Towards Formally Verified Path ORAM in Coq

Hannah Leung, Talia Ringer, Christopher W. Fletcher
University of Illinois Urbana-Champaign
Urbana, Illinois, USA

Abstract

Oblivious RAM is a class of randomized algorithms that break the association between a program's memory access pattern and that program's data. Path Oblivious RAM is a specific ORAM algorithm that is both theoretically interesting and practically efficient. This abstract describes our plans to verify Path ORAM in Coq, and discusses the design decisions and tradeoffs involved.

1 Introduction

Suppose that you are storing your private data/program in a remote untrusted server. Ideally, this data would remain private. In theory, encryption schemes can help with this. In practice, the server's *memory access patterns*—the sequences of memory locations that it visits—can betray your privacy, leaking secret data.

For example, with the following code snippet, the attacker can transmit the secret by dereferencing the pointer that points to the value.

```
*secret = value ;
```

Oblivious RAM (ORAM) solves this data leakage problem by constantly shuffling the contents of the (encrypted) memory and (re)encrypting data as it is accessed. The security goal is to make any two memory access patterns computationally indistinguishable, which means that given any two access patterns of the same length, there is a negligible probability for a polynomial attacker to distinguish them better than random guessing.

This abstract describes our plans to formally verify one implementation of ORAM: Path ORAM [Stefanov et al. 2013]. It describes how Path ORAM works (Section 2), as well as the key functional correctness and security properties we plan to prove about it (Section 3). It concludes with a discussion of the challenges we anticipate, in hopes to solicit feedback from the proof engineering community (Section 4).

2 Path ORAM

The trivial ORAM scans memory to perform each access, performing a linear amount of dummy work in the size of the memory. This incurs too much overhead to be practical. Modern ORAM algorithms seek to make the dummy work per access poly logarithmic in the memory size, making ORAM more practical.

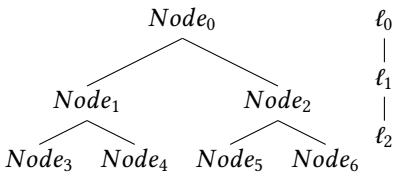
Relative to other implementations of ORAM, Path ORAM is simple and efficient, making it both useful and amenable to verification. Path ORAM has a standard RAM interface – at an abstract level, either *read(address, size)* or *write(address, data, size)*. However, we further abstract away the details and give it a uniform interface – $Access(op, addr, data^*)$.

The algorithm for Path ORAM takes as input an operation (read or write), a block ID, new data, a position map, a stash, and the memory. It modifies the memory and, if the operation is "write," it writes back the new data in the ORAM and returns the old data associated with the block ID.

Path ORAM treats the memory on the remote server as a binary tree (for simplicity, we assume it is a perfect binary tree parameterized by its height ℓ). Each node (*bucket*), in the tree has capacity for Z *blocks*, and each has the capacity for B bits of data (the *block size*). The algorithm temporarily stores some blocks in a client-side stash S. It also maintains a map from blocks to positions in the tree, called the *position map*. $\mathcal{P}(x, \ell)$ denotes node along path from leaf node x, at level ℓ.

For each memory access, Path ORAM maintains a key invariant: at any time, the position map maps each block to a uniformly random leaf node in the tree, and if the block does not live in the stash, it is guaranteed to map to nodes along the path to the mapped leaf node. When there is an access (e.g., reading a block from the server), the algorithm reads all the nodes along the path into the stash, then assigns the block to a different leaf node from a uniform distribution, then finally writes back blocks from the stash back to the memory.

Algorithm 1 contains the pseudo code for the algorithm (taken from the original Path ORAM paper [Stefanov et al. 2013]), which we walk through with a concrete example.



Imagine our remote server looks like the binary tree above, and to begin with, each of the nodes contains no blocks. When we issue a request $access(op = \text{`}wr\text{''}, addr = 11, data^* = \text{``}dataNew\text{''})$ to the server, Line 1 reads the leaf that Block ID a maps to in the position map. In the example, assume that $block_{11}$ maps to $Node_4$ in the tree, then x in $P(x, \ell)$ will be 4. Line 2 remaps this block to another leaf node in the tree, e.g., to Node6, so we can assume it remaps to $Node_6$.

Line 4 reads all the blocks in each node along the path determined by a given leaf node ($Node_4$ in this case) into the Stash S. S will contain all blocks in nodes $Node_4$, $Node_1$, and

$Node_0$. Line 8 updates the data associated block a in S. The value of block a will be updated to $dataNew$.

Lines 10 - 15 write back as many blocks as possible from S to the remote server while upholding the placement invariant. Starting at the leaf level, we find blocks that satisfy the following criteria: at a given level ℓ, the node that a given block maps to shares the same ancestor with the original leaf node also at level ℓ ($Node_4$ in this case). If another block also maps to $Node_4$ at level ℓ_2, then this block can be written back to $Node_4$ if there is still space in the node. If another block maps to $Node_3$, then the first possible node it can be written to is $Node_1$ at ℓ_1. If there is no space in $Node_1$, we come to check ℓ_0, $Node_0$ can take the block if there still is space left.

Blocks are kept in the stash iff there is no room in them along any node from the root to their assigned leaf. In this fashion, the stash can conceptually be thought of as an extension of the root node.

Algorithm 1 Access(op, addr, data*)

1: $x \leftarrow position[a]$
2: $position[a] \leftarrow UniformRandom(0...2^L - 1)$
3: **for** $\ell \in \{0, 1, ..., L\}$ **do**
4: $\quad S \leftarrow S \cup ReadBucket(\mathcal{P}(x, \ell))$
5: **end for**
6: data $\leftarrow$ Read block a from S
7: **if** $op = write$ **then**
8: $\quad S \leftarrow (S - \{(a, data)\}) \cup \{(a, data^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, ..., 0\}$ **do**
11: $\quad S' \leftarrow \{(a', data') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(position[a'], \ell)\}$
12: $\quad S' \leftarrow Selectmin(|S'|, Z)$ blocks from S'
13: $\quad S \leftarrow S - S'$
14: $\quad WriteBucket(\mathcal{P}(x, \ell), S')$
15: **end for**
16: **return** data

3 Verifying Path ORAM

We give the formal specification of the functional correctness of Algorithm 1 and what it means for Path ORAM to be secure.

Functional Correctness. We plan to verify that the ORAM interface behaves like a RAM from the client's perspective. That is, an ORAM client read operation to some address a, i.e., $access(\text{“}rd\text{”}, a, \perp)$, should return the data most recently written to that address, i.e., through $access(\text{“}wr\text{”}, a, data)$.

Security. The key security property we plan to prove is that, given any two client access patterns of the same length, there is a negligible probability for the attacker to distinguish them. This leaves no room for the attacker to learn the association of physical locations and data on the server. Formally, let $A(\vec{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\vec{y}$. An ORAM construction is said to be secure if (1) for any two data request sequences $\vec{y}$ and $\vec{z}$ of the same length, their access patterns $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable by anyone but the client [Stefanov et al. 2013].

There are two possible ways to approach the proof of security: (1) show the pseudo-code in Algorithm 1 is functionally correct and guarantees that memory access patterns are computationally indistinguishable, or (2) use traditional mathematical complexity analysis to demonstrate that the stash $S's$ capacity is bounded to never exceed $O(\log N)$ blocks, where N is the number of blocks stored in the ORAM tree. These two approaches essentially prove the same thing, however, they are complementary.

Proof Engineering. We are just getting started on the proof engineering effort. Good proof engineering hinges on identifying the right invariants and lemmas. Along these lines, we have started by writing an informal implementation[1] in Python, breaking down the code into units that correspond to lemmas and invariants that we anticipate needing. We expect this to make the code more amenable to verification when we move to formal proof. Python modeling has served us in many ways. For example, doing it right in Python helps us with building a comprehensive understanding of the nitty-gritty details of the Path ORAM algorithm. It also helps us with rapid testing of functions and fast prototyping of Algorithm 1.

Some of the invariants and lemmas that we plan to prove come from the Path ORAM paper. For example, the paper states a system invariant: at any time, each block should be mapped to a leaf node, and unstashed blocks should always be placed in a node along the path from the mapped leaf to the root node. The paper also identifies some lemmas, like the bounded stash size lemma: the size of the stash S should be at most $O\log(N)$, with N denoting the total number of blocks.

We are also identifying other invariants and lemmas that we anticipate helping us. Building the Python model in an interactive playground gives us opportunity to spot these intermediate invariants, which corresponds to the form of lemmas and theorems in Coq. Throughout our informal development process, we are generalizing our unit tests into parameterized tests, then documenting natural language invariants, which we hope to later formalize. We also anticipate making heavy use of tree lemmas to simplify the proof effort, since the Path ORAM algorithm makes heavy use of tree algorithms. We appreciate feedback on other invariants and lemmas that may help us prove these theorems, and on how to best state our theorems to reduce the overall proof engineering effort.

4 Next Steps

As we move to a formally verified implementation in Coq, we hope to solicit feedback from the proof engineering community on four important challenges we foresee:

[1] https://github.com/PalindromeLeung/PathORAM

Proving complexity. Several of the informal proofs rely on complexity results. We are aware of one framework that may help us with verifying complexity in Coq [McCarthy et al. 2018], but we know that verification of intensional properties is challenging. Should we prove the complexity results in Coq, or push them out to traditional complexity analysis? And how can we make it easy to change our minds?

Proof reuse. We would prefer to reuse existing libraries and frameworks when possible and to build something that is itself reusable. What libraries and frameworks should we use? And what libraries and frameworks may we wish to build ourselves that may prove useful for other proof engineers?

Reasoning with probability. Our definition of security relies heavily on computational indistinguishability, or equivalently, the probability distribution at its core. Prior work on building libraries and frameworks, such as *The Foundational Cryptography Framework* [Petcher and Morrisett 2015] and *Formal Certification of Code-Based Cryptographic Proofs* [Barthe et al. 2009] for reasoning about probabilities in Coq can be adopted to build infrastructure toward this verification work. Work on embedding probabilistic oblivious computation in lambda calculus not only allow reasoning about deterministic programs but also probabilistic ones [Darais et al. 2020]. SSProve [Haselwarter et al. 2021] is another thread of work that is equipped with a probabilistic relational program logic for formalizing low-level details of constructing game-based cryptographic proofs in Coq, which might give us insights on how to reason with probability in Coq. Other work [Barthe et al. 2019] enhanced separation logic with probabilistic reasoning and demonstrated how this logic system applied to Oblivious RAM. How do we tackle the challenge of reasoning with probability specifically in the context of proving security properties of Path ORAM? How much could we generalize it for it to be useful for other security verification work?

Verified hardware. We would eventually like to verify not just the Path ORAM algorithm, but a hardware implementation in RTL. What is the best way to accomplish this? How do we ensure that what we get is performant? Will prior work for verification of hardware in Coq (like Kami [Choi et al. 2017]) help us with this? And what design principles can we use to make it easier to start by verifying just the algorithm, and then incrementally move toward performant, verified hardware?

Our talk will solicit feedback on these four challenges, share our progress on the verification effort over the upcoming months, and open the floor for discussion.

References

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Savannah, GA, USA) (POPL '09). Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/1480881.1480894

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A Probabilistic Separation Logic. Proc. ACM Program. Lang. 4, POPL, Article 55 (dec 2019), 30 pages. https://doi.org/10.1145/3371123

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and its Modular Verification. In ICFP'17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming. http://adam.chlipala.net/papers/KamiICFP17/

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2020. A Language for Probabilistically Oblivious Computation. In Proceedings of the ACM Conference on Principles of Programming Languages (POPL).

Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. Cryptology ePrint Archive, Paper 2021/397. https://eprint.iacr.org/2021/397 https://eprint.iacr.org/2021/397.

Jay McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. 2018. A Coq library for internal verification of running-times. Science of Computer Programming 164 (2018), 49–65. https://doi.org/10.1016/j.scico.2017.05.001 Special issue of selected papers from FLOPS 2016.

Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In Principles of Security and Trust, Riccardo Focardi and Andrew Myers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–72.

Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path O-RAM: An Extremely Simple Oblivious RAM Protocol. In Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS '13) (2013).