguishable” can refer to computational, statistical, or perfect indistinguishability.

⁶ Buckets are implicitly assumed to be freshly encrypted every time they are uploaded.

Tree-based ORAM Figure 2 illustrates the core idea of constructing (tree-based) ORAMs. Remote storage is used to maintain a full binary tree of buckets of height h^7 , where every logical address is associated with a leaf sampled uniformly at random, and the data associated with this logical address is stored in some node along the *path*⁸ from the root to that leaf. The association relation between logical addresses and leaves is globally kept in a *position map*, which is securely stored on the client side. For simplicity, we assume that the size of logical memory is exactly the number of leaves, i.e., $n = 2^h$.

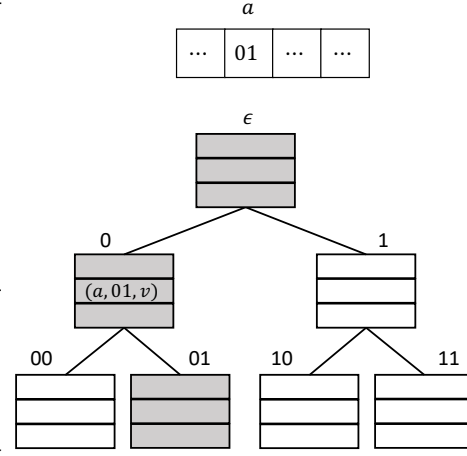


Fig. 2: Tree-based ORAM Construction.

Each node in the tree can be identified/indexed by a bitstring of length up to h where, starting from the root, 0 indicates going to the left child and 1 indicates going to the right child. We make such indices correspond to parameter p in the **put/get** operations we defined above, so that these operations can be easily seen to correspond to setting and retrieving nodes in a tree.

Each node is a *bucket* containing at most Z *blocks*, where Z is a small constant. Each block is a tuple of the form (a, p, v) , where a is a logical address, p is the path from the root to the associated leaf, and v is the value in the memory cell on the logical address a . As a relaxation of the tree structure, in some ORAM constructions, there is also a *stash* containing R blocks that cannot be stored on the tree, where R is another small constant.

The main invariant for the ORAM tree is that blocks in the tree are always stored in some bucket along the path from the root to the leaf identified by its entry in the position map. At a high level, the protocol of tree-based ORAMs is roughly as follows: 1) get a block from the tree, revealing its current position map entry; 2) update the position map to a new random leaf and possibly modify the block on the client side; 3) put the block back in the tree using a procedure that (i) maintains the invariant, (ii) does not reveal the new position map entry, and (iii) respects the size limit of each bucket.

The tree structure allows moving away from a linear scan of the storage and performing only a logarithmic number of physical accesses. Moreover, if nodes had infinite capacity, the correctness and obliviousness of tree-based ORAM would be relatively straightforward to establish. However, computational overhead (a.k.a. network bandwidth) and client-side space constraints require the size of buckets and the stash to be small. So the challenge is to use limited resources and still justify that overflow in buckets and stash is very unlikely. The most intricate parts

⁷ The height counts from 0. For example, the height of a single-node tree is 0.

⁸ In this paper, unless explicitly stated otherwise, *path* refers to a root-to-leaf path.

of ORAM correctness and security proofs reside in proving such overflow bounds, which call for sophisticated probabilistic analysis. Note also that characterizing leakage in the presence of errors is generally hard, so typically the probability of error must be negligible in practice.

3 A mechanized proof for Simple ORAM

We now present our mechanized proof for Simple ORAM. We begin with the formal model and proof overview, and then explain the mechanization of the proof. The notations, formal models, and proofs will serve as a basis for Path ORAM in Section 4.

Notational conventions Here and in the rest of this paper, we write $K \multimap V$ for the type of finite partial maps from K to V . For such a map o , we write $o[a]$ for the lookup at a , $a \in \text{dom } o$ for membership, $o \cup o'$ for the union of two compatible maps, and $\emptyset$ for the empty map. For a list ℓ , we write $[]$ for the empty list, $x :: \ell$ for prepending x to ℓ . For a boolean predicate P , we write $\mathbb{1}[P]$ for its indicator (1 if P holds, 0 otherwise). We write $\text{Ber}(p)$ for the Bernoulli distribution with success probability p , and, for a distribution D , D^n for the product distribution of n independent samples from D .

3.1 Formal Model

We assume two abstract types **addr** and **value** for logical addresses and the values stored at them, and an integer parameter h for the height of the tree. The type **value** comes with a designated default $\perp$.

We then define two subtypes for positions in the tree: **node**, the path to a node, a bit-string of length at most h ; and **path**, a root-to-leaf path, a bit-string of length exactly h . For two paths p_1, p_2 , we write $p_1 \sqsubseteq p_2$ for the prefix order and $p_1 \sqcap p_2$ for their longest common prefix.

Following our definition in the previous section, the *leakage* records the sequence of remote-storage accesses an adversary observes during execution. Each access has a kind, **Get** or **Put**, and targets a node of the tree, so a leakage is a list of pairs (k, u) with $k \in \{\text{Get}, \text{Put}\}$ and u a node, accumulated during execution.

Our proofs are then carried out over the following representation of the Simple ORAM state:

$$\text{record} := \{ \text{path} : \text{path}; \text{depth} : [0, h]; \text{value} : \text{value} \}, \quad \text{oram} := \text{addr} \multimap \text{record}.$$

Here a Simple ORAM state is a finite partial map on logical addresses to *records* — each record has all the metadata of a logical address (or, equivalently, a block). We keep the information about which path an address is assigned to, at which depth it is stored, and which value is associated with it. This data type captures, by construction, many invariants that are kept by the Simple ORAM algorithm, such as the fact that a block is placed at a node sitting along the path to the assigned leaf in the tree.

SimpleORAM.Fetch(o, a)	SimpleORAM.Flush(o)
1: $p \xleftarrow{\$} \{0, 1\}^h; v \leftarrow \perp$ 2: if $a \in \text{dom } o$ then $p \leftarrow o[a].\text{path}$ 3: for $d = 0$ to h do 4: $u \leftarrow p[0..d]; b \leftarrow \text{getNode}(o, u)$ 5: if $a \in \text{dom } b$ then 6: $v \leftarrow b[a].\text{value}$ 7: $b \leftarrow b \setminus \{a\}$ 8: $o \leftarrow \text{setNode}(o, u, b)$ 9: $L \leftarrow (\text{Put}, u) :: (\text{Get}, u) :: L$ 10: return (o, v)	1: $p \xleftarrow{\$} \{0, 1\}^h; s \leftarrow \emptyset$ 2: for $d = 0$ to h do 3: $u \leftarrow p[0..d]$ 4: $s \leftarrow s \cup \text{getNode}(o, u)$ 5: $b \leftarrow \{r \in s \mid p \sqcap r.\text{path} = d\}$ 6: $o \leftarrow \text{setNode}(o, u, b)$ 7: $s \leftarrow \{r \in s \mid p \sqcap r.\text{path} > d\}$ 8: $L \leftarrow (\text{Put}, u) :: (\text{Get}, u) :: L$ 9: return o
SimpleORAM.PutBack(o, a, v)	SimpleORAM.Compile(P)
1: $p \xleftarrow{\$} \{0, 1\}^h$ 2: $o[a] \leftarrow (\text{path}: p, \text{depth}: 0, \text{value}: v)$ 3: $L \leftarrow (\text{Put}, \epsilon) :: (\text{Get}, \epsilon) :: L$ 4: return o	1: $o \leftarrow \emptyset; vs \leftarrow []; L \leftarrow []$ 2: foreach $i \in P$ do 3: $(o, v) \leftarrow \text{Access}_{(\text{inf}/\text{bnd})}(o, i)$ 4: $vs \leftarrow v :: vs$ 5: return vs
SimpleORAM.Access _{inf} (o, i)	SimpleORAM.Access _{bnd} (o, i)
1: $a \leftarrow i.\text{addr}$ 2: 3: $(o, v) \leftarrow \text{Fetch}(o, a)$ 4: if i is a write then $v \leftarrow i.\text{value}$ 5: 6: $o \leftarrow \text{PutBack}(o, a, v)$ 7: 8: $o \leftarrow \text{Flush}(o)$ 9: return (o, v)	1: $a \leftarrow i.\text{addr}; v \leftarrow \perp$ 2: if $\neg \text{Overflow}(o)$ then 3: $(o, v) \leftarrow \text{Fetch}(o, a)$ 4: if i is a write then $v \leftarrow i.\text{value}$ 5: if $\neg \text{Overflow}(o)$ then 6: $o \leftarrow \text{PutBack}(o, a, v)$ 7: if $\neg \text{Overflow}(o)$ then 8: $o \leftarrow \text{Flush}(o)$ 9: return (o, v)

Fig. 3: Simple ORAM algorithm, and its variant for analysis, Infinite Simple ORAM. The only difference between the two variants is that, when an overflow occurs, Simple ORAM aborts and Infinite Simple ORAM continues.

Figure 3 shows the Simple ORAM algorithm and its variant, Infinite Simple ORAM, as an analysis artifact. The entry point is `Compile`, which goes through the program and calls `Access` to handle each instruction. `Access` has two versions, `Accessinf` and `Accessbnd`, which differ in that the latter aborts when overflowing. For each instruction, `Access` carries out three subroutines: `Fetch`, `PutBack`, and `Flush`. `Fetch` gets the block of the requested address from the tree, reading through

the path assigned to the address and removing the block when it is found. If the address is accessed for the first time, a uniformly random path is traversed instead. `PutBack` updates the block with a fresh random path and puts it back into the root of the tree. `Flush` randomly selects a path and pushes down the blocks along the path to the leaf as deep as possible, while respecting the main invariant – each block should stay on the path assigned. Throughout the algorithm, list L accumulates the leakage. Here, *overflow* is defined as the existence of a node containing more than Z blocks.

Remark. In Appendix A, we justify our claim that the models over which we obtain our results for ORAM provide a realistic path for future work to establish a connection to implementations. There we show how we used `EasyCrypt` to prove equivalence between the Simple ORAM model we present here and a lower-level model, closer to implementations, to which the correctness and security results can be naturally transferred by transitivity. In Section 5.2, the same concretization is also applied to Path ORAM, and it will go further towards real-world implementation.

3.2 Proof Overview

The proof goes in three orthogonal aspects: functional correctness, obliviousness, and the bound for overflow probability.

Functional correctness is ensured by the main invariant throughout the algorithm: each block is placed at a node along the path to the assigned leaf in the tree.

For *obliviousness*, we show that (1) leakages can be compressed in terms of paths traversed, and (2) the paths traversed are a partial permutation of the random tape, and, thus, the leakages are uniform.

For *bounding the overflow probability*, we first apply the technique of union bound over all nodes, and separately consider the cases for internal nodes and leaves. For a leaf, the overflow bound follows directly from a union bound over overflowing time and a tail bound of a binomial distribution. For an internal node, we consider counting two kinds of blocks separately – the blocks belonging to a path into the left subtree and a path into the right subtree. Overflowing at this node implies that either of the two counts exceeds half of the capacity, and the analysis of any count can be reduced to a so-called “dart game”: in each round, we throw a black dart and a white dart, both darts hit the bullseye with the same probability; the game ends when a white dart hits the bullseye, and the winning condition is that many enough black darts have been accumulated. To reduce the overflow of internal nodes to the dart game, we first apply a union bound over the starting time of the block stacking interval, and then take assigning a block in the subtree as a black dart hitting and flush through the subtree as a white dart hitting. The winning probability of the dart game comes from its symmetry between black and white darts: each winning dart sequence has $2^Z - 1$ losing sequences of the same probability, which indicates the winning probability is no more than 2^{-Z} .

3.3 Obliviousness of Infinite Simple ORAM

ObliviousAbs.Step(pm, a)	ObliviousAbs.Compile(P)
$ \begin{array}{l} 1: \quad rp \xleftarrow{\$} \{0, 1\}^h; pf \leftarrow rp \\ 2: \quad \text{if } a \in \text{dom } pm \text{ then } pf \leftarrow pm[a] \\ 3: \quad ppb \xleftarrow{\$} \{0, 1\}^h; pm[a] \leftarrow ppb \\ 4: \quad pfl \xleftarrow{\$} \{0, 1\}^h \\ 5: \quad \text{return } (pm, pf, pfl) \end{array} $	$ \begin{array}{l} 1: \quad pm \leftarrow \emptyset; L \leftarrow [] \\ 2: \quad \text{foreach } i \in P \text{ do} \\ 3: \quad \quad (pm, pf, pfl) \leftarrow \text{Step}(pm, i.\text{addr}) \\ 4: \quad \quad L \leftarrow pf :: pfl :: L \end{array} $

Fig. 4: Obliviousness Abstraction for Simple ORAM.

For the obliviousness proof, we first reduce the main model to the abstraction in Figure 4. This is a slicing of the Infinite Simple ORAM code to retain only leakage-related code. It tracks the leakage during execution, and maintains only a position map, as the leakages from fetch are determined by the position map. A crucial choice for the subsequent proof steps is that leakages are compressed as paths, rather than explicit get/put accesses to concrete positions in remote storage. Indeed, this abstraction captures the essence of leakage-production: for every access to address a , the old path associated to a is leaked—on first access a fresh path is taken—and the position map at a is updated by sampling a new path; then, a fresh path for flush is sampled and leaked as well. This abstraction eliminates probabilistic dependencies between different leakage observations in the more realistic model, and the later analysis becomes more tractable.

The soundness of the abstraction is established by the following two properties:

- the position map extracted from the main model is lockstep identically distributed with the position map in the abstraction, and
- there exists a simulation function that transforms the leakages in the abstraction to the leakages in the main model: from a path to the sequence of physical accesses along that path.

We establish obliviousness in our abstract model by proving a stronger property than obliviousness. Our result says that, for any fixed program size, the probability of any leakage is a constant. This implies leakage uniformity and, consequently, leakage simulatability from only the program length. In Section 3.6, we will see that the full obliviousness is a simple corollary of Theorem 3.

Theorem 3. *For any program P and any leakage list l , the probability that ObliviousAbs leaks exactly l is $2^{-2h|P|}$ when $|l| = 2|P|$, or 0 otherwise.*

Proof (Sketch). The key intuition of the proof is to show that the leakage list is a partial permutation of a random tape. First, we show that all random samplings can be eagerly moved to the beginning as a random tape. This is not easy to

prove from scratch, as samplings are performed within the compile loop and used in subsequent steps. Luckily, the PROM library of EasyCrypt has generically established that samplings can be deferred until needed. We instantiate PROM by treating the original samplings as the points of use, and introducing an initialization loop to eagerly sample a random tape. In this way, the original samplings can be treated as being deferred until needed, and they are equivalent to be sampled eagerly at the beginning. The transformation follows from the general PROM theorem.

Second, we show that the leakage is a partial permutation of the random tape. We do this by constructing a function from indices of the leakage list to indices of the random tape and proving its injectivity. The crux of the proof establishes a relation between the index of a memory access a and the index of the last time a is accessed. Thus, any element in the leakage list can be traced back to a unique element in the random tape.

Third, we prove that, since the random tape consists of independent, identically and uniformly distributed samples, any partial permutation of it is still uniform. In our case, each sample is a bitstring of size h , and each access produces two leakages (pf and pfl), so the probability of any leakage list of size $2|P|$ is $2^{-2h|P|}$.

3.4 Functional Correctness of Infinite Simple ORAM

RAM.Step(i)	RAM.Compile(P)
1 : $a \leftarrow i.\text{addr}; v \leftarrow \text{mem}[a]$	1 : $\text{mem} \leftarrow \emptyset; \text{vs} \leftarrow []$
2 : if i is a write then	2 : foreach $i \in P$ do
3 : $v \leftarrow i.\text{value}; \text{mem}[a] \leftarrow v$	3 : $\text{vs} \leftarrow \text{Step}(i) :: \text{vs}$
4 : return v	4 : return vs

Fig. 5: RAM model (reference for correctness).

We establish the functional correctness of Infinite Simple ORAM by proving an equivalence between the semantics of the compiled program $\mathcal{C}(P)$, i.e., the trace of values read and written from logical memory, and those of the original program P . Technically, this is a proof in pRHL, where two experiments are proved to execute in lockstep: on one side we have the execution of the compiled Infinite Simple ORAM program, while on the other we have the source program P interacting with an ideal memory, i.e., the ideal semantics, as in Figure 5. The proof is straightforward and follows from a relational invariant that is preserved throughout, and which states that the Infinite Simple ORAM state of $\mathcal{C}(P)$ unambiguously encodes the ideal map (mem) serving P .

3.5 Bound of Overflow Probability

This subsection presents the analysis of the overflow probability that separates Simple ORAM from Infinite Simple ORAM. As mentioned earlier, we use a union bound that splits two cases: overflow occurring in a leaf node, or overflow occurring in an internal node.

Overflow at Internal Nodes We first apply a union bound over all internal nodes. To upper-bound this probability, we introduce an abstraction that computes an over-approximation of the overflow event, by modelling the *potential* number of records that may come to contribute to the block count in the bucket—either because they are actually in the bucket, or because they are in one of its ancestors and associated with a prefix-consistent path—rather than an exact number of records stored in the bucket at each point in time. Furthermore, the abstraction splits the potential contributing records into two subsets: those that may be flushed down the left subtree, and those that may be flushed down the right subtree, i.e., the subtrees rooted at the left and right child of the node—since we are dealing with an internal node, one of these subsets may totally disappear if a flush operation passes through the node. We note that, if an overflow occurs in the internal node, i.e., if more than Z records fall in that bucket, then this implies that the number of records in one of the subsets above is at least $\lceil (Z + 1)/2 \rceil$.

InternalAbs.Step(<i>base</i>)	InternalAbs.Compile(<i>T</i> , <i>base</i>)
1 : $p \xleftarrow{\$} \{0, 1\}^h$; $c \leftarrow c + \mathbb{1}[base \sqsubseteq p]$	1 : $c \leftarrow 0$
2 : $p \xleftarrow{\$} \{0, 1\}^h$; if $base \sqsubseteq p$ then $c \leftarrow 0$	2 : for $i = 1$ to T do
	3 : Step(<i>base</i>)

Fig. 6: Counting Abstraction for Internal Nodes.

Figure 6 shows the abstraction. Here, *base* refers to the path leading to one of the children of the inner node we are considering, whereas c is the counter that keeps track of how many records are in the corresponding subset of contributing records. The abstracted operations are intuitive: 1) fetch can only decrease the count, so we ignore it; 2) putback contributes to the count if the path associated to the record passes through the correct subtree; 3) flush resets the counter because all potential contributing nodes disappear. We establish soundness of this abstraction by proving the observation we made above: 1) an overflow occurring in the considered node implies that one of the subsets that partitions the records in the bucket exceeds cardinality $\lceil (Z + 1)/2 \rceil$; and 2) for each of these two subsets, the abstraction computes the cardinality of a set that contains (at least) all the elements that may contribute to this subset, and hence it is sufficient to compute the probability that the counter c in the abstraction exceeds $\lceil (Z + 1)/2 \rceil$. This soundness lemma is, again, a pRHL proof maintaining the invariant above.

From the abstraction above, we return to the intuition of the proof in [15] and reduce to the *dart game* in Figure 7, where one has an unbounded number of white darts and black darts. In each round of the game, an iteration of the loop, we first throw a black dart, and then a white dart. Each dart independently hits the bullseye with probability p . The game finishes when enough darts have hit the bullseye. The winning condition is that all the hitting darts are black.

Note that we deviate slightly from the formulation of the dart game in [15]: we compute the winning condition in the middle of the round (after throwing a black dart and before a white dart). This change is crucial to match overflow events with the Simple ORAM algorithm, because overflow can happen right after putback (before flush). One may also wonder why we do not need to check for overflow at the end of the round; the answer comes from the way in which we defined the abstraction: because we are over-approximating by computing the potential of overflow, the failure event can be detected immediately at putback, and after putback the counter may only decrease due to flush.

<pre> 1 : hits ← 0; win ← true 2 : while hits < k do 3 : black $\overset{\\$}{\leftarrow}$ Ber(p) 4 : hits ← hits + $\mathbb{1}[black]$ 5 : if hits < k then 6 : white $\overset{\\$}{\leftarrow}$ Ber(p) 7 : win ← win ∧ ¬white 8 : return win </pre>	DartGame.Play(k, p)
---	---

Fig. 7: Dart Game.

To reduce the abstraction to the dart game, [15] applies a union bound over the time when overflow occurs. At the overflow time, the left/right subset must contain enough blocks, which means that, looking backwards, there must be enough consecutive putback hits (and no flush hits between) in the subtree. Reversing the execution order is appealing on paper, but inconvenient for mechanization.

Our reduction, instead, introduces an intermediate procedure that exposes the full sampling history as two i.i.d. lists of length T — each drawn coordinatewise from $\text{Ber}(p)$: the black-dart samples $lb \overset{\$}{\leftarrow} \text{Ber}(q)^T$ and the white-dart samples $lw \overset{\$}{\leftarrow} \text{Ber}(q)^T$, where q is the probability that the proposition $base \sqsubseteq p$ holds for a uniform path p . We are interested in the existence of *offending intervals*, in which there are at least $\lceil (Z + 1)/2 \rceil$ bits in lb with value 1 and no 1-bit in lw . We then prove soundness of this abstraction step by proving that overflow at time t is equivalent to the existence of an offending interval ending at time t and, furthermore, that the occurrence of an overflow over the whole history is equivalent to the existence of this offending time interval.

To analyze the probability that an offending time interval exists, one can perform a union bound over the ending instant or starting instant: these two ways are mathematically equivalent, due to time symmetry. To conclude the connection to the dart game, we perform a union bound over all *starting instants* for the offending interval, which allows us to consider the natural progression of time in our analysis: we can match the samplings of the dart game with samplings performed by the offending interval in our abstraction.

We finally compute the winning probability of Dart Game using Expectation Hoare Logic (eHL). The computation is performed inductively over the loop: at

each iteration, the probability that no white dart hits the bullseye is at most $2^{-hits+1}$. From this we can derive that the probability of winning the dart game is at most $2^{-\lceil(Z+1)/2\rceil+1}$, which equals $2^{-\lfloor Z/2 \rfloor}$ by a case analysis on the parity of Z . Note that the number of internal nodes in the full binary tree is $2^h - 1 = n - 1 \leq n$. Combining all possibilities over time, left/right subtree, and all internal nodes, the concrete bound that overflow ever happens at any internal node and at any time is $2^{-\lfloor Z/2 \rfloor+1} \cdot nT$.

Comparison with the paper bound This bound is slightly different from the one in [15], which is $2^{-Z/2} \cdot nT$. The difference can be attributed to three small inaccuracies in the paper proof: 1) when a node overflows, the correct bound on one of the counters that guarantees soundness of the counting abstraction is $\lceil(Z+1)/2\rceil$ instead of $Z/2$; 2) the fact that one must account for the union bound over two offending subsets of records seems to have been overlooked,⁹ and this introduces a factor of 2 for probability; 3) overflow may be triggered at two points of an access: right after putback but before flush, or in the flush itself. Summing up, 1) makes the bound better while 2) and 3) makes the bound worse by one bit, which makes the actual bound worse by about one bit.

Overflow at Leaves For the case of leaves, we directly apply a union bound over all leaves and overflow times, and then abstract as in Figure 8.

LeafAbs.Step(pm, a)	LeafAbs.Compile(P)
1 : $pp \xleftarrow{\$} \{0, 1\}^h$	1 : $pm \leftarrow \emptyset$
2 : $pm[a] \leftarrow pp$	2 : foreach $i \in P$ do
3 : return pm	3 : $pm \leftarrow \text{Step}(pm, i.\text{addr})$
	4 : return pm

Fig. 8: Counting Abstraction for Leaves.

This abstraction maintains a position map: for each instruction, we associate the accessed address a with a freshly sampled path. Then soundness is proved, again by pRHL, using the observation that the blocks on the leaf are no more than the blocks on the path to that leaf, which are all the blocks assigned to that leaf. Hence, if overflow occurs in a leaf, then the position map must have at least $Z + 1$ addresses associated with that leaf.

From the abstraction, we first compress the program into a distribution expression which directly samples a position map, and then apply Chernoff bound to over-approximate the probability that at least $Z + 1$ addresses are

⁹ We refer here to the union bound “ $\Pr[a + b \geq Z + 1] \leq \Pr[a \geq \lceil(Z+1)/2\rceil] + \Pr[b \geq \lceil(Z+1)/2\rceil]$ ”, which is justified by “if $a + b \geq Z + 1$, then $a \geq \lceil(Z+1)/2\rceil$ or $b \geq \lceil(Z+1)/2\rceil$ ”.

associated with the same leaf in the position map. As all positions are uniformly and independently selected, the expected number of elements mapping to any leaf is $0 \leq \mu \leq 1$, and by a standard multiplicative version of Chernoff bound, the probability that at least $Z + 1$ elements map to this leaf is no more than $(e^d / (1 + d)^{1+d})^\mu$, where $d = (Z + 1) / \mu - 1$. Then, by analyzing derivatives, we can further loosen this bound to $2^{-\lfloor Z/2 \rfloor}$.

Note that the number of leaves in the full binary tree is exactly the size of the logical memory, i.e., $2^h = n$. We can therefore conclude that the probability of overflow at any leaf and at any time is upper-bounded by $2^{-\lfloor Z/2 \rfloor} \cdot nT$.

Combining the two cases for internal nodes and leaves, the probability of overflow in the whole tree and over the whole history is bounded by $2^{-\lfloor Z/2 \rfloor + 2} \cdot nT$.

3.6 Top-level Theorems

Combining the results in the previous subsections, the formal statements of obliviousness and correctness of Simple ORAM are as follows.

$\text{OBL_IND}_{\text{O}, \mathcal{A}}(b, T)$	$\text{COR_IND}_{\text{O}, \mathcal{A}}(P)$
1 : $b' \leftarrow \text{false}; (P_0, P_1) \leftarrow \mathcal{A}.\text{Find}()$	1 : $vs \leftarrow \text{O}.\text{Compile}(P)$
2 : if $ P_0 = P_1 \wedge P_0 \leq T$ then	2 : return $\mathcal{A}.\text{Dist}(vs)$
3 : $\text{O}.\text{Compile}(P_b)$	
4 : $b' \leftarrow \mathcal{A}.\text{Dist}(\text{O}.L)$	
5 : return b'	

Fig. 9: Distinguishing games for obliviousness (left) and correctness (right).

The obliviousness distinguishing game OBL_IND is defined in Figure 9. An ORAM scheme O provides Compile and exposes its leakage L ; a distinguisher $\mathcal{A}$ provides Find , which outputs two programs, and Dist , which decides from the leakage which one was chosen by the experiment.¹⁰ For an ORAM scheme O , a distinguisher $\mathcal{A}$, and a maximum program length T , the obliviousness advantage is defined as

$$\text{Adv}_{\text{O}}^{\text{OBL-IND}}(\mathcal{A}) := \left| \Pr \left[\text{OBL_IND}_{\text{O}, \mathcal{A}}(0, T) \right] - \Pr \left[\text{OBL_IND}_{\text{O}, \mathcal{A}}(1, T) \right] \right|.$$

The final obliviousness theorem is stated and proved as follows.

Theorem 4 (Obliviousness). *Let T be the maximum length of the program, and n be the size of the memory. For any adversary $\mathcal{A}$,*

$$\text{Adv}_{\text{SimpleORAM}}^{\text{OBL-IND}}(\mathcal{A}) \leq 2 \cdot 2^{-\lfloor Z/2 \rfloor + 2} \cdot nT.$$

¹⁰ In *EasyCrypt*, both are module types, and one can quantify over all implementations of a module type; we use this to quantify over adversaries for security results, and over schemes for generic relations between security notions.

Proof (Sketch). We use an up-to-bad argument to account for the overflow event of `Compile`, which happens with probability at most $2^{-\lfloor Z/2 \rfloor + 2} \cdot nT$. Otherwise, SimpleORAM is equivalent to Infinite Simple ORAM and the obliviousness of Infinite Simple ORAM applies. Note that the failure probability is doubled because we need to exclude the overflow event for both runs.

For correctness, Figure 5 defines the standard RAM model as an address-value dictionary, read and updated with the memory access instructions. The correctness distinguishing game `COR_IND` is defined in Figure 9, which is general for any ORAM: the scheme `O` is either an ORAM or the RAM, and the distinguisher $\mathcal{A}$ tries to tell from the trace of values whether the output is from a normal RAM. For an ORAM scheme `O`, a distinguisher $\mathcal{A}$, and a program P , the correctness advantage is defined as

$$\text{Adv}_{\text{O},P}^{\text{COR-IND}}(\mathcal{A}) := \left| \Pr[\text{COR_IND}_{\text{O},\mathcal{A}}(P)] - \Pr[\text{COR_IND}_{\text{RAM},\mathcal{A}}(P)] \right|.$$

The final correctness theorem is stated and proved as follows

Theorem 5 (Correctness). *For any well-defined program of length at most T , and any adversary $\mathcal{A}$,*

$$\text{Adv}_{\text{SimpleORAM},P}^{\text{COR-IND}}(\mathcal{A}) \leq 2^{-\lfloor Z/2 \rfloor + 2} \cdot nT.$$

Proof (Sketch). Excluding the overflow event, which happens with probability at most $2^{-\lfloor Z/2 \rfloor + 2} \cdot nT$, Simple ORAM is equivalent to Infinite Simple ORAM. Then, the result follows from the previously established functional correctness of Infinite Simple ORAM.

4 A mechanized proof for Path ORAM

Here we present our mechanized proof for Path ORAM. We begin with the formal representation, the high level proof overview, and then explain the mechanized proofs, focusing on the difference from the original paper proof [32]. The main difference from Simple ORAM to Path ORAM is the analysis of the overflow bound, which we detail in Section 4.3. We follow the same notation as in Section 3.

4.1 Formal Model

With the basic types (including addresses, values, paths, leakages, etc.) defined in Section 3.1, the Path ORAM state is represented as follows.

$\text{record} := \{ \text{path} : \text{path}; \text{depth} : [0, h] \cup \{\perp\}; \text{value} : \text{value} \}, \text{oram} := \text{addr} \multimap \text{record}.$

Based on the Simple ORAM formalization, the difference is that now a block may be stashed, rather than being stored in the tree. To capture this, we relax

the `record` type so that the depth field may be undefined. In other words, a block is always associated with a path (its position map entry) and we signal the fact that it is stashed by setting depth to $\perp$. Like for Simple ORAM, this data type also captures by construction important invariants, such as, after the first access, a logical address is either mapped to a block in the stash or is stored in a node along the assigned path.

PathORAM.Access(o, i)	PathORAM.Compile _{inf} (P)
<pre> 1 : (tree, stash) ← split(o) 2 : a ← i.addr; p $\xleftarrow{\\$}$ {0, 1}^h 3 : if a ∈ dom o then p ← o[a].path 4 : for d = 0 to h do 5 : stash ← stash ∪ tree[p[0..d]] 6 : L ← (Get, p[0..d]) :: L 7 : if i is a write then v ← i.value 8 : else v ← stash[a].value 9 : p' $\xleftarrow{\\$}$ {0, 1}^h 10 : stash[a] ← (path: p', value: v) 11 : for d = h downto 0 do 12 : u ← p[0..d] 13 : b ← select_Z{r ∈ stash u ⊆ r.path} 14 : tree ← setNode(tree, u, b) 15 : stash ← stash \ b 16 : L ← (Put, u) :: L 17 : return (tree ∪ stash, v) </pre>	<pre> 1 : o ← ∅; vs ← []; L ← [] 2 : foreach i ∈ P do 3 : 4 : 5 : (o, v) ← Access(o, i) 6 : vs ← v :: vs 7 : return vs </pre>
	PathORAM.Compile _{bnd} (P)
	<pre> 1 : o ← ∅; vs ← []; L ← [] 2 : foreach i ∈ P do 3 : v ← ⊥ 4 : if ¬Overflow(o) then 5 : (o, v) ← Access(o, i) 6 : vs ← v :: vs 7 : return vs </pre>

Fig. 10: Path ORAM algorithm.

Figure 10 shows the Path ORAM algorithm. The entry point is `Compile`, which processes the program one operation at a time by calling `Access`. `Access` first stashes all blocks stored in the relevant path of the tree—immediately updating the accessed block if needed—and then writes back the blocks in the resulting stash along the same path from leaf to root. At each level, we identify the stashed blocks sharing the prefix up to this level with p , i.e., those that can be stored at this level respecting the invariant. If there are more than Z such blocks, only Z of them are placed; the choice is arbitrary, and we write `selectZ(S)` for an arbitrary subset of S of size $\min(Z, |S|)$, following the `select` operator of [32]. The leftover blocks after processing the whole path constitute the final value of the stash after the access is concluded. Here, we fix a stash capacity threshold $R > 0$ and define *overflows* as more than R blocks are stashed.

4.2 Proof Overview

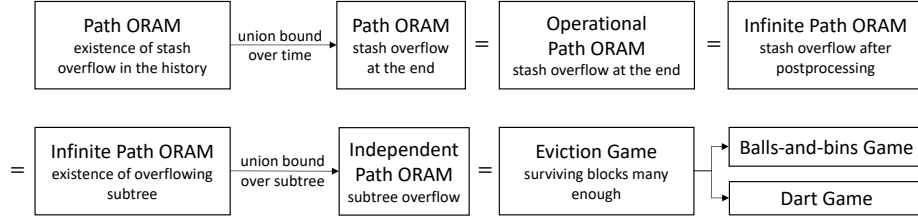


Fig. 11: Proof structure of the overflow bound of Path ORAM.

The proof is structured similarly to the Simple ORAM proof, addressing functional correctness, obliviousness, and bounding the overflow probability. The first two parts are very close to the proof for Simple ORAM, so we focus on how we bound the overflow probability and illustrate the structure of this proof in Figure 11. We first give an overview of the proof, and then provide more details of the main steps, clarifying differences to the paper proof at that point.

The first step is to fix the *time* overflow happens by applying a union bound over the history — we reduce the event that overflow ever happens during the whole history to the event that overflow happens at the end of a program. Then, we reduce the stash overflow event to a block-counting event in an *Infinite Path ORAM* — an abstraction very similar to that used in the Simple ORAM proof. The reduction starts with a restatement of the Path ORAM algorithm in terms of a sequence of tree operations — updating a block, evicting (flushing down) through a path, and pulling up along a path — where the bucket capacity is temporarily ignored after eviction and maintained again by pulling up. This *Operational Path ORAM* is then reduced to *Infinite Path ORAM*, where the algorithm is composed of a sequence of update and eviction, and the bucket capacity is ignored through the whole execution, but recovered by a global pull-up postprocessing algorithm. For Infinite Path ORAM, the stash overflow event is equivalent to the existence of a *rooted subtree*¹¹ where the total block capacity of the tree is exceeded.

From this point on, we work with a fixed rooted subtree T , by applying a union bound over all such subtrees. For a fixed T , we argue that the worst case with respect to overflow is the case where each address is accessed exactly once. This leads us to *Independent Path ORAM*, where each flush and pull-up relies on independently sampled random coins. From here, we reverse the time and reduce to *Eviction Game*, which counts the blocks surviving within a rooted subtree (assigned to this rooted subtree and not evicted out). The analysis of the tail bound of the eviction game is based on the standard moment-generating-function (MGF) technique. We apply negative association to separately consider each node

¹¹ A *rooted subtree* is a connected subgraph of the tree containing the root.

in the rooted subtree and then split the case — a leaf reduces to the balls-and-bins game, and an internal node reduce to the dart game. Combining all the cases together, and carefully choosing the MGF parameter, we get the final bound of the overflow probability, $0.37236 \cdot (2^Z / (2^{Z+1} - 8))^R$.

4.3 Overflow Bound

This subsection presents the analysis of the overflow probability of Path ORAM in more detail. The key observation is that, if we define Infinite Path ORAM where the bucket capacity is ignored, the blocks in the original stash must come from a rooted subtree of Infinite Path ORAM. Figure 11 gives an overview of the proof, which can be roughly split into two halves: we will first apply a series of reductions transforming the overflow event of Path ORAM into an *eviction game*, which is a normalized form of counting blocks in a rooted subtree in Infinite Path ORAM. We then analyze the tail bound of the eviction game upon two elementary games, namely, the balls-and-bins game and the dart game.

Our mechanization follows, at the high level, the paper proof [32], but we refine/improve the argument in the following aspects which we could not fully reconstruct from the written proof" 1) We clarify the definition of Operational Path ORAM (called “Real Path ORAM” in the paper) and the equivalence argument between Path ORAM and Operational Path ORAM; 2) We simplify the equivalence argument between Operational Path ORAM and Infinite Path ORAM, with the help of a different non-redundant postprocessing algorithm that visits each node only once; 3) Instead of reducing the eviction game to an “infinite eviction game”, we reduce it to the dart game—this needs a more sophisticated negative association result, but will, together with a more precise analysis, lead us to a tighter bound; 4) Besides a tighter bound, we also generalize the bound to bucket size $Z \geq 5$, rather than $Z = 5$ as in [32]. Now, we will go into the details.

Reduction to Eviction Game The first half of the proof is to reduce the Path ORAM algorithm to an eviction game, which essentially counts the blocks within a rooted subtree. The main steps are as follows.

Operational Path ORAM. We define the following operations, which will be used to re-describe the Path ORAM Access procedure, and, later, define Infinite Path ORAM. As shown in Figure 12, Access can be viewed as a sequence of three operations — updating the block accessed, flushing blocks through the path as deep as possible, temporarily breaking the bucket capacity limitation, and then pulling up the excess blocks so that bucket capacities are respected. More in detail the operators are as follows: (1) $\text{update}(a, y)$: assign leaf label y to block a and place the block into the stash. (2) $\text{evict}(x)$: starting with the blocks in the stash, along the path from the root to leaf x , push each block as far toward the leaf as possible. (3) $\text{pull}(x)$: starting from the leaf x toward the root, ensure every bucket holds at most Z blocks; whenever it does not, move excess blocks up to the parent, or the stash if there is no parent. WLOG, we assume a global

ordering of addresses to decide which blocks stay and which are moved: staying blocks have *smaller* addresses than moved blocks. Note that we explicitly need to account for the stash in these definitions, which is omitted in the paper [32].

The equivalence between Path ORAM and Operational Path ORAM is established step-by-step from leaf to root, where the blocks put at each node along the tree are the same for the greedy placement of **Access** and for evict followed by pull. The main refinement in the proof wrt [32], already stated above, is that we need to take the stash into consideration when running $\text{evict}(x)$ and $\text{pull}(x)$ for the equivalence to hold.

Infinite Path ORAM with postprocessing. Infinite Path ORAM is then defined based on Operational Path ORAM as shown in

Figure 13. The crucial difference is that pull is no longer used: a stash is only constructed at the end of the execution by running a postprocessing algorithm G that is defined inductively over the tree, with respect to the bucket capacity Z : each node accepts the excess blocks from its children (if it has children), and the stash accepts the excess blocks from the root. This definition removes the redundancy in the definition in [32], which pulls all leaves in an arbitrary order and one node may be pulled through more than one time. With this definition, we are able to have the following simpler proof.

The equivalence between Operational Path ORAM and Infinite Path ORAM with postprocessing is established via the following lemma.

Lemma 1. *Suppose address a is associated with path x . The following two operational sequences are functionally equivalent:*

- $G, \text{update}(a, y), \text{evict}(x), \text{pull}(x);$
- $\text{update}(a, y), \text{evict}(x), G.$

The corresponding lemma and its proof in [32] are quite involved, with a technique of pausing the two executions at two selective instances. Here, we simplify the lemma statement and the proof with the help of the concise definition of G . Atop this lemma, the main equivalence can be obtained by induction on the program.

Proof (Sketch.). Each G can be interpreted as two steps: (1) pulls up all excess blocks until reaching the path x , and (2) pulls up all blocks along the path x bottom up to the root.

OperationalPathORAM.Access(o, i)

```

1 :  $a \leftarrow i.\text{addr}$ 
2 :  $p \xleftarrow{\$} \{0, 1\}^h$ 
3 : if  $a \in \text{dom } o$  then  $p \leftarrow o[a].\text{path}$ 
4 :  $np \xleftarrow{\$} \{0, 1\}^h$ 
5 :  $o \leftarrow \text{update}(o, a, np)$ 
6 :  $o \leftarrow \text{evict}(o, p)$ 
7 :  $o \leftarrow \text{pull}(o, p)$ 
8 : return  $o$ 
```

Fig. 12: Operational Path ORAM.

InfinitePathORAM.Access(o, i)

```

1 :  $a \leftarrow i.\text{addr}$ 
2 :  $p \xleftarrow{\$} \{0, 1\}^h$ 
3 : if  $a \in \text{dom } o$  then  $p \leftarrow o[a].\text{path}$ 
4 :  $np \xleftarrow{\$} \{0, 1\}^h$ 
5 :  $o \leftarrow \text{update}(o, a, np)$ 
6 :  $o \leftarrow \text{evict}(o, p)$ 
7 : return  $o$ 
```

Fig. 13: Infinite Path ORAM.

Consider the effect of the first G in the first operational sequence. The effect of step 2 will be cancelled by $\text{evict}(x)$. The effect of step 1 will not be touched by $\text{update}(a, y)$ and $\text{evict}(x)$, noting that the block of x lies on the path to x , and then, the step 1 can be delayed over $\text{update}(a, y)$ and $\text{evict}(x)$ and merged with $\text{pull}(x)$ to constitute another G .

Independent Path ORAM. To move to Independent Path ORAM, and following the intuition in the paper proof, we need to prove that the overflow probability is maximized when accessed addresses are distinct. This is achieved by inductively removing the duplicate accesses to the same address. The difficulty here for mechanization is that the coupling for the relational logics in EasyCrypt cannot be naturally built because the samplings in the two programs are not aligned

IndependentPathORAM.Access(o, i)

```

1 :  $a \leftarrow i.\text{addr}$ 
2 :  $p \xleftarrow{\$} \{0, 1\}^h$ 
3 :  $np \xleftarrow{\$} \{0, 1\}^h$ 
4 :  $o \leftarrow \text{update}(o, a, np)$ 
5 :  $o \leftarrow \text{evict}(o, p)$ 
6 : return  $o$ 

```

Fig. 14: Independent Path ORAM.

once a sampling is removed on one side. To resolve this, we need to use the PROM EasyCrypt library to delay the redundant sampling. Also, we prove that the probability is maximized when each address is accessed at least once—we reconstructed this part of the argument from scratch, as details are omitted in [32]. The argument is that, as the missing address accessed means the corresponding blocks are also missing in the rooted subtree T , if we add the missing addresses at the beginning of the program, the storage of T will get larger, and the probability of overflow will also get higher. The combination of these two worst-case statements allows us to restrict our input program to have each address exactly once. In this case, the paths newly assigned to blocks and the paths used for eviction are mutually independent, hence the naming Independent Path ORAM shown in Figure 14.

Eviction Game. Figure 15 gives the definition of the eviction game, which is essentially playing Independent Path ORAM reversely. In previous steps, the overflow event is reduced to the event that there are enough blocks accumulating on the subtree T . Here, we reverse the algorithm for the convenience of counting the surviving blocks. Again, this part of the proof follows the intuition of the paper proof, but we needed to reconstruct some missing details.

Formally, a rooted subtree T is represented as a function from **path** to $\{0, \dots, h\}$. We define $\text{exit}(T, p)$ as the shallowest node on path p that falls outside T , and $\perp$ if there is no such a node (the whole path is inside T). The game plays n rounds, where n is the size of the address space. A series of counters $\{X_u\}$ is maintained to track the number of blocks caught by node u during eviction. At the beginning, all nodes are closed, and the counters are set to 0. At each round, we sample an eviction path ep and an assignment path ap . If ep goes through an exit node, we mark it as open. If ap is fully inside T , we increase the counter of its leaf; if it goes through a closed exit node, we increase the counter

of that node. Define $X = \sum_u X_u$ to be the total number of blocks staying in T . The equivalence between Infinite Path ORAM and Eviction Game is established by induction over the number of loop iterations in the ambient logic.

```

EvictionGame.Play( $T$ )
1 :  $open \leftarrow \emptyset$ 
2 : foreach node  $u$  do  $X_u \leftarrow 0$ 
3 : for  $i = 1$  to  $n$  do
4 :    $ep \xleftarrow{\$} \{0, 1\}^h$ ;  $ap \xleftarrow{\$} \{0, 1\}^h$ 
5 :   if  $exit(T, ep) \neq \perp$  then  $open \leftarrow open \cup \{exit(T, ep)\}$ 
6 :   if  $exit(T, ap) = \perp$  then  $X_{ap} \leftarrow X_{ap} + 1$ 
7 :   else if  $exit(T, ap) \notin open$  then  $X_{exit(T, ap)} \leftarrow X_{exit(T, ap)} + 1$ 
8 : return  $(X_u)_u$ 

```

Fig. 15: Eviction Game.

Negative Association. At this point in the proof we are now able to separately consider the counting for each node in the rooted subtree. Although they are not independent, fortunately, they are negatively dependent, as follows. (The negative association lemma in [32] splits the accumulation on exit nodes and leaves, while here we take a fine-grained decomposition over all nodes.)

Lemma 2. *For all $t \geq 0$, $\mathbb{E}[e^{tX}] \leq \prod_u \mathbb{E}[e^{tX_u}]$.*

Proof. Let $a_{i,u}$ to be 1 if the assign path at step i goes through u , where u is an exit node or a leaf, and 0 otherwise. And $b_{i,u}$ to be 1 if the path evicting at step i does not go through u , where u is an exit node or a leaf, and 0 otherwise. By [18, Lemma 8] (zero-one lemma for negative association) and a dual argument, we have that $\{a_{i,u}\}$ (resp. $\{b_{i,u}\}$) is negatively associated. Noting that $a_{i,u}$ and $b_{i,u}$ are independent, [18, Proposition 7(1)] implies negative association of $\{a_{i,u}, b_{i,u}\}$.

Now, note that, for any node u , there are three cases: 1) u is an internal exit node, then $e^{X_u} = e^{\sum_{1 \leq i \leq N} a_{i,u} \prod_{i \leq j \leq N} b_{j,u}}$; 2) u is a leaf in T , then $e^{X_u} = e^{\sum_{1 \leq i \leq N} a_{i,u}}$; 3) otherwise, u is not at the boundary of T , i.e., $e^{X_u} = 1$. In all the three cases, e^{X_u} is a non-decreasing function over $\{a_{i,u}, b_{i,u}\}$. Finally, [18, Lemma 2] tells us $\mathbb{E}[e^{tX}] = \mathbb{E}[\prod_u e^{tX_u}] \leq \prod_u \mathbb{E}[e^{tX_u}]$.

Reduction to Dart Game. We reduce the exit node cases to the dart game, similarly to what happens in Simple ORAM. In the dart game, one has an unbounded number of white and black darts. In each round of the game, one throws a white dart and a black dart; each dart independently hits the bullseye with a probability. The game ends when a white dart hits the bullseye. In each

iteration, ap corresponds to a black dart, and ep corresponds to a white dart. Note that the eviction game has N rounds while the dart game has infinitely many iterations. Let $\hat{X}_u$ to be the number of black darts hitting, then $X_u \leq \hat{X}_u$.

Now, we are going to compute the MGF of $\hat{X}_u$, conditioned on the depth of the node, d . Let W be the number of rounds until the first white dart hit. W follows a geometric distribution with success probability $p = 2^{-d}$ and $\Pr[W = i] = (1-p)^i p$. In each of those W rounds, the black dart hits with probability p , which means the number of black dart hits follow a binomial distribution with W trials. Therefore, $\mathbb{E}[e^{t\hat{X}_u} | d] = \sum_{i=0}^{\infty} \Pr[W = i] \cdot \mathbb{E}[e^{t\hat{X}_u} | W = i, d] = 1/(2 - 2^{-d} + (2^{-d} - 1)e^t)$.

Reduction to Balls-and-Bins Game. We reduce the leaf cases to the balls-and-bins game. This game throws n balls independently and uniformly into 2^h bins (note that we have $n = 2^h$).

Let $\hat{X}_u$ to be the number of balls in a bin, then $X_u = \hat{X}_u$.

To compute the MGF of $\hat{X}_u$, note the game follows a binomial distribution with n trials and success probability $p = 2^{-h}$, thus $\mathbb{E}[e^{t\hat{X}_u}] = (1 - p + pe^t)^n \leq e^{e^t - 1}$.

Combining the results. Finally, we consider the global overflow event, which can be decomposed as the existence of a rooted subtree T such that more than $|T| \cdot Z + R$ blocks survive in the eviction game of T . We bound the probability of the overflow event (1) via the union bound over all rooted subtrees (2), a standard moment-generating-function argument plus Markov's inequality (3), and negative association together with stochastic domination (4).

$$\Pr[\exists T. X \geq |T| \cdot Z + R] \quad (1)$$

$$\leq \sum_T \Pr[X \geq |T| \cdot Z + R] \quad (2)$$

$$\leq \sum_T e^{-t(|T| \cdot Z + R)} \cdot \mathbb{E}[e^{tX}] \quad (3)$$

$$\leq e^{-tR} \sum_T \prod_{u \in T} e^{-tZ} \cdot \mathbb{E}[e^{t\hat{X}_u}] \quad (4)$$

To analyze $\sum_T \prod_{u \in T} e^{-tZ} \cdot \mathbb{E}[e^{t\hat{X}_u}]$, we reason recursively over the tree. Let the total weight of subtrees at most as shallow as d as $A(d)$, which satisfies the following recurrences:

$$A(d) = \begin{cases} e^{-tZ} \cdot (1/(2 - e^t + 2^{-(d+1)} \cdot (e^t - 1)) + A(d+1))^2, & 0 \leq d < h \\ e^{-tZ} \cdot e^{e^t - 1}, & d = h \end{cases}$$

The first case says that, if the left subtree of T is empty, then it contributes as an exit node; otherwise, it contributes weights recursively, and the same holds for the right subtree of T . The second case says that, if the node is a leaf, then it contributes as a leaf. Thus, by careful induction, we show that $A(0) \leq 0.37236$. In [32], the analysis applies a union bound over all subtrees and considers the

maximum overflow probability over any subtree, while here the probability of all subtrees can be more precisely summarized.

In the end, by choosing $t = \ln((2^{Z+1} - 8)/2^Z)$, we have the final bound: $0.37236 \cdot (2^Z/(2^{Z+1} - 8))^R$. The paper proof [32] considers the case of $Z = 5$. Our bound is more general and strictly tighter. If we set $Z = 5$, we will get $0.37236 \cdot (4/7)^R$, which is better than $14 \cdot 0.6002^R$.

4.4 Top-level Theorems

The obliviousness distinguishing game and the advantage are the same as in Section 3.6. In the game, the distinguisher provides two programs, and then decides which one is chosen by the experiment. The final obliviousness theorem is stated as follows. The proof is the same as the one for Simple ORAM (Theorem 4).

Theorem 6 (Obliviousness). *Let T be the maximum length of the program, and R be the capacity threshold of the stash. For any adversary $\mathcal{A}$, we have*

$$\text{Adv}_{\text{PathORAM}}^{\text{OBL-IND}}(\mathcal{A}) \leq 2 \cdot 0.37236 \cdot (2^Z/(2^{Z+1} - 8))^R \cdot T.$$

The correctness distinguishing game is also the same as the one in Section 3.6. In the game, the distinguisher tries to tell whether the output—the trace of values read from and written to memory—comes from a normal RAM or an ORAM construction. The final correctness theorem is stated as follows. Again, the proof follows along the same lines as for Simple ORAM (Theorem 5).

Theorem 7 (Correctness). *For any well-defined program of length at most T , and any adversary $\mathcal{A}$, we have*

$$\text{Adv}_{\text{PathORAM,P}}^{\text{COR-IND}}(\mathcal{A}) \leq 0.37236 \cdot (2^Z/(2^{Z+1} - 8))^R \cdot T.$$

5 Recursive Composition and Practice-Oriented Refinement

A standard technique to reduce client-side storage in ORAM constructions is to outsource the position map to the server using another instance of ORAM. Intuitively, one can reduce client storage at level i from n to n/k by introducing a lower level $i - 1$ ORAM supporting n/k addresses and storing k position map values at each ORAM entry. Section 5.1 provides a general framework where we can fit our results for Simple ORAM and Path ORAM as particular cases and then, in Section 5.2, we provide an instantiation that provides a low-level specification for real-world implementations—targeting the pattern used by Signal [20].

5.1 A General Composition Framework

We start by fixing a general interface for ORAM constructions that enables the modular reasoning required by a general composability result. The functionality of the ORAM interface comprises an initialization function and a main function `Access`. The memory access request is unified as read-modify-write, instead of read/write, to ensure that the number of accesses to the remote server does not blow up during recursion.¹² Furthermore, the interface includes also proof-oriented query functions to model composed leakages and global overflow events. We also fix general forms for leakage simulators and overflow bounds for a single access in order to permit packaging the necessary security and correctness properties of the individual oram constructions modularly. An ideal non-oblivious RAM satisfies this interface as a particular case, and serves as the base level for recursion.

The composability result is stated as follows. Consider two instances of the ORAM interface under the assumptions that both are correct and oblivious in the absence of overflow. One serves as the ORAM tree at the current level i and the other serves as the position map (a lower $i - 1$ level ORAM tree). Then:

1. the overflow probability of the composed construction is upper bounded by the sum of the two individual overflow bounds;
2. the obliviousness (up to overflow) of the composition holds, with the natural simulator given by the join of both parts' simulators;
3. the correctness (up to overflow) of the composition holds.

This result holds for any logical memory size n and any compression ratio k , can be instantiated with arbitrary ORAM algorithm instances, and used for an arbitrary number of accesses and arbitrary recursion depths.

5.2 Towards Real-World Implementation

At this point we have introduced all the results that we need to capture the ORAM construction deployed by Signal [20]. To make this claim easier to check, we also carried out a refinement step that permits stating our results for a specification of a 2-level Path ORAM construction whose operations are specified operationally over lists and clearly separating the code for client and server. We believe that such a specification is a good candidate to serve as target for the formal verification of ORAM implementations. The code for client, server and the top-level statements are given in Appendix B.

6 Conclusion and Future Work

We have used the EasyCrypt proof assistant to establish an end-to-end mechanized security result for Simple ORAM and Path ORAM. Our proof is the

¹² Each level of ORAM is able to read and modify the position map stored in lower level ORAMs with one access.

first mechanization to cover the overflow bound in ORAM constructions. There exist multiple directions for future work. The first direction is to mechanize other variants of ORAM, including Circuit ORAM, doubly oblivious ORAM, and hash-based ORAMs, and for more realistic scenarios where server-side memory is encrypted. For the latter, we can benefit from EasyCrypt support for hybrid arguments. The second direction is to connect our mechanized security proofs with implementations, namely that currently used by Signal. This can be done by modelling the implementation in EasyCrypt and proving that our model captures the behavior and leakage of the implementation.

References

1. Almeida, J.B., Arranz-Olmos, S., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Léchenet, J., Low, C., Oliveira, T., Pacheco, H., Quaresma, M., Schwabe, P., Strub, P.: Formally verifying kyber - episode V: machine-checked IND-CCA security and correctness of ML-KEM in easycrypt. In: Reyzin, L., Stebila, D. (eds.) *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference*, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part II. pp. 384–421. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-68379-4_12, https://doi.org/10.1007/978-3-031-68379-4_12
2. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: Optimal Oblivious RAM. *Journal of the ACM* **70**(1), 1–70 (Feb 2023). <https://doi.org/10.1145/3566049>, <https://dl.acm.org/doi/10.1145/3566049>
3. Avanzini, M., Barthe, G., Grégoire, B., Moser, G., Vanoni, G.: Hopping proofs of expectation-based properties: Applications to skiplists and security proofs. *Proc. ACM Program. Lang.* **8**(OOPSLA1) (Apr 2024). <https://doi.org/10.1145/3649839>, <https://doi.org/10.1145/3649839>
4. Bahi, A., Minaud, B., Moataz, T.: Resizable oblivious ram. In: *Advances in Cryptology – EUROCRYPT 2026: 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Rome, Italy, May 10–14, 2026, Proceedings, Part V. p. 273–302. Springer-Verlag, Berlin, Heidelberg (2026). https://doi.org/10.1007/978-3-032-25330-9_10, https://doi.org/10.1007/978-3-032-25330-9_10
5. Bao, J., Docherty, S., Hsu, J., Silva, A.: A bunched logic for conditional independence. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. pp. 1–14. IEEE (2021). <https://doi.org/10.1109/LICS52264.2021.9470712>, <https://doi.org/10.1109/LICS52264.2021.9470712>
6. Bao, J., D’Osualdo, E., Farzan, A.: Bluebell: An alliance of relational lifting and independence for probabilistic reasoning. *Proc. ACM Program. Lang.* **9**(POPL), 1719–1749 (2025). <https://doi.org/10.1145/3704894>, <https://doi.org/10.1145/3704894>
7. Bao, J., Gaboardi, M., Hsu, J., Tassarotti, J.: A separation logic for negative dependence. *Proc. ACM Program. Lang.* **6**(POPL), 1–29 (2022). <https://doi.org/10.1145/3498719>, <https://doi.org/10.1145/3498719>
8. Barbosa, M., Barthe, G., Doczkal, C., Don, J., Fehr, S., Grégoire, B., Huang, Y., Hülsing, A., Lee, Y., Wu, X.: Fixing and mechanizing the security proof

- of fiat-shamir with aborts and dilithium. In: Handschuh, H., Lysyanskaya, A. (eds.) *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference*, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V. pp. 358–389. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-38554-4_12, https://doi.org/10.1007/978-3-031-38554-4_12
9. Barbosa, M., Dupressoir, F., Grégoire, B., Hülsing, A., Meijers, M., Strub, P.: Machine-checked security for rxmxss as in RFC 8391 and $\mathsf{SPHINCS}^+$. In: Handschuh, H., Lysyanskaya, A. (eds.) *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference*, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V. pp. 421–454. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-38554-4_14, https://doi.org/10.1007/978-3-031-38554-4_14
 10. Barthe, G., Grégoire, B., Heraud, S., Zanella-Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. pp. 71–90. Lecture Notes in Computer Science, Springer (2011). https://doi.org/10.1007/978-3-642-22792-9_5, https://doi.org/10.1007/978-3-642-22792-9_5
 11. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. *SIGPLAN Not.* **44**(1), 90–101 (Jan 2009). <https://doi.org/10.1145/1594834.1480894>, <https://doi.org/10.1145/1594834.1480894>
 12. Barthe, G., Hsu, J., Liao, K.: A probabilistic separation logic. *Proc. ACM Program. Lang.* **4**(POPL), 55:1–55:30 (2020). <https://doi.org/10.1145/3371123>, <https://doi.org/10.1145/3371123>
 13. Boyle, E., Chung, K., Pass, R.: Oblivious parallel RAM and applications. In: Kushilevitz, E., Malkin, T. (eds.) *Theory of Cryptography - 13th International Conference, TCC 2016-A*, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9563, pp. 175–204. Springer (2016). https://doi.org/10.1007/978-3-662-49099-0_7, https://doi.org/10.1007/978-3-662-49099-0_7
 14. Cash, D., Drucker, A., Hoover, A.: A lower bound for one-round oblivious RAM. In: Pass, R., Pietrzak, K. (eds.) *Theory of Cryptography - 18th International Conference, TCC 2020*, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12550, pp. 457–485. Springer (2020). https://doi.org/10.1007/978-3-030-64375-1_16, https://doi.org/10.1007/978-3-030-64375-1_16
 15. Chung, K.M., Pass, R.: A simple ORAM. *Cryptology ePrint Archive*, Paper 2013/243 (2013), <https://eprint.iacr.org/2013/243>
 16. Darais, D., Sweet, I., Liu, C., Hicks, M.: A language for probabilistically oblivious computation. *Proc. ACM Program. Lang.* **4**(POPL), 50:1–50:31 (2020). <https://doi.org/10.1145/3371118>, <https://doi.org/10.1145/3371118>
 17. Davies, G.T., Janson, C., Martin, D.P.: Client-oblivious opram. In: *Information and Communications Security: 22nd International Conference, ICICS 2020*, Copenhagen, Denmark, August 24–26, 2020, Proceedings. p. 312–330. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-61078-4_18, https://doi.org/10.1007/978-3-030-61078-4_18
 18. Dubhashi, D., Ranjan, D.: Balls and bins: a study in negative dependence. *Random Struct. Algorithms* **13**(2), 99–124 (Sep 1998)

19. Garg, S., Mohassel, P., Papamanthou, C.: Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In: Proceedings, Part III, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9816. p. 563–592. Springer-Verlag, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_20, https://doi.org/10.1007/978-3-662-53015-3_20
20. gconnell: Technology deep dive: Building a faster ORAM layer for enclaves. Signal Blog (Aug 2022), <https://signal.org/blog/building-faster-oram/>, published August 19, 2022. Accessed April 12, 2026
21. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. p. 182–194. STOC '87, Association for Computing Machinery, New York, NY, USA (1987). <https://doi.org/10.1145/28395.28416>, <https://doi.org/10.1145/28395.28416>
22. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. *J. ACM* **43**(3), 431–473 (May 1996). <https://doi.org/10.1145/233551.233553>, <https://doi.org/10.1145/233551.233553>
23. Heath, D., Kolesnikov, V., Ostrovsky, R.: Epigram: Practical garbled RAM. In: Dunkelman, O., Dziembowski, S. (eds.) *Advances in Cryptology - EURO-CRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I. *Lecture Notes in Computer Science*, vol. 13275, pp. 3–33. Springer (2022). https://doi.org/10.1007/978-3-031-06944-4_1, https://doi.org/10.1007/978-3-031-06944-4_1
24. Lago, U.D., Davoli, D., Kapron, B.M.: On separation logic, computational independence, and pseudorandomness. In: 37th IEEE Computer Security Foundations Symposium, CSF 2024, Enschede, Netherlands, July 8-12, 2024. pp. 80–95. IEEE (2024). <https://doi.org/10.1109/CSF61375.2024.00040>, <https://doi.org/10.1109/CSF61375.2024.00040>
25. Li, J.M., Ahmed, A., Holtzen, S.: Lilac: A modal separation logic for conditional probability. *Proc. ACM Program. Lang.* **7**(PLDI), 148–171 (2023). <https://doi.org/10.1145/3591226>, <https://doi.org/10.1145/3591226>
26. Li, J.M., Aytac, J., Johnson-Freyd, P., Ahmed, A., Holtzen, S.: A nominal approach to probabilistic separation logic. In: Sobocinski, P., Lago, U.D., Esparza, J. (eds.) *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024*. pp. 55:1–55:14. ACM (2024). <https://doi.org/10.1145/3661814.3662135>, <https://doi.org/10.1145/3661814.3662135>
27. Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Maliciously secure multi-client ORAM. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings. Lecture Notes in Computer Science*, vol. 10355, pp. 645–664. Springer (2017). https://doi.org/10.1007/978-3-319-61204-1_32, https://doi.org/10.1007/978-3-319-61204-1_32
28. Miao, P.: Cut-and-choose for garbled ram. In: *Topics in Cryptology – CT-RSA 2020: The Cryptographers’ Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*. p. 610–637. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-40186-3_26, https://doi.org/10.1007/978-3-030-40186-3_26

29. Patel, S., Persiano, G., Seo, J., Yeo, K.: Latoram: Orams from lateral stashes and delayed shuffling. In: 2026 IEEE Symposium on Security and Privacy (SP). pp. 1483–1502. IEEE (05 2026). <https://doi.org/10.1109/SP63933.2026.00075>
30. Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious RAM. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 415–430. USENIX Association (2015), <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-ling>
31. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology – ASIACRYPT 2011. pp. 197–214. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
32. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. p. 299–310. CCS ’13, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2508859.2516660>, <https://doi.org/10.1145/2508859.2516660>
33. Wang, X., Chan, H., Shi, E.: Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 850–861. ACM, Denver Colorado USA (Oct 2015). <https://doi.org/10.1145/2810103.2813634>, <https://dl.acm.org/doi/10.1145/2810103.2813634>
34. Yan, P., Murray, T., Ohrimenko, O., Pham, V., Sison, R.: Combining classical and probabilistic independence reasoning to verify the security of oblivious algorithms. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I. pp. 188–205. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-71162-6_10, https://doi.org/10.1007/978-3-031-71162-6_10
35. Zhou, M., Shi, E., Chan, T.H., Maimon, S.: A theory of composition for differential obliviousness. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14006, pp. 3–34. Springer (2023). https://doi.org/10.1007/978-3-031-30620-4_1, https://doi.org/10.1007/978-3-031-30620-4_1

A Refining the Simple ORAM Model

A natural question to ask is whether the model over which we obtain our results for ORAM can realistically be used, in future work, to establish a connection to ORAM implementations. We justify our claim by establishing a connection to a refined model of tree-based ORAM that clearly separates client-side and server-side states: the server-side tree is represented as a map from nodes to a list of blocks, and the client-side position map is represented as a finite partial mapping from logical address to paths in the tree.

$$\text{coram} := \{ \text{posmap} : \text{addr} \rightarrow \text{path}; \text{tree} : \text{node} \rightarrow (\text{addr} \times \text{path} \times \text{value}) \text{ list} \}.$$

We formally verified that the bounded model upon this state is equivalent to our model in Section 3: the two models are equivalent: their observable behavior is identical (leakage and semantics) up to overflow, and the probability of overflow is also identical in both models. With this equivalence, the top-level properties we established for Simple ORAM, almost-obliviousness and almost-correctness can be carried over naturally to this more refined model.

The proof relies on establishing the following relational invariant in lockstep: 1) two oram states coincide (containing same set of addresses, where the same address is associated with the same path and value); 2) in the concrete oram state, tree and position map are consistent; 3) the tree in the concrete oram state is well-formed as a organization of memory, e.g., the blocks on the tree are unique with respect to their addresses.

B Path ORAM Construction Capturing Signal's Deployment

Here we describe the Path ORAM construction capturing the implementation deployed by Signal. The target is a two-level recursive Path ORAM, which matches the shape of Signal's implementation: the first level is of height 19, the second level is of height 12, and the base position map is of size 2048.

Figure 16 gives the general recursion combinator, the ideal RAM, and a client/server split of the Path ORAM construction.

The two-level instance is defined as

$\text{RecPathORAM} := \text{RecursiveORAM}(\text{Client}_1, \text{RecursiveORAM}(\text{Client}_2, \text{IdealRAM})),$

where $h_1 = 19$ and $h_2 = 12$.

Now, we can give the top-level theorems as follows.

Theorem 8 (Correctness of the Two-level Path ORAM). *Let T be the maximum length of the program, $Z \geq 5$ the bucket capacity and R the stash bound. For any program P and any adversary $\mathcal{A}$,*

$$\text{Adv}_{\text{RecPathORAM}, P}^{\text{COR-IND}}(\mathcal{A}) \leq 2 \cdot 0.37236 \cdot \left(2^Z / (2^{Z+1} - 8)\right)^R \cdot T.$$

Theorem 9 (Obliviousness of the Two-level Path ORAM). *Let T be the maximum length of the program, $Z \geq 5$ the bucket capacity and R the stash bound. For any program P and any adversary $\mathcal{A}$,*

$$\text{Adv}_{\text{RecPathORAM}}^{\text{OBL-IND}}(\mathcal{A}) \leq 4 \cdot 0.37236 \cdot \left(2^Z / (2^{Z+1} - 8)\right)^R \cdot T.$$

RecursiveORAM(B, PM).Access(a, f)	Server.Get(u)
1: $rp \xleftarrow{\$} (\{0, 1\}^h)^X$; $np \xleftarrow{\$} \{0, 1\}^h$ 2: $g \leftarrow \lambda e$. 3: if $e = \perp$ then $rp[a \bmod X \mapsto np]$ 4: else $e[a \bmod X \mapsto np]$ 5: $e \leftarrow \text{PM.Access}(a/X, g)$ 6: if $e = \perp$ then $e \leftarrow rp$ 7: $p \leftarrow e[a \bmod X]$ 8: return B.Access(a, f, p, np)	1: $L \leftarrow (\ell, \text{Get}, u) :: L$ 2: return $tree[u]$
	Server.Put(u, b)
	1: $tree[u] \leftarrow b$ 2: $L \leftarrow (\ell, \text{Put}, u) :: L$
IdealRAM.Access(a, f)	Client.Access(a, f, p, np)
1: $v \leftarrow mem[a]$; $mem[a] \leftarrow f(v)$ 2: return v	1: for $d = 0$ to h do 2: $stash \leftarrow stash \cup \text{Server.Get}(p[0..d])$ 3: $v \leftarrow \perp$ 4: if $a \in \text{dom } stash$ then 5: $v \leftarrow stash[a].\text{value}$ 6: $stash[a] \leftarrow (\text{path}: np, \text{value}: f(v))$ 7: for $d = h$ downto 0 do 8: $u \leftarrow p[0..d]$ 9: $b \leftarrow \text{select}_Z\{r \in stash \mid u \sqsubseteq r.\text{path}\}$ 10: $stash \leftarrow stash \setminus b$; $\text{Server.Put}(u, b)$ 11: $overflow \leftarrow overflow \vee stash > R$ 12: return v

Fig. 16: The two-level Path ORAM construction. The main procedure is **Access**, which accepts an address a and a modifier f . Top-left: the combinator for ORAM recursion, **RecursiveORAM(B, PM)**, which accepts two ORAM constructions—**B** serves as the ORAM tree at this level and **PM** serves as the position map. Bottom-left: the ideal RAM. Right: a Path ORAM construction with an explicit client/server split, at recursion level ℓ .