EFFICIENT VERIFICATION OF UNTRUSTED SERVICES

by

Ioanna Tzialla

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NEW YORK UNIVERSITY

SEPTEMBER, 2022

Professor Michael Walfish

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Michael Walfish. Without him, this dissertation would literally not exist. I am forever grateful to him for accepting me as a PhD student six years ago, and for his unwavering patience, support, and encouragement ever since joining his group. Thoughout my PhD, he was always looking out for me, helping me develop new skills and teaching me how to produce high-quality work while giving me the freedom to pursue my own interests and ideas. He has deeply influenced the way I think and work and helped me grow both professionally and personally.

This dissertation also owes its existence to Srinath Setty and Aurojit Panda. They were both second advisors to me and I am extremely grateful to them for all of their help, support, and advice the past years. Working with them has always been both fun and educational. I deeply appreciate their kindness and patience, even when I was making rookie mistakes; they have both profoundly impacted me, in their own way, and changed how I think about solving problems.

I am thankful to my committee members, Jinyang Lee and Joseph Bonneau, for their great feedback on my proposal, defense, and previous drafts of this dissertation, and for everything they have taught me throughout the years in the PhD. I would also like to thank Oded Regev for his help and support during some particularly challenging times.

I am also indebted to my collaborators: Sebastian Angel, Abhiram Kothapalli, Bryan Parno, abhi shelat, Justin Thaler, Riad S. Wahby, Jeffery Wang, and Jenny Zhu. Collaborating with them has been both a pleasure and a great honor, and I have learned a lot from them. Abhiram Kothapalli,

Bryan Parno, and Jenny Zhu worked with me on Verdict and Karousos, and their contributions to these works were invaluable.

I am grateful to all my friends from New York and Greece that have been by my side throughout this journey and made it much more enjoyable.

Last, I would like to thank my family for always being on my side, tolerating me, and supporting me all these years.

ABSTRACT

Using a third-party service today requires trusting that it is executing as promised. Meanwhile, the correct execution of services is regularly impeded by failures, bugs, misconfigurations, operational mistakes, and insider attacks. Is it possible to *verify*, instead of trust, that a third-party service executes correctly?

This dissertation studies this question for two services that execute on remote servers: transparency dictionaries, a foundational infrastructure for end-to-end encryption and other applications, and event-driven web applications. For each of these two services, we leverage their workloads to introduce a *practical* system that allows a verifier to get a *strong security* guarantee that the service executes correctly.

In the case of a transparency dictionary, this guarantee is in the form of a cryptographic proof provided by the service. Producing cryptographic proofs typically requires high resource costs. This dissertation shows that tailoring the cryptographic tools used by the transparency dictionary for its use case mitigates these costs and results in a system, Verdict, that scales to dictionaries with millions of entries while imposing modest overheads on the service and its clients.

In the case of outsourced event-driven web applications, the verifier gets the required guarantee by replaying the requests on a trusted machine using Karousos, a novel record-replay system in which the service has the role of the untrusted recorder. Karousos takes advantage of the particular characteristics of event-driven web applications to enable the replayer (the verifier) to use less computational resources than the recorder (the service), while imposing tolerable overheads on the

recorder and keeping communication small.

CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

1 | INTRODUCTION

How can we gain assurance that an untrusted service operates as promised?

This question is becoming increasingly relevant as services provided by third parties are becoming an integral part of our personal and professional lives. Examples of such services include authentication, cloud computing, messaging, and software distribution. The convenience of these services makes them appealing to their users. However, users may not always want to trust that a service operates as promised: the service might not execute correctly due to internal problems such as insider attacks [91, 130], bugs [80], misconfigurations [4], unexpected failures [20], and operational mistakes [11]. Moreover, the service's incentives may not always align with those of its clients.

The goal of this dissertation is to develop tools that allow users to detect when a service is not executing as promised. The problem is *execution integrity*: giving a verifier assurance that a service is executing according to a given program. This problem is distinct from but complementary to the problem of program verification, which is about ensuring that a program meets a given specification.

There is a lot of prior work on this problem (§2.1). However, many previous solutions make strong assumptions or impose high resource costs on the service or the service's auditors. In contrast, we restrict our attention to systems that solve the problem while having the following properties:

1. *Strong Model*: The security of the system should not rely on any assumptions about the service's failure modes nor require any trusted components in the service's software and hardware stack. However, we do assume that the service is computationally bounded and cannot break

standard cryptographic assumptions.

2. *Provable Security*: The system's security properties should be formally defined and proved. Although security proofs do not provide the certainty of traditional mathematical proofs [94], we consider them important for two reasons: First, the process of writing rigorous proofs leads the developer to better understand their system's properties and, more importantly, it reveals security problems. Second, a formal proof of a claim is essential to its validation: it is considerably more difficult to confirm or refute a statement without knowing the arguments that corroborate it.
3. *Practicality*: Systems should be able to actually run: the audit procedure should impose reasonable overheads on the service, the amount of communication between the service and the auditor should be small, and the audit procedure should be less expensive than the execution at the service.

There are two classes of systems in the literature (§2.1) that provide the above security properties while targeting general services. However, both achieve practicality only under certain conditions.

The first class comprises systems that employ probabilistic proofs [28, 29, 43, 72, 77, 78, 89, 117]. These are protocols that allow a prover to provide a proof to a verifier that they executed a computation correctly. State-of-the-art probabilistic proof systems provide strong security guarantees and admit proof sizes and verification times that are sub-linear in the size of the computation. However, proof generation generally imposes intolerable overheads on the service: it requires explicitly representing the computation as a static circuit by translating every operation that the computation does to a number of boolean or arithmetic gates, causing time to turn into space (with bad constants); then, the time to generate a proof scales with the size of this circuit. As a result of high overheads, the only computations for which this machinery is practical are those that have small circuit representations.

The second class of systems that can meet our desiderata is inspired by Orochi [146]. In these systems, verifying that the service executes some workload correctly involves re-executing the

workload on a trusted executor. This executor is assumed to execute each operation of the program according to its specification. As opposed to systems that employ probabilistic proofs, these systems relieve the service and the verifier from the burden of encoding the execution at the service as a flow through a general purpose circuit. Thus, they are able to satisfy the desired security properties presented above and impose small overheads on the service, even when the service is executing legacy programs. However, execution at the verifier requires less computational resources than the service only if there is redundancy in the execution at the service. In this case, the verifier is able to save work by deduplicating the execution of identical instructions during re-execution.

In this dissertation, we target two important services. The first service is a transparency dictionary [51, 53, 85, 115, 150]: an untrusted service that maintains a mapping from labels[1] to values that users can query and update. Securing transparency dictionaries is important because they have been proposed as a foundation for several security-critical applications such as PKIs, end-to-end encryption (for example, for email or messaging), and software updates. The second service that we target is an outsourced event-driven web application. This problem is motivated by scenarios where people run their applications on remote rented servers such as cloud providers. In particular, we are interested in event-driven web applications due to the popularity that event-driven frameworks such as Node.js [14] have gained in the area of web development in recent years.

These two services have different program and workload characteristics, which lead us to develop two very different systems: Verdict, a system that employs probabilistic proofs to enable efficient auditing of transparency dictionaries, and Karousos, an Orochi-inspired system that audits outsourced event-driven web applications.

Verdict. In a nutshell, Verdict is a transparency dictionary that creates proofs attesting that it executes all updates and queries correctly. A transparency dictionary executes updates correctly with respect to an application-dependent function F when it only updates each value associated

[1] A transparency dictionary is similar to a key-value store, but we avoid using key, since the term has many other meanings in the cryptography context.

with a label by applying this function. Moreover, the dictionary executes queries correctly when it replies to each query with the value associated with the requested label in the dictionary's state.

Verdict's update proofs are *publicly verifiable*, meaning that each client can check these proofs to get assurance that the dictionary updates its whole state by only applying the function F. Verdict produces these proofs using probabilistic proofs, yet is able to impose modest proving overheads on the service.

Verdict owes its efficiency to a novel cryptographic accumulator [35] – a commitment scheme with support for proving membership of an element against a commitment. The dictionary uses this accumulator to produce a commitment to its state. This commitment is published on a public timeline (for example, a public blockchain [151]). Our accumulator yields efficient ($O(\log n)$ where n is the size of the dictionary) proofs of membership and non-membership, which the dictionary uses to demonstrate correct execution of queries without employing a probabilistic proof system. Meanwhile, the accumulator is designed so that correct execution of updates can be efficiently represented as a circuit ($\approx$ 25k gates per update for dictionaries with one billion labels).

Verdict also owes its efficiency to Phalanx, a novel probabilistic proofs system that is optimized for Verdict's workloads. Similar to prior work [5, 51, 115] in Verdict, the dictionary does not execute updates one by one but, instead, works in epochs: during each epoch, the service buffers all updates that the users request and, at the end of the epoch, it applies them to its state and produces a proof using Phalanx that it applied all updates correctly. Phalanx is a Succinct Non-interactive Argument of Knowledge (SNARK) [40, 73, 74], a special type of probabilistic proof system that yields proofs that are static strings (non-interactive) and have size sub-linear in the size of the computation (succinct). Phalanx leverages the epoch-based nature of Verdict to admit amortized constant-sized proofs and verification times. Furthermore, Phalanx leverages the data-parallel nature of the statement proven in each epoch to reduce proof-generation costs by over an order of magnitude compared to prior systems [135].

We implement Verdict and show its practicality by evaluating it against state-of-the-art sys-

tems with similar security properties. Our evaluation shows that Verdict scales to dictionaries with millions of labels while imposing modest overheads on the service and clients.

Karousos. Karousos is a system that aims to audit event-driven web applications running on remote servers. It allows a verifier that has the trace of requests to and responses from the server (see prior work [26, 146] for ways to obtain such a trace) to verify that the remote server operates correctly, meaning that executing the event-driven application on the inputs (the requests in the trace) produces the alleged outputs (the responses in the trace). The verifier's audit relies on untrusted advice that the server sends to the verifier to help the verifier re-execute correctly and efficiently, meaning using less computational resources than the server. Efficiency at the verifier is achieved by using *SIMD-on-demand* [146]: the verifier re-executes the requests in an accelerated manner by batching the execution of suitable requests and deduplicating the execution of identical instructions across each batch.

An essential question that this dissertation studies is at what granularity the verifier should batch requests. We observe that there is there is a tradeoff between re-execution throughput and advice size, and each batching granularity corresponds to a point in the tradeoff curve. Karousos identifies a point that exposes many deduplication opportunities while keeping the advice small: the verifier batches the execution of requests that have the same tree of events regardless of the order in which event handlers are executed at the server. To identify this point we rely on an essential observation: if during execution two operations are guaranteed to be re-executed in the same order, then these operations can in principle be re-executed at the verifier correctly. Karousos leverages this observation to introduce a novel record-replay algorithm in which the server dynamically decides whether to log operations depending on whether the replayer could re-execute them out-of-order.

Additionally, Karousos introduces mechanisms to rule out misbehavior by an untrusted server. An audit procedure in which the verifier merely re-executes the requests based on the advice and checks the produced responses against the given trace could be exploited by a misbehaving

server, leading to the verifier accepting physically impossible executions. Karousos handles server misbehavior with several techniques for inferring edges (representing ordering) in a graph of alleged events, and then checking whether this graph is acyclic.

Other than supporting event-driven applications, Karousos makes some first steps in handling weakly consistent external shared state. Prior work in this area [26, 27, 146], assumes that external shared state such as databases and KV stores are strictly serializable [123]. However, in practice many databases default to weak isolation levels and some don't even offer strict serializability [30]. Specifically, Karousos is able to handle transactional KV stores whose isolation level is serializability, read committed, or read uncommitted. The main challenge that Karousos has to overcome is that existing algorithms for testing isolation assume that the verifier knows the actual history of execution at the KV store. This assumption does not hold in the context of Karousos: the verifier only has access to an alleged history that the server sends as part of the advice. Karousos responds to this challenge by running Adya's checks [16] against the alleged database history, and then checking the consistency of the contingent history with the rest of the server's alleged execution.

To evaluate our techniques we implement Karousos as a system that audits Node.js applications that use MySQL as a transactional KV store. We find that Karousos achieves significant speedups at the verifier while imposing tolerable overheads to the service.

Roadmap. The rest of this dissertation is organized as follows: Chapter 2 summarizes related work on execution integrity (§2.1), prior work related to Verdict (§2.2), and prior work related to Karousos (§2.3). Chapter 3 presents the problem of verifying transparency dictionaries and how Verdict solves it (§3.1), details Verdict's cryptographic accumulator (§3.2) and SNARK (§3.3), and discusses Verdict's implementation (§3.4) and evaluation (§3.5). Chapter 4 provides necessary background on verifying outsourced web applications (§4.1) and on event-driven applications (§4.2), and presents Karousos's design (§4.3), implementation (§4.4), and evaluation (§4.5). Chapter 5 concludes by discussing the limitations of our work and future directions.

2 | RELATED WORK

We discuss related work on execution integrity in Section 2.1 and evaluate prior systems with respect with our desiderata (Ch. 1). We discuss previous systems that realize transparency dictionaries in Section 2.2. Last, we give an overview of systems related to Karousos including record-replay systems in Section 2.3.

2.1 EXECUTION INTEGRITY

Execution integrity – giving a principal assurance that a given program executes correctly – is a broad topic for which many different solutions have been proposed.

Replication. One possible solution is Byzantine replication [49, 111, 162], a technique with many applications in distributed systems, including consensus and fault tolerance. When used for execution integrity, the principal replicates the execution of the program across many machines and checks that a super-majority of these machines produce the same outputs. However, this requires trusting that a super-majority of replicas executes fault-free. With a similar assumption that some nodes must execute fault-free, PeerReview [82] and FullReview [63] provide accountability to general distributed systems, by maintaining a log of each node's inputs and outputs and requiring nodes to audit each other's logs to detect misbehaviors.

Attestation and Enclaves. In attestation, including TPM-based systems [52, 83, 113, 114, 125, 131, 134, 144] and SGX [88]-based systems [24, 32, 87, 132, 141, 143], the remote substrate proves that it is running the expected software stack. However, this does not guarantee that the remote program is actually executing faithfully: vulnerabilities in the stack may be exploited to undermine correct execution. Using an enclave, the principal can place a program's code and data in a an encrypted memory region that the CPU hardware isolates strongly from the rest of the machine's hardware and software. However, enclaves that provide strong integrity guarantees have limited-sized memory regions [58] and, thus, cannot accommodate applications that have large memory needs.

Probabilistic proofs. Execution integrity has received attention from complexity theorists and cryptographers in the context of probabilistic proofs – which include Interactive Proofs [28, 77, 78] and Arguments [43, 72, 89, 117]. Probabilistic proof systems were considered wildly impractical for decades due to their astronomical proof generation and verification costs. However, the past decade has seen orders of magnitude improvements to their costs [158] which have resulted in mushrooming implementations (see Thaler's manuscript for a survey [149, Ch.19]). Despite these improvements, as we discussed previously (Ch. 1) state-of-the-art probabilistic proof systems still incur high proving costs.

Re-execution-based audit. Another class of systems [25–27, 64, 81, 146, 155] verify that a remote program executes correctly given a trace of the program's inputs/outputs by re-executing the program on a trusted executor. Then, they check the trusted executor's outputs against the given trace. Among these systems, Orochi [146] and the systems that build atop it [26, 27], are the only ones in which re-execution is accelerated and, thus, admit audit procedures that require less computational resources than execution at the remote executor. We review Orochi in Section 4.1. EAR [27] expands Orochi's applicability to C++ programs and improves the performance of its verifier by

letting a developer place portions of an application in an enclave, to reduce the re-execution burden, under the restriction that the enclave and executing server are co-located. At a lower system level, in DIVA [25], a trusted checker performs an accelerated re-execution of an untrusted processor core, but it is geared to uniprocessor systems.

In AVM [81], an untrusted hypervisor records an execution while a trusted replayer uses something akin to Karousos's trace, together with VM replay [45, 68], to validate the execution. In Ripley [155], a web server re-executes client-side code. Dickerson et al. [64] propose a system in which miners in a blockchain network execute transactions in parallel, and validators verify the miners' work by re-executing the transactions in each block deterministically and concurrently. In all above works, the recorder is untrusted, but the replay is not accelerated.

2.2 Verifying transparency dictionaries

Verdict is a transparency dictionary that admits publicly verifiable proofs (Ch. 1) with efficient verifiability, meaning that verifying a batch of operations scales sublinearly with the number of operations. Moreover, verification does not require a trusted setup. Trusted setup means that a party (or a group of parties of which at least one is honest) is trusted by all clients. That party produces public parameters for a proof system using a secret trapdoor. However, anyone with the knowledge of the trapdoor can forge false proofs.

Transparency Dictionaries. Many prior works provide a transparency dictionary. However, they either don't offer public and efficient verifiability, or they require a trusted setup. For example, CONIKS [115] only provides *private*-verifiability (each user only verifies a subset of the updates that the dictionary executes). Very recently, Merkle^2 [85] improves on CONIKS by partially proving that the service updates its state correctly with a publicly-verifiable proof. However, it still produces a privately-verifiable proof for each update submitted by a client. It is possible to combine all the

privately-verifiable proofs produced by the untrusted service to construct a publicly-verifiable proof. However, such a proof would be neither succinct nor efficiently-verifiable. SEEMless [51] provides publicly verifiable proofs, but does not offer efficient verifiability. Similarly, Keybase [5] maintains a key directory and periodically publishes a commitment to its state on a public blockchain. However, proofs are not publicly verifiable: each user only checks their own updates. It would be straightforward to modify Keybase to offer public verifiability, by requiring the service to expose to users the sequence of operations that transformed one state to the other and have each client check this sequence. However, in this case, the data that each client would need to download and the work that each client would need to do are impractical. AAD [150] produces publicly-verifiable proofs with efficient verifiability. However, it requires a trusted setup and imposes high concrete costs on the service to produce proofs (§3.5). Finally, recent concurrent works [53, 152] explore the use of SNARKs to reduce costs for clients of transparency dictionaries. However, these solutions incur high prover costs to produce proofs, and they require a trusted setup.

In *verifiable state machines (VSMs)*, an untrusted service proves the correct execution of state machine transitions using SNARKs [136]. Due to their generality, VSMs imply transparency dictionaries. However, existing work does not optimize core ingredients to efficiently realize a transparency dictionary. Specifically, several works [44, 71] compose Merkle trees with SNARKs, but they target general computation. Spice [104, 136] composes a multiset-based data structure with SNARKs. However, it must treat both reads and writes in a uniform manner, so the high cost of SNARKs is incurred for both reads and writes. In Verdict, by contrast, reads do not require the use of SNARKs. Similarly, Ozdemir et al. [122] compose RSA accumulators with SNARKs. However, as we discuss later (Section 3.2), RSA accumulators require $O(n)$ computation for the service to respond to a lookup operation over an n-sized dictionary. Also, RSA accumulators require a trusted setup.

Untrusted storage systems. A large body of work offers untrusted storage systems [69, 107, 110, 112, 127]. However, they target scenarios where a small number of clients collectively read and write data stored at the service. Furthermore, they require clients to effectively execute all updates processed by the service. Concerto [23] provides a verifiable key-value store that that can be used to realize an untrusted storage system. However, Concerto's verifier must replay all operations executed by an untrusted service. Hence, it does not provide efficient verifiability. Furthermore, verification requires periodically scanning the entire state maintained by the service.

Blockchain-based solutions. While Verdict shares some of its tools with blockchains, it differs from them in several respects. First, blockchains employ a single global hashchain to order blocks of transactions, whereas Verdict uses a hashchain for each user to order updates (§3.1). Second, while blockchains can be used to build a transparency dictionary [7, 129] (for example, Microsoft's ION [7] records operations submitted by users in Bitcoin's blockchain), to retrieve the latest public key associated with a particular identity, a client must effectively re-execute all operations in the order they are recorded on the blockchain. In contrast, with Verdict, an untrusted service provides a response to a lookup along with a proof. Verdict's service can equivocate and expose different commitments to different clients, but this can be prevented by having the service use a blockchain to disseminate a single sequence of commitments to clients (note that the use of blockchain still does not require Verdict's clients to re-execute all operations processed by the system).

Persistent authenticated dictionaries (PADS) [21, 60, 128]. PADS provide specialized data-structures such as skip-lists and red-black trees. PADs provide succinct proofs for each operation (logarithmic in the size of the state). However, to prove that a batch of updates were executed correctly, the proof size grows linearly with the number of updates, similar to the two naive baselines against which we evaluate Verdict in Section 3.5. In contrast, Verdict uses SNARKs to produce a succinct proof for a batch of updates.

2.3 VERIFYING EVENT-DRIVEN WEB APPLICATIONS

Record-replay systems. At its core, Karousos is a record-replay system. Record-replay is a vast area, with several excellent surveys [56, 57, 65]. Karousos supports the combination of an untrusted recorder, accelerated replay, and executions with concurrency. Only in Karousos and a few works mention earlier (§2.1) the recorder is untrusted.

Poirot [92] and Shortcut [66] provide an accelerated replayer. In these systems the recorder captures hints that the replayer uses to re-execute (in Poirot's case, in a batch, as in Karousos), but the hints necessitate trust in the recorder.

A large body of record-replay work aims to re-execute concurrent executions while controlling the size of advice supplied to the replayer. These works trust the recorder and do not achieve acceleration. Here, we will specifically cover the works that relate to Karousos's logging algorithm (§4.3.2); for other record-replay work that targets concurrency, see JaRec [75], Respec [102], DoublePlay [154], and citations therein.

In Netzer's work [120], implemented in hardware by FDR [160] (see also [126]), when a data race occurs, the recorder logs the conflicting operations; the goal is to log a minimal set of such races. The replayer synchronizes these data races to reproduce the original order. In Bugnet [119], implemented in software by PinSEL [118] (see also [38, 126]), the recorder applies memory store operations to main memory and a *shadow memory*; on a load, if the main and shadow values disagree, the recorder infers that the memory was concurrently modified (for example by DMA, or an interrupt), and logs the load. This technique also appears in Jalangi [133], which re-executes JavaScript and also shares some of Karousos's approaches to handling calls to native code (§4.4).

These techniques are reminiscent of the way that Karousos decides dynamically whether to log an access to a program variable. However, they handle only *physical* concurrency, not concurrency at the replayer due to out-of-order execution (Ch. 1). Notice that concurrency at the replayer is strictly harder: if two accesses are not physically concurrent, and hence are reconstructible with

a traditional re-execution, they could nevertheless be concurrently executed during out-of-order re-execution, and hence need logging in our context. Furthermore, it's not clear how to extend these techniques to ensure consistency in the face of a possibly-cheating server.

Another approach to keep logging small when re-executing concurrent executions is to log enough information for the replayer to reconstruct a thread schedule equivalent to the original. In CREW systems [55, 67, 97, 101, 163], the recorder logs, for read operations, a current version number; for write operations, the recorder logs the number of readers *before* this write. The replayer then blocks a given write until all prior readers execute their read. As in Karousos, this approach has the freedom to re-order concurrent reads. However, CREW reproduces a schedule equivalent to the original physical one, and thus cannot handle general out-of-order re-execution.

LEAP [86] has a similar log structure to Karousos (§4.3.2), and thus would be amenable to out-of-order re-execution. However, LEAP trusts the recorder and is not designed to control the size of the logs. ORDER [161] improves on LEAP; it statically analyzes a program to determine which accesses need logging. Karousos could borrow these techniques to identify which variables are accessed by operations that can be re-ordered during re-execution, as this identification is currently a manual process (§4.4).

Other related techniques. Karousos's techniques for ensuring well-ordered executions (§4.3.3–4.3.4) relate to prior work in memory checking and consistency checking, where typically there is a dependency graph that the checker wants to ensure is acyclic [16, 18, 19, 22, 37, 76, 123, 140, 142, 145]. Karousos also builds explicitly on Adya's algorithms [16]. Karousos, however, must rely on untrusted advice separate from the input/output trace (§4.1.1), which complicates validation.

Cobra [147], Elle [93], and Litmus [159] ensure the serializability of an untrusted database, in setups which are closer to that of Karousos, and (in the case of Cobra and Elle) rely on dependency graphs. These works, of course, do not validate an entire application, only the database component.

JARDIS [31] is a time-travel debugger for JavaScript; it allows (among other things) stepping

from the execution of a callback backward to the handler that registered that callback. To do so, JARDIS wraps each handler, to pass in information about where it was registered, enabling the debugger to "walk up the activation stack." This is similar to Karousos's use of handler labels (§4.4).

3 | TRANSPARENCY DICTIONARIES

This chapter studies the problem of verifying transparency dictionaries.

A distinctive aspect of transparency dictionaries is that they support efficient membership *and* non-membership queries. To better articulate the problem and the importance of non-membership queries, let's consider the case where the dictionary is used for key transparency.

In this case the dictionary maps unique identities to their public keys (such as RSA encryption keys), and enables the following message exchange: To send a "top secret" document D to `bob@dom.org`, Alice picks a symmetric secret key k, encrypts D using k, and then encrypts k using the public key(s) associated with `bob@dom.org` in the transparency dictionary; she then sends both ciphertexts to `bob@dom.org` via an untrusted channel (for example, the cloud). However, if the dictionary can return "rogue" public keys (that is, public keys that Bob does not control), then even perfect cryptography cannot protect the secrecy of Alice's document. Non-membership queries are important for this scenario as well: they allow validating the absence of a label from the dictionary prior to its insertion. Without validating that each insertion inserts a label that does not already exist, the service's dictionary might contain more than one tuple for the `bob@dom.org` label, allowing the service to show different tuples to different clients.[1]

We are interested in building a transparency dictionary that can prove to its clients that it updates its state correctly, meaning that the service has some application-dependent function and

[1] There is an orthogonal but important requirement that the service should not be able to *equivocate* that is, show different *versions* of its dictionary to different clients such that each version is well behaved. In general, this is impossible to prevent with a fully untrusted service [112]. However, in prior systems [51, 115, 150] and this work, if the service equivocates, it can be eventually detected by clients (§3.1, Appendix A.2).

only updates each value associated with a label by applying this function on the prior value. In the context of key transparency, suppose that the first public key registered for `bob@dom.org` was legitimate. Then the application update function might require that all subsequent updates must be authorized using one of the existing public keys.

A natural question here is: how can the untrusted service prove to its clients that it updates its dictionary correctly? We would like the service's proofs to be *publicly verifiable* (Ch. 1), which, in this context, means that any client can verify independently that the service updates its whole state correctly regardless of actions of other clients in the system. In the key transparency example, public verifiability implies that when Alice verifies the service's proof, she knows that Bob's key has not been subverted, even if Bob is offline or fails to actively monitor updates to his own key. Hence, Alice can safely encrypt sensitive data using the public key the service returns for Bob. Moreover, we want the system to be practical (Ch. 1). Concretely, we require proofs to be *efficiently verifiable*: the cost of verifying proofs should be lower than re-executing requests processed by the service, both asymptotically and concretely (this also implies that proofs should be succinct). Satisfying the latter requirement makes it possible for any light client to download and verify proofs. Prior work (§2.2) do not provide a dictionary abstraction [100], do not support public verifiability [85, 115], and/or impose significant proving/verification overhead [51, 53, 106, 150, 152].

In contrast, we describe Verdict, the first transparency dictionary with publicly and efficiently verifiable proofs of correct operation as well as modest proving overheads. Verdict achieves this by employing SNARKs. As explained in Chapter 1, a SNARK is a cryptographic primitive that enables a *prover* to prove that it has executed a computation correctly. In particular, the prover proves knowledge of a witness to an NP statement by producing a proof such that the size of the proof and the time to verify it are both *sub-linear* in the size of the statement.[2] Verdict employs SNARKs as

[2]SNARKs provide an additional property called zero-knowledge, where the verifier learns nothing about the prover's witness beyond what is implied by the proven statement. Verdict's focus is on succinct verification property of SNARKs, but with modern SNARKs (including the one that Verdict employs), achieving zero-knowledge is "free" in terms of additional costs to the prover, the verifier, and proof sizes.

follows. At the end of epoch i, the untrusted service publishes a succinct cryptographic commitment to C_i the current state of its dictionary. Informally speaking, a cryptographic commitment scheme enables a *sender* to commit itself to a value by sending a short commitment and then later reveal the value. A commitment scheme is binding (that is, the sender cannot reveal a value different from what it originally committed), and hiding (that is, a commitment does not reveal anything about the committed value). It also produces a succinct SNARK proof demonstrating that the information in C_i is a superset of that in the dictionary committed to by C_{i-1}. Stated at this level, the use of SNARKs seems like an obvious solution.

Of course, the problem with such an "obvious" solution is well-known (Ch. 1): the resource costs to produce SNARK proofs are, in general, excessive. However, we address this problem in our specific context, obtaining orders of magnitude speedups over a naïve application of SNARKs. First, we design *Indexed Merkle Trees*, a SNARK-friendly data structure (§3.2) that can be used to efficiently prove that each value associated with a label is only updated by applying the application-specific function. Specifically, rather than directly prove (at significant expense) that each update to a label's value was applied correctly (§2.2), Verdict uses an Indexed Merkle Tree to maintain, for each label, an *append-only* list of updates, which a client can apply locally to arrive at the current value. To prove that the dictionary is updated correctly, the service proves that the Indexed Merkle Tree is updated in an append-only fashion [51, 53, 85, 115, 150]: the set of labels in the Indexed Merkle Tree grows monotonically and so do the lists associated with labels.

Second, we design Phalanx, a new SNARK that leverages Verdict's particular workload characteristics to significantly drive down costs of the untrusted service and of the clients (§3.3). Specifically, by leveraging the epoch-based nature of Verdict, Phalanx produces amortized constant-sized proofs and verification times. Furthermore, it leverages the data-parallel nature of the statement proven in each round to substantially reduce proof-generation costs. Phalanx may be of independent interest to other epoch-based services.

We implement Verdict in Rust as a generic library for constructing transparency dictionaries

(§3.4). To demonstrate the concrete utility of Verdict, we apply it to our running example of key transparency, creating Keypal, a service for translating user identities to public keys without trusting the hosting service.

We evaluate Verdict (§3.5) and compare it to three baselines that provide similar security properties: AAD [150], and two variants of Merkle-Patricia trees. Unlike AAD, Verdict incurs low overheads even for large dictionaries with millions of labels, and unlike the Merkle-Patricia baseline, Verdict produces efficiently-verifiable proofs of correct operation. We also find that our workload-specific SNARK optimizations improve proving costs by an order of magnitude. Together, when executing in a single CPU core, Verdict achieves about 4 updates/sec and about 2 inserts/sec, with a per-epoch (amortized) proof size of 651 bytes and a verification time of about 3 ms (for a dictionary with 2^{20} label-value tuples); Verdict can achieve about 18–22 updates/sec and 9–11 inserts/sec, with a per-epoch proof size of 290 bytes and a verification time of 161μs for the same dictionary size at the cost of deferred guarantees (§3.5). This represents over an order of magnitude improvement over prior state-of-the-art, general proof systems for stateful services [104, 135, 136, 138] and over three orders of magnitude improvement over AAD [150].

Despite the fact that Verdict is general, it is optimized for read-heavy workloads; we are interested in read-heavy workloads because they are common in many applications such as key transparency and certificate transparency. Verdict's principal limitation is that the service still incurs significant CPU costs to produce proofs. Less fundamentally, although the proof-generation process is highly parallelizable, our current implementation of Verdict does not leverage multiple CPUs. Nevertheless, Verdict makes transparency dictionaries with efficiently-verifiable proofs affordable.

In summary, Verdict contributes:

1. A transparency dictionary that scales well asymptotically and concretely to large dictionaries.
2. A SNARK-friendly accumulator (Ch. 1) with $O(\log n)$ proofs of membership and non-membership.
3. A SNARK that provides amortized constant-sized proofs and verification times and that lever-

ages particular workload characteristics (for example, computations with repeated substructure) to speed up proof generation.

3.1 The Verdict Transparency Dictionary

This section introduces the problem of building a dictionary service that can prove its own correctness with succinct proofs. We then provide an overview of Verdict, a system that meets these requirements.

3.1.1 Problem Statement and Definitions

Our goal is to build Verdict, a service that exposes a dictionary abstraction and can cryptographically prove to its clients the correctness of every request it executes. Specifically, the service's state is a label-value map that clients can query and update: clients can insert a new label-value pair, update the value associated with an existing label, and look up the value associated with an existing label. By correct operation, we mean that the value associated with a label is only updated according to an application-specific function. We refer to this primitive as a transparency dictionary. As a concrete application of Verdict, we construct Keypal (§3.4), a public key directory that enables clients to register their public keys under an identifier such that the keys can be retrieved by other clients who can be assured that the public keys they retrieve are legitimate.

Threat Model. We assume that the service and any subset of clients can misbehave arbitrarily. For example, the service can misbehave when processing requests, or show different views of its state to different clients. Furthermore, we assume that the network can arbitrarily duplicate, drop, or reorder messages. However, we assume that the untrusted service and clients cannot break cryptographic hardness assumptions. We do not aim to protect against denial of service (an honest service can use standard techniques, such as rate-limiting or proof-of-work, to defend itself) nor to ensure

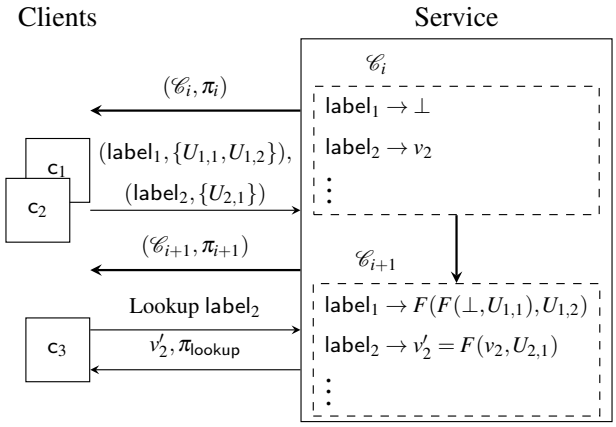


Figure 3.1: Example showing how Verdict's service updates its state in epoch i for some application-specific function F. At the end of epoch i, the service broadcasts a new commitment $\mathscr{C}_{i+1}$ and a proof π_{i+1} after incorporating requests from clients c_1 and c_2. Later, client c_3 can query the server to retrieve the latest value (v_2') associated with label label_2.

liveness (although if the service is honest and the network is reliable, then requests from correct clients will be executed). Finally, we do not aim to *prevent* all misbehavior from the untrusted service. Indeed, there are fundamental limits: it is impossible to prevent an untrusted service from equivocating [112]. However, as in prior work [51, 115], Verdict's clients can eventually detect such equivocation through client-side mechanisms (such as gossip), or by requiring the service to disseminate its commitments and proofs through a public blockchain (Appendix A.2).

More formally, let D_i for $i \in \mathbb{N}$ denote a mapping from labels to values. Labels are bitstrings and values belong to some domain $\mathscr{D}$ that depends on the application. For instance, in the context of key transparency, each value is a (possibly empty) set of public keys. Also, let $\mathscr{C}_i$ be a hiding and binding commitment to D_i. We assume that $D_0 = \bot$ and that $\mathscr{C}_0$ is a well-known commitment to D_0.

Moreover, let $F : \mathscr{D} \times \mathscr{U} \to \mathscr{D}$ a function that takes as input a label's old value, a request, and outputs a new value for the label. When a client wants to update the value v associated with some label label, it sends a request $U \in \mathscr{U}$ and the service updates the value associated with label to $F(v, U)$. Both $\mathscr{U}$ and F depend on the application. For example, in the context of key transparency

$$\mathscr{U} = \{(t, pk, s) | t \in \{\text{“revoke”, “add”}\} \text{ and } pk \text{ is a public key and } s \text{ is a signature }\}$$

```
1: procedure F(v: set of keys, (t: op type, pk: public key, s: signature) )
2:     // Check that the operation is signed by some key in v
3:     assert(∃k ∈ v s.t. VerifySignature((t, pk), k) = true)
4:     if t = revoke then
5:         return v ∖ pk
6:     else
7:         // t = add
8:         return v ∪ {pk}
```

Figure 3.2: Example F in the case of Key Transparency

In this case F validates that s is a valid signature for (t, pk) using one of the existing keys, and produces a new set of keys by either adding or removing a key from the input set of keys (Figure 3.2).

Let λ denote the security parameter. Throughout this chapter, the depicted asymptotics depend on λ, but we omit this for brevity.

A transparency dictionary is a tuple of algorithms

$$(\mathsf{Setup}, \mathsf{ApplyUpdates}, \mathsf{VerifyUpdates}, \mathsf{Lookup}, \mathsf{VerifyLookup})$$

with the following semantics.

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$: Returns public parameters pp used to produce and verify proofs.
- $(D_{t+1}, \mathscr{C}_{t+1}, \pi_{t+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_t, \mathscr{C}_t, \mathtt{U})$: Takes as input a dictionary D_t, commitment $\mathscr{C}_t$, and a sequence of insert/update requests $\mathtt{U}$. Outputs a dictionary D_{t+1}, a commitment $\mathscr{C}_{t+1}$ to D_{t+1}, and a proof of valid update π_{t+1}.
- $\{0,1\} \leftarrow \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_t, \mathscr{C}_{t+1}, \pi_{t+1})$: Takes as input a pair of commitments $(\mathscr{C}_t, \mathscr{C}_{t+1})$, and a proof of valid update π_{t+1}. Outputs 1 if for each label in the dictionary committed to by $\mathscr{C}_{t+1}$, the associated value equals the value associated with the label in $\mathscr{C}_t$, *or* π_{t+1} proves the knowledge of a sequence of requests $(U_0, \ldots, U_{\ell-1})$ such that $F(v_i, U_i) = v_{i+1}$ for $0 \leq i < \ell$, and where v_0 is the value associated with the label in $\mathscr{C}_t$ (or $\perp$ if the label did not exist in $\mathscr{C}_t$) and v_ℓ is the value associated with the label in $\mathscr{C}_{t+1}$.
- $(v, \pi_{\mathsf{lookup}}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, D_t, \mathscr{C}_t, \mathsf{label})$: Takes as input a dictionary D_t, commitment $\mathscr{C}_t$, and label label. Outputs a value v, and a proof of valid lookup π_{lookup}.

- $\{0,1\} \leftarrow \mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}, \mathsf{label}, v, \pi_{\mathsf{lookup}})$: Takes as input a commitment $\mathscr{C}$, a label label, a value v, and a proof π_{lookup}. Outputs 1 if v is the value associated with label in $\mathscr{C}$.

As in prior work [51, 85, 115, 150], the service in a transparency dictionary operates in a sequence of (fixed) time epochs. In each epoch, the service collects requests submitted by clients, uses the ApplyUpdates procedure to update its internal state and produce a new commitment and a proof of valid update, which it publishes to clients. Clients collect the sequence of commitments and the associated proofs published by the service at each time epoch and use the VerifyUpdates to check if the service correctly updated its state. Similarly, for lookup requests, the service uses the Lookup procedure to produce a response along with a proof, which the clients can verify by applying the VerifyLookup procedure to the proof and the latest service commitment they have received.

We formally define the properties of a transparency dictionary in Appendix A.1 but we summarize them here.

- **Completeness.** If the service is honest, clients do not reject responses produced by the service.
- **Update soundness.** The values associated with labels are updated only by applying the function F.
- **Lookup soundness.** The service cannot return an incorrect value for any label included in a commitment.
- **Fork consistency.** If the service equivocates at some point in time by presenting different sequences of commitments to different sets of clients, it cannot undetectably merge their views.

Additionally, we desire the following properties:

- **Public verifiability.** Any client can check proofs produced by the service to verify that that values are updated only by applying the function F.
- **Efficient verifiability.** Compared with re-executing every request processed by the service, the cost to verify a proof is more resource efficient (in terms of CPU, network bandwidth, storage, etc.), both asymptotically and concretely. This implies that proofs produced by the service are

succinct.

3.1.2 VERDICT: ARCHITECTURAL OVERVIEW

At its core, Verdict utilizes cryptographic SNARKs (Ch. 1) to prove that the untrusted service correctly maintains its dictionary abstraction. Generically, a SNARK allows a prover to demonstrate that an arbitrary polynomial relation $\mathscr{R}(w,x)$ holds, where w is a (possibly secret) witness, and x is a public input. The prover produces a proof π such that the size of π and the time to verify it are both sub-linear in the size of the circuit that computes $\mathscr{R}$.

At the end of epoch $t+1$, Verdict uses SNARKs to prove that for each (label, v) in the dictionary committed to by $\mathscr{C}_t$, one of the following holds: either (label, v) exists in the dictionary committed to by $\mathscr{C}_{t+1}$; or there is a sequence of requests $U_0, \ldots, U_\ell$ such that $v' = F(\ldots(F(v, U_0)\ldots, U_\ell)$ and (label, v') is in $\mathscr{C}_{t+1}$.

Implemented in a naïve manner, the approach above would be prohibitively expensive. First, for each value updated, the service must prove, using an expensive SNARK, that it has faithfully executed the function F for some sequence of requests `U`. For example, in the context of a key directory, when updating a public key associated with an identity, the service must prove that it knows of valid digitally-signed requests in the sense that the signatures can be verified using one of the non-revoked keys associated with the identity (Figure 3.2). Verifying such cryptographic operations via a SNARK is quite costly [62, 104, 122]. Even worse, as described above, to prove that its $O(N)$ dictionary is updated correctly, the size of the circuit that the service must prove using a SNARK is $\Omega(N)$. In other words, even when the number of values updated in a given epoch is far less than N, the service's cost for each epoch is still $\Omega(N)$. Finally, even non-cryptographic computations are costly to prove using SNARKs.

We now discuss how Verdict addresses these issues. Figure 3.3 compares Verdict's asymptotic complexity with that of prior work.

	space (life-time)	updates (per-epoch) prover	proof size	verifier	lookups (per-op) prover/proof size/verifier	assumption
AAD [150]⋆	N	$\beta \cdot \log^3 n$	$\log n$	$\log n$	$\log^2 n$	q-type
SEEMless [51]	N	$\beta \cdot \log N$	$\beta \cdot \log N$	$\beta \cdot \log N$	$\log N$	CRHF
Verdict	N	$\beta \cdot \log n$	$\log(\beta \cdot \log n)/k$	$\log(\beta \cdot \log n)/k$	$\log n$	SXDH
Verdict-lazy	N	$\beta \cdot \log n$	$\log(\beta \cdot \log n)/k$	$\beta \cdot \log n/k$	$\log n$	DLOG

⋆ Requires a trusted setup

Figure 3.3: Asymptotic costs for updates and lookups under Verdict and its baselines. All costs also depend on the security parameter λ but we omit this for brevity. β is the number of updates per epoch, n is the maximum number of labels in the dictionary, ℓ is the maximum number of operations associated with a label. We let $N = n \cdot \ell$. Lookup responses include an additive cost of $\Omega(\ell)$. Verdict variants use indexed Merkle trees (§3.2) with different variants of Phalanx (§3.3). k denotes the number of epochs over which costs are amortized (see 3.3.3 for details).

Validating Value Updates via Hashchains. To reduce the service's overhead, instead of directly proving that it updates each value by applying F, Verdict employs a simpler and cheaper alternative. In Verdict, the service maintains for each label an append-only hashchain of requests, where nodes store the request U as well as the cryptographic hash of the previous node in the chain. For example, in the context of key transparency, each node in the hashchain is (t, pk, s, h) where t, pk and s are defined above and h is the hash of the previous node. A hashchain c is valid if each node includes a correct hash of the previous node.

The service proves that all dictionary updates correspond to applications of the function F by showing that hashchains are never deleted and that each hashchain grows monotonically. The service replies to lookup queries with the hashchain c rather than the value v associated with a label. When a client retrieves a hashchain c associated with a label, it can quickly repeatedly apply F using the requests recorded in the hashchain to construct the current value v associated with the label, checking the validity of the cryptographic hashes along the way. This design supports a richer class of application-specific functions F without requiring the service to prove that it correctly applies F using SNARKs.

Succinct Commitments and Proofs via Indexed Merkle Trees. Verdict commits to the mapping from labels to values by committing to the mapping from labels to hashchains. We now discuss how Verdict commits to the latter so that it can efficiently produce: (i) lookup proofs, and (ii)

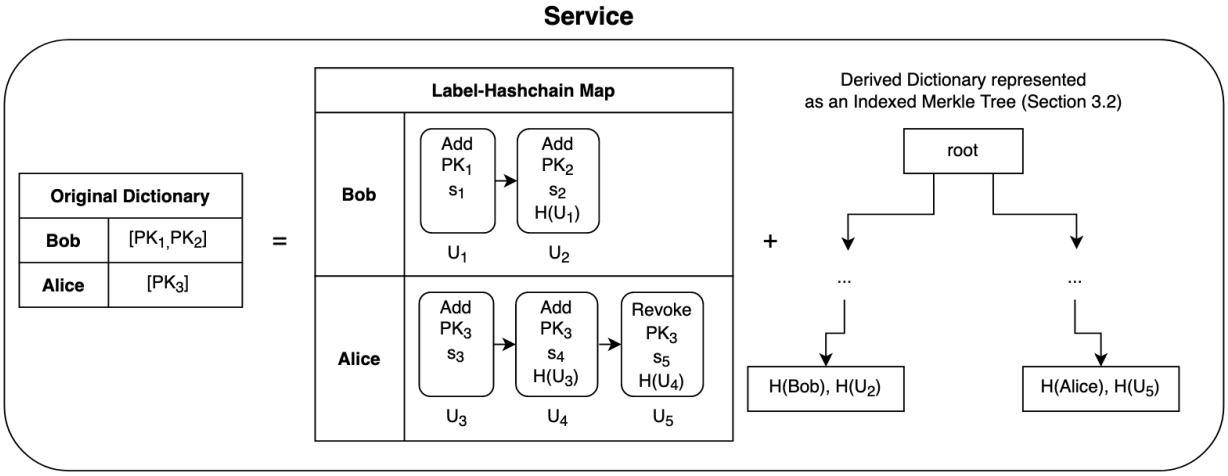


Figure 3.4: Internal State maintained by the service under Verdict. The service commits to the derived dictionary by publishing the root of the indexed merkle tree. The service replies to lookup proofs with the hashchain that corresponds to the requested label (if it exists) and (non-)membership proof of the label in the Indexed Merkle Tree.

update proofs. First, Verdict's service stores the mapping from labels to hashchains in a commodity storage service. Next, it creates a "derived" dictionary that stores a map from the cryptographic hash of a label to the cryptographic hash of the last node in the hashchain c associated with the label. In particular, for each $(\mathsf{label}, \mathsf{c})$ tuple in the service's state, the derived dictionary contains $(H(\mathsf{label}), h)$, where H is a cryptographic hash function and h is the cryptographic hash of the last node in c. Figure 3.4 shows an example of the state maintained by the dictionary when the dictionary is used for key transparency.

Observe that to commit to its mapping from labels to hashchains the service only needs to commit to its derived dictionary. This is because each $(\mathsf{label}, \mathsf{c})$ tuple in the derived dictionary is cryptographically bound (via the collision-resistance of H) to a unique tuple in the original dictionary. More crucially, this choice ensures that the labels and values that Verdict needs to commit are constant-sized and in particular short (for example, they are 32 bytes each when using SHA-256 as the hash function).

To commit to a derived dictionary we design a cryptographic accumulator using textbook Merkle trees, which we refer to as *indexed Merkle trees* (§3.2). As we mentioned in Chapter 1, a crypto-

graphic accumulator is a commitments scheme with support for proving membership of an element against a commitment. We provide details in Section 3.2; for now, a distinguishing aspect of indexed Merkle trees is that they support *both* efficient membership proofs (that is, proofs for statements of the form $(\mathsf{label}, v) \in \mathscr{C}$, where (label, v) is a label-value pair and $\mathscr{C}$ is a commitment to a dictionary) and non-membership proofs (that is, proofs for statements of the form $(\mathsf{label}, v) \notin \mathscr{C}$). In particular, for a dictionary of size N, it produces $O(1)$-sized commitments (for instance, they are 32 bytes when using SHA-256), and $O(\log N)$-sized proofs of membership and non-membership.

Thus, at the end of each epoch, Verdict's commitment is of size $O(1)$. Furthermore, for a lookup request issued by a client, π_{lookup} is either a proof of membership (if the hash of the requested label exists in the derived dictionary) or a proof of non-membership (otherwise). Note that the indexed Merkle tree itself is stored in a commodity storage service, so producing a lookup proof does not require the use of SNARKs, so the throughput of the service for lookup operations is bound purely by the throughput of the underlying storage service to retrieve proofs of membership (or non-membership).

Another crucial aspect of indexed Merkle trees is that, by leveraging proofs of non-membership, they provide efficient ($O(\log N)$) proofs of insertion and update.

Condensing Merkle Proofs via SNARKs. As described above, Verdict uses Merkle proofs to reduce the cost of proving/verifying dictionary updates from $\Omega(N)$ to $O(\log(N))$. This improvement does not lead to efficiently verifiable proofs (§3) as for an epoch that consists of β insert/update operations, the proof length and verification time are both $O(\beta \cdot \log N)$. Cost-wise, this is equivalent to requiring the service to broadcast all updates it processes in order to prove that it maintains the desired append-only properties. Verdict mitigates these costs by employing SNARKs.

Specifically, for $\beta > 0$ insert/update operations, the service processes each operation sequentially, and for each operation, the service produces a new commitment and an $O(\log N)$-sized proof. The untrusted service however does not publish these intermediate commitments nor the $O(\log N)$-

sized proofs. Instead, it employs SNARKs to prove that it *knows* indexed Merkle tree proofs that a verifier would accept, thereby achieving a proof of valid updates whose size and verification time are both sub-linear in $\beta \cdot \log N$ (Figure 3.3).

As discussed in Section 3.2, our indexed Merkle trees are carefully designed to be SNARK-friendly.

Accelerating SNARKs in Verdict's Context. Thus far, Verdict could be instantiated with any generic SNARK system. However, despite the proof-friendly nature of our indexed Merkle trees, the costs would still remain high. Hence, we introduce Phalanx, a new SNARK that leverages particular workload characteristics to provide amortized constant-sized proofs and to reduce proof generation costs (§3.3 provides details).

3.2 Instantiating Verdict's Cryptographic Accumulator

This section describes the SNARK-friendly accumulator that Verdict employs to maintain state. We discuss properties that we desire from an accumulator, limitations of existing solutions, and our design of a Merkle-tree variant that suffices for Verdict.

3.2.1 Requirements and Possible Instantiations

Let λ denote the security parameter, and let $\texttt{negl}(\lambda)$ denote a negligible function in λ. Let "PPT algorithms" refer to probabilistic polynomial time algorithms.

A cryptographic accumulator [35] enables a *prover* to commit to a collection D (for instance, a dictionary with a set of label-value pairs) by sending a succinct commitment $\mathscr{C}$ to a *verifier*. Such digests are *binding*, meaning that it is computationally infeasible to identify a different collection of items with the same digest. In addition, cryptographic accumulators support succinct *proofs of*

membership: for any item (that is, a label-value tuple) $x \in D$, the prover can produce a succinct proof π such that an honest verifier accepts π, and for any $x \notin D$ and any purported proof π produced by a PPT algorithm, Pr[the verifier accepts π] $\leq \texttt{negl}(\lambda)$.

A classic example of an accumulator is a Merkle tree, where the root of the Merkle tree commits to items stored at leaf nodes and provides $O(\log n)$-sized proofs of membership, where n is the size of the collection.

As we discuss in Section 3.1.2, for Verdict, we require accumulators with succinct proofs for membership, inserts, and updates. Specifically, suppose that the prover commits to its collection D with a commitment $\mathscr{C}$:

- *Proof of insertion.* For any $x \notin D$, the prover can produce a new commitment $\mathscr{C}'$ and prove that $x \notin D$ and $\mathscr{C}'$ commits to $D \cup \{x\}$ with a succinct proof π such that a verifier accepts π. If $x \in D$ or if $\mathscr{C}'$ does not commit to $D \cup \{x\}$, then for any purported proof π produced by a PPT algorithm, Pr[the verifier accepts π] $\leq \texttt{negl}(\lambda)$.
- *Proof of update.* For any $x \in D$, the prover can produce a new commitment $\mathscr{C}'$ and prove that $\mathscr{C}'$ commits to $D \cup \{x'\} - \{x\}$ for some x' with a succinct proof π such that a verifier accepts π. If $\mathscr{C}'$ does not commit to $D \cup \{x'\} - \{x\}$, then for any purported proof π produced by a PPT algorithm, Pr[the verifier accepts π] $\leq \texttt{negl}(\lambda)$.

Note that an important building block for a proof of insertion is a a *proof of non-membership*: for any $x \notin D$, the prover can prove that $x \notin D$ by producing a succinct proof π such that the verifier accepts π, and for any $x \in D$ and any purported proof π produced by a PPT algorithm, Pr[the verifier accepts π] $\leq \texttt{negl}(\lambda)$.

In addition to the above properties, we desire a cryptographic accumulator where these proofs are *SNARK-friendly*; that is, where the proof verification algorithm can be efficiently encoded in the input language of a modern SNARK. This can be thought of as an arithmetic circuit over a large field, although in practice most modern SNARKs use a generalization known as R1CS. The

instantiation	membership			non-membership/insert/update			SNARK friendly?
	prover	proof size	verifier	prover	proof size	verifier	
Merkle trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	✓
RSA accumulators	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	✗
Merkle-Patricia trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	✗
Merkle-AVL trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	✗
Indexed Merkle trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	✓

Figure 3.5: Comparison of asymptotic costs of Merkle proofs of membership, non-membership, inserts, and updates under different instantiations of cryptographic accumulators, where n denotes the number of items in the committed collection.

efficiency of this encoding matters, since, as described in Section 3.1.2, the Verdict service relies on SNARKs to condense many insert/update proofs into a single succinct proof of correct operation. To accomplish this, the service first updates the cryptographic accumulator to produce a sequence of insert/update proofs, one for each operation. The service then proves that it *knows* a sequence of valid proof of insert/updates using a SNARK.

We examine existing accumulators in the literature and folklore. Figure 3.5 summarizes our findings, where n is the number of items in a collection. Briefly:

- Merkle trees [41, 116] do not support succinct proofs of non-membership nor insertion.
- RSA accumulators [48] support $O(1)$-sized membership and insert/update proofs, but they impose $O(n)$ costs on the prover to produce them; this is expensive, both asymptotically and concretely. Also, they require a trusted setup as well as big number arithmetic that is inefficient to encode as an arithmetic circuit.
- Merkle-Patricia trees (or more generally, Sparse Merkle trees) (for example, [51, 61, 99, 121]) support $\Theta(\log n)$-sized proofs of membership, non-membership, insertion, and updates. However, by design, paths in these trees are of variable length, so devising arithmetic circuits to verify Merkle proofs introduces significant complexity. Of course, one can use fixed-depth trees (for instance, depth-256) to make them SNARK-friendly, but this incurs over an order of magnitude higher costs at our collection sizes (2^{20}–2^{30} items).
- Like Merkle-Patricia trees, Merkle-AVL trees are not SNARK-friendly as they require rebalanc-

ing upon insertion of new nodes.

3.2.2 INDEXED MERKLE TREES

We now describe *indexed Merkle trees*, our SNARK-friendly variant of "textbook" Merkle trees, with support for efficient non-membership and insertion proofs.

An indexed Merkle tree is a standard Merkle tree in the following way: each item in a collection is stored at a leaf node in a Merkle tree. This ensures that indexed Merkle trees support $O(1)$-sized commitments, $O(\log n)$-sized proofs of membership, and $O(\log n)$-sized proofs of update. To support $O(\log n)$-sized proofs of non-membership and proofs of insertion, we encode additional metadata at each leaf node and maintain an invariant upon insertions and updates. We now elaborate.

Suppose that a collection is a set of label-value tuples, where each label and each value is of a fixed size w. This is the case for Verdict's derived dictionary (§3.1.2).[3] WLOG, we assume that labels can be sorted (for example, with a bitwise ordering of labels). Unlike a textbook Merkle tree, a leaf node in an indexed Merkle tree is of the form: $\langle \mathsf{active}, \mathsf{label}, v, \mathsf{next} \rangle$, where active is a bit indicating whether the leaf node holds a valid tuple. If active is 1, then label and v are respectively the label and its associated value stored at the leaf node, and next is a label in the tree that is larger than label.

Verifiable Initialization. We now discuss how the prover can initialize an empty indexed Merkle tree and how the verifier can efficiently compute a commitment to such an empty tree. Suppose that the indexed Merkle tree maintained by the prover has a capacity of $n \geq 2$ (that is, it has n leaf nodes), where each leaf node stores the same tuple $\langle 0, 0^{\mathsf{w}}, 0^{\mathsf{w}}, 0^{\mathsf{w}} \rangle$ (we later discuss how the prover can double the capacity of a Merkle tree and how the verifier can efficiently verify that). Any verifier—without help from the prover—can compute the root of such a tree with $O(\log n)$ hash computations. WLOG, Verdict designates two labels as reserved: 0^{w} and 1^{w}. These denote

[3] For other contexts, one can employ the same technique as in Verdict to derive a new collection with fixed-sized labels and values.

respectively the lowest and the highest values of labels in the system. Furthermore, the prover picks a designated leaf node (WLOG, the left-most node in the initial Merkle tree) in the initial indexed Merkle tree and updates it to hold the following tuple: $\langle 1, 0^{\mathsf{w}}, 0^{\mathsf{w}}, 1^{\mathsf{w}} \rangle$. The verifier—without help from the prover—can compute the root of the updated Merkle tree in $O(\log n)$ time.

Invariant. A core invariant that an indexed Merkle tree maintains is that for any "active" leaf node $N = \langle 1, \mathsf{label}, v, n \rangle$, $N.n$ is either: (i) 1^{w}, or (ii) there exists some leaf node in the tree with label $N.n$ and there are no active leaf nodes in the tree with labels in the range $(N.\mathsf{label}, N.n)$. Observe that this invariant holds for the indexed Merkle tree computed at the end of the initialization step. Below, we discuss how the invariant holds despite inserting new label-value pairs or updates to existing label-value pairs.

Proof of Non-Membership. A core building block for proofs of insertion are proofs of non-membership. To prove the absence of a particular label label in an indexed Merkle tree with commitment $\mathscr{C}$, the prover furnishes a proof of membership for a unique leaf node with contents $\langle 1, \mathsf{low}, v, \mathsf{high} \rangle$ such that $\mathsf{low} < \mathsf{label} < \mathsf{high}$. This proof can be verified using commitment $\mathscr{C}$. It is easy to see that this constitutes a proof of non-membership given the invariant stated above.

Maintaining the Invariant. In Verdict, there are only two types of operations.

(1) Updates. For a leaf node $N = \langle 1, \mathsf{label}, v, \mathsf{next} \rangle$, updating $N.v$. This trivially upholds the desired invariant. The proof of update is a proof of membership of the old leaf node against the old commitment. Verification involves verifying the proof of membership and then locally computing an updated commitment using the updated leaf node.

(2) Inserts. As shown in Figure 3.6, to insert a new label-value pair (label, v) into an indexed Merkle tree, the proof of insertion is produced as follows.

- The prover identifies an "inactive" leaf node;[4] that is, a leaf node of the form $N = \langle 0, 0^{\mathsf{w}}, 0^{\mathsf{w}}, 0^{\mathsf{w}} \rangle$.

[4] Verdict's service maintains a separate index of inactive leaf nodes; the choice of which inactive leaf node to use is

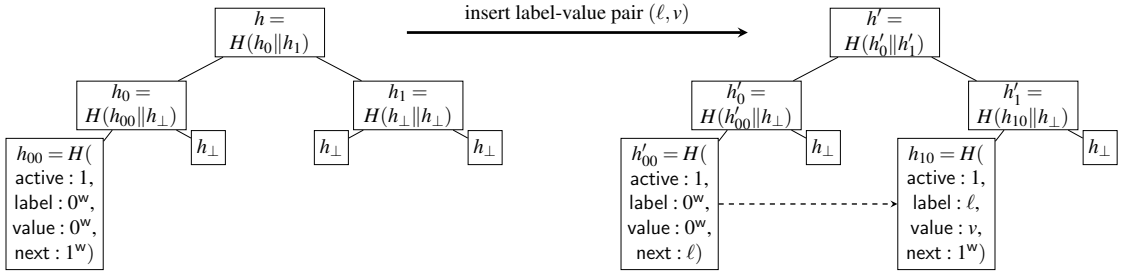


Figure 3.6: Indexed Merkle trees in action. We depict state changes from inserting a new label-value pair (ℓ, v) to an empty indexed Merkle tree with capacity 4. The choice of which non-active leaf node holds (ℓ, v) is arbitrary (see the text for details). $h_\perp = H(\mathsf{active} : 0, \mathsf{label} : 0^w, \mathsf{value} : 0^w, \mathsf{next} : 1^w)$. Dotted arrow indicates a leaf's "pointer" to the next active label. In the updated tree, to prove non-membership of ℓ' where $\ell' < \ell$, the prover presents the leaf node h'_{00} and its proof of membership.

If this fails, invoke the capacity doubling procedure described below and then retry this step.

- The prover produces a proof of non-membership of label in the indexed Merkle tree, that is, a unique leaf node $N^\star = \langle 1, \mathsf{low}, v, \mathsf{high} \rangle$ such that $\mathsf{low} < \mathsf{label} < \mathsf{high}$.
- The prover updates $N^\star$ to hold the value $\langle 1, \mathsf{low}, val, \mathsf{label} \rangle$, and produces a proof of update.
- The prover updates the previously inactive node N to hold $\langle 1, \mathsf{label}, v, \mathsf{high} \rangle$, and produces a proof of update.

Supporting Dynamic Capacity. The number of leaf nodes of an indexed Merkle tree can be doubled at any time such that a verifier can verify that the new Merkle tree contains all of the data from the original tree with $O(\log n)$ computation. Specifically, given the root r of an existing 2^i-sized indexed Merkle tree, the prover and the verifier can compute the (unique) root of the 2^{i+1}-sized Merkle tree as: hash(r, r'), where r' is the root of the Merkle tree where all leaf nodes have the same default value of $\langle 0, 0^w, 0^w, 0^w \rangle$. Note that r' can be computed by the verifier using $O(i)$ hashes. This does not require any precomputation or amortization as the cost is comparable to the cost of verifying a membership proof, which is logarithmic in the size of the collection.

arbitrary. In particular, note that leaf nodes in an indexed Merkle tree are *not* sorted, so if a new label falls between two existing labels in the dictionary, the new label can be inserted at any "inactive" leaf node in the tree.

3.3 Reducing Costs of SNARKs with Phalanx

In a Verdict epoch with β update operations, the untrusted service produces β indexed Merkle update proofs. A straightforward approach to prove the correctness of these indexed Merkle proofs with a SNARK is to use an arithmetic circuit (say of size C) that verifies one update proof, replicate it β times, and then employ a generic SNARK (such as Spartan [135] or Xiphos [138]) on the combined circuit. The result would be a prover that runs in time $O(C \cdot \beta)$. The per-epoch proof sizes and verification times are $O(\sqrt{C \cdot \beta})$ when using Spartan and $O(\log(C \cdot \beta))$ when using Xiphos.

To make Verdict concretely efficient, we develop Phalanx, a SNARK that substantially reduces resource costs imposed by prior SNARKs. Specifically, Phalanx produces (amortized) constant proof sizes and verification times. Additionally, Phalanx does not require a trusted setup. Given these, Phalanx is of independent interest (for instance, to other epoch-based services [104, 122]).

3.3.1 Overview of Phalanx

Phalanx targets epoch-based services where the prover in each epoch proves the satisfiability of the same N-sized circuit (with different witness values). In Verdict, the circuit verifies β indexed Merkle proofs.

In more detail, Phalanx's prover and verifier maintain a $O(1)$-sized *running instance*, which collectively represents the circuit satisfiability instances of *all* prior epochs (the verifier and the prover initialize the running instance to the first epoch's instance). At the end of each epoch, the prover produces an $O(1)$-sized proof that enables the verifier to combine the circuit satisfiability instance of that epoch with the running instance; the verifier incurs $O(1)$ computation to update its running instance. At the end of each epoch, the prover also produces an $O(\log N)$-sized proof to prove the satisfiability of the running instance. Verifying the logarithmic-sized proof (in addition to verifying constant-sized proofs for all prior epochs) verifies that the circuit satisfiability instances from all prior epochs are satisfiable.

In Verdict's context, a client must verify constant-sized proofs every epoch, but it need not verify the logarithmic-sized proof in every epoch. Specifically, a client only needs to verify the logarithmic-sized proof of the most recent epoch before issuing lookup operations. Furthermore, in Verdict, we observe that the running instance is data-parallel (as the circuit verifies β indexed Merkle proofs); Phalanx leverages this to substantially speedup proof-generation costs.

3.3.2 PRELIMINARIES

SNARKs. Recall (Ch. 3) that a SNARK is a cryptographic primitive that enables a prover to demonstrate its knowledge of a witness to an NP statement (such as a circuit satisfiability instance) with a proof that can be verified in time sub-linear (ideally, polylogarithmic) in the time to check the NP witness; this also implies that the proof is sub-linear in the size of the NP witness. For our purpose, SNARKs enable proving the correct execution of (stateful) computations, since those executions can be represented with NP statements (a formal definition of SNARKs is in Appendix A.1.1)

R1CS. Rank-1 constraint satisfiability (R1CS) is an NP-complete language that generalizes arithmetic circuits [34, 73, 137]. R1CS is a popular target for toolchains that compile programs in high-level languages [33, 44, 95, 104, 124, 136, 139, 156]. In more detail, let $\mathbb{F}$ denote a finite field (such as the set $\{0,1,\ldots,p-1\}$ for a large prime p, with addition and multiplication operations). An R1CS instance is a tuple $((\mathbb{F},A,B,C,m,n,\ell),\mathsf{X})$, where $\mathsf{X} \in \mathbb{F}^{\ell}$ is the public input and output of the instance, $A,B,C \in \mathbb{F}^{m\times m}$, $m \geq |\mathsf{X}|+1$, and there are at most $n = \Omega(m)$ non-zero entries in each matrix. A witness $W \in \mathbb{F}^{m-\ell-1}$ satisfies an R1CS instance $((\mathbb{F},A,B,C,m,n,\ell),\mathsf{X})$ if $(A\cdot Z)\circ(B\cdot Z) = C\cdot Z$, where $Z = (W,\mathsf{X},1)$. In a nutshell, the matrices encode the structure of a circuit whose gates can compute an arbitrary bilinear operation over Z. We refer to n as the size of the R1CS instance.

SIMD R1CS. To capture data-parallel (or SIMD) computations, we introduce a natural extension of R1CS that we refer to as SIMD R1CS. Informally, SIMD R1CS considers the same circuit (represented with matrices A, B, C) over β witness vectors $\{w_1, \ldots, w_\beta\}$ and the corresponding input/output vectors $\{\mathsf{x}_1, \ldots, \mathsf{x}_\beta\}$, representing β data-parallel units. We assume that β, m, and n are powers of 2, and ℓ is even; in practice, this can always be enforced by padding the matrices and vectors.

More formally, a SIMD R1CS instance is a tuple $\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X})$, where each of the matrices $A, B, C \in \mathbb{F}^{m \times m}$ has at most $n = \Omega(m)$ non-zero values and $\mathsf{X} \in \mathbb{F}^{\ell}$ is the public input/output. A witness (W, x), where $W \in \mathbb{F}^{(m-\ell-1)\times\beta}$ (each column of W is a purported witness for a separate data-parallel unit) and $\mathsf{x} \in \mathbb{F}^{\ell\times\beta}$ (each column of x is a purported input/output for a separate data-parallel unit), satisfies ϕ if $(A \cdot Z) \circ (B \cdot Z) = C \cdot Z$, where $Z = (W, \mathsf{x}, \vec{1})^\top$. In Verdict, we additionally require *IO consistency*, that is, that the input of each data-parallel unit is the output of the previous unit. Without loss of generality, we assume that for each i, $|\mathsf{x}_i|$ is even and that the first half of x_i and the second half of x_i are respectively the input and output of the ith data-parallel unit. Formally, we require that

$$\begin{aligned} \mathsf{x}_i[\ell/2:] &= \mathsf{x}_{i+1}[:\ell/2] \text{ for all } 1 \le i \le \beta - 1 \\ \mathsf{x}_1[:\ell/2] &= \mathsf{X}[:\ell/2] \\ \mathsf{x}_\beta[\ell/2:] &= \mathsf{X}[\ell/2:]. \end{aligned} \tag{3.1}$$

3.3.3 Constant Proof Sizes and Verification Times

Phalanx builds on recent work [96] that enables the prover and the verifier to combine two N-sized R1CS instances into a single N-sized instance with an $O(1)$-sized proof, such that the prover only needs to prove the validity of the combined instance. Such a protocol is referred to as a *folding scheme* and is used to realize incrementally verifiable computation [153]. We extend prior work [96] to design a folding scheme for SIMD R1CS. Specifically, in each Verdict epoch, Phalanx's prover

and verifier fold a SIMD R1CS instance (encoding the statement proven in the epoch) into a running instance. Below, we describe details of Phalanx's folding procedure.

A Folding Scheme for SIMD R1CS. Consider a SIMD R1CS instance

$$\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}).$$

While it is unclear how to fold two such instances in general, it is possible to fold two SIMD R1CS instances with the same structure (that is, the same A, B, C matrices) by first "relaxing" the instance. Specifically, we define a "slack" matrix $E = \vec{0}^{m \times \beta}$ and scalar $u = 1$ and transform the satisfiability check into checking

$$AZ \circ BZ = u \cdot CZ + E, \tag{3.2}$$

where $Z = (W, \mathsf{x}, u)^\top$.

More generally, a relaxed SIMD R1CS instance is a tuple

$$\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}, E, u)$$

where $\mathbb{F}, A, B, C, m, n, \ell, \beta, \mathsf{X}$ are defined as in the case of SIMD R1CS instances, $E \in \mathbb{F}^{m \times \beta}$ and $u \in \mathbb{F}$. A witness (W, x) satisfies ϕ if

$$AZ \circ BZ = u \cdot CZ + E, \tag{3.3}$$

where $Z = (W, \mathsf{x}, u)^\top$, and x satisfies IO consistency with respect to X as defined in the case of SIMD R1CS instances.

Now, given two relaxed SIMD R1CS instances $\phi_1 = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}_1, E_1, u_1)$, and $\phi_2 = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}_2, E_2, u_2)$ defined over the same matrices A, B, C with corresponding witnesses (W_1, x_1) and (W_2, x_2), the verifier can fold them into a new instance ϕ by randomly

sampling $r \in \mathbb{F}$ and taking a random linear combination:

$$\phi \leftarrow ((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}_1 + r \cdot \mathsf{X}_2, E_1 + r \cdot T + r^2 \cdot E_2, u_1 + r \cdot u_2)$$
$$(W, \mathsf{x}) \leftarrow (W_1 + r \cdot W_2, \mathsf{x}_1 + r \cdot \mathsf{x}_2)$$

where $T \leftarrow AZ_1 \circ BZ_2 + AZ_2 \circ BZ_1 - u_1 CZ_2 - u_2 CZ_1$ for $Z_1 \leftarrow (W_1, \mathsf{x}_1, u_1)$ and $Z_2 \leftarrow (W_2, \mathsf{x}_2, u_2)$. Observe that the resulting instance has the same size and A, B, C matrices as the input instances but different E matrix that depends on the witnesses of the input instances.

With textbook algebra, it is easy to show that (W, x) satisfies check (3.3) with respect to ϕ if (W_1, x_1) and (W_2, x_2) satisfy check (3.3) with respect to ϕ_1 and ϕ_2 respectively. Conversely, as with most batching techniques, soundness holds due to the randomness of the linear combination, which ensures with high probability that if (W, x) satisfies check (3.3) with respect to ϕ then (W_1, x_1) and (W_2, x_2) also satisfy check (3.3). The same reasoning holds for IO consistency: if matrices x_1 and x_2 both satisfy their respective IO consistency checks with respect to X_1 and X_2, then a folded input

$$\mathsf{x} \leftarrow \mathsf{x}_1 + r \cdot \mathsf{x}_2$$

for some randomly sampled $r \leftarrow_R \mathbb{F}$, also satisfies the IO consistency check with respect to X. Conversely, if x satisfies the IO consistency check, then it must hold with high probability that both x_1 and x_2 also satisfy the input consistency check with respect to X_1 and X_2.

For efficiency, the prover treats $(E_1, E_2, W_1, W_2, \mathsf{x}_1, \mathsf{x}_2)$ as the witness and provides additively homomorphic commitments to these values as part of the instance. Formally, the instance that both the prover and the verifier hold is a *committed relaxed SIMD R1CS instance*: a tuple

$$\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), (\mathsf{X}, \bar{\mathsf{x}}, \overline{E}, u, \overline{W})$$

where $(\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}, u$ are defined as in the case of relaxed SIMD R1CS instances, and $\bar{\mathsf{x}}, \overline{E}, \overline{W}$ are commitments. A witness E, W, x satisfies ϕ if $\overline{E}$ is a commitment to E, $\overline{W}$ is a commitment to W, $\bar{\mathsf{x}}$ is a commitment to x, and (W, x) is a witness that satisfies the relaxed SIMD R1CS instance $((\mathbb{F}, A, B, C, m, n, \ell, \beta), \mathsf{X}, E, u)$.

To efficiently fold two committed relaxed SIMD R1CS instances, the prover provides a commitment to T, instead of sending it directly. Then, instead of computing (linearly sized) E, W, and x, the verifier homomorphically computes commitments to E, W, and x as part of the new instance. Because the folding scheme is public coin,[5] we make it non-interactive via the Fiat-Shamir transform [70]. If the commitments are constant-sized (as is the case with our choice, see §3.3.5), then Phalanx achieves (per epoch) constant-sized proofs and verification times.

Observe that, as in the case of committed relaxed SIMD R1CS instances, the resulting instance depends on the witnesses of the input instances. In the context of Verdict, this implies that if the proving and verifying keys needed to demonstrate the satisfiability of the running instance depend on the part of the instance that changes at each epoch (for example, $\overline{E}, \overline{W}$), then new such keys need to be generated at each epoch. This would be prohibitively expensive. To avoid this cost, in Verdict, we follow prior work [96] to design a tranpsarent SNARK whose proving and verifying keys only depend on the part of the instance that remains the same across epochs (that is, the size of the instance, and the A, B, C matrices).

Bootstrapping and Inter-Epoch IO Consistency. At initialization, the running instance in Phalanx is the committed relaxed SIMD R1CS derived from, first, relaxing the initial Verdict epoch using default values of u and E (that is, $u = 1$ and $E = \vec{0}^{m \times \beta}$) and, then, committing to it. For epoch i ($i \geq 2$), the prover and the verifier use the folding scheme described above to fold the SIMD R1CS instance of epoch i with the running instance. Note that the the instance that is folded into the running instance is a SIMD R1CS instance, so it is first relaxed by using the default values of u and E and, then, committed.

In addition to the running instance, the verifier maintains $y_{last} \in \mathbb{F}^{\ell/2}$, which represents the output of the last SIMD R1CS instance that was folded. At initialization, $y_{last} = \mathsf{X}[\ell/2 :]$, where X is the public input/output of the SIMD R1CS of the first Verdict epoch (the verifier additionally

[5] An interactive protocol is public coin if the verifier's challenges are chosen uniformly at random.

checks that $\mathsf{X}[:\ell/2]$ holds the well-known initial input; for example, in Verdict, $\mathsf{X}[:\ell/2]$ must hold the Merkle root of an empty indexed Merkle tree of a pre-defined size). In epoch i (where $i \geq 2$), the verifier checks that $y_{last} = \phi.\mathsf{X}[:\ell/2]$, where ϕ is the SIMD R1CS instance of epoch i, and then after running the folding procedure, it updates y_{last} to $\phi.\mathsf{X}[\ell/2:]$.

3.3.4 Proving the Satisfiability of Running Instance

The previous subsection describes how Phalanx's prover and verifier fold a SIMD R1CS instance (of each epoch) into a running instance. We now describe how the prover produces a succinct proof of the satisfiability of the running instance in each epoch. To accomplish this, Phalanx relies on techniques from Spartan [135].

As background, Spartan [135] combines the sum-check protocol [108] with polynomial commitments [90] to obtain a SNARK. Alternatively, Spartan [135] can be viewed as combining a public-coin *polynomial interactive oracle proof (IOP)* [47] for R1CS with polynomial commitments [105]. A polynomial IOP is an interactive proof [77] where in each round the prover sends a polynomial as an oracle and the verifier may request an evaluation of the polynomial at a point in its domain. A public-coin polynomial IOP can be compiled into a public-coin interactive argument of knowledge using a polynomial commitment scheme. Instead of sending a polynomial, the prover sends a commitment to its polynomial, and when the verifier requests an evaluation of the polynomial, the prover sends an evaluation along with a proof that the evaluation is consistent with the prior commitment. The resulting interactive argument can then be turned into a SNARK in the random oracle model [70]. We refer the reader to prior work [105, 135, 138, 148] for details. Thaler [148] in particular provides detailed background as well as descriptions of several SNARKs, including Spartan.

Our main contribution is to provide a polynomial IOP for (committed relaxed) SIMD R1CS, adapting the polynomial IOP for R1CS from Spartan. To create a new polynomial IOP for committed relaxed SIMD R1CS, we first encode a committed relaxed SIMD R1CS instance as a set

of polynomials. For this, we first interpret matrices and vectors as functions that map bits to elements of $\mathbb{F}$. For example, a vector V of length m over $\mathbb{F}$ can be viewed as function with signature: $\{0,1\}^{\log m} \to \mathbb{F}$. We then take *multilinear extensions* of these functions. A multilinear extension of a function is the unique multilinear polynomial whose evaluations match that of the function over the domain of the function. For example, a multilinear extension of the aforementioned V viewed as a function is a polynomial $\widetilde{V} : \mathbb{F}^{\log m} \to \mathbb{F}$, where $\widetilde{V}(i) = V(i)$ for all $i \in \{0,1\}^{\log m}$.

Let $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$, represent the multilinear extensions of A, B, C respectively. Consider a purported witness $(W, E, \mathsf{x}) \in (\mathbb{F}^{m-\ell-1 \times \beta}, \mathbb{F}^{m \times \beta}, \mathbb{F}^{\ell \times \beta})$. Let $\widetilde{W}$, $\widetilde{E}$, $\widetilde{\mathsf{x}}$ denote the corresponding multilinear extensions, and let $\widetilde{Z}$ denote the multilinear extension of matrix $Z = (W, \mathsf{x}, \vec{1})^\top$. As part of the running instance, Phalanx's prover and verifier hold X, u, and commitments $\overline{W}$, $\overline{E}$, and $\bar{\mathsf{x}}$ to the prover's witness $\widetilde{W}$, $\widetilde{E}$, and $\widetilde{\mathsf{x}}$ respectively.

Given these polynomials, we define a polynomial F that evaluates to zero iff a given SIMD R1CS instance is satisfiable. Let $s = \log m$ and $c = \log \beta$.

$$\begin{aligned} F(k,i) = &\Bigg(\sum_{j \in \{0,1\}^s} \widetilde{A}(i,j) \cdot \widetilde{Z}(k,j) \Bigg) \cdot \\ &\Bigg(\sum_{j \in \{0,1\}^s} \widetilde{B}(i,j) \cdot \widetilde{Z}(k,j) \Bigg) - \\ &u \cdot \Bigg(\sum_{j \in \{0,1\}^s} \widetilde{C}(i,j) \cdot \widetilde{Z}(k,j) \Bigg) + \widetilde{E}(k,i) \end{aligned} \tag{3.4}$$

Lemma 1. If (W, x, E) is a satisfying assignment to a committed relaxed SIMD R1CS instance $((\mathbb{F}, A, B, C, m, n, \ell, \beta), (\mathsf{X}, \bar{\mathsf{x}}, \overline{E}, u, \overline{W}))$, then $F(k,i) = 0$ for all $k \in \{0,1\}^c$ and $i \in \{0,1\}^s$.

It is unclear how the prover can efficiently prove that F evaluates to zero over the Boolean hypercube defined by k and i, so we instead define a *multilinear* polynomial Q:

$$Q(t_1, t_2) = \sum_{k \in \{0,1\}^c} \sum_{i \in \{0,1\}^s} F(k,i) \cdot \widetilde{\mathsf{eq}}((k,i),(t_1,t_2))$$

where $\widetilde{\mathsf{eq}}$ is the multilinear extension of the function eq defined as follows: $\forall x, y$ over the domain of eq, $\mathsf{eq}(i,j) = 1$ if $i = j$ and 0 otherwise.

Due to the multilinearity of Q and the observation that if Z is a satisfying witness, then $Q(k,i) = 0$ for all $k \in \{0,1\}^c$ and $i \in \{0,1\}^s$, we have that Q is the zero polynomial iff Z is a satisfying witness. Therefore, it is sufficient for the prover to prove $Q(\tau_1, \tau_2) = 0$, where $\tau_1, \tau_2 \leftarrow_R \mathbb{F}$ are chosen by the verifier. Furthermore, the instance to be proven is in a form suitable for the application of the sum-check protocol [108]: an interactive proof system for proving $T = \sum_{i \in \{0,1\}^s} G(i)$, where G is an s-variate multivariate polynomial over $\mathbb{F}$ and $T \in \mathbb{F}$.

Phalanx's interactive argument proceeds as follows:

1. The verifier sends $(\tau_1, \tau_2) \leftarrow_R \mathbb{F}^c \times \mathbb{F}^s$.
2. The prover and the verifier use the sum-check protocol to reduce the task of checking

   $$Q(\tau_1, \tau_2) = 0$$

   to checking

   $$F(r_k, r_i) = e$$

   where $(r_k, r_i) \in \mathbb{F}^{c+s}$ are chosen by the verifier over the course of the sum-check protocol and $e \in \mathbb{F}$.
3. The prover and the verifier use the sum-check protocol to reduce the task of checking

   $$F(r_k, r_i) = e$$

   to checking claimed evaluations of $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at (r_i, r_j), $\widetilde{E}(r_k, r_i)$, and $\widetilde{Z}(r_k, r_j)$, where $r_j \in \mathbb{F}^s$ is once again sampled by the verifier during the sum-check protocol. The first three are evaluated by the verifier locally (or by using a (sparse) polynomial commitment scheme [135]).[6] The prover proves evaluations of $\widetilde{E}$, $\widetilde{W}$, and $\widetilde{\mathrm{x}}$. Below, we show how the verifier can efficiently evaluate $\widetilde{Z}(r_k, r_j)$ using an evaluation of $\widetilde{W}$ and $\widetilde{\mathrm{x}}$.

[6] Even in the case the verifier evaluates these polynomials locally, the cost is $O(n)$ and is independent of β, whereas without a sparse polynomial commitment scheme, Spartan would require $O(n \cdot \beta)$ time.

Computing $\widetilde{Z}(r_k, r_j)$. WLOG, assume that for each k, $|W[k]| = |\mathsf{x}[k]| + 1 = m/2$. For all $0 \leq k < \beta$ and $0 \leq j < m$, we have

$$Z_{k \cdot m + j} = \begin{cases} \mathsf{x}[k \cdot m/2 + j/2], \text{ if } j < m/2 \\ W[k \cdot m/2 + j/2], \text{ otherwise} \end{cases} \tag{3.5}$$

Let b be the binary representation of $(k \cdot m + j)$. We use $b[i]$ to refer to the bit i of b with $b[0]$ corresponding to the MSB. Then by equation (3.5), we have that $\forall b \in \{0,1\}^{c+s}$

$$Z[b] = \begin{cases} \mathsf{x}[b[0, 1, \ldots, (c-1), (c+1), \ldots, (c+s)]], & \text{if } b[c] = 0 \\ W[b[0, 1, \ldots, (c-1), (c+1), \ldots, (c+s)]], & \text{otherwise} \end{cases}$$

Thus, for the multilinear extensions of Z, W and x:

$$\widetilde{Z}(r) = (1 - r[c]) \cdot \widetilde{\mathsf{x}}(r[0, \ldots, (c-1), (c+1) \ldots]) + \quad r[c] \cdot \widetilde{W}(r[0 \ldots (c-1), (c+1) \ldots])$$

Thus, the prover sends an evaluation of $\widetilde{W}$ and $\widetilde{\mathsf{x}}$ at point $r' = r[0, \ldots, (c-1), (c+1), \ldots]$ along with a proof of correct evaluation. This aids the verifier with completing the final step of the interactive argument depicted above.

Proving IO Consistency Checks. The prover can send x and the verifier can check: (1) x is consistent with the commitment $\bar{\mathsf{x}}$ it holds as part of the running instance; (2) x satisfies the desired IO consistency; and (3) x is consistent with the public input/output X in the running instance. However, this incurs $O(\beta)$ proof sizes and verifier times. Instead, Phalanx does the following: At the time of folding, instead of committing to x, the prover commits to a "deduplicated version" of x (which obviates the need to prove check (2)). The prover then uses a simplified Spartan to prove the knowledge of x such that checks (1) and (3) hold; the prover also proves the evaluation of the multilinear extension of x necessary to compute $\widetilde{Z}(r_k, r_j)$ described above. The proof sizes are $O(\log(\beta))$ and verification times requires $O(\beta)$ finite field operations. Using sparse polynomial commitments from Spartan [135], the verification times can also be made $O(\log \beta)$ cryptographic

operations; however, doing so provides benefits only when β is large (for instance, when $\beta > 2^{15}$).

3.3.5 Phalanx's Commitment Scheme and Phalanx-lazy

Phalanx needs a polynomial commitment scheme. For this, Phalanx uses Dory [103], which results in the following asymptotics. Phalanx produces $O(1)$-sized proofs and verification times for each Verdict epoch (§3.3.3). Furthermore, for proving the running instance (§3.3.4) with β data-parallel units each of size c, Phalanx produces $O(\log(c \cdot \beta))$-sized proofs that can be verified in $O(\log(c \cdot \beta))$ time.

Another commitment scheme choice is Hyrax-PC [157]. With this choice, Phalanx provides $O(1)$-sized proofs and verification times for each Verdict epoch (§3.3.3), as before. However, it provides a different tradeoff between the proving costs and verification costs of running instances. Specifically, in the context of Verdict, Phalanx with Hyrax-PC has $\approx 5\times$ lower proving costs than Phalanx with Dory. However, proofs of running instances are $O(c \cdot \beta)$-sized and verifying them takes $O(c \cdot \beta)$ time, which is high. So, they can be verified only infrequently. Given this, we refer to this variant of Phalanx as "Phalanx-lazy" since it provides *deferred* guarantees meaning that the guarantees hold only when clients, in some future epoch, check a succinct proof produced by the prover.

3.4 Implementation and Applications

We implement Verdict in about 7,400 lines of Rust. This consists of an implementation of the Verdict service, which uses Redis [9] to store its state (a map from labels to append-only hashchains and an indexed Merkle tree that in turn stores a derived dictionary), and a client library that exposes VerifyUpdates and VerifyLookup procedures to verify proofs produced by the service. This is about 3,600 lines of Rust. We implement Phalanx as a library by extending libSpartan [10] and Xiphos [138] with about 6,000 lines of Rust.

Implementing Verdict requires constructing a SIMD R1CS instance that verifies a batch of indexed Merkle proofs. However, libSpartan provides only a low-level API that accepts R1CS matrices, which is unusable for more complex applications such as Verdict. We address this by leveraging bellman [2, 3], which provides R1CS "gadgets" for hash functions and other primitives that can be composed and extended to build higher-level apps. Specifically, we implement an adapter that lets a programmer compose and use existing bellman gadgets with libSpartan; this is about 1,000 lines of Rust. For the hash function in Verdict's SIMD R1CS, we use MiMC [17], a SNARK-friendly hash function.

Application: Key Transparency. As a concrete application of Verdict, we design Keypal, a public-key directory, where the service's state is a label-value map in which labels are client identifiers (such as email addresses) and values are the set of public keys associated with the identifier. Keypal supports four types of requests from clients: **(1)** register a new id and associate an initial key, **(2)** add a new key to an existing id, **(3)** revoke a key associated with an existing id, and **(4)** look up the set of keys associated with an existing key. For adding or revoking keys, Keypal uses a simple policy that such requests must be digitally signed by one of the existing, unrevoked keys associated with the identity. Keypal's implementation uses Ed25519 signatures [8].

3.5 Evaluation

Our principal experimental questions are:

- What is Verdict's performance compared to prior work?
- How do Verdict's techniques improve its costs?

We run our experiments on Azure Standard F64s_v2 (64 vCPUs, 2.70 GHz Intel(R) Xeon(R) Platinum 8168, 128 GB RAM) running Ubuntu 18.04. However, we only utilize one CPU core in our experiments.

To summarize our results, Verdict's lookup proofs (a few thousands of bytes) are shorter than its baselines and their verification is fast (a few milliseconds of CPU-time). Producing succinct proofs for updates with Verdict takes at most a few minutes (on a single CPU core) even when the service needs to apply thousands of updates on dictionaries with a million labels. Additionally, we find that Verdict's techniques improve over the proving costs of Spartan [135] by an order of magnitude. Together, Verdict achieves about 4 updates/sec/CPU-core and about 2 inserts/sec/CPU-core, with a per-epoch (amortized) proof size of 651 bytes and a verification time of about 3 ms (for a dictionary with 2^{20} label-value tuples); for the same workload, Verdict-lazy achieves about 18–22 updates/per/sec/CPU-core and 9–11 inserts/sec/CPU-core, with a per-epoch proof size of 290 bytes and a verification time of 161μs at the cost of deferred guarantees. This is over an order of magnitude improvement over prior state-of-the-art, general proof systems for stateful services [104, 136].

3.5.1 Comparison with Prior Work

We compare Verdict and Verdict-lazy with three baselines and use Keypal as the concrete application.

The first baseline we compare with is AAD [150], a prior transparency dictionary with asymptotic and security properties similar to Verdict. Unlike Verdict, AAD requires a trusted setup.

The second baseline we use is a system in which the service organizes its dictionary in a Merkle-Patricia tree. In epoch t, the prover sends a commitment $\mathscr{C}_t$ to its updated state and the following: for each update, the label, the new value, the old value (if it exists), and a membership proof. The verifier uses the membership proof to verify that the old value (or its absence) is consistent with $\mathscr{C}_{t-1}$ and the new value is consistent with $\mathscr{C}_t$. We call this baseline *Naive*.

The third baseline we use is an optimized version of the Naive baseline in which the service applies a batch of updates at once, and instead of producing a separate proof for each update, it merges the individual proofs and deduplicates them when possible. We call this baseline *Naive++*.

Note that Naive and Naive++ are more efficient than prior work such as SEEMless [51], which

additionally protects privacy, so the service cannot directly send Merkle proofs to clients and hence incurs additional expense. Similarly, other privately-verifiable works [85, 115] when transformed to produce publicly-verifiable proofs would incur more expense than Naive and Naive++.

We are interested in the performance of two operations: **(1)** lookups, and **(2)** a batch of inserts/updates. For lookups, our evaluation metrics are: **(a)** the size of a lookup proof; and **(b)** the verifier's cost of verifying a lookup proof. For updates, our metrics are: **(a)** the prover's cost of processing a batch of updates and producing an update proof; **(b)** the size of an update proof; and **(c)** the verifier's cost to verify an update proof. We do not focus on the cost of producing a lookup proof since in Verdict and the naive baselines, the cost is based purely on the performance of the underlying storage system used to maintain state (recall that in Verdict, a lookup proof is produced by retrieving appropriate nodes in an indexed Merkle tree stored in a commodity storage service). Verdict supports thousands of lookup requests per second on a commodity VM, and it can be scaled up with standard systems techniques.

To measure the performance of AAD [150], we use its C++ implementation [6]. Unfortunately, it only supports small dictionaries (for example, 8,192 label-value tuples); for larger dictionaries, the prover time is several hours or more, so we use results from smaller experiments and the authors' cost models to extrapolate its costs for larger dictionary sizes. Furthermore, we assume that there is only one value associated with the requested label as the lookup proof sizes and verification times grow with the number of operations associated with the requested label. For updates/inserts, AAD's performance depends on the number of trees in the forest that comprises the directory, so for a desired dictionary size, we measure the cost of doubling the dictionary and compute the amortized per-operation cost (this is optimistic for AAD).

For the naive baselines, we report costs based on microbenchmarks and cost models. We assume that the Merkle-Patricia trees use SHA-256 for the hash function.

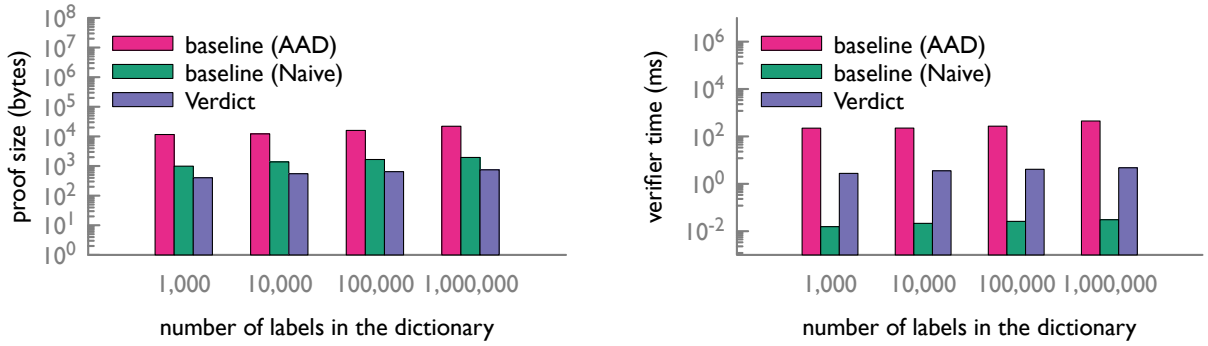


Figure 3.7: Proof sizes and verification times for lookups, with varying number of labels in a dictionary. Results for Naive++ and Verdict-lazy are the same as the results for Naive and Verdict respectively, so we do not depict them.

Results for Lookups. Figure 3.7 depicts our results. First, Verdict provides the shortest lookup proofs. Verdict has 28–29× smaller proofs than AAD. Proofs in the Naive baseline are ≈2.5× larger than Verdict's proofs because each membership proof in a Merkle-Patricia tree contains more data in intermediate nodes.

Second, Verdict's lookup proof verification times are not the fastest, but they are fast: 2.5–5 ms for Verdict and tens of microseconds for the naive baselines. This is because the naive baselines use SHA-256, whereas Verdict uses a SNARK-friendly hash function, which is more expensive on x86. However, Verdict's verification times are more than 80× cheaper than AAD's.

Results for Inserts/Updates. We experiment with Verdict and its baselines using $\beta = 128$ with varying number of labels in the dictionary. To demonstrate scalability, we also experiment with varying batch sizes for a dictionary with 2^{24} labels. For the latter large-scale experiments, we use a different VM with larger RAM, which is an Azure Standard E64-16ds_v4 with 504 GB RAM (we use only one CPU core and its performance is identical to that of the VM listed earlier); also, Verdict uses $< 50\%$ of the available memory in all experiments.

Figure 3.8 depicts our results. We find the following.

Verdict's per-epoch constant-sized proofs are 651 bytes and take about 3 ms to verify. Verdict's logarithmic-sized proofs for the running instance are 26–32 KB for a dictionary with 2^{24} labels,

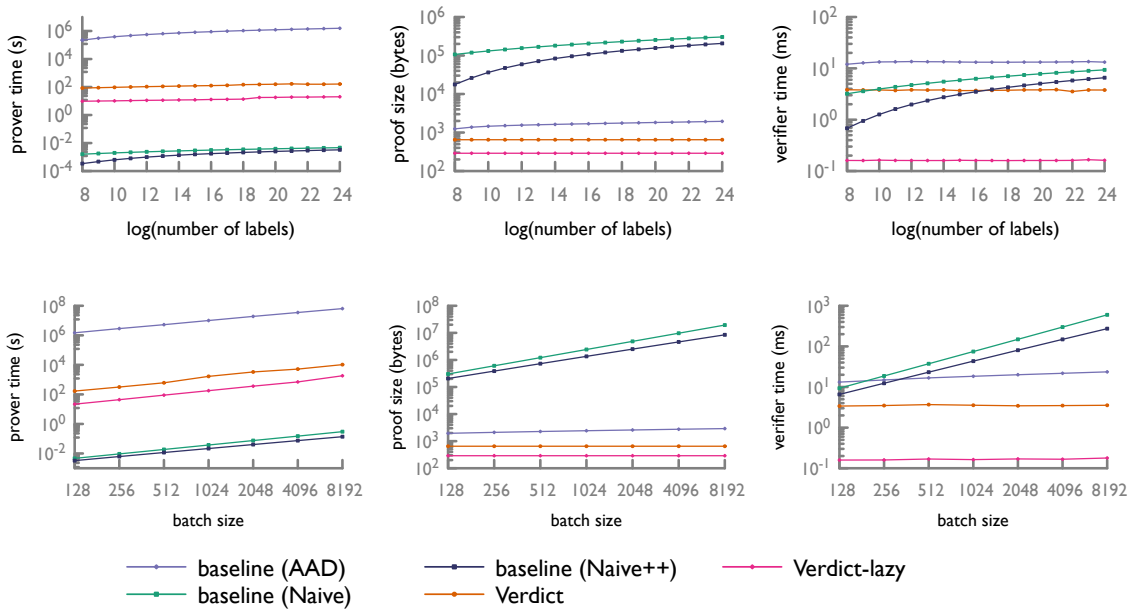


Figure 3.8: Per-epoch prover time, proof sizes, and verifier time for updates/inserts. The first row depicts costs for a batch size of $\beta = 128$ with varying number of labels in a dictionary (n), and the second row depicts costs for $n = 2^{24}$ and varying β. For Verdict, the depicted prover time includes the cost of producing both a constant-sized proof and a logarithmic-sized proof, while the depicted proof sizes and verification times are for the constant sized proofs (since clients can amortize the the cost of verifying logarithmic-sized proofs); similarly, for Verdict-lazy, we depict per-epoch costs; §3.5.1 provides details of amortization.

when the batch size varies from $\beta = 2^7$ to $\beta = 2^{13}$; verifying these proofs takes 44 ms for $\beta = 128$ and 77 ms for $\beta = 2^{13}$.

Recall that Verdict's clients only need to verify the logarithmic-sized proof of the most recent epoch (before issuing lookup operations). However, we find that, for a dictionary of size 2^{24}, even if clients check these proofs in *every* epoch, Verdict's clients are more efficient than the ones of Naive and Naive++ (which employ a fast hash function) as long as $\beta \geq 2^{10}$. Furthermore, at this configuration, Verdict's proof size is over an order of magnitude shorter. Specifically, it is ≈30 KB whereas Naive's proof size is ≈2.4 MB and Naive++'s proof size is ≈1.4 MB. Verdict's logarithmic-sized proofs are concretely larger than AAD's, but Verdict's clients can incur lower amortized proof sizes and verification times (for instance, if clients verify these logarithmic-sized proofs after 14 epochs when the batch size is 2^{13}). For prover times, Verdict is slower than both Naive and Naive++, but Verdict is faster than AAD by 2–3 orders of magnitude.

Verdict-lazy produces shorter per-epoch proofs than Verdict: proofs are 290 bytes and verification times are 161μs. However, generating proofs for running instances is over 3$\times$ slower than in Verdict (for example, for dictionaries with 2^{24} labels and $\beta = 128$ Verdict-lazy takes takes 303 s whereas Verdict takes 69 s). Finally, verifying these proofs is slower than in Verdict by over an order of magnitude: for the aforementioned workload, the verifier under Verdict-lazy takes 29s. So, unlike Verdict, Verdict-lazy needs hundreds of epochs to achieve faster verification times than the naive baselines.

Storage Costs. Compared to a baseline that stores only a dictionary, Verdict incurs $\approx$2$\times$ overhead from maintaining an indexed Merkle tree. This overhead is similar to those of Naive and Naive++, but much smaller than AAD, Merkle^2, CONIKS, or SEEMless. Concretely, for a dictionary with 2^{20} labels, Verdict uses 128 MB to store the indexed Merkle tree. Moreover, Verdict maintains a hashchain for each label. Each node in the chain is 167 bytes and contains an operation (49 bytes), a signature (78 bytes), and the hash of the previous node (40 bytes). Finally, for the service to produce proofs, it stores SNARK parameters. Phalanx's parameters are smaller than Phalanx-lazy's parameters, both asymptotically and concretely. Concretely, for a dictionary with 2^{24} labels and $\beta = 1024$, the parameters are about 103 MB under Phalanx and 2 GB under Phalanx-lazy.

3.5.2 Improvements From Verdict's Techniques

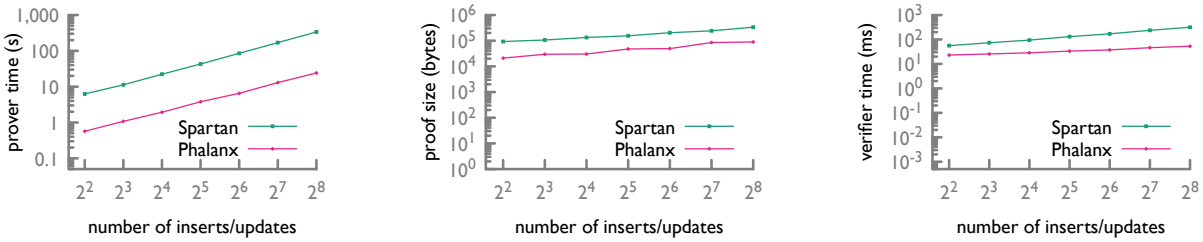


Figure 3.9: Prover time, proof sizes, and verifier time under Phalanx and its baseline Spartan [135] for a SIMD R1CS instance with varying number of insert/update operations on a dictionary with 2^{20} labels (see the text for details).

	Phalanx-eager	Phalanx
Prover time	72 s	168 s
Proof size (per-epoch)	26 KB	651 bytes
Verifier time (per-epoch)	44 ms	3.4 ms
Proof size (fixed)	N/A	26 KB
Verifier time (fixed)	N/A	44 ms

Figure 3.10: Performance of Phalanx-eager and Phalanx for proving a batch of 128 insert/update operations on a dictionary with 2^{24} labels. See the text for details.

Benefits of Phalanx's polynomial IOP for SIMD R1CS. We consider a SIMD R1CS instance where each sub-circuit performs one update/insert operation on a dictionary with 2^{20} labels. We measure the performance of Phalanx with a varying number of update/insert operations, and our performance metrics are: (1) the prover time, (2) the proof size, and (3) the verifier time. Our baseline is Spartan [135]. To focus performance on the polynomial IOP, we configure Phalanx to use the same polynomial commitment scheme as Spartan, which is Hyrax-PC configured to produce $O(\sqrt{m})$-sized commitments for m-sized multilinear polynomials.

Figure 3.9 depicts our results. We find that Phalanx is more efficient than Spartan, with Phalanx's prover faster by over an order of magnitude compared to Spartan. Prior to this work, Spartan offers the fastest zkSNARK [135], so for SIMD computations, Phalanx has the fastest prover.

Cost and benefits of Phalanx's folding scheme. To evaluate the benefits of Phalanx's folding scheme, we consider a variant of Phalanx that proves SIMD R1CS instances but does not employ Verdict's folding scheme. We refer to this variant as "Phalanx-eager". Asymptotically, for an N-sized SIMD R1CS, the prover times are $O(N)$ under both Phalanx and Phalanx-eager; the proof sizes and verifier times are $O(\log N)$ under Phalanx-eager whereas they are $O(1)$ under Phalanx. For proving running instances, Phalanx however produces $O(\log N)$-sized proofs that take $O(\log N)$ time to verify, but this cost can be amortized across epochs at each client's discretion (§3.3.5).

Figure 3.10 depicts our results for applying a batch of 128 inserts/updates on a dictionary with 2^{24} labels (the relative results for other dictionary sizes and batch sizes are similar). Since Phalanx

must commit to an additional vector T (§3.3.3), its prover incurs higher costs than Phalanx-eager by about $2.3\times$. In exchange, the per-epoch verifier time and proof sizes are both lower by over an order of magnitude under Phalanx compared with Phalanx-eager. Even when accounting for fixed costs incurred by Phalanx, Phalanx still incurs lower verifier costs and proof sizes than Phalanx-eager as long as the fixed costs are amortized over ≥ 2 epochs.

4 | KAROUSOS

This chapter studies the problem of verifying outsourced event driven web applications. To illustrate the problem, consider the following realistic scenario.

Cam has source code for a web application (for example, written in Node.js), which Cam has written or otherwise trusts. Cam deploys that code on a remote server (for example on AWS, Azure, or GCP). But Cam doesn't trust the server to execute the application *faithfully*, meaning according to its code (see Chapter 1 for sources of unfaithful execution). How can Cam get assurance that the intended application is executing faithfully?

Orochi [146] explicitly targets Cam's scenario. In Orochi, a *verifier* (a machine under Cam's control) performs a comprehensive audit. The verifier is given as ground truth a *trace* [146, §1,§4.1,§7] of exactly the inputs to, and outputs from, the server (see also [26, 27]). Then, the verifier *re-executes* from the inputs in the (trusted) trace, checking that the re-executed outputs match the outputs in the trace. Crucial to this process is (untrusted) *advice* that the verifier receives from the server. This advice enables the verifier to accelerate re-execution versus naive replay, by re-executing requests in suitable *batches*, deduplicating instructions that are identical across the batch. The advice also helps the verifier make sense of concurrent executions.

On the one hand, Orochi inspires us; we borrow the setup just described. On the other hand, Orochi has a restricted execution model, which limits applicability to a small subset of web applications. First, each client request in Orochi must be handled within a single execution context, as in PHP. This rules out web applications that use event-driven frameworks such as Tornado [13],

Node.js [14], and Phoenix [15]. Without taking a position in the eternal events-versus-threads debate, we note that most modern web application frameworks are written in the event-driven style. Second, Orochi assumes that little state is shared between execution contexts; if more state were shared, Orochi's protocols would require the server to send an impractically large quantity of advice to the verifier. Third, external state in Orochi, such as a database, is assumed to meet the strong condition of strict serializability [123]; yet, many external data stores default to weaker isolation levels and may not even offer strict serializability [30].

Addressing these restrictions introduces new technical problems. Defining and solving them is the work of this paper, which we do in the context of a system called *Karousos*. Like Orochi, Karousos is a record-replay system [56, 57, 65] (§2.3). Karousos makes the following contributions:

A new record-replay technique for event-driven systems, which balances re-execution throughput and server logging. The more the verifier can batch requests and deduplicate instructions, the higher the throughput of re-execution. But the more batching is permitted, the more the re-execution can be reordered versus the original execution. And the more reordering, the more the server has to log and transmit to the verifier (in the advice) to facilitate faithful re-execution.

Karousos shifts the tradeoff curve and identifies a point on the shifted curve, using several interlocking ideas. First, Karousos's verifier batches together requests that induce the same trees of events (§4.3.1), regardless of the order in which the corresponding handlers were originally executed. Second, Karousos introduces a notion of *R-ordered* (§4.3.2): two dependent operations during execution (for example, a write of a program variable followed by a read of that variable) are *R*-ordered if they are guaranteed to be *re*-executed in that same order. The server then logs only operations that are not *R*-ordered. Third, for unlogged operations, the verifier consults a *version history* that it constructs while re-executing (§4.3.2).

Ensuring that re-executions are sensible. Without further mechanism, a misbehaving server could make the verifier accept executions (as embodied in traces) that are inconsistent with having executed the original code (§4.3.3, §4.3.4). At a high level, Karousos handles server misbehavior by requiring that the order in which requests are served, program variables are accessed, and database accesses are issued are consistent with each other—and with the program. Doing so requires care because the verifier must check consistency across three sources of ordering, only one of which (the trace) is trusted; the others must be validated with specific kinds of crosschecks or through the course of re-execution.

At a lower level, Karousos uses several techniques. For program variables, the Karousos verifier reconstructs an alleged partial order of variable accesses while re-executing (§4.3.3). For databases (§4.3.4), the principal correctness conditions are different isolation levels (serializability, read-committed, and so on). On the one hand, there are algorithms for testing isolation, notably Adya's work [16]. On the other hand, these works are predicated on the assumption that database history and internal state is available to the verifier. The Karousos verifier runs Adya's algorithms against an alleged history, thereby contingently justifying that history, and then ensures that the contingent history is consistent with the rest of the execution.

Proof of correctness. We prove (Appx. B) that Karousos's algorithms are complete (the verifier accepts faithful executions) and sound (the verifier rejects unfaithful ones) (§4.1.1).

Implementation. Our implementation of Karousos supports web applications that are written in Node.js and use MySQL as a transactional key-value store. As we explain later (§4.4), developers wanting to use our implementation need to annotate portions of their code. Our implementation supports a core of JavaScript, disallowing certain other constructs.

We have evaluated Karousos on three model web applications (§4.5). For realistic workloads, a Karousos application has response latencies that are between 1.4–3.5$\times$ higher than an unmodified

baseline; we believe this is a reasonable price to pay for execution integrity. On traces with more than 50 requests, the Karousos verifier is between 1.3–14× faster than an alternative that sequentially re-executes requests. The advice produced by the Karousos applications are of reasonable size (200KB to 6MB for 100 requests). Finally, Karousos's core techniques are key to the re-execution throughput and the manageable advice size.

These results are encouraging, but Karousos has clear limitations. First, JavaScript workers are disallowed; this is not fundamental. More fundamentally, timers are disallowed, range queries on transactional state are not supported, and snapshot isolation is not supported. Addressing these restrictions would require extending our algorithms and proofs; we leave this to future work. Finally, at the level of architecture, Karousos verifies only a single web application, not its interactions with other server-side components such as other web services (at least not in a verifiable way).

The bottom line, however, is that Karousos takes a big step forward: it shows how to get assurance about the execution of event-driven web applications in realistic scenarios.

4.1 Setup and background

4.1.1 Problem: comprehensive server audit

Here, we define our problem abstractly, to showcase the challenge while avoiding distracting details. Later (§4.4), we will translate it to event-driven web applications. Our presentation is inspired by, and has some textual debts to, Orochi [146, §2].

Figure 4.1 depicts the problem. Some *principal* (like Cam) deploys a *program P* on an untrusted *server* (for example, running on a cloud platform). Clients make *requests* to the server. Requests can be concurrent with each other, and *P* can be a concurrent program. A *response* is allegedly the result of invoking *P* against the corresponding request.

In this setting, the principal has access to a *trace* of the actual requests and (possibly unfaith-

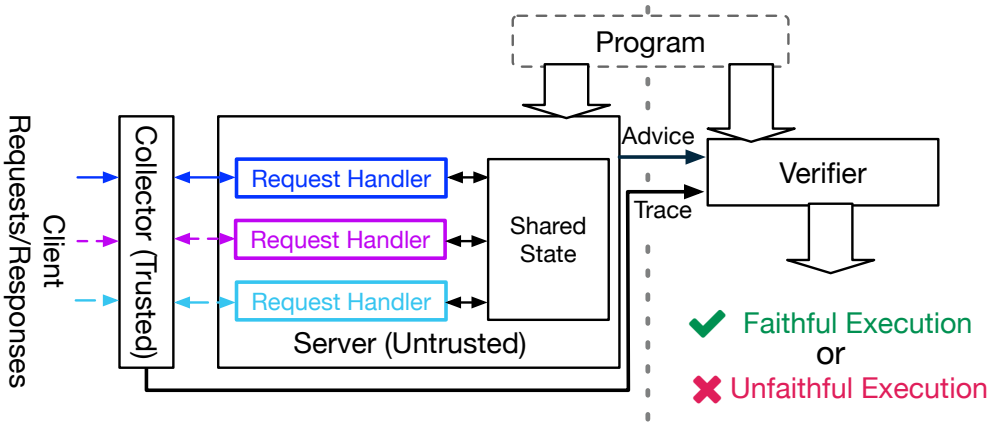


Figure 4.1: The problem: efficiently auditing an untrusted server.

ful) responses. The trace is provided by a *collector* that is computationally less demanding than the server, and which is assumed to work correctly (see Orochi [146, §1,§4.1,§7], Dog [26], and EAR [27]) for deployment scenarios for trace collection). We can think of the trace as the ground truth record of what enters and leaves the server.

The server is supposed to follow a defined reporting procedure during execution, which produces *advice*. However, the server is untrusted and could either decline to produce advice, or generate adversarial advice designed to deceive the system.

Using the (ground truth) trace and the (untrusted) advice, a *verifier* that the principal controls periodically conducts an *audit*, to determine whether the responses in the trace were in fact produced by executing P on the requests in the trace. A constraint is that the (local) verifier has much less computational capacity than the (remote) server. Likewise, the verifier and server are connected by a network with limited capacity. The verifier and the advice should satisfy these properties:

- *Completeness.* If the server behaved during the time period of the trace (which includes collecting advice honestly), then the verifier must accept the given trace.
- *Soundness.* The verifier must reject if the server misbehaved. Specifically, the verifier accepts only if there is some schedule S of (possibly concurrent) executions, such that: (a) executing the given program against the inputs in the trace, while following S, reproduces exactly the respective

outputs in the trace, and (b) S is consistent with the ordering in the trace. (Appendix B.2 states Soundness precisely.) This property means that the server can pass the audit only by behaving in a way that is, to external observations, indistinguishable from actually executing the program on the received requests.

- *Efficiency.* This means several things in our context. (a) The verifier, being computationally weaker than the server, needs to perform less computation than the server; in particular, the work of the verifier should be computationally less costly than naively re-executing each request in the trace one-by-one. (b) The advice sent from the server to the verifier needs to be kept small. (c) Advice collection should not significantly impact the server's response latency. We are willing to tolerate some computational overhead at the server, as we expect auditability to cost something.

4.1.2 Our starting point: Orochi

As stated in the introduction, Orochi [146] re-executes all requests in a trace, checking that the produced responses match the outputs in the trace.

Orochi addresses the challenge of a computationally limited verifier by exploiting an aspect of web applications: many executions follow the same code paths [92, 146]. The Orochi server is supposed to track control flow, and then specify (in the advice) *control flow groups*, meaning which requests have the same control flow as each other. The verifier then re-executes a single control flow group as a batch, using *SIMD-on-demand*. If an instruction has the same operands across a batch, the verifier re-executes that instruction only once, and otherwise executes the opcode for each request in the batch. This technique is facilitated by a datatype called a multivalue, which collapses when all of the entries in the multivalue are identical, and expands into a vector when needed. If the execution within the group diverges, the verifier rejects.

Given batching (which can group together a later request with an earlier one), a read operation

may be re-executed before the dictating write operation is re-executed. Consequently, the advice should tell the verifier how to re-execute the read. Yet, the advice is untrusted; it could be wrong. This is one way in which Completeness, Soundness, and Efficiency are in tension: the advice is necessary (for Efficiency and Completeness), but possibly wrong (threatening Soundness).

Orochi's solution includes a technique called *simulate-and-check*. The advice allegedly contains, for each object that is shared among requests, a linear log of the values read and written. When re-executing a read operation, the verifier feeds that operation from the most recent write, according to the log. When re-executing a write operation, the verifier checks that the value produced by re-execution matches what is in that object's log, thereby validating the values that have fed, or will feed, reads.

Despite this technique, the server could still arrange responses and advice to cause the verifier to accept bogus executions [146, §3.4]. Consequently, another technique in Orochi is *consistent ordering verification*, in which the verifier builds a graph that includes every operation, request arrival, and response delivery, with edges indicating various kinds of order (time-order between requests, program order between operations, operation order from the logs). The verifier then insists that the graph is acyclic.

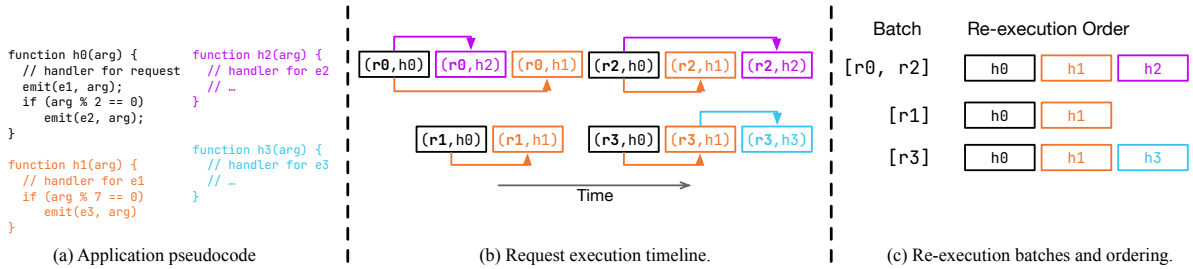
Orochi's techniques provably result in Completeness and Soundness. However, Orochi makes a number of simplifying assumptions. First, although the server is concurrent, requests themselves are handled mostly in isolation, in straight-line fashion (with the unrealistic assumption that when a response is delivered, the request has no further effect). This rules out many web application architectures and all event-driven frameworks. Second, Orochi's approach to logging would lead to unacceptably verbose logs (contra the Efficiency goal; §4.1.1) in a setting where a lot of state is shared between discrete execution units (for example, program variables that are accessed by multiple event handlers). Third, external state such as databases must be strongly consistent, and must be accessed synchronously; this too rules out many deployment scenarios. The sections ahead delve into the technical work required to relax these assumptions.

4.2 EXECUTION MODEL

We define an execution model, *KEM*, for unmodified concurrent web applications. In subsequent sections we use KEM to describe our core algorithm; the proofs presented in the Appendix also build on KEM. KEM is intended to capture the semantics of Node.js programs, and it does not model the behavior of a database or external state store. We defer a discussion of how Karousos models and handles interactions with databases to §4.3.4. Furthermore, KEM models a runtime that can have multiple concurrent threads executing at a time. This is more general than the Node.js runtime (and indeed other JavaScript runtimes) which is single-threaded, allowing us to minimize assumptions about the runtime.

KEM models the state of a program as a set of variables, a set of zero-or-more pending *events* and a set of zero-or-more *event handlers*. Program code can read or update any in-scope variable. However, similar to JavaScript, functions and closures capture variables by reference. Consequently, all variables in scope when a function or closure is defined are in scope for the body of the function; even local variables might be accessed from multiple functions. As a result, a variable might be concurrently accessed and updated by multiple concurrent threads. KEM assumes all accesses are sequentially consistent [98]. This assumption reflects reality: existing JavaScript runtimes, including Node.js, are single-threaded and most extant JavaScript code is written assuming sequentially consistent access.

Events in KEM are associated with a name and a type. Multiple events of the same type can occur during execution, and the set of pending events can contain multiple events of the same type at a time. Events can be added to the pending events by the runtime or by user code. The runtime adds I/O events, including ones for new user requests or when a database query has finished running. Program code can add to the set of pending events by calling a designated *emit* function. Events are removed from the set of pending events by the runtime's dispatch loop: each iteration of the loop non-deterministically selects an event from the set, removes it from the set, and then uses the



(a) Application pseudocode

(b) Request execution timeline.

(c) Re-execution batches and ordering.

Figure 4.2: Grouped re-execution in Karousos: (a) Pseudocode for a simple application; (b) An example execution trace. A directed arrow between handlers h_0 and h_1 indicates that $h_0 \prec_A h_1$, that is, the arrows represent the activation partial order; (c) Re-execution groups and the order in which groups are re-executed. Observe that requests r_0 and r_2 are batched together for replay despite executing handlers h_1 and h_2 in different orders.

selected event's type to identify and call the appropriate event handlers.

As in JavaScript, KEM event handlers are closures. Program code can add and remove handlers by calling designated *register* and *unregister* functions. Both functions take as input an event type and a closure, which we sometimes refer to, loosely, as the *function* associated with the event. Event handlers in KEM can perform computation, modify in-scope variables, emit events and register or unregister handlers. KEM assumes that event handlers run to completion and that a handler's execution is not interrupted when it emits an event. Our exposition uses the term *handler activation* to refer to the act of the runtime's dispatch loop calling an event handler.

Programs in KEM begin execution by calling a designated initialization function. This models the fact that JavaScript programs, including Node.js programs, generally place initialization code outside of a function body. We assume that the initialization function is deterministic (and later rely on this assumption for replay fidelity).

Activation partial order. The execution model described above induces a partial order on handler activations, A. Given a handler activation h_0, define the relation activator$(h_0) = h_1$ if and only if h_0 emitted the event e resulted in the activation h_1 or h_0 issued the I/O request or database requests whose completion resulted in h_1's activation. Observe that our execution model ensures that activator$(h) \neq h$ and that a handler activation h has at most one activator. Furthermore, our definition of the *activation* relation implies that any handler activation h without an activator (that

is, $\nexists h'$ s.t. activator$(h) = h'$) must have been run in response to a user request. For analytical convenience, we treat the initialization function's execution as a handler activation I. Furthermore, we use I as the activator for all user request activations. Thus any handler activation $h \neq I$ has a unique activator(h) and activator$(h) \neq h$. Given this definition of the activator relation, we define the partial order A as the transitive closure of the activator relation. That is, we say $(h, h') \in A$ if activator$(h') = h$ or there exists $h = h_0, h_1, h_2, \ldots, h_n = h'$ such that $1 \leq i \leq n$, activator$(h_i) = h_{i-1}$. We sometimes write $(h, h') \in A$ as $h \prec_A h'$.

One can visualize each user request activation as inducing a tree of handlers, with edges given by the activator relation.

Related work. KEM extends λ_{JS} [79] which provides a semantic model for JavaScript. While λ_{JS} does not model events, prior work [109] shows extensions that model several event-driven frameworks. Similarly, KEM extends λ_{JS} by adding constructs for registering and unregistering event handlers (or listeners) and for emitting events. Unlike these works, KEM does not make assumptions about the order in which event handlers are executed nor about the number of concurrently executing event handlers. As a result, KEM models a more general execution environment. Thus, our algorithms, which are designed to check execution integrity for all KEM executions, can be used with other languages and event-driven frameworks. Furthermore, this generality means that Karousos can be used even with future Node.js runtimes that adopt different event dispatch loops or use multiple threads.

4.3 AUDITING EVENT-DRIVEN SERVERS

As in Orochi [146] (§4.1.2), the Karousos server collects advice that tells the verifier how to re-execute groups of requests simultaneously, which the verifier does using the SIMD-on-demand technique (§4.1.2). Karousos must address a key question: how should it group code to be re-

executed? There is an essential trade-off: the more batching that is permitted (and hence the more opportunity for re-execution efficiency), the more there can be reordering in re-execution (relative to the original execution). However, the more reordering, the more the server has to collect advice to facilitate faithful replay; for example, if a read of a program variable is re-executed before the dictating write for that read, then the re-executed read would have to be somehow fed from advice.

To highlight the trade-off, consider two extremes. Karousos could conceivably chop each request into small pieces, and re-execute structurally identical basic blocks from multiple requests simultaneously; this would require logging enough information so that each basic block has enough "context" to be re-executed faithfully. At the other extreme, Karousos could group together only identical requests that invoke identical handlers in the identical order and do not share state with each other; this would require essentially no logging.

Karousos aims for the midpoint of this trade-off: we want to enable a lot of reordering (to expose batching opportunities) while controlling the burden of logging. In the remainder of this section, we deal with each side of this issue: we describe the choice of batching granularity (§4.3.1) and how Karousos facilitates faithful replay of operations on program variables (§4.3.2), assuming an honest server. We then describe how Karousos defends against an untrusted server (§4.3.3). Section 4.3.4 extends the design to transactional state. We will present the key mechanisms abstractly, deferring concrete details to Section 4.4. Full algorithms, with proofs of correctness, are in the Appendix.

4.3.1 Batched re-execution in Karousos

In Karousos, a re-execution group comprises requests that have the same tree of handlers—that is, the same A relation (§4.2)—and the same in-handler control flows, meaning that corresponding handlers in different requests follow the same branches. Re-execution respects the A relation and program order within a handler but does not respect temporal order. Specifically, later requests can be re-executed before, or simultaneously with, temporally earlier ones. For example, in Figure 4.2, r_2 is later than r_0 and r_1, yet r_2 is re-executed together with r_0 and before r_1. Similarly, handlers

within a request, if not ordered by A, can be reordered during re-execution; for example, (r_0, h_1) and (r_0, h_2) in Figure 4.2.

Section 4.4 describes how the server tracks the A relation and the control flow within a handler. Having done so, the server places in the advice a tag for each request in the trace (§4.1.1), where requests with the same tags allegedly belong in the same re-execution group. Of course, the verifier does not trust that the server is honest about the claimed grouping. However, the verifier expects the server to include in its advice a description of the activation partial order A, and the verifier validates that description as it executes.

Specifically, the advice is supposed to include, for each request, a *handler log*, consisting of entries for each emit, register, and unregister (§4.2) operation. An entry specifies the alleged activator (the handler), the alleged event, and (for register and unregister) the allegedly registered/unregistered function. When re-executing an emit, the verifier "trusts" the handler log, which implicitly indicates which functions are registered for the given event (all functions that have been registered but not unregistered before the emit). When re-executing register and unregister operations, the verifier checks that these operations are exactly the ones that appear in the log, thereby vindicating the "trust" placed in the handler log when re-executing emits. There are additional checks, for example that all emit entries listed in the handler log correspond to events that materialize during re-execution. The Appendix contains the full details, including the necessary bookkeeping.

4.3.2 Trusted recorder, out-of-order replay

How does Karousos faithfully re-execute reads of program variables? As we saw (§4.3.1), requests can be re-executed in the opposite order from what happened originally, which means that a read can be re-executed before the corresponding write. This section assumes that the server is honest.

What the (honest) Karousos server does. As a strawman, the server could include in its advice the values of all read operations, which the verifier could use to feed each re-executed read operation [86,

92, 146] (see also Section 2). However, in logging every read or write of a program variable, this solution conflicts with the goal of controlling the log size (§4.1.1).

In contrast, the Karousos server decides dynamically whether to actually log a given operation. Karousos introduces the concept of *R-ordered*: two operations are R-ordered if one of them is guaranteed to be re-executed before the other under any possible grouping during re-execution. (We say that they are *R-concurrent* if they are not R-ordered.) More formally, we define a partial order R over operations. We say that $(o, o') \in R$ or $o \prec_R o'$ if (a) o was executed as a part of handler activation h, o' was executed as a part of handler activation h', and $(h, h') \in A$; or (b) o and o' were both executed as part of handler activation h and o was executed before o'. Observe that R can be regarded as the union of A and the program order, and that R is formalizing the constraints on re-execution that were stated in Section 4.3.1.

With this definition, we can now say what the Karousos server puts in the advice: as depicted in Figure 4.3, the server logs reads of program variables that are not R-ordered with respect to the dictating write. The server also logs writes of program variables that are not R-ordered with respect to the overwritten write; this helps validate executions from untrusted servers, as explained later (§4.3.3). In both cases, the server logs the function location and value written by the dictating or predecessor write.

In more detail, the principal (§4.1.1) must statically identify and annotate *loggable variables*. A variable is loggable if it might be accessed by R-concurrent operations; each such variable gets a *variable log*. If a variable is not loggable, then operations on it are simply re-executed during replay. If a variable is loggable, then the server logs only if the access is not R-ordered with respect to the dictating or overwritten write. We note that marking a variable that has no R-concurrent operations loggable impacts performance but has no effect on Karousos's soundness or completeness (§4.1.1). Conversely, not annotating a loggable variable does not impact Karousos's soundness (all unfaithful executions will be rejected) but compromises completeness (some faithful executions might not be accepted).

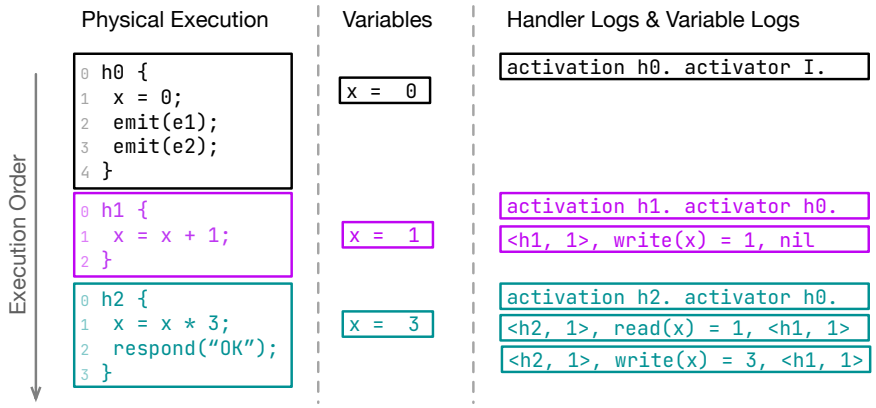


Figure 4.3: For program variables, Karousos only logs reads and writes that are not R-ordered. Execution points in the figure are depicted as a <handler, line number> tuple, so `<h2, 1>` means line 1 in handler h_2. When logging reads in a variable log, the server records the locations of the read and the dictating write. When logging writes, the server records the locations of the write and the write that is being overwritten.

Re-execution. Re-execution works as follows. For a given program read of a loggable variable, if the value of that read is indeed in the variable log, then the re-executor feeds the value from the log to the re-execution. If that read is not in the variable log, then (because the server is assumed to be honest), the read must be R-ordered with its dictating write. This implies, by definition of R-ordered, that by the time the read happens, the write was already re-executed, which means that in principle the read can be fed from that write.

We say "in principle" because Karousos must solve a problem: feeding the re-executed read with the correct write operation. To illustrate the challenge, consider the naive solution of simply applying re-executed writes to a reconstructed copy of the variable, and feeding non-logged reads from that variable. In the re-execution depicted in Figure 4.4, this naive solution would cause h_1 to incorrectly read `x=3` (the most recently re-executed write) rather than `x=0`, which is the value faithful to the original execution (Figure 4.3).

To eliminate this confusion, the Karousos re-executor keeps, for each loggable variable, all values written during the re-execution, indexed by the identifier of the handler and the operation number within the handler. We call this versioned variable the *variable's dictionary*. Figure 4.4 depicts the technique. The re-executor knows that if a read is unlogged, then originally that read

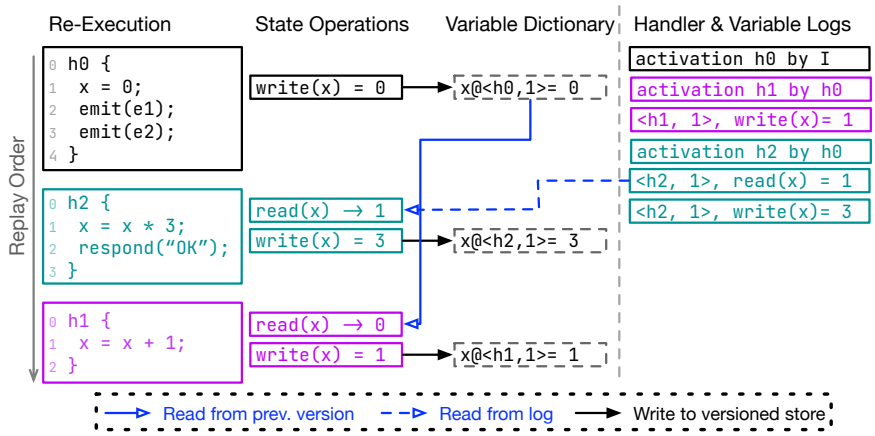


Figure 4.4: Re-execution in Karousos: During re-execution Karousos maintains a dictionary for each variable (the figure shows the dictionary for x) that contains previous values. Any logged reads return the logged value, while any unlogged reads use the most recent value (as defined by $\prec_R$) from the variable's dictionary. The figure depicts an abridged version of the logs from Figure 4.3. The notation `x@<h0, 1>` represents, for example, the value of `x` after line 1 of handler h_0 was executed.

must have observed a write that was prior according to R. To find that dictating write, the re-executor looks for the latest write in the variable's dictionary, where "latest" refers to the R relation. One can think of this as starting at the current handler, looking for the last write (if any) to the given variable by the current handler, and then repeating this step for each successive ancestor in the A tree until one encounters a write to the variable.

Here is a sketch for why this approach works; a full proof is in the Appendix (§B.3.1). If a read r is logged, then re-execution of course gets the correct value. If r is not logged, the dictionary interrogation, to be correct, needs to find the immediately prior causal write w that happened during the original execution. Meanwhile, we have $w \prec_R r$, otherwise the read would have been logged. But the dictionary interrogation is following R in reverse. Thus, if the dictionary interrogation stops at a different write $w' \neq w$, then we have $w' \prec_R r$, which together with $w \prec_R r$ and the fact that each operation has exactly one immediate predecessor in R, implies $w \prec_R w' \prec_R r$. Now, by definition of R (activation partial order A and program order), in the original execution, r would have observed w' not w, a contradiction.

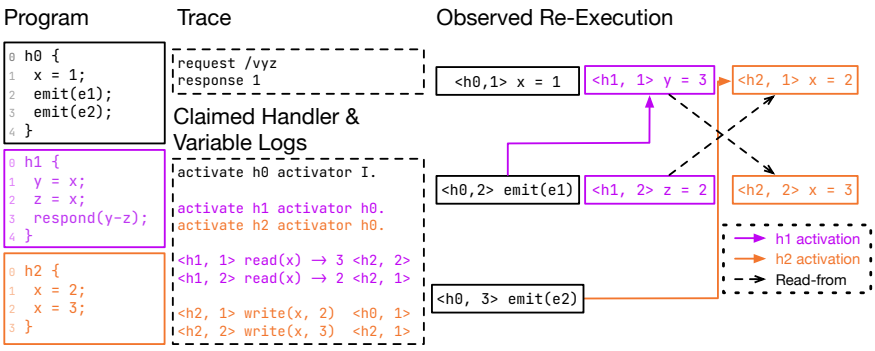


Figure 4.5: An example of what the Karousos verifier could observe when re-executing the given program based on a dishonest server's advice. Karousos allows out-of-order re-execution by design, and can thus observe the execution shown. However, this execution is physically impossible: based on the execution model (§4.2) and the possible interleavings, the response should never be positive. So, the verifier ought to reject it.

Discussion. Recall our goal of conserving log space (§4.1.1). First, and most important, the server places in the variable logs only what is necessary, given the possible reorderings that can happen from batched re-execution.

Second, we have designed the batching scheme so that logging is infrequently needed. In particular, looking at a tree of handlers where each handler touches state, a common pattern is that only the reads are concurrent with each other: consider, for example, an execution with one or more writes in a handler h, followed by a set of n reads, each in a handler h'_i, where h activates each $h'_1, \ldots, h'_n$. In this example, there is no logging required because each read is R-ordered: during the original execution, each read observes a write from h, which is an ancestor of the given h'_i. Notice that the preceding holds regardless of whether the h'_i are re-ordered during re-execution. Overall, this leads to good batching opportunities without much logging (§4.5.4).

4.3.3 Untrusted recorder, out-of-order replay

This section relaxes the assumption of a well-behaved server. To motivate the relevant mechanisms in Karousos, we will consider several attacks. However, the soundness of the protocol (§4.1.1) is not based on reasoning about each thing that can go wrong but instead on an end-to-end proof (§B.3.2).

Absent further mechanism, the adversarial server could place arbitrary values in a variable log,

thereby causing re-executed reads (§4.3.2) to be detethered from the expected program. Thus, the verifier checks that the values of re-executed writes match alleged writes in the variable log. This is essentially Orochi's simulate-and-check (§4.1.2), adapted to Karousos's log structure.

At this point, one might conclude that the worst the server can do is incriminate itself by failing to justify an execution. But in fact, by creating both bad advice and bogus outputs, the server could fool the verifier into accepting impossible executions, with semantically invalid responses. Figure 4.5 depicts a small example. Other misbehavior is possible too, for example, the server could arrange for the verifier to wrongly validate "reads from the future", which would enable the server to rationalize an allegedly-read but wrong value, provided some later request writes that value to a shared variable.

To ensure that the executions reproduced by the verifier are physically possible and consistent with external observations (meaning the trusted trace; §4.1.1), the verifier has a postprocessing phase, where it creates an execution graph G covering its entire audit. The graph establishes an alleged ordering among operations, and the verifier checks that it is acyclic. This technique creates a cycle in the example: $(h1,2) \rightarrow (h2,2) \rightarrow (h1,1)$. This technique is inspired by other systems, including Orochi [146] (§4.1.2); see also Section 2. Like Orochi, the Karousos verifier includes edges for time precedence (referring to the ordering of requests in the trace; §4.1.1) and program order of operations.

The novel aspects in Karousos are as follows. First, Karousos includes edges that reflect the alleged activation partial order, A, based on the handler logs (§4.3.1). Second, the Karousos verifier embeds in G the alleged operation history of all loggable variables. Notice that the history of accesses to a variable in the original execution should be a write, followed by zero or more reads, followed by a write, followed again by zero or more reads, and so on. The verifier reconstructs this partial order from a combination of re-executing and the variable logs.

Specifically, for each loggable variable and each write w to that variable, the verifier maintains during re-execution a list of `read_observers`: all the reads r that observe w in re-execution,

inferred from the variable log (if r was allegedly not R-ordered with w), or the versioned variable (if r was not present in the variable log). The verifier also maintains for each write w a `write_observer`: the write w' that succeeds w. Because w' and w might not be R-ordered—and thus the `write_observer` of w might not be inferrable from re-execution—write-write pairs are logged, as stated in Section 4.3.2. Now, after re-executing, the verifier uses these lists to embed edges in the graph G: WR (read-from edges, using `read_observers`), WW edges (write-write, using each write's `write_observer`), and RW (anti-dependency edges, connecting a given write's `read_observers` to that write's `write_observer`). Intuitively, these edges encode the history type mentioned in the prior paragraph.

Provided G has no cycles (and together with the verifier's other checks), the entire execution (of all requests in the audit) is well-ordered and physically possible, thus meeting the requirement of soundness (§4.1.1).

4.3.4 Transactional state

Model. We consider a transactional key-value store (KV store) that provides one of the following isolation levels: serializability, read committed, or read uncommitted [30]. Snapshot isolation is future work (§4). Each request issues operations to the KV store: `tx_start`, `tx_commit`, `tx_abort`, `PUT`, or `GET`. A transaction might be split across multiple handlers, but we assume that if multiple handlers issue operations on the same transaction, these handlers are not concurrent; in practice, the principal can efficiently check that the program meets this restriction before outsourcing the program.

Adya's isolation testing. To check that an execution is consistent with an isolation level, we build on Adya's algorithms [16]. For transactional KV stores, Adya's algorithms take as input the *history* of execution that comprises: (a) the *event order* at the KV store, which in this paper we call *TxOp order* to avoid confusion. This is a partial order of all operations in the KV store that preserves the

order of operations within each transaction and includes the dictating write for each read, and (b) a *version order*: for each key, a total order of all committed values.

To test for an isolation level, these algorithms construct a graph H from the history. This is distinct from the graph G from earlier (§4.3.3), though both encode kinds of operation orders. The nodes of H correspond to the committed transactions in the TxOp order. H contains a *read-depend* edge $\langle T_1, T_2 \rangle$ if some operation in transaction T_2 reads from an operation in transaction T_1. It contains a *write-depend* edge $\langle T_1, T_2 \rangle$ if transaction T_1 writes some version of a key and transaction T_2 installs the next version. It contains an *anti-depend* edge $\langle T_1, T_2 \rangle$ if transaction T_1 reads some version of a key and transaction T_2 installs the next version.

Each isolation level is defined in terms of properties of H and the history. For example, a history is serializable if: (1) the graph H has no cycles, (2) a committed transaction never reads from an uncommitted transaction in the TxOp order and (3) if a committed transaction T_2 reads a value of a key that is written by a transaction T_1, that value is the last modification (per the version order) that T_1 makes to that key.

Advice collection. To adapt Adya's algorithms to Karousos, we augment the server's advice to include (a) the (alleged) TxOp order at the KV store, and (b) an (alleged) global order of writes (which implies an Adya version order). The alleged TxOp order is encoded as a list, for each transaction, of operations and the dictating `PUT` for each `GET`; we call such a list a *transaction log*. We call the alleged global order of writes, the *write order*.

Advice validation. The verifier executes Adya's algorithms on the transaction logs and write order to *provisionally* verify the isolation level. Depending on the expected isolation level, the verifier checks for the relevant phenomena by generating the graph H (see above) and checking for acyclicity. This verification is provisional because Adya's algorithms take as input the true history at the KV store. But the server is untrusted, so the transaction logs and write order may not

correspond to the true history. The verifier thus needs to perform additional checks, as follows.

First, similar to Section 4.3.3, the verifier ensures that all operations in the transaction logs are produced during re-execution. Second, the verifier ensures that the transaction logs are well-formed; specifically the verifier checks, by comprehensively inspecting the transaction logs, that transactions observe their own writes. Third, the verifier ensures consistency between the transaction logs and write order by checking that the operations in the write order are the last operations of committed transactions in the transaction logs.

Finally, the verifier needs to check that the transaction logs correspond to a legal KV store execution history that is consistent with the rest of the advice. Consider a server that claims that request r_1 issues the following operations, where k is a key in the KV store and x is a program variable: $op_1 = \texttt{GET}(k); op_2 = \text{write}(x,1)$, and request r_2 issues: $op_3 = \text{read}(x); op_4 = \texttt{PUT}(k,1)$. Additionally, the server claims that the dictating write of op_3 is op_2 and that op_1 reads from op_4. But op_3 reading from op_2 implies that op_2 originally preceded op_3, which implies that op_1 precedes op_4. Thus, the server is claiming, preposterously, that op_1 read from an operation that, according to the rest of the advice, was executed after it. To detect these types of misbehaviors, the verifier expands the graph G (§4.3.3) with nodes for external state operations, and adds write-read edges from `PUT`s to the corresponding `GET`s.[1]

4.4 IMPLEMENTATION

This section describes how the design in Section 4.3 is instantiated in a built system for auditing Node.js applications that optionally use MySQL as a transactional KV store.

Our system uses a transpiler to reduce the amount of effort that the principal needs to expend when using Karousos. We implemented our transpiler by extending the Babel [1] JavaScript tran-

[1] It would be wrong to augment G with write-write edges or read-write edges between external state operations as Karousos does for program variables (§4.3.3). Program variables are sequentially consistent, whereas external state operations are more weakly ordered even in valid executions. These types of edges would thus constrain TxOp order artificially, causing the verifier to mark such executions as invalid, undermining Completeness (§4.1.1).

spiler, via Babel's plugin mechanism. Our transpiler supports a core subset of Node.js, however we currently do not support some features, including JavaScript workers and timers, and monkey patching.

The transpiler does not entirely automate the process of using Karousos. The principal must: (a) Identify and annotate all loggable variables (§4.3.2); (b) Change the application to use a Karousos-provided version of the Knex library for database accesses (the Knex library included with Karousos is augmented to collect the TxOp order; §4.3.4); and (c) Annotate all handlers that are activated by user requests (§4.2). One can in principle extend the Karousos transpiler to automate some of these tasks. Below, we describe implementations of some of Karousos's mechanisms.

Identifying batches (§4.3.1). Recall that the Karousos server has to group requests (§4.3.1) with the same A relation and the same control flow within the handlers. To encode the A relation in a way that is invariant across requests, the server assigns an identifier to each function (functionID), and computes a *handlerID* as a digest of the functionID, the event that activates the handler, and the activator's handlerID. Notice that a handlerID is unique only within a request, and that if two requests have the same set of handler IDs, they have the same handler tree. To encode control flow within a handler, the server (as in Orochi [146, §4.3] and EAR [27, §3.1]) computes a *control flow digest*, updating it according to which branches are taken by the handler. Then, the server computes the top-level *tag* of a request (§4.3.1) as a digest of all handler IDs and their corresponding control flow digests.

Accelerated re-execution (§4.1.2, §4.3.1). Karousos borrows *SIMD-on-demand* (§4.1.2) from Orochi [146] but implements it differently. Whereas Orochi modified a PHP runtime to expose *multivalue* (§4.1.2) versions of primitive types, we use the transpiler to turn program variables into multivalues.

Testing A, computing the activator relation (§4.2, §4.3.2). The Karousos server needs an efficient check of whether two handlers are ordered by A; similarly, the verifier needs to efficiently compute a handler's activator, when interrogating the variable dictionary (§4.3.2). For these purposes, the implemented server assigns a *label* to each handler so that two handlers are ordered by A iff the label of the one is a prefix of the other. In contrast to handlerIDs, labels do not correspond across requests; handler labels encode only enough information to check the A relation and compute activator. Mechanically, a handler's label is computed at runtime as `parent_label/num` where `num` is the number children of the parent that have executed so far.

Non-determinism. Node.js programs often use non-deterministic operations, which Karousos handles as other record-replay systems do [56, 57, 65]: the server records the result of each non-deterministic operation in the advice, and, during re-execution, the verifier supplies the recorded information in response to the operation. Karousos does not currently give soundness guarantees about non-deterministic operations, but prior works show how to implement basic checks of well-formedness [24, 32, 54, 84, 146, 164].

Transactional state (§4.3.4). Karousos uses MySQL as a transactional KV store by requiring individual queries to SELECT or UPDATE only a single row, specified by the row's primary key. This maps naturally to the abstract `PUT-GET` interface from Section 4.3.4. The server generates the *transaction log* (§4.3.4) by logging operations when they are executed by the application. The server captures the dictating `PUT` of each `GET` operation by storing each row's last writer in the row itself. Our implementation obtains the *write order* (§4.3.4) by repurposing MySQL's binary log, or binlog. This involved some engineering work because the binlog, being intended for a different purpose (state replication), is in an internal format that is not well-documented, and which contains information that is extraneous for our purposes.

LOC, challenge, and limitation. Our implementation comprises 16,200 lines of JavaScript. In addition to the transpiler (6,100 lines), the implementation includes a library of helper functions used by the transpiled server and verifier (10,000 lines) and a program that periodically processes the MySQL binlog (100 lines of JavaScript).

Maintaining the activation partial order was a significant source of implementation overheads, and is also a significant source of runtime overheads (§4.5.1). It requires endowing each handler activation with knowledge of its activator's ID, and passing that information to all functions called by the handler. Meanwhile, many JavaScript functions are implemented in native code, and the transpiled code cannot change their call signatures or semantics. Our transpiler adopts a variety of strategies for this purpose.

A limitation is that our implementation assumes that handlers that are registered by the initialization function (§4.2) cannot be unregistered. This is not fundamental and can be addressed by treating handler registration and unregistration as state operations. Our proofs in Appendix B.3 also assume this restriction, and thus apply to our current implementation.

4.5 EVALUATION

We evaluate Karousos by answering the following questions:

1. What is the overhead of collecting advice (§4.5.1)?
2. What is the size of the advice that the server sends to the verifier? (§4.5.2)
3. What is the speedup of batching requests during verification when compared to a baseline that sequentially re-executes requests? (§4.5.3)
4. What is the impact of each of Karousos's batching and logging techniques on advice size and verification time? (§4.5.4)

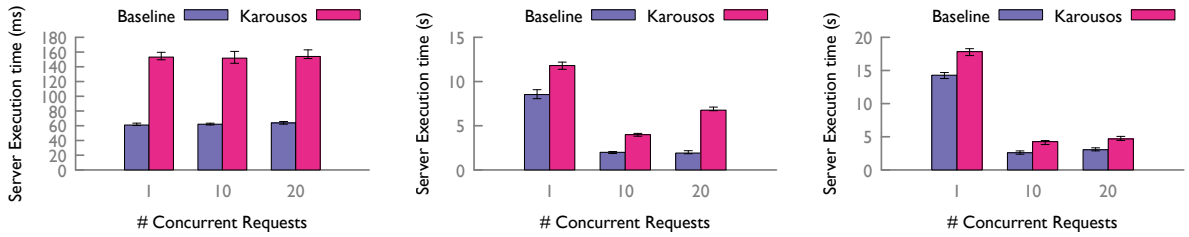
Applications. We evaluate Karousos with three Node.js applications that we developed:

Inventory management: Users can add new products, change a product's availability and stock level, and look up product information. Product information is stored in a database table indexed by product ID. The application uses a single SQL statement when adding a new product or looking up information. Updates to product information require issuing a transaction with two SQL statements: the first reads the existing information and the second updates the information.

Stack dump logging: Users can submit stack dumps, count how many times a stack dump has been reported, and get a list of unique stack dumps. Stack dumps and the number of times they have been reported is stored in a database table indexed by their digest. When a stack dump is submitted, the application checks if it is unique, in which case it adds it to the database. Otherwise the application increments the number of times the trace has been submitted. The application maintains a variable containing the set of all digests stored in the database, and uses this set to implement the list request. In particular, when listing unique traces the application issues a query for each digest in the set. Thus the application can implement the list request while using the transactional key-value store interface described in §4.3.4.

Message of the day: Users can retrieve or set a "message of the day" (MOTD). When setting the MOTD, a user can specify whether the message should be displayed every day or only on a particular day. Messages and metadata are stored in a local hashmap rather than in the database.

Workloads. We use three types of workloads for each application: (a) read-heavy with 90% read requests and 10% write requests; (b) write-heavy with 90% write requests and 10% read requests; and (c) mixed with 50% write requests and 50% read requests. For all three workload types, write requests for the inventory application are split so that half of them insert a new product and the other half update information about an existing product. Similarly, across all workloads, write requests to the stack dump application are split so that 10% of them report a new stack dump and the remaining 90% report a previously reported one. In our experiments, we vary the total number of requests and the number of concurrent requests. Unless otherwise specified, our graphs show the median from 10



(a) Message of the Day with mixed workload. (b) Stack Dump with mixed workload. (c) Stack Dump with write heavy workload

Figure 4.6: Response latency for a Karousos server when compared to an unmodified baseline. Each graph shows total time taken to execute 140 requests. Across all applications and workloads we observe overheads of 1.4–3.5×.

experiments, and we use errors bars to show 5^{th} and 95^{th} percentile values from these experiments.

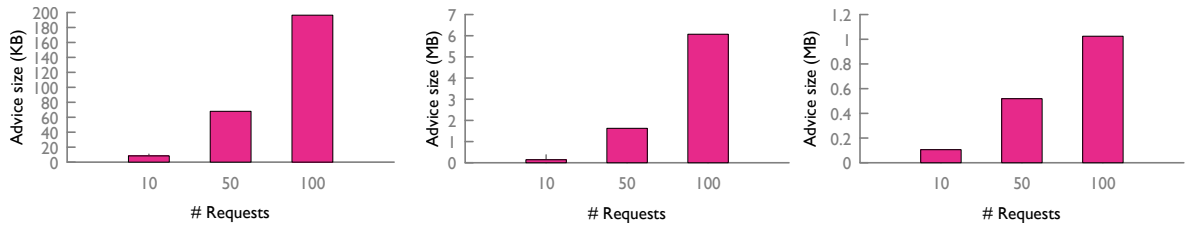
Testbed. We run all our experiments on servers equipped with a 3.7GHz Intel(R) Xeon(R) E5-1630 v3 (4-core) CPU with 32GB RAM and 1TB SSD, and running Ubuntu 16.04. We run the server and verifier using the Node.js v12.20.0 runtime, and use MySQL 8.0.19 as our database server. In our experiments the database and application are co-located on the same server and use up to 10 concurrent connections.

We use Python scripts to generate the request headers and bodies for our workloads and issue the requests using `wrk` [12].

4.5.1 Advice collection overheads

We first evaluate the overhead of advice collection by comparing the response latency of a Karousos server to that of an unmodified baseline. We believe the reported overheads are reasonable, as we expect auditability to cost something.

In each experiment, we first warm up the application using 60 requests from the target workload. We measure the time taken to serve a 140-request trace while varying the number of concurrent requests. Figure 4.6 depicts the results for two applications. In the MOTD application (Figure 4.6a), the Karousos server has response latency that is 2.4–2.5× larger than the baseline. A differential analysis (not shown) indicates that a majority (60–70%) of this overhead is from tracking the

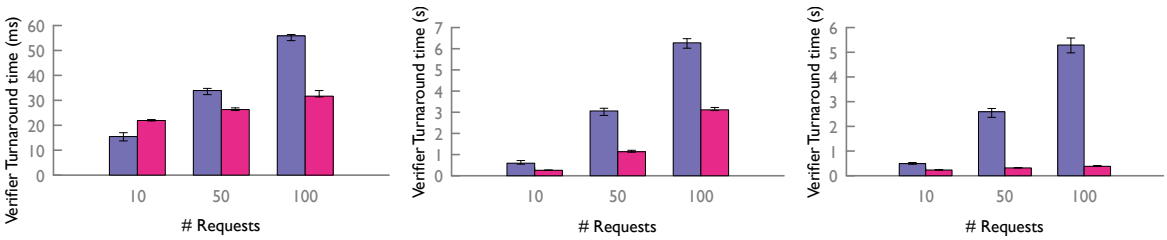


(a) Message of the Day (b) Stack Dump (c) Inventory

Figure 4.7: Size of the advice generated by the Karousos server. The reported values are from serving requests from a mixed workload.

activation partial order (§4.4). The overheads for this application do not vary with the number of concurrent requests nor across different workloads. Therefore, we show results only from the mixed workload trace.

In the stack dump application (Figures 4.6b, 4.6c), the Karousos server has response latency that is between 1.4–3.5× higher than the baseline. As with the MOTD application, a majority of the overhead is from tracking the activation partial order. In contrast to the MOTD application, overheads here increase with additional concurrent requests. This is because each request requires making at least one call to the database, allowing the single-threaded Node.js runtime to switch to serving other requests. Consequently, Karousos needs to track the activation order for multiple requests simultaneously, and the cost of accessing and maintaining this state increases as the number of concurrent requests increases. For this application, the mixed workload (Figure 4.6b) has higher overheads than the write-heavy workload (Figure 4.6c). This is because for the write-heavy workload, the database is the bottleneck and Karousos does not add to the database's query execution time. In our setup, database updates take nearly an order of magnitude more time than database reads ($\sim$ 80ms vs $\sim$ 1ms). We omit results from the inventory application; they are similar to the stack dump application.



(a) Message of the Day (b) Stack Dump (c) Inventory

Figure 4.8: Karousos verification time vs sequentially re-executing the trace. We use the same colors as Figure 4.6. The results are from a mixed workload run.

4.5.2 Advice size

Figure 4.7 quantifies the size of the advice sent by a Karousos server to a Karousos verifier. In this case we use a workload with no concurrent requests, and measure size as we vary the number of requests. (Counterintuitively, Karousos's algorithm ensures that the number of concurrent requests has no impact on advice size; this is because accesses to state shared between requests are always *R*-concurrent, as requests can be arbitrarily re-ordered.)

We use a slight variation on the workload described above when evaluating the stack dump application: across all workload types any write requests add a new stack dump. We made this alteration because we found that the MOTD and stack dump applications had identical results otherwise. We found that, regardless of workload or application, advice size scales linearly with the number of requests. Handler logs are a majority (70–90%) of the advice. In our experiments advice sizes ranged from 8KB to 6MB total, with a majority totaling 1MB or less. We believe these sizes are reasonable in practice. Furthermore, Karousos servers send periodic advice updates to verifiers, and can thus tradeoff update size for frequency.

4.5.3 Verification performance

Ideally, we would compare our performance against another verifier that checks execution integrity without relying on any advice from the untrusted server and without batched re-execution. However,

we do not know of an existing system that does so. Instead, we use as the baseline the time taken by the unmodified application to re-execute the trusted trace. This baseline serves as a best-case estimate for the performance of any verifier that uses re-execution and does not batch requests. We run this evaluation using the same workload that we used in Section 4.5.2. Figure 4.8 depicts the results.

Across all applications, the Karousos verifier performs better than the baseline when auditing traces with 50 or more requests: Karousos re-executes the trace 1.3–14× faster. However, for the 10-request traces, results are equivocal. For MOTD, (Figure 4.8a), the baseline is about 25% faster than Karousos ($\sim$ 15.5ms vs $\sim$ 22ms): the overheads from checking and using the advice during re-execution are larger than the benefits from the limited batching opportunities offered by a short trace. For the stack dump (Figure 4.8b) and inventory (Figure 4.8c) applications, batching reduces overheads even for 10-request traces, showing that batching opportunities and overheads differ across applications and workloads. In practice, we expect that deployed web applications serve several clients per second, and thus in the common case, the Karousos verifier will be processing long traces, with significant opportunity for batching (as also observed by Orochi [146] and Poirot [92]).

4.5.4 KAROUSOS LOGGING AND BATCHING TECHNIQUES

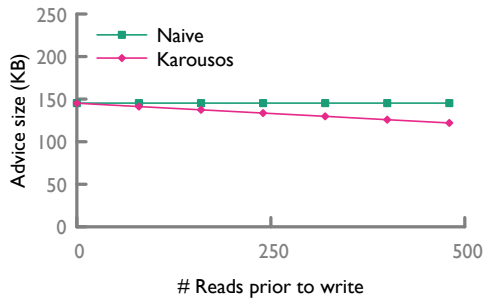


Figure 4.9: A comparison between a naive advice generation approach that logs all accesses and Karousos's dynamic logging.

Karousos logging techniques. Figure 4.9 compares Karousos's approach of dynamically deciding what accesses to log to a naive approach that, for loggable variables, logs *all* accesses, not just *R*-concurrent ones (§4.3.2). We use a simple web application that creates 500 concurrent handlers for each request. All but one read from a shared variable; the remaining handler writes the shared variable. We measure log size, varying the number of times the variable is read (these reads, if

executed before the write, are not *R*-concurrent because the given request observes the value written by the initializer; §4.2). The figure shows advice sizes for a trace where 30 such requests are executed. While advice size is constant for the naive approach, it decreases linearly for Karousos as the write happens later (which suppresses the degree of *R*-concurrency).

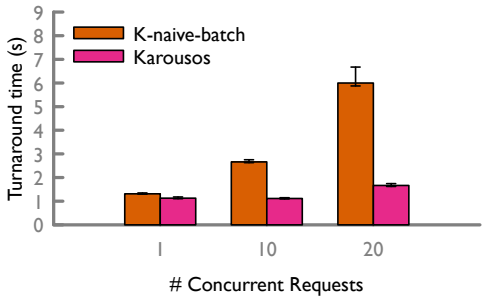


Figure 4.10: A comparison of Karousos's approach of batching requests with different handler execution order to a naive approach.

Karousos batching techniques. We evaluate Karousos's approach of placing requests into the same re-execution group even if they executed handlers in different orders (§4.3.1). We compare re-execution time to a naive version of the Karousos verifier, which we call K-naive-batch. K-naive-batch imposes the additional grouping requirement that requests in the same group should have executed handlers in the same order. Figure 4.10 shows the time taken to verify the stack dump application on a trace of 200 requests drawn from the mixed workload, varying the number of concurrent requests. While the Karousos verifier's performance does not vary as the degree of concurrency increases, K-naive-batch takes longer with more concurrent requests, for example 3.6× longer when there are 20 concurrent requests. This is because a larger number of concurrent requests increases the likelihood that handlers are re-ordered, decreasing the grouping opportunities for K-naive-batch, while leaving Karousos unaffected, thus demonstrating the importance of this design decision.

5 | Summary and Next Steps

In this dissertation we studied how to audit two essential services: transparency dictionaries and outsourced event-driven web applications. We introduced Verdict and Karousos, two practical systems that allow a *single* principal to verify that the respective services work as promised, without making any assumptions about the execution at the services or at other auditors.

Specifically, Chapter 3 describes Verdict, a transparency dictionary that proves to its clients that it is executing faithfully (§3.1) by employing a novel cryptographic accumulator based on Merkle Trees (§3.2) and a new SNARK tailored to the dictionary's workloads (§3.3); Verdict scales to dictionaries with millions of users (§3.4–§3.5). Chapter 4 introduces Karousos, a system that comprehensively audits remote servers executing event-driven web applications (§4.1) by using a novel record-replay algorithm. This algorithm accelerates the replayer, balances logging with replay efficiency (§4.3.1–§4.3.2) and, at the same time, supports an untrusted recorder (§4.3.3). Additionally, Karousos takes some first steps in supporting weakly consistent databases (§4.3.4). Karousos's evaluation shows that it achieves an efficient verifier while imposing reasonable overheads on the service (§4.4–§4.5).

Both Verdict and Karousos have clear limitations (see also §3 and §4) that indicate possible next steps.

Verdict. Compared to its predecessors, Verdict has lower CPU costs for producing proofs. However, further mitigating these costs is essential to the adoption of verifiable transparency dictionaries

in the real world. A possible way do this is to combine Phalanx with a recently proposed accumulator [152] based on Sparse Merkle Trees. This accumulator admits $\approx 20\%$ smaller circuits to verify updates on large dictionaries than Indexed Merkle Trees (§3.2), but has larger membership proofs and non-membership proofs.

Moreover, as discussed in Chapter 3, despite the fact that Verdict is highly parallelizable, the current implementation of Verdict does not leverage multiple CPUs. Parallelizing the proof generation process under Verdict is essential for its adoption in the real world where transparency dictionaries are required to serve millions of requests every minute.

Verdict could also benefit from further reducing the costs of light clients. In Verdict, a light client that is offline for k epochs needs to do $O(k)$ work when it comes back online to validate the epochs it missed. It would be desirable to reduce this work to $O(1)$. One could possibly achieve this without introducing large overheads on the service or requiring a trusted setup using recent work in the area of recursive arguments [42, 46, 96].

Karousos. Karousos's current implementation requires the programmer to manually annotate all loggable variables. This requirement could hinder Karousos's applicability in the real world as manual annotation is error-prone. However, we could automate the process of identifying and annotating loggable variables by using a static analyzer, in particular, escape analysis [86, 161]. Escape analysis is conservative and, thus, could erroneously add annotations to variables that are not loggable. This does not undermine the security of Karousos (§4.3.2) but might impact the server's performance. We leave this direction to future work.

Another limitation of Karousos's implementation is that it supports only a subset of Node.js, which means that it cannot be immediately applied to arbitrary applications. Despite the fact that the subset of Node.js that our current implementation supports is representative, extending our system to support all Node.js features requires significant programming effort due to the broad set of supported semantics.

Karousos takes an important first step in handling weakly consistent shared state, but there is still a long way to go. First, Karousos does not support Snapshot Isolation (SI) [36], a widely used isolation level that has better performance than serializability but still avoids most of the concurrency anomalies that serializability avoids. One way to add support for SI to Karousos, is to have the verifier use Adya's algorithms [16]. However, these algorithm require the server to provide start and commit timestamps for all executed transactions. A potential way to avoid the overheads of generating and communicating these timestamps is to have the verifier check for SI using an algorithm that does not depend on timestamps [50, 59]. Furthermore, Karousos only supports transactional KV stores and not general databases. Extending Karousos to handle general databases is a challenging problem: it requires developing and implementing an efficient algorithm to check if predicate-based reads and writes are consistent with an isolation level.

Despite their limitations, Verdict and Karousos make a significant step forward in verifying two important services while introducing several techniques that may be of independent interest.

A | VERDICT'S CORRECTNESS PROOFS

This appendix formally defines the correctness properties of a transparency dictionary (§A.1) and the correctness properties of a SNARK (§A.1.1), and then proves that Verdict satisfies them (§A.2).

A.1 CORRECTNESS PROPERTIES

We formally define the completeness and soundness properties of a transparency dictionary. In the definitions, we assume that both the initially empty dictionary $D_0 = \perp$ and its commitment $\mathscr{C}_0$ are well-known. A transparency dictionary satisfies the following properties.

- **Update Completeness.** Informally, clients do not reject update proofs produced by an honest service. Formally, for any epoch t, and sequence of update operations $U_1, \ldots, U_t$, and application-specific update policy F, the following probability is 1:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ \forall i < t: \\ (D_{i+1}, \mathscr{C}_{i+1}, \pi_{i+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_i, \mathscr{C}_i, U_i) \\ \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \end{array}\right]$$

- **Update Knowledge Soundness.** Informally, the values are updated only according to an application-specific function. Formally, for any application-specific update function F and any PPT adversary

$\mathscr{A}$, there exists an extractor $\mathscr{E}$, such that for any epoch t the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathscr{C}_i, \pi_i\}_{1 \le i \le t}, v, \pi_{\mathsf{lookup}}, \mathsf{label}) \leftarrow \mathscr{A}(\mathsf{pp}, \rho) \\ (\{U_i\}_{1 \le i \le k-1}) \leftarrow \mathscr{E}(\mathsf{pp}, \rho) \text{ for some } k \le t \\ \forall i < k : v_{i+1} \leftarrow F(v_i, U_i) \text{ where } v_1 = \perp \\ \forall i < t : \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \\ v \neq v_k \end{array}\right]$$

where ρ denotes the input randomness for adversary $\mathscr{A}$.

- **Lookup Completeness.** Informally, if the service is honest, then clients accept lookup proofs. Formally, for any application-specific update policy F, epoch t, label label, and sequence of requests $U_1, \ldots, U_t$, the following probability is 1:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ \forall i < t : \\ (D_{i+1}, \pi_{i+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_i, \mathscr{C}_i, U_i) \\ (v, \pi_{\mathsf{lookup}}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, D_t, \mathscr{C}_t, \mathsf{label}) \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \end{array}\right]$$

- **Lookup Soundness.** Informally, the service cannot return an incorrect value for any label included in a given commitment. Formally, for any application-specific update policy F, any PPT adversary $\mathscr{A}$, and epoch t, the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathscr{C}_i, \pi_i\}_{1 \le i \le t}, v, v', \pi_{\mathsf{lookup}}, \pi'_{\mathsf{lookup}}, \mathsf{label}) \leftarrow \mathscr{A}(\mathsf{pp}) \\ \forall i < t : \\ \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, \mathsf{label}, v', \pi'_{\mathsf{lookup}}) = 1 \\ v' \neq v \end{array}\right]$$

- **Fork consistency.** Informally, if the service equivocates at some time t by presenting two different commitments to different sets of clients, it cannot forge a proof that merges the two different commitments. Formally, for any application-specific update policy F, any PPT adversary $\mathscr{A}$ and epoch t the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathscr{C}_i, \pi_i, \mathscr{C}'_i, \pi'_i\}_{1 \le i < t},) \leftarrow \mathscr{A}(\mathsf{pp}) \\ \forall i < t : \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \\ \forall i < t : \mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}'_i, \mathscr{C}'_{i+1}, \pi'_{i+1}) = 1 \\ \exists i < t : \mathscr{C}_i \neq \mathscr{C}'_i \\ \mathscr{C}_t = \mathscr{C}'_t \end{array}\right]$$

A.1.1 Formal Properties of a zkSNARK

To make our proofs self contained, we briefly recall SNARK-related definitions.

A zero-knowledge succinct non-interactive argument of knowledge (zkSNARK [39, 74]) for a circuit $\mathscr{R}$ has the following semantics

- $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathscr{R})$: Returns prover parameters pp and verifier parameters vp used to produce and verify proofs respectively for circuit $\mathscr{R}$, where λ is the security parameter.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathsf{X}, w)$: Takes as input IO X and secret witness w and returns a proof π that $\mathscr{R}(\mathsf{X}, w) = 1$.
- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vp}, \mathsf{X}, \pi)$: Takes as input IO X and proof π and returns 1 if π attests that the prover knows secret w such that $\mathscr{R}(\mathsf{X}, w) = 1$.

and satisfies the following properties:

- **Completeness.** Informally, a verifier does not reject an honest proof. Formally, for IO X, and

witness w such that $\mathscr{R}(\mathsf{X}, w) = 1$ the following probability is 1:

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathscr{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathsf{X}, w) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{X}, \pi) = 1\end{array}\right]$$

- **Soundness.** Informally, if there exists no valid witness, then the prover cannot produce an accepting proof. Formally, for circuit $\mathscr{R}$, and IO X, if there exists no w such that $\mathscr{R}(\mathsf{X}, w) = 1$, then for any probabilistic polynomial time (PPT) adversary $\mathscr{A}$, the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathscr{R}) \\ \pi' \leftarrow \mathscr{A}(\mathsf{pp}, \mathsf{vp}, \mathsf{X}) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{X}, \pi') = 1\end{array}\right]$$

- **Knowledge Soundness.** Informally, if a prover produces an accepting proof, then it must know a valid witness. Formally, for any circuit $\mathscr{R}$, and PPT adversary $\mathscr{A}$, there exists PPT $\mathscr{E}$ such that the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathscr{R}) \\ (\pi', \mathsf{X}') \leftarrow \mathscr{A}(\mathsf{pp}, \mathsf{vp}, \rho) \\ w \leftarrow \mathscr{E}(\mathsf{pp}, \mathsf{vp}, \pi', \mathsf{X}', \rho) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{X}', \pi') = 1 \text{ and } \mathscr{R}(\mathsf{X}', w) \neq 1\end{array}\right]$$

  where ρ is the input randomness for $\mathscr{A}$.

- **Zero Knowledge.** Informally, the prover does not reveal any information about its witness in the proof. Formally, for circuit $\mathscr{R}$ and $(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathscr{R})$, there exists a PPT simulator $\mathscr{S}$ such that for all PPT $\mathscr{V}^*$ with input randomness ρ, and IO X and witness w such that $\mathscr{R}(\mathsf{X}, w) = 1$, $\mathsf{Prove}(\mathsf{pp}, \mathsf{X}, w)$ is computationally indistinguishable from $\mathscr{S}(\mathsf{pp}, \mathsf{vp}, \mathsf{X}, \rho)$.

- **Succinctness.** For circuit $\mathscr{R}$ and for any proof π output by Prove, $|\pi|$ is sublinear in $|\mathscr{R}|$.

A.2 PROOFS

Verdict's update and lookup completeness follows in a straightforward manner from its construction. Below, we prove the soundness properties.

Lemma 2. Verdict is a transparency dictionary that satisfies update knowledge soundness.

Proof. For security parameter λ and for some application-specific update policy F, let $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$. Suppose that PPT adversary $\mathscr{A}$ on input pp and some randomness ρ outputs

$$\{C_i, \pi_i\}_{1 \le i \le t}, v, \pi_{\mathsf{lookup}}, \mathsf{label}$$

for some epoch t such that

$$\mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \quad \text{for all } i < t \tag{A.1}$$

and

$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1. \tag{A.2}$$

We must show that there exists PPT extractor $\mathscr{E}$ that on input pp and ρ, outputs for some $k \le t$

$$U_1, \ldots, U_{k-1}$$

such that for $v_{i+1} \leftarrow F(v_i, U_i)$, where $v_1 \leftarrow \perp$, we have that $v = v_k$ with probability $1 - \mathtt{negl}(\lambda)$. Intuitively, this would mean that the extractor has been able to extract a valid sequence of updates that, starting from the empty dictionary, result in value v (thus proving the adversary's knowledge of such a sequence). To show this, we will first show that $\mathscr{E}$ can extract *some* sequence of requests by the knowledge soundness of the underlying SNARK. Next, we will show that the sequence of requests provided in π_{lookup} (and the final resulting value), is valid with respect to the application specific update policy. Finally, we will show that the requests extracted by $\mathscr{E}$ must be equal to the requests provided in π_{lookup} by the binding property of the indexed Merkle tree, thus showing that $\mathscr{E}$ has indeed extracted a valid sequence of requests.

We start by showing that $\mathscr{E}$ can extract some sequence of requests: Because Equation A.1 holds by the knowledge soundness of the underlying SNARK and folding scheme, for label $H(\mathsf{label})$, the adversary knows a sequence of values $U'_1, \ldots, U'_{k'-1}$ for some $k' < t$ such that

$$h'_{i+1} \leftarrow H(U'_i, h'_i)$$

for all $i < k' - 1$ (where $h_1 = \perp$) such that the indexed Merkle tree with root $\mathscr{C}_t$ contains $h'_{k'}$ under $H(\mathsf{label})$. Thus, $\mathscr{E}$ can extract these U'_i for all $i < t$.

Next, we show that the sequence of requests provided in π_{lookup} must be valid: Because Equation A.2 holds, the included hashchain must be well-formed with probability $1 - \mathtt{negl}(\lambda)$. In particular, the hashchain contains values $U_1, \ldots, U_{k-1}$ for some $k < t$ such that

$$h_{i+1} \leftarrow H(U_i, h_i)$$

for all $i < k - 1$ such that $h_1 = \perp$ and the indexed Merkle tree with root $\mathscr{C}_t$ contains h_k under label $H(\mathsf{label})$. Also, because Equation A.2 holds, we must have that

$$v_{i+1} \leftarrow F(U_i, v_i)$$

for all $i < k$, where $v_1 = \perp$, and that

$$v_k = v. \tag{A.3}$$

Now, we show that the requests extracted by $\mathscr{E}$ must be equal to the requests provided in π_{lookup}: Because Equation A.1 holds, the insert invariant of the indexed Merkle tree must hold with probability $1 - \mathtt{negl}(\lambda)$. In particular, the label $H(\mathsf{label})$ can only be inserted once under a single leaf node. But from the above reasoning we know that the indexed Merkle tree with root $\mathscr{C}_t$ must contain h_k under label $H(\mathsf{label})$, and must contain $h'_{k'}$ under label $H(\mathsf{label})$. By the binding property of indexed Merkle trees, this implies that $h_k = h'_{k'}$. Therefore, by the binding property of H, we must have that $k = k'$ and moreover that $U'_i = U_i$ for all $i < k$ with probability $1 - \mathtt{negl}(\lambda)$. Thus

we must have

$$v_{i+1} = F(U_i', v_i)$$

for all $i < k$. From Equation A.3, we have that $v_k = v$. Thus, the material extracted by $\mathscr{E}$ is valid with probability $1 - \mathtt{negl}(\lambda)$. □

Lemma 3. Verdict is a transparency dictionary that satisfies lookup soundness.

Proof. For security parameter λ and for some application-specific update policy F, let $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$. Suppose that PPT adversary $\mathscr{A}$ on input pp outputs

$$\{\mathscr{C}_i, \pi_i\}_{1 \le i \le t}, v, v', \pi_{\mathsf{lookup}}, \pi'_{\mathsf{lookup}}, \mathsf{label} \tag{A.4}$$

such that

$$\mathsf{VerifyUpdates}(\mathsf{pp}, \mathscr{C}_i, \mathscr{C}_{i+1}, \pi_{i+1}) = 1 \quad \text{for all } i < t \tag{A.5}$$

and

$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, v, \mathsf{label}, \pi_{\mathsf{lookup}}) = 1 \tag{A.6}$$

$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathscr{C}_t, v', \mathsf{label}, \pi'_{\mathsf{lookup}}) = 1. \tag{A.7}$$

We must show that $v' = v$ with probability $1 - \mathtt{negl}(\lambda)$. To do so, we will show that the sequence of requests provided in π_{lookup} must be equal to the sequence of requests provided in π'_{lookup} due to the binding property of indexed Merkle trees.

We first consider the sequence of requests provided in π_{lookup}: Because Equation A.6 holds, the hashchain included in π_{lookup} contains requests $U_1, \ldots, U_{k-1}$ for some $k < t$ such that

$$h_{i+1} \leftarrow H(U_i, h_i)$$

$$v_{i+1} \leftarrow F(U_i, v_i)$$

for all $i < k-1$ such that

$$h_1 = \bot \tag{A.8}$$

$$v_1 = \bot \tag{A.9}$$

$$v_k = v \tag{A.10}$$

and that the indexed Merkle tree with root $\mathscr{C}_t$ contains h_k under label $H(\mathsf{label})$.

Symmetrically, we consider the sequence of requests provided in π'_{lookup}: Because Equation A.7 holds, the hashchain included in π'_{lookup} contains requests $U'_1, \ldots, U'_{k-1}$ for some $k' < t$ such that

$$h'_{i+1} \leftarrow H(U'_i, h'_i)$$

$$v'_{i+1} \leftarrow F(U'_i, v'_i)$$

for all $i < k'-1$ such that

$$h'_1 = \bot \tag{A.11}$$

$$v'_1 = \bot \tag{A.12}$$

$$v'_{k'} = v' \tag{A.13}$$

and that the indexed Merkle tree with root $\mathscr{C}_t$ contains $h'_{k'}$ under label $H(\mathsf{label})$.

Now, we show that the requests provided in π_{lookup} must be equal to the requests provided in π'_{lookup}: Because Equation A.5 holds, the insert invariant of the indexed Merkle tree must hold with probability $1 - \mathtt{negl}(\lambda)$. In particular, the label $H(\mathsf{label})$ can only be inserted once under a single leaf node. But from the above reasoning, we know that the indexed Merkle tree with root $\mathscr{C}_t$ must contain h_k under label $H(\mathsf{label})$, and must contain $h'_{k'}$ under label $H(\mathsf{label})$. By the binding property of indexed Merkle trees, this implies that $h_k = h'_{k'}$ with probability $1 - \mathtt{negl}(\lambda)$. Therefore, by the binding property of H, we must have that $k = k'$ and moreover that $U'_i = U_i$ for all $i < k$ with probability $1 - \mathtt{negl}(\lambda)$. By Equations A.10 and A.13, this implies that $v = v'$. □

Achieving Fork Consistency. We have thus far shown that Verdict satisfies lookup soundness and update soundness. These do not prevent the service from presenting a stale view of its current state (for example, an older state that does not include recent updates).

To mitigate this issue, the service can publish its commitments (and proofs) to a public bulletin board (such as a blockchain). Alternatively, clients can detect inconsistent views via a peer-to-peer gossip protocol. In the latter case, the service cannot fork the clients' views indefinitely. But, it does not prevent the service from quietly forking the clients views for a brief period and later merging the views by replaying updates from both forks. We prevent this as follows: We require the service to include in its published commitment $\mathscr{C}_i$ not only the root of the indexed Merkle tree but also a hash h_i that is $\perp$ for $i = 0$ and $H(\mathscr{C}_{i-1})$ otherwise where H is a collision resistant hash function. We also extend VerifyUpdates to check that $\mathscr{C}_i.h_i = H(\mathscr{C}_{i-1})$;

Lemma 4. Verdict is a transparency dictionary that satisfies fork consistency.

Proof. Denote $\mathscr{C}_i.h_i$ as h_i and $\mathscr{C}'_i.h_i$ as h'_i Since for all $j < t$,

$$\mathsf{VerifyUpdates}(pp, \mathscr{C}_j, \mathscr{C}_{j+1}, \pi_j) = 1 \tag{A.14}$$

and

$$\mathsf{VerifyUpdates}(pp, \mathscr{C}'_j, \mathscr{C}_{j+1}, \pi_{j+1}) = 1 \tag{A.15}$$

it is sufficient to show that in order for the adversary to produce $\mathscr{C}_t = \mathscr{C}'_t$ and $\mathscr{C}_i \neq \mathscr{C}'_i$ for some $i < t$, it needs to find a collision for the hash function H.

Equations A.14 and A.15 imply that for all $j \leq t$,

$$h_j = H(\mathscr{C}_{j-1})$$

and

$$h'_j = H(\mathscr{C}'_{j-1})$$

If there exists some $i < t$ for which $\mathscr{C}_i \neq \mathscr{C}'_i$ but $\mathscr{C}_t = \mathscr{C}'_t$ there exists some $k \in (i,t)$ s.t. $\mathscr{C}_k \neq \mathscr{C}'_k$

and $\mathscr{C}_{k+1} = \mathscr{C}'_{k+1}$. The latter implies that $h_{k+1} = h'_{k+1}$ and, thus, $H(\mathscr{C}_k) = H(\mathscr{C}'_k)$ as requested.

□

B | KAROUSOS ALGORITHMS AND CORRECTNESS PROOFS

This section contains the detailed description of Karousos's algorithms (§B.1), the formal definition of the desired correctness properties (§B.2), and the proof that Karousos satisfies these properties (§B.3)

B.1 ALGORITHMS

In the following, a *global handler* is a handler that is registered by the initialization function (§4.2).

B.1.1 ANNOTATING LOGGABLE VARIABLES

The principal annotates a program P (§4.3.2) by identifying all loggable variables S and placing a special annotation called OnInitialize right after the initialization of each variable v in S.

Then, the Karousos compiler, produces an annotated program P_a by taking the program with the OnInitialize annotations and doing the following modifications:

- It replaces each read of a variable v that has an OnInitialize annotation (and is, thus, in S) with a special annotation called OnRead
- Right after each write of a variable v that has an OnInitialize annotation, it places a special annotation called OnWrite

We use the term *annotated operation* to refer to an annotation in P_a along with its corresponding variable operation if it exists (that is, if the annotation is OnInitialize or OnWrite).

B.1.2 Request IDs, Handler IDs, Variable IDs, and Transaction IDs

During execution, each request has a globally unique id which we denote *rid*.

Also, the honest server assigns a handler id to each handler that is running. This handler id is unique within a request and is a tuple $(\textit{functionID}, \textit{parent\_hid}, \text{opnum})$ where *functionID* is a globally unique identifier of the handler function (piece of code), *parent_hid* is the id of the handler that activates this handler and opnum is the index of the event that activates the handler within the parent handler. For instance, if a handler with *functionID* f is activated by the third operation of handler with id hid_2, this handler is assigned handler id $(f, hid_2, 3)$. Because each handler function can only be registered once for each event, handler ids are unique within a request, but not across requests.

Also, the honest server assigns a globally unique variable ID to each variable, and a globally unique transaction id to each transaction.

B.1.3 Advice collection

The honest server collects the following advice:

- The control flow groupings (C) (§4.3.1).
- The handler logs HLs (§4.3.1): for each request, the ordered log of handler operations that the request issued. Each entry in the log is one of the kinds below. For all of these, *hid* is the id of the handler that issues the operation and opnum is the order of this operation among all operations that the handler issues:
  - register operations are tuples $(hid, \text{opnum}, \textit{functionID}, \textit{eventNames})$, where *functionID* is the

```
1: logs are of type (requestid, handlerid, ℕ) → ({read, write}, value, requestid, handlerid, ℕ)
2: READ entries contain references to the write that they observe.
3: WRITE entries contain the value written.
4: procedure OnInitialize(rid, hid, opnum, v)
5:     Let v.log ← empty VL
6:     Let v.value ← nil     //the most recent written value
7:     //Store the most write operation (rid, hid, opnum)
8:     Let v.rid ← rid
9:     Let v.hid ← hid
10:    Let v.opnum ← opnum

11: procedure OnRead(rid, hid, opnum, v)
12:    if Rconcurrent((rid, hid, opnum), (v.rid, v.hid, v.opnum)) then
13:        //Check that the write that we read from has already been logged. If it has not, log it.
14:        if v.log{v.rid, v.hid, v.opnum} = nil then
15:            Let v.log{v.rid, v.hid, v.opnum} ← (write, v.value, nil, nil, nil)
16:        //Log the read
17:        Let v.log{rid, hid, opnum} ← (read, nil, v.rid, v.hid, v.opnum)
       return v

18: procedure OnWrite(rid, hid, opnum, opcontents, v)
19:    if Rconcurrent((rid, hid, opnum), (v.rid, v.hid, v.opnum)) then
20:        //Check that the write observed by this one has already been logged. If it has not, log it.
21:        if v.log{v.rid, v.hid, v.opnum} = nil then
22:            Let v.log{v.rid, v.hid, v.opnum} ← (write, v.value, nil, nil, nil)
23:        //Log the write
24:        Let v.log{rid, hid, opnum} ← (write, opcontents, v.rid, v.hid, v.opnum)
25:    //This write is the most recent write. So set the v fields value, rid, hid, opnum
26:    //to those of this write operation.
27:    Let v.value ← opcontents
28:    Let v.rid ← rid
29:    Let v.hid ← hid
30:    Let v.opnum ← opnum
```

Figure B.1: Pseudocode for server's logic on reaching an annotation.

id of the function, and *eventNames* is the set that contains the names of the events that the handler is registered for.

- emit operations are tuples $(hid, \text{opnum}, eventName)$, where *eventName* is a string that corresponds to the name of the event. An emit operation activates all functions that are registered for the event with name *eventName* (For more details on events and handler operations check Section 4.2).
- unregister operations are tuples $(hid, \text{opnum}, functionID, eventName)$, where *functionID* is the id of the function that is unregistered from event name *eventName*.
- Check operations is a class of operations that inspect the handlers and the events. The server logs such operations as tuples $(hid, \text{opnum}, opInfo)$, where *opInfo* is the name of the operation

and any arguments that the operation is called with.

- The variable logs *VL*s (§4.3.2): We denote the variable log of a variable id *v* as *VLv*. *VLv* is a map from triplets (request id, handler id, opnum) to tuples of type (t: AccessType, v: Value, prec_rid: request id, prec_hid: handler id, prec_anum: Int). These are created during execution; on each variable access, the server follows the algorithms of Figure B.1. AccessType is READ or WRITE. READ entries contain references to the write that they observe. WRITE entries contain the value written.
- The transaction logs *TXL*s (§4.3.4): for each transaction id, an ordered log of all operations that the transaction executes. Each entry is of the form:

$$(hid, \text{opnum}, \text{optype}, key, \text{opcontents})$$

  where

  - *hid* is the id of the handler that executes this operation
  - opnum is the order of this operation among all other operations that the handler executes.
  - optype is the type of operation, namely `tx_start`, `tx_commit`, `tx_abort`, `PUT` or `GET`,
  - *key* is the key for `PUT` and `GET` operations and null otherwise.
  - opcontents are null except for `PUT` and `GET` operations: For `PUT` operations they are the contents that are written and for `GET` operations they are the position in the logs of the write that they read from.
- *writeOrder* (§4.3.4): a single log that allegedly reflects the order in which the server applied the writes to shared external state.
- *responseEmittedBy*: a map from request ids to tuples (hid, opnum) s.t. the handler with id *hid* is the one that sends back the response and opnum is the number of operations that *hid* had issued prior to sending the response.
- *opcounts*: a map from the id (rid, hid) of every handler that is executed to the total number of

operations that the handler issues (may be zero).

B.1.4 VERIFIER

The verifier's algorithms are in Figures B.2, B.4, B.5, B.6, B.7, B.8, B.9:

```
1: Input Trace Tr, Input Advice A, Input Isolation level I
2: Global Graph G

3: Global Map OpMap : (requestid, handlerid, ℕ) → ("handler_log", requestid, ℕ) ∪ ("tx_log", txid, ℕ):
4:     maps the i-th operation of a handler to the location of this operation in the logs.

5: Global Map activatedHandlers: (rid, hid, i) → Set of invoked hids:
6:     defined over (rid, hid, i) s.t. the i-th operation of handler (rid, hid) is an emit operation (§B.1.3); maps these
       triples to the set of hids they invoke.

7: Global Set Committed: a set of tuples (requestid, txid) of purported committed transactions,

8: Global Map ReadMap: Map from write ops to the read ops that read from them

9: Global Set GlobalHandlers: Set of tuples (e, f) s.t. f is a global handler listening for e

10: Global Map lastModification: Map from (requestid, handlerid, key) to an integer representing the order of the
11:     last operation of the transaction that modifies this key among all other operations that the transaction issues
12:
13: procedure Audit
14:     Preprocess()
15:     ReExec() // Figure B.6
16:     Postprocess()
17:
18: procedure Preprocess
19:     Check Tr is balanced.
20:     Run the initialization phase and log all global handlers.
21:     G_Tr ← CreateTimePrecedenceGraph() // See [146, Figure 6]
22:     SplitNodes(G_Tr) // See [146, Figure 5]
23:     AddProgramEdges()
24:     AddBoundaryEdges() // Figure B.3
25:     AddHandlerRelatedEdges() // Figure B.4
26:     AddExternalStateEdges() // Figure B.4
27:     IsolationLevelVerification() // Figure B.5
28:
29: procedure Postprocess
30:     AddInternalStateEdges() // Figure B.9
31:     if CycleDetect(G) then REJECT
32:
33: procedure AddProgramEdges
34:     //This procedures adds all the nodes of each handler and program edges
35:     //between consecutive operations within a handler.
36:     for all (rid, hid) in A.opcounts do
37:         if rid does not appear in Tr then REJECT
38:         //Add the handler end, start nodes
39:         G.add_node((rid, hid, 0))
40:         G.add_node((rid, hid, ∞))
41:         for i ← 1, ..., A.opcounts[(rid, hid)] do
42:             G.add_node((rid, hid, i))
43:             G.add_edge((rid, hid, i − 1), (rid, hid, i))
44:         G.add_edge((rid, hid, A.opcounts[(rid, hid)]), (rid, hid, ∞))
```

Figure B.2: Pseudocode for verifier's audit procedure in Karousos.

```
1:  // Global Variables are the ones in Figure B.2
2:
3:  procedure AddBoundaryEdges
4:      // For all (rid, hid) that are request handlers, add edge from (rid, 0) to (rid, hid, 0)
5:      for all (rid, hid) in A.opcounts do
6:          if hid.parent_hid = null then
7:              G.add_edge((rid, 0), (rid, hid, 0))
8:      // For each rid, (rid, ∞) represents delivering the response. For the handler (rid, hid_r)
9:      // that delivers the response for rid (according to A), add an edge to (rid, ∞)
10:     // from the operation of hid_r just prior to delivering the response, and an edge from (rid, ∞) to the
11:     // operation of hid_r just after delivering the response.
12:     for all rid in Tr do
13:         if A.responseEmittedBy[rid] = null or A.responseEmittedBy is not of type (handler id, i) where i ∈ ℕ then
14:             REJECT
15:         Parse A.responseEmittedBy[rid] as (hid_r, opnum_r)
16:         if (rid, hid_r, opnum_r) ∉ G.Nodes then REJECT
17:         G.add_edge((rid, hid_r, opnum_r), (rid, ∞))
18:         if opnum_r = A.opcounts[(rid, hid_r)] then
19:             //In this case the handler's next operation is handler exit
20:             G.add_edge((rid, ∞), (rid, hid_r, ∞))
21:         else
22:             G.add_edge((rid, ∞), (rid, hid_r, opnum_r + 1))
```

Figure B.3: Pseudocode for verifier's AddBoundaryEdges procedure in Karousos.

```
1: // Global Variables are the ones in Figure B.2
2:
3: procedure AddHandlerRelatedEdges
4:     //add edges between consecuting operations in handler logs and activation edges
5:     for all rid in A.HL do
6:         if rid does not appear in Tr then REJECT
7:         Registered ← new Set()
8:         for i ← 1,...,A.HL_rid.length do
9:             op ← A.HL_rid[i]
10:            CheckOpIsValid(rid, op)
11:            OpMap[(rid, op.hid, op.opnum)] ← ("handler_log", rid, i)
12:            //Add the handler op precedence edge
13:            if i ≠ 1 then
14:                Let prev_op ← (rid, HL_rid[i−1].hid, HL_rid[i−1].opnum)
15:                G.add_edge(prev_op, op)
16:            if op is a register operation then
17:                for all eventName in op.eventNames do
18:                    Registered.add(eventName, op.functionID)
19:            else if op is an unregister operation then
20:                Registered.remove(op.eventName, op.functionID)
21:            else if op is an emit operation then
22:                for all (op.eventName, functionID) in Registered ∪ GlobalHandlers do
23:                    hid′ ← (functionID, op.hid, op.opnum)
24:                    //Check that the server has reported the activated handler
25:                    if A.opcounts[(rid, hid′)] = ∅ then REJECT
26:                    activatedHandlers[rid, op.hid, op.opnum].add(hid′)
27:                    //add the activation edge
28:                    G.add_edge((rid, op.hid, op.opnum), (rid, hid′, 0))
29:
30: procedure AddExternalStateEdges
31:     //Bookkeeping for external state and edges described in Section 4.3.4
32:     for all (rid, tid) in A.TXL do
33:         //Check if the transaction is allegedly committed or not
34:         if last operation in the log TXL_(rid,tid) is of type commit then
35:             Committed.add(rid, tid)
36:         Initialize map MyWrites
37:         for all i ← 1,...,TXL_(rid,tid) do
38:             Let op ← TXL_(rid,tid)[i]
39:             CheckOpIsValid(rid, op)
40:             OpMap[(rid, op.hid, op.opnum] ← ("tx_log", tid, i)
41:             if i ≠ 1 then
42:             if op.optype = GET then
43:                 Let (rid_w, tid_w, i_w) ← op.opcontents
44:                 Let op_w ← A.TXL_(rid_w,tid_w)[i_w]
45:                 CheckOpIsValid(rid, op_w)
46:                 G.add_edge((rid_w, op_w.hid, op_w.opnum), (rid, op.hid, op.opnum) //Add a read-from edge
47:                 // Add this op to the dictating write's list of readers
48:                 if op_w.optype ≠ PUT ∨ op_w.key ≠ op.key then REJECT
49:                 ReadMap[(rid, tid_w, i_w)].add(rid, tid, i)
50:                 //Make sure that if it reads a key that it has modified, it reads the last modification
51:                 if op.key ∈ MyWrites ∧ MyWrites[key] ≠ (rid_w, tid_w, i_w) then REJECT
52:             else if op.optype = PUT then
53:                 //update MyWrites
54:                 MyWrites[op.key] ← (rid, tid, i)
55:                 if (rid, tid) ∈ Committed then
56:                     lastModification[rid, tid, key] ← i
57:
58: procedure CheckOpIsValid(rid: request id, op: operation)
59:     if A.opcounts[(rid, op.hid)] = ∅ then REJECT
60:     if op.opnum < 1 ∨ op.opnum > A.opcounts[(rid, op.hid)] ∨ OpMap[(rid, op.hid, op.opnum)] exists then
61:         REJECT
```

Figure B.4: Pseudocode for verifier's AddHandlerRelatedEdges and AddExternalStateEdges in Karousos

```
1: // Global Variables are the ones in Figure B.2
2:
3: procedure IsolationLvlVer
4:     Initialize DG to an empty graph
5:     //Add a node for each committed transaction
6:     for all (rid, tid) ∈ Committed do
7:         DG.add_node((rid, tid))
8:     writeOrderPerKey ← ExtractWriteOrderPerKey()
9:     if I = READ UNCOMMITTED then
10:        AddWriteDependencyEdges(writeOrderPerKey)
11:        if CycleDetect(DG) then REJECT
12:    else if I = READ COMMITTED then
13:        AddWriteDependencyEdges(writeOrderPerKey)
14:        AddReadDependencyEdges()
15:        if CycleDetect(DG) then REJECT
16:    else if I = SERIALIZABILITY then
17:        AddWriteDependencyEdges(writeOrderPerKey)
18:        AddReadDependencyEdges()
19:        AddAntiDependencyEdges(writeOrderPerKey)
20:        if CycleDetect(DG) then REJECT
21:
22: procedure ExtractWriteOrderPerKey
23:    if writeOrder.length ≠ |lastModification| then REJECT
24:    Initialize writeOrderPerKey ← Map from keys to lists
25:    for all (rid, tid, i) in A.writeOrder in order do
26:        Let op ← TXL_(rid,tid)[i]
27:        if lastModification[(rid, tid, op.key)] ≠ i then REJECT
28:        writeOrderPerKey[op.key].append(rid, tid, i)
29:    return writeOrderPerKey
30:
31: procedure AddReadDependencyEdges // w-r edges
32:    for all (rid_w, tid_w, i_w) in ReadMap do
33:        //check that if the write is not the last modification, no committed transaction reads from it
34:        if (rid_w, tid_w, i_w) ∉ writeOrder then
35:            for all (rid_r, tid_r, i_r) in ReadMap[(rid_w, tid_w, i_w)] do
36:                if (rid_r, tid_r) ∈ Committed then REJECT
37:        else
38:            for all (rid_r, tid_r, i_r) in ReadMap[(rid_w, tid_w, i_w)] do
39:                if (rid_w, tid_w) ∈ Committed ∧ (rid_w ≠ rid_r ∨ tid_w ≠ tid_r) then
40:                    DG.add_edge(⟨(rid_w, tid_w), (rid_r, tid_r)⟩)
41:
42: procedure AddWriteDependencyEdges(writeOrderPerKey) // w-w edge
43:    for all key ∈ writeOrderPerKey do
44:        Let o ← writeOrderPerKey[key]
45:        for j = 1, ..., o.length − 1 do
46:            //check that there's only one version per transaction
47:            DG.add_edge(⟨(o[j].rid, o[j].tid), (o[j + 1].rid, o[j + 1].tid)⟩)
48:
49: procedure AddAntiDependencyEdges(writeOrderPerKey) // r-w edges
50:    for all k ∈ writeOrderPerKey do
51:        Let o ← writeOrderPerKey[k]
52:        for j = 1, ..., o.length − 1 do
53:            for all (rid, tid, _) ∈ ReadMap[o[j]] do
54:                Let T_1 = (rid, tid) and T_2 = (o[j + 1].rid, o[j + 1].tid)
55:                if T_1 ≠ T_2 ∧ T_1 ∈ Committed then
56:                    DG.add_edge(⟨T_1, T_2⟩)
```

Figure B.5: Pseudocode for verifier's isolation level verification in Karousos (§4.3.4)

```
1: //Global Variables are the ones in Figure B.2
2: procedure ReExec
3:     Re-execute Tr in groups according to A.C
4:         (1) Initialize a group as follows:
5:             Read in inputs for all requests in the group. Let in be these inputs
6:             Allocate program structures for each request in the group
7:             Initialize active: a queue of tuples (handler id, inputs)
8:             Find the functionIDs of the request handlers.
9:             if the functionIDs of the request handlers don't line up across requests then REJECT
10:            for all functionID in functionIDs do
11:                Let hid ← (functionID, null, 0)
12:                active.Enqueue(hid, in)
13:                if ∃rid in the group s.t. A.opcounts[(rid, hid)] = ∅ then REJECT
14:        (2) Execute the requests in the group with SIMD-on-demand:
15:        while active ≠ ∅ do
16:            (a) The runtime picks the next handler c to execute
17:                if c ≠ null then
18:                    Compute hid from the functionID of the function, the parent handler and the event.
19:                    if hid ∉ active then
20:                        continue;//Do not execute this handler.
21:                    else
22:                        Name the handler hid and set the inputs to the ones associated with hid in active
23:                        Remove hid from active
24:                        idx[hid] ← 1
25:                        Execute the activated handler for all requests in the group
26:                else
27:                    //Pick the next handler to be executed from active
28:                    (hid, in) ← active.Dequeue
29:                    idx[hid] ← 1
30:                    Execute the function hid.functionID for all requests in the group with inputs in
31:            (b) ReExecute hid for all requests:
32:                if execution within the group diverges then REJECT
33:                if the group makes an external state operation then
34:                    optype ← the type of state operation
35:                    for all rid in the group do
36:                        opcontents, tid, txnum ← parameters from execution
37:                        s ← CheckStateOp(rid, hid, idx[hid], optype, tid, txnum, key, opcontents)
38:                        if optype = GET then
39:                            state op result ← s
40:                    idx[hid] = idx[hid] + 1
41:                if the group reaches an annotated operation then
42:                    For all rid in the group:
43:                        if opnum > A.opcounts[(rid, hid)] then REJECT
44:                        if it is a write or initialization then
45:                            Execute the operation
46:                        Execute the annotation according to Figure B.8 where opnum is set to idx[hid]
47:                    idx[hid] = idx[hid] + 1
48:                if the group makes a handler operation then
49:                    optype ← the type of handler operation
50:                    for all rid in the group do
51:                        info ← parameters from execution
52:                        CheckHandlerOp(rid, hid, idx[hid], optype, info)
53:                    if optype = emit then ActivateHandlers(hid, idx[hid], active)
54:                    Execute the handler operation
55:                    idx[hid] = idx[hid] + 1
56:                if the group sends back a response then
57:                    if ∃rid in the group s.t. A.responseEmittedBy[rid] ≠ (hid, idx[hid]) then REJECT
58:                    Write out the produced outputs
59:            (c) When the execution of the handler hid exits
60:                if ∃rid in the group s.t. idx[hid] < A.opcounts[(rid, hid)] then REJECT
61:        (3) for all rid in the group do
62:            if the produced outputs are not exactly the responses in Tr then REJECT
63:            //Check that there are no handlers in the advice that we did not execute
64:            if ∃rid s.t. ∃hid : A.opcounts[(rid, hid)] but (rid, hid) was not executed by ReExec then REJECT
65:    return ACCEPT
```

Figure B.6: Pseudocode for verifier's ReExec in Karousos

```
1: //Global Variables are the ones in Figure B.2
2:
3: procedure CheckStateOp(rid, hid, opnum, optype, tid, txnum, key, opcontents)
4:     //Simulate and check logic for state operations (§4.1.2, §4.3.4)
5:     if opnum > A.opcounts[(rid, hid)] then REJECT
6:     Let (t, tid_c, txnum_c) ← OpMap[(rid, hid, opnum)]
7:     if t ≠ "tx_log" ∨ tid_c ≠ tid ∨ txnum_c ≠ txnum then REJECT
8:     Let op ← A.TXL_rid tid[txnum]
9:     if op.optype ≠ optype ∧ op.optype ≠ tx_abort ∧ optype ≠ tx_commit then REJECT
10:    if op.key ≠ key then REJECT
11:    if optype ≠ GET then
12:        if op.opcontents ≠ opcontents then REJECT
13:    else
14:        Let (rid_w, tid_w, i_w) ← op.opcontents
15:        Let op_w ← A.TXL_rid_w tid_w[i_w] return op_w.opcontents
16:
17: procedure CheckHandlerOp(rid, hid, opnum, optype, info)
18:    //Check that the handler operation matches the entry in the logs (§4.3.1)
19:    if opnum > A.opcounts[(rid, hid)] then REJECT
20:    Let (t, rid_c, i) ← OpMap[(rid, hid, opnum)]
21:    if t ≠ "handler_log" ∨ rid_c ≠ rid then REJECT
22:    Let op ← A.HL_rid[i]
23:    if info does not match the fields in op then REJECT
24:
25: // The following procedure is called by ReExec while it is executing a control flow group
26: // when it encounters an emit operation.
27: // It checks that all requests in the group induce the same handlers,
28: // and adds the handlers to active.
29: procedure ActivateHandlers(hid, i, active)
30:    //Check that (hid, i) activates the same handlers across all requests, according to the advice (§4.3.1)
31:    if exist rid_1, rid_2 in the group s.t. activatedHandlers[rid_1, hid, i] ≠ activatedHandlers[rid_2, hid, i] then REJECT
32:    Let in the set of values of the emit operation across all requests.
33:    for all hid' ∈ activatedHandlers[rid, hid, i] for some rid in the group do
34:        active.Enqueue(hid', in)
```

Figure B.7: Pseudocode for check op routines and activateHandlers routine of Karousos verifier

```
1:  all_variables ← {} // A set of all variables.
2:  procedure OnInitialize(rid, hid, opnum, v)
3:      Let v.log ← VLv.variableID
4:      Let v.rid ← rid
5:      Let v.hid ← hid
6:      Let v.opnum ← opnum
7:      Let v.var_dict ← {} // Map from rid, hid, opnum to values.
8:      Let v.read_observers ← {} // maps from a write op to all readers who allegedly observed that op,
9:      // based on both server-supplied advice, and re-execution.
10:     Let v.write_observer ← {} // maps from a write op to 0 or 1 writers who allegedly observed that op,
11:     // based on either server-supplied advice or re-execution.
12:     Let v.initializer ← nil
13:     all_variables.insert(v)
14:
15: procedure OnRead(rid, hid, opnum, opcontents, v)
16:     if v.log.contains(rid, hid, opnum) then
17:         // if a read is logged, then the server was supposed to
18:         // have logged the dictating write. So find the dictating
19:         // write in the log, and feed its value to the read.
20:         op, _, rid_op, hid_op, opnum_op ← v.log{rid, hid, opnum}
21:         if op is not read or !v.log.contains(rid_op, hid_op, opnum_op) then
22:             return nil
23:         op, value, _, _, _ ← v.log{rid_op, hid_op, opnum_op}
24:         if op is not write then
25:             return nil
26:         v.read_observers{(rid_op, hid_op, opnum_op)}.insert((rid, hid, opnum))
27:         return value
28:     else
29:         //Below FindNearestRPrecedingWrite returns the last write by the nearest ancestor handler
30:         //by climbing up the handler tree and checking v.var_dict.
31:         Let rid_p, hid_p, opnum_p, value ← FindNearestRPrecedingWrite(v, rid, hid, opnum)
32:         if rid_p = nil and hid_p = nil then
33:             return nil
34:         v.read_observers{(rid_p, hid_p, opnum_p)}.insert((rid, hid, opnum))
35:         return v
```

Figure B.8: Code that verifier executes upon an annotated operation (§4.3.3), I

```
1:  procedure OnWrite(rid, hid, opnum, opcontents, v)
2:      Let v.var_dict{(rid, hid, opnum)} ← opcontents
3:      if v.log.contains(rid, hid, opnum) then
4:          Let op, value, rid_o, hid_o, opnum_o ← v.log{rid, hid, opnum}
5:          if op is not write or value ≠ opcontents then
6:              return false // Operations or values don't agree.
7:          if rid_o ≠ nil and hid_o ≠ nil and opnum_o ≠ nil then
8:              if v.write_observer{rid_o, hid_o, opnum_o} ≠ nil then
9:                  return false // Two handlers cannot overwrite the same value.
10:             else
11:                 Let v.write_observer{rid_o, hid_o, opnum_o} ← (rid, hid, opnum)
12:                 return true
13:     else
14:         Let (rid_p, hid_p, opnum_p, value) ← FindNearestRPrecedingWrite(v, rid, hid, opnum)
15:         if rid_p ≠ nil and hid_p ≠ nil and opnum_p ≠ nil then
16:             Let v.write_observer{rid_p, hid_p, opnum_p} ← (rid, hid, opnum)
17:         else
18:             Let v.initializer ← (rid, hid, opnum)
19:         return true

20: procedure AddInternalStateEdges
21:     for all v ← all_variables do
22:         Let (rid, hid, opnum) ← v.initializer
23:         while rid ≠ nil and hid ≠ nil and opnum ≠ nil do
24:             // Add WR (write-read) edges.
25:             for all (rid_r, hid_r, opnum_r) ← v.read_observers{rid, hid, opnum} do
26:                 G.add_edge((rid, hid, opnum), (rid_r, hid_r, opnum_r))
27:             if v.write_observer{rid, hid, opnum} ≠ nil then
28:                 // Add RW (anti-dependency) edges.
29:                 for all (rid_r, hid_r, opnum_r) ← v.read_observers{rid, hid, opnum} do
30:                     G.add_edge((rid_r, hid_r, opnum_r), v.write_observer{rid, hid, opnum})
                    // Add WW edge.
31:                 G.add_edge((rid, hid, opnum), v.write_observer{rid, hid, opnum})
32:             Let (rid, hid, opnum) ← v.write_observer{rid, hid, opnum}
```

Figure B.9: Code that verifier executes upon an annotated operation (§4.3.3) II

B.2 CORRECTNESS PROPERTIES

Definition 1 (Request/Response trace Tr)**.** An ordered list of the request and response events. The events appear in the list in chronological order. A request event is a tuple (REQ, rid, x) where *rid* is the request id of the request that was issued and x is the input data. A response event is a tuple (RESP, rid, y) where *rid* is the request id of the request that corresponds to this response and y are the contents of the response.

Definition 2 (Completeness)**.** An advice collection procedure and an audit procedure are defined to be *Complete* if the following holds: If the server serves the requests according to the annotated program P_a and executes the given advice collection procedure, then the given audit procedure (applied to the resulting trace and advice) passes.

Definition 3 (Request Schedule)**.** A request schedule is an ordered list of request ids that models the execution schedule. Notice that request ids are permitted to repeat in the schedule.

Definition 4 (Operation-wise execution)**.** Consider a model where, instead of requests arriving and departing, the executor has access to all request ids in a trace Tr and their inputs. Operation-wise execution means executing the program P by following a request schedule S; the output of operation-wise execution is a trace Tr'. Specifically:

- The executor runs the initialization process of P.
- Then, for each request id *rid* in the request schedule S in order:
  - If it is *rid*'s first appearance, the executor reads in the request's inputs x, appends (REQ, rid, x) to Tr' and initializes the active handlers set of *rid* with the request handlers for this request
  - Otherwise, the executor non-deterministically chooses one of the handlers in the active handlers set of *rid* and runs it up to and including its next special operation.

After the execution of a request's handler, the request is held, until the executor reschedules it.

If a request is scheduled but the request has no active handlers, the executor immediately yields and chooses the next *rid* in S.

- At the end, output Tr$'$.

Our operation-wise execution differs from the one in given in Orochi [146] in that it explicitly constructs an alternate trace instead of consulting the observed one.

Moreover, because of the non-deterministic choices flagged above, this procedure can produce multiple output traces for the same starting schedule S.

Definition 5 (O_S)**.** For a request schedule S, O_S is the set of all possible output traces that Operation-wise execution on request schedule S can generate.

Definition 6 (Soundness)**.** An advice collection procedure and an audit procedure are defined to be sound if the following holds: If the given audit procedure accepts a trace Tr and advice A, then there exists a request schedule S such that Tr $\in O_S$.

B.3 Proofs

We need the following definitions:

Definition 7 (R-precedes)**.** An operation $op = (rid, hid, \text{opnum})$ *R-precedes* an operation $op' = (rid', hid', \text{opnum}')$, written $op <_R op'$, iff

- $rid = rid'$ and $hid = hid'$ and opnum $<$ opnum$'$, or
- $rid = rid'$ and hid is an ancestor of hid' in the handler tree.

Definition 8 (R-ordered, R-concurrent)**.** Two operations op and op', with $op \neq op'$, are *R-ordered* iff $op <_R op'$ or $op' <_R op$. They are *R-concurrent* iff $op \not<_R op'$ and $op' \not<_R op$.

Definition 9 (Op Schedule)**.** An op schedule is a map:

$$S : \mathbb{N} \to requestid \times (\{0, \infty\} \cup \{handlerid \times (\mathbb{N} \cup \{\infty\})\})$$

For example:

$$(1,0),(23,0),(1,hid_1,0),(23,hid_2,0),(1,\infty),(1,hid_1,1)\ldots$$

where hid_1, hid_2 are handler ids as defined in Section 3.2 of the paper, and the natural number domain is implicit in the order.

Definition 10 (Well formed op schedule)**.** An op schedule S is well formed (with respect to a trace Tr and set of advice A) if:

1. it is a permutation of the graph G that is constructed by Preprocess,
2. it respects program order (that is, if there exists a program edge added by AddProgramEdges or a boundary edge added by AddBoundaryEdges in G from node n_1 to node n_2, then n_1 appears before n_2 in S), and
3. it respects activation order (that is, if there exists an activation edge from node n_1 to node n_2 in G, n_1 appears before n_2 in S)

Remark.. Notice that any topological sort of the graph G constructed by Preprocess in Audit(Tr,A) is well-formed. This is immediate from the definition.

OOOAudit. This procedure is shown in Figure B.10.

Lemma 5 (Equivalence of well formed op schedules)**.** For all op schedules S_1, S_2 that are well-formed (with respect to Tr and advice A)

$$\text{OOOAudit}(\text{Tr},A,S_1) = \text{OOOAudit}(\text{Tr},A,S_2).$$

Proof. The schedule does not affect the OOOAudit until the line where OOOExec is invoked. So up until then, either both executions accept or both reject.

Now, assume that OOOExec(S_1) and OOOExec(S_2) are equivalent, meaning that (a) either both accept or both reject and (b) they access the same variables setting the *initializer*, *write_observer*

```
1: //Global Variables are the ones in Figure B.2

2: procedure OOOAudit(op schedule S)
3:     Preprocess() // Figure B.2
4:     OOOExec(S)
5:     Postprocess() // Figure B.2
6:
7: procedure OOOExec(op schedule S)
8:     for each op in S do
9:         if op = (rid, 0) then
10:            Read inputs in of the request
11:            Allocate program structures
12:            active[rid] ← new Map
13:            Find the functionIDs of the request handlers
14:            for all functionID in functionIDs do
15:                Let hid ← (functionID, null, 0)
16:                active[rid][hid] ← in
17:                if A.opcounts[(rid, hid)] = ∅ then REJECT
18:        else if op = (rid, ∞) then
19:            Let hid ← A.responseEmittedBy[rid].hid
20:            Run the handler (rid, hid) until the next event
21:            if the next event is not a send response operation then REJECT
22:            write out the produced outputs
23:        else if op = (rid, hid, i) then
24:            if i = 0 then
25:                if (hid is not in active[rid]) then REJECT
26:                //It is the first operation
27:                Set the handler's inputs to active[rid][hid].
28:                Allocate structures for running the handler
29:            else if i = ∞ then
30:                Run the handler (rid, hid) until the next event
31:                if it is not a handler exit operation then REJECT
32:                Remove hid from active[rid]
33:            else
34:                Run the handler (rid, hid) until the next event
35:                if the next event is an external state operation then
36:                    optype ← the type of state operation
37:                    opcontents, tid, txnum ← parameters from execution
38:                    s ← CheckStateOp(rid, hid, i, optype, tid, txnum, opcontents)
39:                    if optype = GET then
40:                        state op result ← s
41:                else if the next event is an annotated operation then
42:                    if it is a write or initialization then
43:                        Execute the operation
44:                    Execute the annotation according to Figure B.8 where opnum is set to i
45:                else if the next event is a handler operation then
46:                    info ← parameters from execution
47:                    CheckHandlerOp(rid, hid, i, optype, info)
48:                    if the event is an emit operation then
49:                        for all hid′ ∈ activatedHandlers[(rid, hid, i)] do active[rid][hid′] ← value of the emit
50:
51:    if ∃rid s.t. ∃hid : A.opcounts[(rid, hid)] but (rid, hid) was not executed by OOOExec then REJECT
52:    if the produced outputs exactly match the responses in Tr then return ACCEPT
53:    return REJECT
```

Figure B.10: Pseudocode for OOOAudit in Karousos.

and *read_observers* of each variable to the same values. Now examine the execution of Postprocess. (b) implies that the edges that are added to G by AddInternalStateEdges are the same in both executions and, thus, the constructed graph G is the same in both executions. CycleDetect thus runs the same in both executions. Therefore, either both executions accept or both reject.

Now we need to prove that OOOExec(S_1) and OOOExec(S_2) are equivalent: The two schedules contain the same operations because they are constructed from the same graph G. We need to prove that each operation is executed in the same way in both executions. We will prove this by induction on the operations of each request.

1. Fix a request *rid*.
2. First notice that the only global state that is modified during OOOExec is the *active* map, per-variable dictionaries, lists of read_observers, write_observers, and initializer.
3. Base case: Because both schedules are well-formed, the first operation of a request is $(rid, 0)$: none of the data that this execution depends on get modified throughout OOOExec. So the execution of this operation is independent of its position in the log, and it is executed in the same way in both executions.
4. Induction: If both executions are about to execute operation k of request *rid*, and neither has rejected so far, the execution of operation k will proceed in the same way in both executions.
   - Assume that the next operation is (rid, ∞): The handler *hid* which both executions of OOOExec execute is the one in $A.responseEmittedBy$. Moreover, because the schedules are well formed, the latest operation of (rid, hid) that has been executed so far on both executions is
     $(rid, hid, A.responseEmittedBy[rid].\text{opnum})$. Thus, both executions will execute the handler that allegedly sends back the response, from the (allegedly) last operation prior to the response up until the next operation. Because of the induction hypothesis and the fact that the execution of a handler between operations is deterministic, the two executions will proceed

in the same way up until right before the next event, producing the same state. Moreover, the next event will be the same in the two executions. If this event is not the emission of a response, both executions will reject. Otherwise, because both executions have the same state, the produced outputs will be the same.

- Operations (rid, hid, i):
  - If it is a handler start operation ($i = 0$) then the executions do not depend on state that is modified except for the check in line 25. We will show that either both executions accept or both reject: Assume that this does not hold. Then, without loss of generality assume that the check passes in OOOExec(S_1) and fails in OOOExec(S_2). So in OOOExec(S_2), *hid* is not in $active[rid]$, either because (i) *hid* was in $active[rid]$ and removed from it, or (ii) *hid* was never added to $active[rid]$. We can rule out case (i) because the only place where *hid* could be removed is line 32 which, if it were executed, would mean that (hid, rid, ∞) appears before $(hid, rid, 0)$ in S_2, which is not possible, since S_2 is well-formed and in particular respects program order. So case (ii) holds.

    Now, in OOOExec(S_1), *hid* is in $active[rid]$. There are two places where *hid* could have been inserted: (a) line 16 during execution of $(rid, 0)$ or (b) line 49 during the execution of an emit operation $(rid, parent, j)$. Consider case (a). Because S_1 and S_2 are well-formed, $(rid, 0)$ appears before $(rid, hid, 0)$ in both S_1 and S_2. Also, as argued above, both OOOExec(S_1) and OOOExec(S_2) execute $(rid, 0)$ the same way, initializing $active[rid]$ to the same value. Therefore, if case (a) holds for OOOExec(S_1) then correspondingly, *hid* would have been inserted in OOOExec(S_2) in the same line, in contradiction to case (ii) above. So case (b) holds.

    In this case, because OOOExec(S_1) adds *hid* to $active[rid]$ during the execution of an emit operation $(rid, parent, j)$ at line 49, $hid \in activatedHandlers[(rid, parent, j)]$. This, in turn, implies that there is an activation edge $\langle (rid, parent, j), (rid, hid, 0) \rangle$ in G. So, because S_2 is well-formed, operation $(rid, parent, j)$ appears before $(rid, hid, 0)$

in S_2. This means that OOOExec(S_2) executes operation $(rid, parent, j)$ prior to executing $(rid, hid, 0)$ but during its execution it does not add *hid* to *active*[*rid*]. This can only be the case if during OOOExec(S_2), $hid \notin activatedHandlers[(rid, parent, j)]$. But this is impossible because *activatedHandlers* is the same across both executions (it is initialized during preprocessing and is not modified after preprocessing), and $hid \in activatedHandlers[(rid, parent, j)]$ during OOOExec(S_1).

- If it is a handler end operation ($i = \infty$), the execution does not depend on any objects that are modified during OOOExec so both executions proceed in the same way.
- If it is an external state operation: same argument as $i = \infty$.
- If it is an annotated operation (and hence interacting with, the aforementioned per-variable dictionaries and lists):
  - The parameters of the operation are the same across both executions because of the induction hypothesis and the fact that OOOExec proceeds deterministically from operation to operation.
  - If the operation is in the advice, then the execution proceeds in the same way in both executions.
  - If the operation is not in the advice, then both executions will find the nearest *R*-preceding write. Because of the induction hypothesis, and the fact that both schedules respect activation and program order, the nearest ancestor write will be the same in both executions, *regardless of the order in which concurrent handlers are re-executed*. This means that reading from the nearest ancestor will be the same in both executions (the same ancestor, the same value read) and, for this operation, both executions will add the same value to the variable dictionary (if it's a write operation) and both update *read_observers*, *write_observer* and *initializer* in the same way.
- If it's a handler operation: Same argument as $i = \infty$ and external state.

□

B.3.1 COMPLETENESS

At a high level, we need to show that if the server honestly executes the given program and the advice collection procedure, producing trace Tr and advice A, then Audit(Tr, A) accepts. We will do this in two steps:

1. First, we show that for any well-formed op schedule S, OOOAudit(Tr, A, S) accepts (Lemma 6).
2. Next, we establish that Audit(Tr, A) is equivalent to OOOAudit(Tr, A, S') for a specific well-formed op schedule S' (Lemma 7). We take S' to be the op schedule that results from a "flattened" batch execution.

Lemma 6 (OOOAudit Completeness)**.** If the executor executes the given program (under the execution model given in Section 4.2 and the given advice collection procedure, producing trace Tr and advice A, then for any well-formed op schedule S (with respect to Tr and A), OOOAudit(S) accepts.

Proof. Because of Lemma 5, it is sufficient to prove that there exists some well-formed op schedule S' (with respect to Tr and A) for which OOOAudit(S') accepts.

We will derive the op schedule S' from the online execution at the honest server. Define the following events during online execution:

- A *request event* happens when a request *rid* reaches the server, and is notated as $(rid, 0)$.
- A *response event* happens when the server issues a response for a request *rid*, and is notated as (rid, ∞).
- A *handler start event* happens when the server starts executing a handler (rid, hid), and is notated as $(rid, hid, 0)$.
- A *handler end event* happens when the server finishes executing a handler (rid, hid), and is notated as (rid, hid, ∞).

- A (rid, hid, i) event happens when the server either collects advice associated with a handler op or a state op, or when the handler executes an annotated operation.

We observe that there exists a partial order in which these events happen during online execution. The order is partial because some events may happen concurrently from the perspective of the system; for example, even if the trace shows that a particular event (such as a request's arrival) is earlier than another (such as a different request's arrival or response), the server may have "seen" those two events in the opposite order. Define a total order on these events by ordering concurrent events according to Tr if the events are both request/response events and arbitrarily otherwise. Take the op schedule S' to be this total order.

Sub-lemma 6.1. S' is well-formed, with respect to the Trace Tr and advice A produced by the online execution.

Proof. First, we show that S' is a permutation of the nodes in graph G. Since the server is honest, S' contains exactly one request event and exactly one response event for each request in Tr; so does G (from the logic of CreateTimePrecedenceGraph and SplitNodes). Moreover, S' contains exactly one handler start event and exactly one handler end event for each handler (rid, hid) that is executed; so does G. This follows from the logic of AddProgramEdges, specifically, lines 39 and 40 of Figure B.2, and the fact that the server faithfully executes the advice collection procedure, and sets the entries of $A.opcounts$ to exactly the handlers that are executed during online execution. Last, S' will contain exactly one entry for each handler operation/state operation/annotated operation that it executes. Because the honest server faithfully reports $A.opcounts$, S' will contain exactly one (rid, hid, i) for each $i < A.opcounts$. So will G (line 42 of Figure B.2). S' contains no other entries other than the ones above and G contains no other nodes other than the ones above. Thus, S' is a permutation of the nodes of G, as required.

Moreover, S' respects program order (Definition 10): The server faithfully executes the given program and collects the advice. This means that the relevant order of events within a handler

implied by the opnum field of the corresponding operations in the logs, and the order of the response event relative to the other events of the handler that issues the response implied by the contents $A.responseEmittedBy$ reflect what happened online. As a result, from the logic of AddProgramEdges of Figure B.2 and AddBoundaryEdges of Figure B.3, the existence of a program edge or boundary edge $\langle n_1, n_2 \rangle$ in G implies that n_1 happens before n_2 during online execution. Thus, n_1 also appears before n_2 in S', by construction of S'.

Last, we argue that S' respects activation order (Definition 10). Since S' reflects the order of events during online execution, it is sufficient to show that if there exists an activation edge

$$\langle (rid, parent\_hid, i), (rid, hid, 0) \rangle$$

in G, then the emit event $e = (rid, parent\_hid, i)$ activates handler (rid, hid) during online execution (because during a faithful execution a handler cannot start running until after the event that activates it is emitted). The activation edge is added to G at line 28 of Figure B.4 only if $hid.functionID$ is registered for the event e according to *GlobalHandlers* or according to *Registered*. We will show that in both cases, $hid.functionID$ is registered for the event e during online execution and, thus, e activates handler (rid, hid). In the former case, $hid.functionID$ is registered for e at the end of the initialization procedure at the verifier. Because the initialization procedure is deterministic, $hid.functionID$ is registered for e at the end of the initialization procedure at the online server and, because requests don't modify global handlers, it is still registered when e is emitted, as required. In the latter case, $hid.functionID$ is registered for e according to *Registered* only if there exists a register operation prior to e in HL_{rid}. Because the server executes the advice collection procedure faithfully, the order of operations in HL_{rid} reflects the order in which they are executed at the online server. This implies that $hid.functionID$ is registered for e during online execution. □

Sub-lemma 6.2. Preprocess passes.

Proof. Consider all the lines in which OOOAudit may reject during Preprocess. We need to show that if the server is well-behaved then all of the checks pass.

- Line 19 of Figure B.2: Passes because the honest server always sends back a response for each request it receives.
- Line 37 of Figure B.2: When the server is honest, it does not execute nor collect advice for any requests that are not in Tr.
- Lines 14 and 16 of Figure B.3: The honest server executes exactly the requests in Tr and sends back responses for exactly those requests. Moreover, it faithfully executes the advice collection procedure setting the contents of $A.responseEmittedBy[rid]$ for each *rid* that appears in the trace to a tuple: (hid_r, opnum_r) s.t. $0 \leq \text{opnum}_r \leq A.opcounts[(rid, hid)]$. Consequently, the check of line 14 passes. Moreover, notice that because of the logic of AddProgramEdges and the fact that the honest server correctly sets the $A.opcounts$, $(rid, hid_r, \text{opnum}_r)$ is added to G before the check of line 16 which implies that the check passes.
- Line 6 of Figure B.4: Because the server is well-behaved, it never includes a handler operations log in A for a request that is not in Tr.
- Line 25 of Figure B.4: As argued in the proof of lemma 6.1, if a *functionID* is registered for an event *e* when *e* is emitted according to *Registered* or *GlobalHandlers* during AddHandlerRelated-Edges, this *functionID* is registered for *e* during online execution. This implies that all handler ids for which line 25 is executed are handler ids that are actually activated by this operation during online execution. Moreover, the server, being well-behaved, has these ids as keys in $A.opcounts$. Thus, the check passes.
- Invocation of CheckOpIsValid in Line 10 of Figure B.4: When the server is honest, it correctly sets *opcounts* for each request in Tr and the contents of the logs so that each operation appears exactly once in the logs. Under these conditions the checks pass.
- Line 48 of Figure B.4: When the server is honest, each $\texttt{GET}(key)$ operation reads the contents of a $\texttt{PUT}(key, \cdot)$ operation. Moreover, the honest server correctly logs state operations in $A.TXL$. Under these conditions, the check passes.

- Line 51 of Figure B.4: When the server is well-behaved, the execution at the database is internally consistent (Section C) meaning that if a transaction modifies a *key* and later reads it, it reads its latest modification. Moreover, the well-behaved server correctly sets the opcontents field of each `GET` operation to the position of its dictating write in $A.TXL$s. This implies that for each `GET` operation op that appears in some $A.TXL_t$ after a `PUT` operation op' with $op'.key = op.key$, op.opcontents corresponds to the last `PUT` operation to $op.key$ that precedes op in $A.TXL_t$. Meanwhile, from the logic of AddExternalStateEdges, when a `GET` operation $op \in A.TXL_t$ is processed, $op.key \notin MyWrites$ iff there are no `PUT` operations to $op.key$ prior to op in $A.TXL_t$. Otherwise, $MyWrites[op.key]$ is the last `PUT` operation to $op.key$ that precedes op in $A.TXL_t$. Thus, either $op.key \notin MyWrites$ or $MyWrites[op.key] = op$.opcontents. So, the check passes.
- Lines 23 and 27 of Figure B.5: We show that both checks pass by showing that the entries (rid, tid, i) of $A.writeOrder$ are exactly the set of (rid, hid, i) for which there exists some *key* s.t. $lastModification[rid, tid, key] = i$. First, notice that because the server is well-behaved the entries (rid, tid, i) of $A.writeOrder$ correspond to the `PUT` operations that the server applied to the external state. These are exactly the last modifications of committed transactions: that is, the `PUT` operations op s.t. op belongs to a committed transaction and op is the last operation of the transaction that modifies a key. Moreover, the honest server correctly sets the entries of $A.TXL$s. Thus, the entries (rid, tid, i) of $A.writeOrder$ are exactly the operations $op = A.TXL_{(rid,tid)}[i]$ s.t. (1) the last operation of (rid, tid) is `tx_commit`, and (2) there exists no $j > i$ s.t. $A.TXL_{(rid,tid)}[j]$ is a `PUT` on $op.key$. From the logic of AddExternalStateEdges, these are exactly the (rid, hid, i) s.t. $\exists key : lastModification[rid, tid, key] = i$, as required.
- Line 36 of Figure B.5: First, observe that from the logic of AddExternalStateEdges, *ReadMap* maps each `PUT` operation that appears in the logs to the set of `GET` operations that read from it according to the advice. Thus, to show that this check passes for all `GET` operations in the range of *ReadMap*, we show that for each `GET` operation op that appears in some $A.TXL_t$ either $t \notin Committed$ or op.opcontents $\in A.writeOrder$. Observe that this line is executed only when

the purported isolation is READ COMMITTED or SERIALIZABILITY. Consider the history of execution (Section C) at the honest server. The history is consistent with the isolation level and, thus, does not exhibit phenomena G1a and G1b. Consequently, during online execution, `GET` operations of committed transactions only read from operations that correspond to last modifications of committed transactions. These latter operations are exactly the ones that the honest server places in the $A.writeOrder$. Thus, `GET` operations of committed transactions only read from operations that are in the $A.writeOrder$. Because the server correctly logs state operations in $A.TXL$s, we deduce that for each `GET` operation op that appears in some $A.TXL_t$ either the last operation in $A.TXL_t$ is `tx_abort` and, thus, $t \notin Committed$ or op.opcontents $\in A.writeOrder$, as required.

- Line 11 of Figure B.5: We need to show that if the server is honest, then the graph DG when this line is executed is acyclic. This line is executed only when the purported isolation level is READ UNCOMMITTED. Consider the history of execution H at the honest server (Section C). Because the server is well-behaved, H exhibits READ UNCOMMITTED which implies that H does not exhibit phenomenon G0: $DSG(H)$ contains no cycles consisting of write depend edges. We show that DG is acyclic by showing that DG is the subgraph of $DSG(H)$ that contains only write depend edges. First, DG and $DSG(H)$ have the same nodes: DG has a node for each transaction that commits during online execution whereas $DSG(H)$ has a node for each transaction in $Committed$. Because the honest server collects advice for each transaction that it executes and the last operation of each committed transaction t in $A.TXL_t$ is a `tx_commit` operation, AddExternalStateEdges adds exactly the transactions that commit during online execution to $Committed$ (line 35 of Figure B.4). Thus, DG and $DSG(H)$ have the same nodes, as required. Now we argue that the edges of DG which are the write dependency edges (added at line 47 of Figure B.5) are exactly the write depend edges of $DSG(H)$: Observe that the write depend edges of $DSG(H)$ are the edges$\langle t_1, t_2 \rangle$ s.t. t_1 writes a key and t_2 writes the next version of the key according to H (Section C). That is, $DSG(H)$ has a write depend edge $\langle t_1, t_2 \rangle$ iff there exist

operations $op_1 = (t_1, i)$ and $op_2 = (t_2, j)$ s.t.

1. op_1 appears before op_2 in the version order of H,
2. $op_1.key == op_2.key$, and
3. for each operation op' that appears between op_1 and op_2 in the version order of H, the key that op' writes is not key.

Moreover, a well-behaved server sets $A.writeOrder$ to the version order of H, and correctly logs state operations in $A.TXL$s. Thus, the write depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist indexes i and j s.t.

1. (t_1, i) appears before (t_2, j) in $A.writeOrder$,
2. $A.TXL_{t_1}[i].key = A.TXL_{t_2}[j].key$, and
3. for each operation (t, k) that appears between (t_1, i) and (t_2, j) in $A.writeOrder$, it holds $A.TXL_t[k].key \neq A.TXL_{t_1}[i].key$.

Meanwhile, from the logic of ExtractWriteOrderPerKey, t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$ iff they meet the above conditions. Thus, t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$ iff $\langle t_1, t_2 \rangle$ is a write depend edge of $DSG(H)$. Moreover, from the logic of AddWriteDependEdges the edges of DG are exactly the edges $\langle t_1, t_2 \rangle$ s.t. t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$. Thus, the edges of DG are exactly the write depend edges of $DSG(H)$, as required.

- Line 15 of Figure B.5: As in the previous case, we need to show that the graph DG when this line is executed is acyclic. This line is executed only when the purported isolation level is READ COMMITTED. Consider the history of execution H at the honest server (Section C). Because the server is well-behaved, H exhibits READ COMMITTED which implies that H does not exhibit phenomenon G1c: $DSG(H)$ contains no cycles consisting of write depend edges and read depend edges. We show that DG is acyclic by showing that DG is the subgraph of $DSG(H)$ that contains the write depend and read depend edges of $DSG(H)$. Specifically, we show:

1. that DG and $DSG(H)$ have the same nodes,
2. that the write dependency edges of DG (added at line 47 of Figure B.5) are exactly the write depend edges of $DSG(H)$, and
3. that the read dependency edges of DG (added at line 40 of Figure B.5) are exactly the read depend edges of $DSG(H)$.

We show 1 and 2 as above (in the the proof that the check at line 11 of Figure B.5 passes). Now we show 3: First, observe that the read depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ s.t. some operation of t_2 reads a value written by t_1. Moreover, because the server is well-behaved, the history H does not exhibit phenomenon G1b: as explained above (in the proof that the checks at lines 23 and 27 of Figure B.5 pass), this implies that all `GET` operations of committed transactions read from operations that are in H's version order. Thus, the read depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ for which there exist an operation op_1 that t_1 issues and an operation op_2 that t_2 issues s.t.

- op_2 reads the value written by op_1,
- op_1 appears in H's version order,
- t_2 commits, and
- $t_1 \neq t_2$

Because the server is well-behaved, it correctly logs all state operations in the $A.TXL$s and sets $A.writeOrder$ to H's version order. Moreover, as argued above (in the the proof that the check at line 11 of Figure B.5 passes), when the server is honest, *Committed* contains exactly the transactions that commit during online execution. Thus, the read depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist operations (t_1, i) and (t_2, j) s.t.:

- For $op = A.TXL_{t_2}[j]$ it holds that op.optype $=$ `GET` and op.opcontents $= (t_1, i)$,
- $(t_1, i) \in A.writeOrder$,
- $t_2 \in Committed$, and

- $t_1 \neq t_2$

These are exactly the read dependency edges of *DG*: from the logic of AddExternalStateEdges, *ReadMap* maps each PUT operation (t_1, i) to the set of all GET operations (t_2, j) for which $A.TXL_{t_2}[j]$.opcontents $= (t_1, i)$. Moreover, AddReadDependencyEdges examines all (t_1, i) and (t_2, i) for which $(t_2, j) \in ReadMap[(t_1, i)]$ and adds a read dependency edge $\langle t_1, t_2 \rangle$ to *DG* iff $A.TXL_{t_2}[j]$.opcontents $= (t_1, i)$, $(t_1, i) \in A.writeOrder$, $t_2 \in Committed$, and $t_1 \neq t_2$. Thus, the read dependency edges of *DG* are exactly the read depend edges of $DSG(H)$, as required.

- Line 20 of Figure B.5: As in the previous case, we need to show that the graph *DG* when this line is executed is acyclic. This line is executed only when the purported isolation level is SERIALIZABILITY. Consider the history of execution H at the honest server (Section C). Because the server is well-behaved, H exhibits SERIALIZABILITY which implies that H does not exhibit phenomena G1c and G2 and, thus, $DSG(H)$ contains no cycles. We show that *DG* is acyclic by showing that *DG* is exactly $DSG(H)$. Specifically we show 1, 2, and 3 as in the previous case and, additionally, we show that the anti dependency edges of *DG* (added at line 56 of Figure B.5) are exactly the anti depend edges of $DSG(H)$: The anti depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ s.t. t_1 reads some version of a *key* and t_2 writes the next version of *key* according to the H's version order. Thus, the anti depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist a transaction t_3, and operations op_1, op_2, and op_3 issued by t_1, t_2, and t_3 respectively:
  - op_3 appears before op_2 in the version order of H,
  - $op_3.key = op_2.key$,
  - for each operation op' that appears between op_3 and op_2 in the version order of H, the key that op' writes is not $op_3.key$,
  - op_1 reads the value written by op_3,
  - $t_1 \neq t_2$, and

- t_1 commits

Because the server is well-behaved, it correctly logs all state operations in the *A.TXL*s and sets *A.writeOrder* to H's version order. Moreover, as argued above, when the server is honest *Committed* contains exactly the transactions that commit during online execution. Thus, the anti depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ for which there exists a transaction t_3 and operations (t_1, i), (t_2, j), and (t_3, k) s.t.:

- (t_3, k) appears before (t_2, j) in the version order of H,
- $A.TXL_{t_2}[j].keyA.TXL_{t_3}[i].key$
- for each operation (t, ℓ) that appears between (t_3, i) and (t_2, j) in *A.writeOrder*, it holds $A.TXL_t[\ell].key! = A.TXL_{t_3}[i].key$.
- $A.TXL_{t_1}[i]$.optype $=$ `GET` and $A.TXL_{t_1}[i]$.opcontents $= (t_3, i)$,
- $t_1 \neq t_2$
- $t_1 \in$ *Committed*

From the logic of AddExternalStateEdges, ExtractWriteOrderPerKey and AddAntiDependencyEdges, these are exactly the anti dependency edges of *DG*.

□

Sub-lemma 6.3. The invocation of OOOExec(S'):

- reproduces the program state of online execution
- passes all checks

Proof. Proof outline: Induct on S':

Base case:. The first operation in S' has no ancestors in G'. It can only be an operation $(rid, 0)$ for some $rid \in$ Tr. OOOExec handles this operation by allocating structures for running and reading in the inputs. This is the same behavior as online execution. Moreover, OOOExec finds all request

handlers for *rid*, computes their handler ids and checks that for each handler id there is an entry in *opcounts* (Line 17). This check will pass because the honest server sees the same request handlers for the request during online execution, computes their handler ids in the same way as the verifier (the computation is deterministic), and has entries for each of them in *opcounts*.

Inductive step:. Assume that the claim holds for the first $\ell - 1$ operations in S'. Let *op* be the ℓ-th operation in S':

- Case I: $op = (rid, 0)$: Same reasoning as in the base case.
- Case II: $op = (rid, hid, 0)$ where handler (rid, hid) is a request handler (that is, $hid.parent\_hid = null$): Because S' obeys program order, this operation appears after $(rid, 0)$ and before (rid, hid, ∞). This means that, because this handler is a request handler, when OOOExec executes this operation, *hid* has already been added to $active[rid]$ in line 16, has not been removed yet, and $active[rid][hid]$ has been set to the request inputs. Thus, the check of line 25 passes, OOOExec sets the handler's inputs to the request inputs and allocates structures for running the handler. This is the same behavior as online execution.
- Case III: $op = (rid, hid, i)$ where $i = 1$ and handler (rid, hid) is a request handler (that is, $hid.parent\_hid = null$) By the induction hypothesis and the fact that S' obeys program order, OOOExec and online execution had the same program state at $(rid, hid, 0)$. Because the server is well-behaved, both online execution and OOOExec will take the same next step in terms of handler op, handler exit event, external state op, or annotated operation. Since the server is well-behaved and $opcounts[rid][hid] > 1$, the next operation is a handler op, an external state op, or an annotated operation in both executions.
  - Handler Op: Similar arguments to the ones in case III of Sub-lemma 7b of Orochi [146]. The determinism of passing from $(rid, hid, 0)$ to $(rid, hid, 1)$ and the induction hypothesis imply that the program state of online execution and the program state of OOOExec right before executing the handler operation are the same. Being well-behaved, the server recorded this op-

eration correctly in HL_{rid} and this is the operation that the verifier checks in CheckHandlerOp. Moreover, the contents of the log entry (optype and *info*) are the ones produced during online execution and consequently the ones produced during by OOOExec. Under these conditions CheckHandlerOp passes. Moreover, if the operation is an emit operation, the handler ids in *activatedHandlers* are exactly the ones that this operation activated and their inputs in *active* will be set to the inputs during online execution.

- External State Op: First, observe that the operation has the same *tid* and txnum under both executions: If optype is `tx_start` then both online execution and OOOExec compute the same *tid* as (hid, opnum) and set txnum $= 0$, as required. Otherwise, because of the induction hypothesis, both executions have assigned the same *tid* to this transaction. Meanwhile, the operations of a transactions are not concurrent meaning that the order of operations within a transaction is consistent with program order and activation order. Because both online execution and OOOExec(S') follow program order and activation order, the transaction *tid* issues the same number of operations prior to *op* under both executions. Thus, txnum is the same under both executions as required. This implies that during OOOExec(S'), the operation is checked against the entry in the logs that the honest server records for this operation during online execution. Moreover, the determinism of passing from $(rid, hid, 0)$ to $(rid, hid, 1)$ and the induction hypothesis imply that the program state of online execution and the program state of OOOExec right before executing the state operation are the same. This implies that the parameters of the operation $(\text{optype}, \text{opcontents}, key)$ are the same under both executions except in the case where optype $=$ `tx_commit`: in this case the recorded operation in the logs may be `tx_abort` because during online execution, the transaction could not successfully commit. Meanwhile, because the server is well behaved, it correctly logs the operation in $A.TXL_{(rid,tid)}$. Thus, all checks of CheckStateOp pass. Moreover, the well behaved server correctly sets the opcontents field of a `GET` operation to (rid_w, tid_w, i_w) s.t. $A.TXL_{(rid_w,tid_w)}[i_w]$ is the dictating `PUT` operation. This implies that the value that OOOExec reads is the one written

by the dictating `PUT`, which is the value read at online execution. Thus, the two executions have the same program state after executing the state operation.

- Annotated Operation: Both online execution and OOOExec execute the operation and call the annotation with arguments (rid, hid, i) for the same variable v. We will argue that the claim holds after executing the annotation for any handler hid and any i.
  - Initialization: In this case both executions execute the operation and then execute the annotation, which performs no checks. Thus, the two executions result in the same program state.
  - Write operation: Both executions execute the operation, assigning the same value to the v, and then execute the annotation. If during the execution of the annotation by OOOExec the operation is found in the logs, then the server, being well-behaved, has correctly recorded the operation in $v.\log$. As a result, all checks of OOOExec pass. Otherwise, the verifier does no checks.
  - Read operation: We need to show that the value that OnRead returns in OOOExec is the value of v when the annotation is executed by online execution, which is the value written by its dictating write.

    If during the execution of the annotation the read operation is in $v.\log$ then the online server, being well-behaved, has correctly logged both the read operation and its dictating write. As a result, all checks pass and OnRead sets the value of v to its value during online execution.

    We will now argue that OnRead returns the value of v at online execution when the operation is not in $v.\log$. If the operation is not in $v.\log$, then this is because when the operation is executed by the honest server, op reads the value written by some operation op' that is not R-concurrent with op (Definition 8). Thus, $op' <_R op$. Meanwhile, because $\text{OOOExec}(S')$ follows program order and activation order, it executes op' prior to executing op. Furthermore, from the induction hypothesis, $\text{OOOExec}(S')$ and online execution have the same

program state when they execute op', meaning that the parameters of op' that OOOExec(S') records in *v.var_dict* (at line 2 of Figure B.9) are exactly the parameters of op' during online execution. Thus, when OOOExec(S') executes op, there exists an entry in *v.var_dict* that maps op' to the value written during online execution. If op' is not the nearest R-preceding write of op in *v.var_dict*, then there is a later ancestor, call it op'', such that op' was re-executed before op'', which was re-executed before op. Since op' and op'' R-precede op and since each handler has only one parent, we must have $op' <_R op'' <_R op$. But $<_R$ never inverts online execution, so op'' was also executed in between op' and op online, in which case op could not have observed op' without violating causality. Thus, there is no such op''. FindNearestRPrecedingWrite therefore returns op' and reads the value written by op' during online execution, as required.

- Case IV: $op = (rid, hid, i)$ where $i \in [2, A.opcounts[(rid, hid)]]$ and handler *hid* is a request handler (that is, $hid.parent\_hid = null$) Same arguments as in case III.
- Case V: $op = (rid, hid, \infty)$ where *hid* is a request handler (that is, $hid.parent\_hid = null$) An argument similar to one made elsewhere (Orochi [146], Sub-lemma 7b, Case II) establishes that the next operation is handler exit both in online execution and in OOOExec. OOOExec handles handler exit events in the same way as online execution.
- Case VI: $op = (rid, hid, 0)$ where *hid* is not a request handler. We need to show that the check of line 25 of Figure B.10 accepts and that the inputs on which the handler is executed by OOOExec are the ones of online execution. Because (rid, hid) is not a request handler it is activated by some emit operation op' during online execution and, since the server is well-behaved, op' is executed before op during OOOExec. From the induction hypothesis, the program state of OOOAudit when it executes op' is the one of online execution. This implies that if line 49 of Figure B.10 is executed for *hid*, $active[rid][hid]$ is set to the handler's inputs according to online execution. In order for this line to be executed for *hid*, it must be that $hid \in activatedHandlers[op']$. This

can only happen if when op' is parsed during AddHandlerRelatedEdges $hid.functionID$ is registered for $op'.eventName$ according to *GlobalHandlers* or *Registered*. We will now argue that this is indeed the case. Because op' activates (rid, hid) during online execution, $hid.functionID$ is registered for $op'.eventName$ when op' is executed at the online server. $hid.functionID$ is either a global handler, or there exists some operation op'' executed by the request rid that registers $op'.eventName$ for $hid.functionID$ during online execution. In the former case per the determinism of the initialization procedure $(op'.eventName, hid.functionID) \in GlobalHandlers$. In the latter case, because the server is well-behaved, op'' appears before op' in HL_{rid} and $(op'.eventName, hid.functionID) \in Registered$ when op is examined during AddHandlerRelatedEdges.

- Case VII: $op = (rid, hid, i)$ where hid is not a request handler and $i \in [1, A.opcounts[(rid, hid)]]$. We can show this using the same arguments as in cases III and IV above.
- Case VIII: $op = (rid, hid, \infty)$ where hid is not a request handler. We can show this using the same arguments as in case V above.
- Case IX: $op = (rid, \infty)$. Because S' is well-formed,

$$(rid, A.responseEmittedBy[rid].hid, A.responseEmittedBy[rid].\text{opnum})$$

is the last operation of $(rid, A.responseEmittedBy[rid].hid)$ that OOOExec executes prior to op. Because the server is well-behaved, it correctly sets the contents of $A.responseEmittedBy$. Because of the induction hypothesis, the program state of handler $(rid, A.responseEmittedBy[rid].hid)$ at the time when op is encountered is the one of online execution. Under these conditions, the next operation of handler $(rid, A.responseEmittedBy[rid].hid)$ is the issue of the response and the check of line 21 passes. Moreover, because execution between operations is deterministic, the produced outputs are the ones of online execution.

Moreover, the check in line 51 passes because S' being well-formed contains operations for all (rid, hid) in $A.opcounts$ and, thus, all $(rid, hid) \in A.opcounts$ are executed by OOOExec.

Last, as argued in case IX, all responses will match the ones of online execution and OOOExec accepts at line 52 of Figure B.10. □

Sub-lemma 6.4. Postprocess passes.

Proof. Postprocess rejects only when the graph G has a cycle. So our task is to show that, when the server is honest, graph G is acyclic. We have already argued (earlier) that the events that happen during online execution have a partial order. Below, we will show that if there exists an edge $\langle n_1, n_2 \rangle$ in G, then n_1 precedes n_2 in that partial order. Now, if there were a cycle in G, that would imply that some event precedes itself in the partial ordering, which contradicts the definition of partial order.

Consider the edges that are added to G during Preprocess:

- Procedure SplitNodes: An edge $\langle (rid_1, \infty), (rid_2, 0) \rangle$ is added to the graph only if the response for rid_1 appears in the trace before the request rid_2 is issued. This implies that the response for rid_1 was issued by the server before the request rid_2 reached the server. Thus, (rid_1, ∞) happened before $(rid_2, 0)$, as required.
- Line 7 of Figure B.3: An edge $\langle (rid, 0), (rid, hid, 0) \rangle$ is added because *hid* is a request handler for *rid*. All handlers for a request start executing after the request reaches the server, so the event $(rid, 0)$ happened before the event $(rid, hid, 0)$ during online execution
- Lines 43 and 44 of Figure B.2: An edge $\langle n_1, n_2 \rangle$ is added to the graph because according to the advice, n_1 preceded n_2 during the execution of a handler. Because the server is honest, n_1 indeed preceded n_2 during online execution.
- Lines 17, 20 and 22 of Figure B.3: For some *rid*, let *hid* be the handler that issued the response according to *responseEmittedBy* and n be the last operation of *hid* prior to issuing the response according to *responseEmittedBy*. Because the honest server correctly sets the contents of *responseEmittedBy* and send_response is a synchronous operation at the server, these lines add edges to indicate that a response is issued after n and before the next event of *hid* during online execution

- Line 15 of Figure B.4: such edges are added because according to the advice a handler operation preceded another handler operation. Since the server is honest, this precedence held during online execution as well.
- Line 28 of Figure B.4: such an edge $\langle (rid, parent\_hid, \text{opnum}), (rid, hid, 0) \rangle$ is added only if according to the advice the emit operation $(rid, parent\_hid, \text{opnum})$ activates handler (rid, hid). Because the server is well-behaved the emit operation $(rid, parent\_hid, \text{opnum})$ activates handler (rid, hid) during online execution, and, because a handler does not start running until the event that activates it is emitted, the handler start operation $(rid, hid, 0)$ happens after operation $(rid, parent\_hid, \text{opnum})$ during online execution as required.
- Line 46 of Figure B.4: Such an edge $\langle n_1, n_2 \rangle$ is added only if the operation n_2 reads a value written by operation n_1 according to the advice. Because the server is well-behaved, n_2 truly reads from n_1 during online execution and, because an operation cannot read from the future, the operation n_1 executes before operation n_2 during online execution. Meanwhile, during online execution, the server collects advice for `PUT` operations before issuing them to the database and it collects advice for `GET` operations after their execution at the database completes. Thus, event n_1 precedes event n_2 during online execution, as required.

Now consider the edges added in G during Postprocess. That is, edges added during AddInternalStateEdges.

First, we argue that if at the beginning of Postprocess for two operations n_1, n_2, there exists some variable v s.t. $v.write\_observer\{n_1\} = n_2$ or $n_2 \in v.read\_observers\{n_1\}$, then n_1 happens before n_2 during online execution. We will only argue this in the case where $v.write\_observer\{n_1\} = n_2$ because the *read_observers* case is similar. $v.write\_observer\{n_1\}$ can be set to n_2 at only two locations during OOOExec(S'). First, in line 11 of Figure B.9 which is executed if n_2 is in the logs and the server has recorded n_1 as the previous write. Because the server is well-behaved, n_1 happens before n_2 during online execution. Second, in line 16 of Figure B.9, which is executed if

n_1 is identified as the nearest write by some ancestor of n_2. In this case n_1 appears before n_2 in S'. Because n_1 and n_2 operate on the same variable and we assume that variables are serializable, n_1 and n_2 are ordered during online execution and, because S' follows the order of online execution on non-concurrent operations, n_1 happens before n_2 during online execution.

Now, we argue that for each edge $\langle n_1, n_2 \rangle$ added during AddInternalStateEdges, it holds that n_1 happens before n_2 during online execution. For ww and wr edges, this follows immediately from our previous argument about *write_observer* and *read_observers*: a ww-edge is added iff $v.write\_observer\{n_1\} = n_2$ for some v and a wr edge is added iff $n_2 \in v.read\_observers\{n_1\}$.

Last, we need to argue this about rw edges. We will do this by contradiction. A rw edge $\langle n_1, n_2 \rangle$ can be added to G only if there exists some $n \notin \{n_1, n_2\}$ s.t. $n_1 \in v.read\_observers\{n\}$ and $v.write\_observer\{n\} = n_2$. Assume toward a contradiction that n_2 happens before n_1 during online execution. Because honest servers don't allow reads from the future, n_1 either (i) reads from n_2 or (ii) reads from some write that happened subsequently to n_2.

In case (i), we claim that $n_1 \in v.read\_observers\{n_2\}$. There are two sub-cases: either n_2 is an ancestor of n_1 during online execution, or n_2 and n_1 are concurrent during online execution. For the first sub-case: because OOOExec(S') follows activation order, n_2 is an ancestor of n_1 during OOOExec and n_1 is added to $v.read\_observers\{n_2\}$ at line 34 of Figure B.8. For the second sub-case: because a faithful server logs concurrent accesses for which at least one is a write, OOOExec adds n_1 to $v.read\_observers\{n_2\}$ at line 26 of Figure B.8. Combining $n_1 \in v.read\_observers\{n_2\}$ with $n_1 \in v.read\_observers\{n\}$ (from the fact of an rw edge $\langle n_1, n_2 \rangle$), we have a contradiction, as an operation is only added to one *v.read_observers*.

In case (ii), there exist $n'_1, \ldots, n'_k$ s.t.

$$v.write\_observer\{n_2\} = n'_1$$
$$v.write\_observer\{n'_{i-1}\} = n'_i, \forall i \in [2,k]$$
$$n_1 \in v.read\_observers\{n'_k\}$$

Notice that because of our previous argument about write observers, the above equations imply that n_2 happens before n'_k during online execution. In order for the rw edge to exist, since n_1 can only appear in one *v.read_observers* it should hold $v.write\_observer\{n'_k\} = n_2$. This implies that n'_k happens before n_2 during online execution which is a contradiction. □

□

Lemma 7 (Equivalence of OOOAudit and Audit)**.** If the server executes the given program and advice collection procedure, producing trace Tr and advice A, then there exists a well-formed op schedule S' (with respect to Tr and A) such that $\text{Audit}(\text{Tr}, A)$ and $\text{OOOAudit}(\text{Tr}, A, S')$ are equivalent.

Proof. We use the control flow groupings to create an op schedule S' as follows: Initially S' is empty. For each control flow group C we add each request's operations in layers as follows:

1. For each request id r in C, append $(r, 0)$ to S'
2. Pick some request id rid^* in the group C
3. Initialize a set R that contains tuples (event name, function ID).
4. Initialize *active* to the ids of the request handlers of rid^*. .
5. $I \leftarrow active$.
6. While $active \neq null$:
   (a) If $I \neq null$:

i. Pick some *hid* from I and remove this *hid* from I.

ii. If $hid \notin active$, go to step 6.

Otherwise, pick some *hid* from *active*.

(b) For opnum $= 0 \ldots A.opcounts[(rid^*, hid)]$:

i. For all requests r in the group, append (r, hid, opnum) to S'.

ii. If $A.responseEmittedBy[rid^*] = (hid, \text{opnum})$, then for all requests r in C, append (r, ∞) to S'.

iii. $(t, rid_c^*, i) \leftarrow \text{OpMap}[(rid^*, hid, \text{opnum})]$.

iv. if $t =$ *"handler_log"* and $A.HL_{rid^*}[i].\text{optype} = \text{register}$, for all $eventName \in A.HL_{rid^*}[i].eventNames$ do:

$$R.add(eventName, A.HL_{rid^*}[i].functionID)$$

v. if $t =$ *"handler_log"* and $A.HL_{rid^*}[i].\text{optype} = \text{unregister}$,

$$R.remove(A.HL_{rid^*}[i].eventName, A.HL_{rid^*}[i].functionID)$$

vi. If $(rid^*, hid, \text{opnum})$ is in *activatedHandlers*,

A. add all hid' in $activatedHandlers[(rid^*, hid, \text{opnum})]$ to *active*.

B. $(t, rid_c^*, i) \leftarrow \text{OpMap}[(rid^*, hid, \text{opnum})]$.

C. $eventName \leftarrow A.HL_{rid^*}[i].eventName$.

D. For all f s.t. $(eventName, f) \in R \cup GlobalHandlers$, add (f, hid, opnum) to I

(c) For all requests r in the group, append (r, hid, ∞) to S'.

(d) Delete *hid* from *active*

Now we must argue that S' is well-formed.

First, we need to show that S' is a permutation of the nodes of G, that is

$$G.nodes = set(S') \tag{B.1}$$

where $set(A) = \{a \mid \exists i A[i] = a\}$.

We do this through two more relations (B.2) and (B.3). Specifically, we will show: that relation (B.2) implies (B.1), that relation (B.3) implies relation (B.2), and finally that relation (B.3) holds. The relations are:

$$\forall rid^* \in R : \{n \mid n \in G.nodes \wedge n.rid = rid^*\} = \{n \mid n \in set(S') \wedge n.rid = rid^*\} \tag{B.2}$$

and

$$\forall rid^* \in R : hid \in active \Leftrightarrow A.opcounts[(rid^*, hid)] \neq null \tag{B.3}$$

where R is the set of rids picked at step 2 above.

First, we show that relation (B.2) implies relation (B.1): Because the server is well-behaved, two requests are in the same group only if they activate the same handlers, take the same control flow path on each handler, and activate the same handlers using corresponding emit operations. Thus, requests in the same group have the same $A.opcounts$ and the same $A.responseEmittedBy$, which implies that they have corresponding nodes in G and corresponding operations in S'. Consequently, if relation (B.2) holds, then relation (B.1) holds, as required.

Now we show that relation (B.3) implies relation (B.2): Specifically, we show that the backward direction of relation (B.3) implies that

$$\forall rid^* \in R : \{n \mid n \in G.nodes \wedge n.rid = rid^*\} \subseteq \{n \mid n \in set(S') \wedge n.rid = rid^*\}$$

and that the forward direction implies that

$$\forall rid^* \in R : \{n \mid n \in set(S') \wedge n.rid = rid^*\} \subseteq \{n \mid n \in G.nodes \wedge n.rid = rid^*\}$$

Consider arbitrary rid^* and denote G_{rid^*} the set of nodes associated with rid^*, that is $\{n \mid n \in$

$G.nodes \wedge n.rid = rid^*\}$. Observe that from the logic of AddHandlerProgramEdges, G_{rid^*} contains the nodes $(rid^*, 0), (rid^*, \infty)$ and (rid^*, hid, i), for $i = 0, \ldots, A.opcounts[(rid^*, hid)], \infty$ for all *hid* s.t. $A.opcounts[(rid^*, hid)] \neq null$.

First, we show that when the backward direction of relation (B.3) holds, then each of the nodes in G_{rid^*} is added to S': First, $(rid^*, 0)$ is added to S' at step 1. Second, the backward direction of relation (B.3) implies that all handler ids that have entries in $A.opcounts$ are added to *active*. Moreover, from the logic of step 6a, steps 6b and 6c are executed for each $hid \in active$. Thus, steps 6b and 6c are executed for each *hid* s.t. $A.opcounts[(rid^*, hid)] \neq null$. Thus, for all *hid* s.t. $A.opcounts[(rid^*, hid)] \neq null$ the operations (rid^*, hid, i), for $i = 0, \ldots, A.opcounts[rid^*][hid], \infty$ are added to S'. Last, denote $A.responseEmittedBy[rid^*]$ as (hid_r, opnum_r). Because the honest server correctly sets the contents of $A.responseEmittedBy$, $A.opcounts[(rid^*, hid_r)] \neq null$. Because the backward direction of relation (B.3) holds, hid_r is added to *active* and step 6b is executed for hid_r. Moreover, because the server is well behaved, $\text{opnum}_r \in [0, A.opcounts[(rid^*, hid_r)]$ which implies that step 6(b)ii is executed for $(rid^*, hid_r, \text{opnum}_r)$ and (rid^*, ∞) is added to S'.

Now we show that when the forward direction of relation (B.3) holds, then each of the operations of S' associated with rid^* are in G_{rid^*}. Assume that the forward direction of relation (B.3) holds. We will argue that in each of the steps in which an operation is added to S', the operation exists in G_{rid^*}. Operations are added to S' in steps 1, 6(b)i, 6(b)ii and 6c. Steps 1 and 6(b)ii add $(rid^*, 0)$ and (rid^*, ∞) respectively to S' and each of these operations appears in G_{rid^*}. Because the forward direction of relation (B.3) holds, each *hid* that is added to *active* has an entry in $A.opcounts$. Moreover, steps 6(b)i and 6c are only executed for $hid \in active$ and, consequently, these steps add operations (rid^*, hid, i) s.t. $A.opcounts[(rid^*, hid)] \neq null$ and $i = 0, \ldots, A.opcounts[rid^*][hid], \infty$. Each of these operations exists in G.

Now we show that relation (B.3) holds. The forward direction holds because an *hid* is added to *active* if it is a request handler or it is in $activatedHandlers[rid^*, hid', i]$ for some hid'. In the former case there is an entry in $A.opcounts[rid^*]$ because the server is well-behaved and in the latter

there is an entry in $A.opcounts[rid^*]$ because all entries in *activatedHandlers* are in $A.opcounts$ as argued in the proof of lemma 6.2. For the backwards direction of (B.3), notice that if *hid* is in $A.opcounts[rid^*]$, then because the server faithfully sets the contents of $A.opcounts[rid^*]$, *hid* is activated for rid^* during online execution. If *hid* is a request handler then it is added to *active* at the beginning of the process. Otherwise let $op_1, \ldots, op_n$ be the sequence of emit operations that led to the activation of *hid* during online execution. Because the server correctly logs the handler operations to reflect what happened during online execution, $activatedHandlers[op_i]$ for each i will contain the handler that emits op_{i+1}. Moreover, op_1 is issued by a request handler. Under these conditions, the above process will add all handlers that execute $op_1, \ldots, op_n$ to *active*, will examine each operation, and when it encounters op_n it will add *hid* to *active*, as required.

Now, we show that S' respects program order (Definition 10). This holds by construction: for each request r, $(r,0)$ appears before any other operation of r in S', all operations of each handler are added in ascending order of opnum and (r,∞) is added right after the last operation of the handler that emits the response prior to emitting the response according to $A.responseEmittedBy[r]$.

Last, we show that S' respects activation order (Definition 10) as follows: Consider a request r of a control flow C where rid^* is the request that "drives" the construction of S' as above. Let an activation edge $\langle (r,hid,i),(r,hid',0)\rangle$. We will show that (r,hid,i) appears before $(r,hid',0)$ in S'. The existence of this activation edge implies that $activatedHandlers[(r,hid,i)] = hid'$. Because an honest server puts in the same group requests that activate the same handlers from corresponding operations and, thus, they have the same entries in *activatedHandlers* we infer that $activatedHandlers[(rid^*,hid,i)] = hid'$. This implies that hid' is added to *active* after (rid^*,hid,i) and (r,hid,i) are added to S'. Because the first operation of hid' is added to S' after *hid* is added in *active*, we conclude that $(r,hid',0)$ appears after (r,hid,i) in S' as required.

Now we need show that OOOAudit(Tr,A,S') and Audit(Tr,A) are equivalent. The two executions are the same except for the following differences between OOOExec and ReExec. These differences are superficial in terms of affecting the program state of execution and the output:

1. ReExec checks that the number of operations that each handler issued matches the purported number of operations in the advice. OOOExec has no such explicit check but it does have an (rid, hid, ∞) case. An argument similar to the one in case (i) of Theorem 10 of Orochi [146] implies that the difference is superficial.
2. OOOExec executes the requests in a Round-Robin fashion whereas ReExec does SIMD-style execution. An argument similar to the one in case (ii) of Theorem 10 of Orochi [146] implies that the difference is superficial.
3. ReExec checks that the execution of requests does not diverge inside each handler. An argument similar to the one in case (iii) of Theorem 10 of Orochi [146] implies that the difference is superficial.
4. When OOOExec starts executing a handler, it checks that it is in *active*. ReExec does not do this check. The difference is superficial because due to the fact that the server is well-behaved, the check always passes during OOOExec.
5. ReExec checks that when a group makes an emit operation, all requests in the group activate the same handlers. This difference does not affect the execution because when the server is honest all requests in the same group activate the same handlers from corresponding emit operations which means that requests in the same group have the same entries in *activatedHandlers* and, thus, ReExec's check passes.
6. ReExec keeps track of the number of ops that a handler has executed so far in *idx*. OOOExec uses the *i* field in the op schedule entry as the number of ops that the handler has issued so far. The difference is superficial: $i = idx$ at all times because *idx* and *i* are both the running counter of operations that the handler has executed so far.
7. When a group sends back a response, ReExec checks that the contents of $A.responseEmittedBy$ match re-execution. In OOOExec there is no such check, but there is a (rid, ∞) case. This difference is superficial: both executions reject if the contents of $A.responseEmittedBy$ do not

match the ones produced during re-execution and reject otherwise.

8. ReExec lets the runtime pick the next handler to execute at line 16 of Figure B.6, whereas OOOExec picks itself the next handler to execute from S'. Observe that from the logic of ReExec and the way S is constructed, in both cases, the handler that is executed is a handler whose id is in *active*. Moreover, the two executions pick the same handler to execute under the condition that the activated handlers under ReExec are exactly the handlers that are in I when the operation is added to S' (during the construction of S'). We now show that this condition holds: Initially I contains exactly the request handlers of the request which are exactly the handlers that the request activates under ReExec. Now, we show that the handlers that each emit operation activates during ReExec are exactly the handlers that are added in I: The handlers that are activated by each emit operation under ReExec are exactly the handlers registered for the event by the request and the global handlers registered for the event. Meanwhile, from the logic of the algorithm that we use to construct S', and the semantics of handler operations, when each operation is executed by ReExec, the set of handlers that are registered by the request contains exactly the entries of R when the operation is added to S'. Thus, the handlers activated by each emit operation are the ones registered for the emitted event in *GlobalHandlers* and the ones in R when the operation is added to S'. These are exactly the handlers that are added in I when the emit operation is added to S', as required.

□

Composing Lemmas 6 and 7, we have proved:

Theorem 1 (Audit Completeness)**.** If the executor executes the given program (under the concurrency model given in Section 4.2 of the paper) and the given advice collection procedure, producing trace Tr and advice A, then Audit(Tr, A) accepts. □

B.3.2 SOUNDNESS

In the following we assume no external state operations. To show that Definition 6 is satisfied, we will show that whenever the verifier accepts an input trace Tr and advice A, there exists a well formed op schedule (with respect to Tr and A) that causes OOOAudit to accept (Lemma 9) which in turn implies the existence of a request schedule RS s.t. Tr $\in O_{RS}$ (Lemma 8).

Lemma 8 (OOOAudit Soundness)**.** Given trace Tr and advice A, if there exists a well-formed op schedule S for which OOOAudit(Tr, A, S) accepts then there exists a request schedule RS s.t. Tr $\in O_{RS}$.

Proof. If OOOAudit(Tr, A, S) accepts, then there are no cycles in graph G. We consider an op schedule S' that is a topological sort of G in which the order of $(rid, 0)$ and (rid, ∞) events matches Tr. We show that such an op schedule exists (Lemma 8.1). S' is well-formed (which follows from the remark after Definition 10). Thus, by Lemma 5, OOOAudit(Tr, A, S') accepts. Then, we define an execution ActualHandlerOps as in Figure B.11. ActualHandlerOps is the same as OOOExec of Figure B.10 but:

1. It does fewer checks
2. It constructs a trace Tr$'$ while it is executing and outputs it
3. It executes handler operations instead of simulating them
4. It skips all annotations. This means that all operations on variables observe the most recently-written value.

Then, we prove that if OOOAudit(Tr, A, S') accepts, then ActualHandlerOps(Tr, A, S') outputs Tr (Lemma 8.2).

Subsequently, we define an execution Actual as in Figure B.12 that is the same as ActualHandlerOps of Figure B.11 except that it executes external state operations against a database instead of simulating them by reading from the $A.TXL$s. Because the execution at the database is non

```
1: //Global Variables are the ones in Figure B.2

2: procedure ActualHandlerOps(op schedule S)
3:     Preprocess()
4:     return ActualHandlerOpsExec(S)
5:
6: procedure ActualHandlerOpsExec(op schedule S)
7:     Tr' ← []
8:     for each op in S do
9:         if op = (rid, 0) then
10:            Read inputs in of the request from Tr
11:            Allocate program structures
12:            Tr'.append((REQ, rid, inputs))
13:            Find the functionIDs of the request handlers
14:            for all functionID in functionIDs do
15:                Let hid ← (functionID, null, 0)
16:                Name the instance of the handler hid
17:        else if op = (rid, ∞) then
18:            Let hid ← A.responseEmittedBy[rid].hid
19:            Run the handler (rid, hid) up to and including the next event
20:            if it is not a send response operation then REJECT
21:            Tr'.append((RESP, rid, outputs))
22:        else if op = (rid, hid, i) then
23:            if i = 0 then
24:                if (hid is not an activated handler) then REJECT
25:                Read in the handler's inputs and allocate structures for running the handler
26:            else if i = ∞ then
27:                Run the handler (rid, hid) until the next event
28:                if it is not a handler exit operation then REJECT
29:            else
30:                Run the handler (rid, hid) until the next event
31:                if the next event is an external state operation then
32:                    optype ← the type of state operation
33:                    opcontents, tid, txnum ← parameters from execution
34:                    s ← CheckStateOp(rid, hid, i, optype, tid, txnum, key, opcontents)
35:                    if optype = GET then
36:                        state op result ← s
37:                else if the next event is an annotated operation then
38:                    if it is a write or initialization then
39:                        Execute the operation and skip the annotation
40:                    else
41:                        Return the current value of the variable
42:                else if the next event is a handler operation then
43:                    Execute the handler operation
44:                    if the operation is an emit operation then
45:                        for all functions that the operation activates do
46:                            hid' ← (functionID, hid, i)
47:                            Name the instance of the handler that is activated hid'
48:
    return Tr'
```

Figure B.11: Pseudocode for ActualHandlerOps

deterministic, each GET operation that Actual issues may return more than one outputs meaning that Actual has many possible output traces.

Then, we show that if ActualHandlerOps(Tr, A, S') outputs Tr then Tr is a possible output of

```
1: //Global Variables are the ones in Figure B.2

2: procedure Actual(op schedule S)
3:     Preprocess()
4:     return ActualExecS)
5:
6: procedure ActualExec(op schedule S)
7:     Tr′ ← []
8:     for each op in S do
9:         if op = (rid, 0) then
10:            Read inputs in of the request from Tr
11:            Allocate program structures
12:            Tr′.append((REQ, rid, inputs))
13:            Find the functionIDs of the request handlers
14:            for all functionID in functionIDs do
15:                Let hid ← (functionID, null, 0)
16:                Name the instance of the handler hid
17:        else if op = (rid, ∞) then
18:            Let hid ← A.responseEmittedBy[rid].hid
19:            Run the handler (rid, hid) up to and including the next event
20:            if it is not a send response operation then REJECT
21:            Tr′.append((RESP, rid, outputs))
22:        else if op ← (rid, hid, i) then
23:            if i = 0 then
24:                if (hid is not an activated handler) then REJECT
25:                Read in the handler's inputs and allocate structures for running the handler
26:            else if i = ∞ then
27:                Run the handler (rid, hid) until the next event
28:                if it is not a handler exit operation then REJECT
29:            else
30:                Run the handler (rid, hid) until the next event
31:                if the next event is an external state operation then
32:                    Execute the state operation against the database
33:                else if the next event is an annotated operation then
34:                    if it is a write or initialization then
35:                        Execute the operation and skip the annotation
36:                    else
37:                        Return the current value of the variable
38:                else if the next event is a handler operation then
39:                    Execute the handler operation
40:                    if the operation is an emit operation then
41:                        for all functions that the operation activates do
42:                            hid′ ← (functionID, hid, i)
43:                            Name the instance of the handler that is activated hid′
44:
       return Tr′
```

Figure B.12: Pseudocode for Actual

Actual(Tr, A, S') (Lemma 8.3).

Last, we show that if one of the possible outputs of Actual(Tr, A, S') is Tr, then Tr $\in O_{RS}$, where RS is the request schedule derived from S' by discarding the handler id and opnum components (Lemma 8.4).

```
1:  procedure ConstructS(graph G)
2:     Initialize S' to empty, a set frontier to the set of all in-degree 0 nodes of G , and set i = 0;
3:     while G is not empty do
4:        while there exists a node v in frontier which is not request/response do
5:           ProcessFrontier(v, G, S', frontier);
6:        Let u be the node that corresponds to Tr[i] in frontier. If u is not in frontier then REJECT
7:        ProcessFrontier(u, G, S', frontier)
8:        i ← i + 1
9:
10: procedure ProcessFrontier(Graph G, Node v, op Schedule S', frontier)
11:    Remove v from frontier and from G. Also remove the outgoing edges of v from G
12:    Append v to S'
13:    Add all nodes of G that have in-degree 0 to frontier
```

Figure B.13: Algorithm for creating S'

Sub-lemma 8.1. If G is acyclic, then there exists a topological sort S' of G in which the order of $(rid, 0)$ and (rid, ∞) events matches Tr.

Proof. In the following we will move between request/response nodes in G (that are also the entries of the op schedule) and request/response events in Tr. We will say that the node of G that corresponds to a request event $(\text{REQ}, rid, \cdot)$ (resp., response event $(\text{RESP}, rid, \cdot)$) in the trace Tr, is the node $(rid, 0)$ (resp., (rid, ∞)) of G and vice versa. We sometimes abuse notation by writing that $(rid, 0)$ or (rid, ∞) is in the trace instead of specifying that we are referring to the entries that correspond to these nodes.

We create an ordered list S' as in Figure B.13.

If the procedure ConstructS of Figure B.13 does not reject, the constructed S' is a topological sort of G with the required property: It is a topological sort because a node v is not added to S' until after all nodes that have a path to v have been removed from G and added to S'. Moreover, from construction, nodes that correspond to request/response events are always added in the order that they appear in Tr.

We now prove that ConstructS of Figure B.13 does not reject. Assume that it does reject. This can happen only if all nodes in *frontier* correspond to request/response events (that is, items in Tr) and none of them is the node u that corresponds to $\text{Tr}[i]$. *Claim:* There exists a request or response node v such that v appears in Tr after u yet v has a directed path in to u in G. We now justify this

Claim. Denote as G_i the graph G at line 3 of Figure B.13 is executed for the i-th iteration. Because u is in G_i but not in *frontier*, u has in-degree larger than 0 in G_i. Because G_i is acyclic (being a subgraph of G), there exists a path in G_i, and hence also in G, to node u from some node v that has in-degree 0 in G_i. By inspection of the algorithm, v is in *frontier*. Because all nodes in *frontier* are request or response nodes, there exists a j s.t. $\mathrm{Tr}[j]$ corresponds to v. Meanwhile, for $j < i$, all nodes that correspond to $\mathrm{Tr}[j]$ are not in G_i (again by inspection of the algorithm). Thus, $j > i$, which implies that the node v appears in Tr after u.

In the following, we use $v_1 \overset{G}{\rightsquigarrow} v_2$ to denote that there is a directed path from v_1 to v_2 in G and $rid_1 <_{\mathrm{Tr}} rid_2$ to denote that $(\mathrm{RESP}, rid_1, \cdot)$ appears before $(\mathrm{REQ}, rid_2, \cdot)$ in the trace Tr. Similar arguments as in the proof of Lemma 2 of Orochi [146] imply that

$$(rid_1, \infty) \overset{G}{\rightsquigarrow} (rid_2, 0) \iff rid_1 <_{\mathrm{Tr}} rid_2 \tag{B.4}$$

Now we use this observation to analyze cases:

1. $u = (rid_1, \infty), v = (rid_2, 0)$: Because u precedes v in Tr, $rid_1 <_{\mathrm{Tr}} rid_2$. The right-to-left direction of relation (B.4) implies that there exists a path from u to v in G. Consequently, G has a cycle, which is a contradiction.
2. $u = (rid_1, \infty)$, $v = (rid_2, \infty)$. From the construction of G, all outgoing edges of v are to nodes $(rid_3, 0)$ s.t. $rid_2 <_{\mathrm{Tr}} rid_3$. Since there exists a path from v to u, there exists a path from some node $v' = (rid_3, 0)$ to u. On the other side,
   (a) $u = (rid_1, \infty)$ appears before $v = (rid_2, 0)$ in Tr,
   (b) Because a request always appears before its corresponding response, $(rid_2, 0)$ appears before (rid_2, ∞) in Tr, and
   (c) Since $rid_2 <_{\mathrm{Tr}} rid_3$, (rid_2, ∞) appears before $(rid_3, 0)$ in Tr.

   These imply that (rid_1, ∞) appears before $(rid_3, 0)$ in Tr and consequently $rid_1 <_{\mathrm{Tr}} rid_3$ and, thus, from the right-to-left direction of relation (B.4), there is a path in G from u to v'. Consequently, G has a cycle, which is again a contradiction.

3. $u = (rid_1, 0)$, $v = (rid_2, \infty)$. Since there is a path from v to u in G, the left-to-right direction of relation (B.4) implies that $rid_2 <_{\text{Tr}} rid_1$. This implies that v appears before u in Tr, which is a contradiction.

4. $u = (rid_1, 0)$, $v = (rid_2, 0)$. From the construction of G, the only incoming edges to u are from nodes (rid_3, ∞) that appear before u in Tr. Thus, $v \overset{G}{\rightsquigarrow} v'$ for some $v' = (rid_3, \infty)$. Meanwhile, v' appears before v in Tr (because v' appears before u and u appears before v), so $v' \overset{G}{\rightsquigarrow} v$, hence a cycle exists between v and v', impossible.

□

.

Sub-lemma 8.2. If OOOAudit(Tr, A, S') accepts, ActualHandlerOps(Tr, A, S') outputs the trace Tr.

Proof. First, we show that the two runs have the same program state after each schedule step by inducting over the sequence S'. Specifically, we show that the executions after processing each operation (that is, OOOAudit at line 50 and ActualHandlerOps at line 48) preserve the following invariants:

1. they have the same program state (program state does not include the list of registered handlers, the list of activated handlers, or the set of emitted events).
2. the set of handler ids in *active* under OOOExec is exactly the set of handler ids that are activated under ActualHandlerOps.

Base case: Before processing any operation, the two runs have the same program state (because we assume that initialization is deterministic), *active* is empty in OOOAudit and there are no activated handlers in ActualHandlerOps. Thus, the invariants hold before processing the first operation of S'. The first operation in S' has the form $(rid, 0)$. Both executions read inputs from Tr, allocate program structures and, subsequently, perform operations that do not affect program state. Thus, since both executions start from the same program state, the two executions have the same

program state after processing *op*. Moreover, because invariant 2 holds prior to processing *op*, it holds after processing *op*: both executions compute the same handler ids for *rid*'s request handlers, which OOOAudit adds to *active* at line 16 of Figure B.10 and ActualHandlerOps uses to name the new activated handlers at line 16 of Figure B.11.

Induction step: Consider the i-th operation of S' and denote it as *op*. Assume that the invariants hold for all operations j s.t. $j < i$. We will show that for any type of *op*, after processing *op* the invariants hold:

- Case $op = (rid, 0)$: A similar argument as the one used in the base case implies that the invariants hold after processing *op*.
- Case $op = (rid, \infty)$: Let $hid = A.responseEmittedBy[rid].hid$. Since invariant 1 holds, prior to processing *op*, the two executions have executed handler (rid, hid) up until the same operation op_l and have the same program state. From the logic of OOOAudit and ActualHandlerOps, the two executions resume the execution of (rid, hid) from op_l until its next special operation. Since both executions proceed deterministically between operations, the program state of OOOAudit when it reaches line 21 of Figure B.10 is the same as the program state of ActualHandlerOps when it reaches line 20 of Figure B.11. This implies that the next operation will be the same in both executions and, thus, either both checks at the aforementioned lines pass or both fail. Because the work that the two executions perform past these checks does not affect program state, we conclude that the two executions have the same program state after *op*.

  Moreover, observe that invariant 2 holds prior to processing *op*, OOOAudit does not modify *active* as part of handling *op* and ActualHandlerOps does not modify the activated handlers as part of handling *op*. These imply that invariant 2 holds after processing *op*.
- Case $op = (rid, hid, 0)$: The two executions handle this operation in the same way except that OOOExec checks if *hid* is in *active* and ActualHandlerOps checks if *hid* is the name of some activated handler. Because from the induction hypothesis the ids of all activated handlers of

ActualHandlerOps are exactly the handler ids in *active*, either both checks pass or both fail. Thus, the two executions reach the same program state after processing *op*. Moreover, invariant 2 holds prior to processing *op* and while processing *op* neither the activated handlers are modified by ActualHandlerOps nor *active* is modified by OOOAudit. Thus, invariant 2 holds after processing *op*.

- Case $op = (rid, hid, \infty)$: The two executions start from the same program state, pick the same handler, run it until the next event and check that it is a handler exit event. Thus, the two executions result in the same program state. Moreover, upon reaching the handler exit event, ActualHandlerOps truly executes the handler exit operation, and this operation removes the handler with id *hid* from the activated handlers. On the other side, OOOAudit removes *hid* from *active* at line 32. This implies that the invariant holds after processing *op*.
- Case $op = (rid, hid, i)$ where *op* is an external state operation: The result follows from the induction hypothesis, the fact that execution proceeds deterministically between operations and the fact that both executions handle external state operations in the same way.
- Case $op = (rid, hid, i)$ where *op* is a handler operation. From the induction hypothesis and the fact that execution proceeds deterministically between operations we conclude that the two executions have the same program state right before they process *op*. The processing of *op* in ActualHandlerOps and OOOAudit does not affect program state (which, recall, excludes the set of registered handlers and emitted events). Thus, invariant 1 holds after processing *op*.

  Now we argue that invariant 2 holds. From the induction hypothesis the invariant holds before processing *op*. If *op* is not an emit operation, ActualHandlerOps does not modify the activated handlers while processing and OOOAudit does not modify *active*. Thus, invariant 2 holds after processing *op*. On the other hand, assume *op* is an emit operation. We will argue that the handler ids that are in $activatedHandlers[(rid, hid, i)]$ (which are the ones added to *active* by OOOAudit at line 49) are exactly the ones activated in ActualHandlerOps at lines 45–47 of Figure B.11. Let

eventName be the event that *op* emits. Let C be the set of function ids c s.t. $(\textit{eventName}, c) \in \textit{Registered} \cup \textit{GlobalHandlers}$ when *op* is processed by AddHandlerRelatedEdges and C' the set of function ids that *op* activates during ActualHandlerOps. In the following we will sometime abuse notation and refer to the handler's function as handler.

Claim: $C = C'$. Denote C_g the set of function ids c s.ts. $(\textit{eventName}, c) \in \textit{GlobalHandlers}$ and C_r the set of function ids c s.t. $(\textit{eventName}, c) \in \textit{Registered}$. Obviously,

$$C = C_g \cup C_r.$$

Moreover, because each function that *op* activates during ActualHandlerOps is either a global handler or a function registered for *eventName* by *rid*,

$$C' = C'_g \cup C'_r,$$

where C'_g is the set that contains the ids of all global handlers that are registered for event *eventName*, and C'_r is the set that contains the ids of all functions that are registered for event *eventName* over the course of *rid*. To establish the Claim, we show that $C_g = C'_g$ and $C_r = C'_r$:

1. $C_g = C'_g$: First, observe that C_g is exactly the ids of the functions registered for *eventName* over the course of the initialization procedure of OOOExec. Because the initialization procedure is deterministic, it registers the same functions for *eventName* under both OOOExec and ActualHandlerOps. Thus, C_g is exactly the ids of the functions registered for *eventName* over the course of the initialization procedure of ActualHandlerOps. Because requests don't modify global handlers, $C_g = C'_g$.
2. $C_r = C'_r$: C'_r contains the ids of the functions that are registered by *rid* for *eventName* at the time when *op* is executed. Because ActualHandlerOps follows S', these are exactly the ids of the functions H s.t.
   (a) There exists an operation op_r that registers H for *eventName* and appears before *op* in S', and

(b) For all operations op' between op_r and op in S', $op'.rid \neq rid$ or op' does not unregister H from *eventName*.

Meanwhile, the induction hypothesis and the fact that execution proceeds deterministically between operations imply that OOOExec and ActualHandlerOps have the same program state right before executing every register and unregister operation that precedes op. This implies that the parameters of each register or unregister operation op' (these are the *functionID* and *eventNames* for register operations, and *functionID* and *eventName* for unregister operations) are the same under ActualHandlerOps and under OOOExec. Moreover, OOOExec checks these parameters against the corresponding entry in $A.HL_{rid}$ (line 23 of Figure B.7). The above implies that C'_r contains exactly the function ids c s.t.:

(a) There exists a register operation op_r with parameters c and *eventNames* in $A.HL_{rid}$ that appears before op in S' and for which $eventName \in eventNames$, and

(b) For all operations op' between op_r and op in S', either $op'.rid \neq rid$ or the entry in $A.HL_{rid}$ that corresponds to op' is not an unregister operation with parameters c and *eventName*.

Meanwhile, because S' is a topological sort of the graph G and G has edges between consecutive handler operations in $A.HL_{rid}$ (line 15 of Figure B.4), the order of the handler ops of *rid* in S' matches their order in $A.HL_{rid}$. Thus, we conclude that C'_r contains exactly the function ids c s.t.

(a) There exists a register operation op_r with parameters c and *eventNames* in $A.HL_{rid}$ that appears before op in $A.HL_{rid}$ and for which $eventName \in eventNames$, and

(b) For all operations op' between op_r and op in $A.HL_{rid}$, op' is not an unregister operation with parameters c and *eventName*.

From the logic of AddHandlerRelatedEdges these function ids are exactly the ones in C_r, as required.

Let

$$C_{id} = \{(functionID, op.hid, op.i) \mid functionID \in C\}.$$

By definition of C, C_{id} is exactly the set of handler ids that AddHandlerRelatedEdges places in $activatedHandlers[(rid, hid, i)]$ at line 26 of Figure B.4, and, because $C = C'$, also exactly the handler ids that ActualHandlerOps uses to name the handlers that op activates at line 47 of Figure B.11. So, these two sets are equal, as required.

- Case $op = (rid, hid, i)$ where op is an annotated operation. Since in this case the activated handlers under ActualHandlerOps and *active* under OOOAudit are not modified, if invariant 2 holds prior to this step, it holds after this step.

  Now, we argue that invariant 1 holds. As argued in some of the previous cases, the induction hypothesis and the determinism of execution between operations implies that the program state right before op is processed is the same across executions. If the annotated operation is either a write or initialization, then both executions execute the operation, which results in the same program state. Then, OOOAudit executes the annotation, which ActualHandlerOps skips. However, the annotation does not affect program state on OOOAudit and, consequently, the two executions have the same program state after executing the annotation, as required. Now, we argue that they have the same program state when op is a read. In this case, ActualHandlerOps reads the current value of the variable whereas OOOAudit reads the value returned by the OnRead function of Figure B.8. We argue that the value of the variable under ActualHandlerOps (which is the most recent value written) is the value returned from the OnRead annotation in OOOAudit. Let op' the write operation that op reads from in OOOAudit and v the variable that these operations access. We will show that op' is the most recent write operation to v prior to op in S'. From the logic of OnRead, $op \in v.read\_observers\{op'\}$. This implies that there exists a read edge $\langle op', op \rangle$ in G. Moreover, because G contains anti-depend edges and write-depend edges, for any other write op'' to v either there exists a path from op'' to op' consisting of write-depend edges or there exists

a path from op to op'' in which the first edge is an anti-depend edge and the rest are write-depend edges. Thus, because S' is a topological sort of G, the last write op to v prior to op in S' is op'. Because ActualHandlerOps follows S', this implies that op' is the most recent write to v prior to op and, thus, the value of v under ActualHandlerOps is the value written by op' as requested.

Since every step preserves program state in the two runs and OOOExec does not reject, ActualHandlerOps also does not reject and thus returns a trace Tr$'$.

Now, we show that Tr$'$ is a permutation of Tr. First, we argue that Tr and Tr$'$ contain entries for the same request ids: This follows from (1) the fact that G's $(rid, 0)$ and (rid, ∞) nodes are exactly those for which $rid \in$ Tr (this follows from the logic of CreateTimePrecedenceGraph and SplitNodes) (2) the fact that S' is a topological sort of G and (3) that Tr$'$ has exactly one request entry for each $(rid, 0)$ node in S' and one response entry for each (rid, ∞) node in S'. Moreover, the request contents of each request in Tr$'$ are those in Tr because of the logic of lines 10 and 12 of Figure B.11. Last, because invariant 1 holds, for each rid, the program state of OOOExec at line 21 of Figure B.10 is the same as the program state of ActualHandlerOps when it reaches line 20 of Figure B.11. This implies that the response contents for rid that ActualHandlerOps writes in Tr$'$ are those that OOOExec checks against Tr at line 52 of Figure B.10.

Last, from the construction of S' (lemma 8.1, Figure B.13), the order of $(rid, 0)$ and (rid, ∞) operations in S' corresponds to their order in Tr. Moreover, the order of the operations in Tr$'$ matches their order in S'. Consequently the order of operations in Tr$'$ matches their order in Tr, and Tr$' =$ Tr as required. □

Sub-lemma 8.3. If ActualHandlerOps(Tr, A, S') outputs the trace Tr, then Tr is a possible output of Actual(Tr, A, S').

Proof. ActualHandlerOps(S') and Actual(S') are the same except that Actual does not simulate external state operations but, instead, it executes them against a database that exhibits the required isolation level. Observe that the execution of the program under Actual is identical to the execution

of the program under ActualHandlerOps under the condition that each GET reads the same value under Actual and under ActualHandlerOps. Thus, if this condition is satisfied, then Actual outputs Tr, as required. Furthermore, observe that the executions at the database in which the dictating write of each GET is the one in $A.TXL$s satisfy this condition. We pick one of these executions and show that it is is a legal database execution (meaning that its history obeys all rules of definition 11) and that it is consistent with the required isolation level (Section C). To fully specify this execution we need to first specify what is the version order of this execution. Second, we need to specify what happens when the server issues a tx_commit operation: upon a tx_commit operation, the database can either execute the tx_commit or, if the transaction cannot commit, the database can instead abort the transaction. We pick the execution whose version order (definition 11) is consistent with $A.writeOrder$ and whose execution of tx_commit operations is consistent with the $A.TXL$s.

Consider the TxOp order E in the above execution: First, because Actual follows S', this TxOp order is consistent with the order of external state operations in S'. Moreover, the contents of the entries in E are consistent with the $A.TXL$s: For tx_commit operations, this follows from the definition of the execution. For the rest of the parameters, this follows from the fact that the parameters of the operations under Actual and under ActualHandlerOps are the same, and ActualHandlerOps checks that these parameters match the ones in $A.TXL$s. Formally, if the i-th external state operation of S is op, then for the i-th entry of E it holds:

- if op.optype $\in \{$tx_start, tx_commit, tx_abort$\}$, it is $(op.rid, op.\text{tid}, op.\text{optype})$,
- if op.optype $=$ PUT, it is $(op.rid, op.\text{tid}, \text{PUT}, op.key, m, op.\text{opcontents})$, where m is the order of op among all PUT operations in $A.TXL_{op.rid}op.\text{tid}$,
- if op.optype $=$ GET, it is $(op.rid, op.\text{tid}, \text{GET}, op.key, op_w.rid, op_w.\text{tid}, m)$, where m_w is the order of op_w among all PUT operations in $A.TXL_{(op_w.rid, op_w.\text{tid})}$ and $op_w = op.\text{opcontents}$.

Moreover, the version order is the sequence of operations V s.t. $V[i] = (op.rid, op.tid, m)$ where $op = A.writeOrder[i]$, and m is the order of op among all PUT operations in $A.TXL_{op.rid}op.\text{tid}$.

We now show that the history $H = (E, V)$ satisfies all the constraints of definition 11:

1. Constraint 1a: First, because CheckStateOp at ActualHandlerOps does not reject when called for external state operations and ActualHandlerOps follows S', the order of the operations of a transaction t in S' is consistent with their order in $A.TXL_t$. This implies that S' preserves the order of all operations within the transaction. Because E is consistent with S', so does E, as required.

2. Constraint 1b: Because S' is a topological sort of G and G contains read-from edges (line 46 of Figure B.4), for each `GET` operation op in S', $op_w = op$.opcontents, precedes op in S'. Because the order of operations in E is consistent with their order in S', op_w precedes op in E, as required. Furthermore, because the check at line 48 of Figure B.4 passes, $op_w.key = op.key$, and op_w.optype = `PUT`, as required.

3. Constraint 1c: From the logic of AddExternalStateEdges (Figure B.4), when the i-th operation op of $A.TXL_t$ is examined, *MyWrites* has an entry for each *key* for which there exists at least one `PUT` operation op' prior to op in $A.TXL_t$ with $op'.key = key$. Moreover, *MyWrites* maps each such *key* to the latest `PUT` operation that modifies it according to $A.TXL_t$. Because ActualHandlerOps passes, the check at line 51 of Figure B.4 passes which implies that for each $op \in A.TXL$s: if op.optype = `GET` and $op.key \in MyWrites$ then op.opcontents = $MyWrites[op.key]$. Thus, the dictating write of each operation op of transaction t that reads a key that has been previously modified by t according to $A.TXL_t$, is the operation op' that last writes this key according to $A.TXL_t$. Meanwhile, because CheckStateOp does not reject when called for external state operations and ActualHandlerOps follows S', the order of the operations of a transaction t in S' is consistent with their order in $A.TXL_t$. Thus, the dictating write of each operation op of transaction t that reads a key that has been previously modified by t according to S', is the last `PUT` operation op' issued by t that modifies *key* according to S'. Furthermore, from the definition of E, the order of operations is consistent with S' and the dictating write of each operation is consistent with the $A.TXL$s. Thus, E is internally consistent, as required.

4. The version order V is a list of unique tuples (rid, tid, m) s.t. $(rid, tid, m) \in V$ iff (a) $(rid, tid,$ PUT$, key, m, v)$ in E, (b) there exists no $(rid, tid,$ PUT$, key, m', \cdot)$ in E with $m' > m$, and (c) $(rid, tid,$ tx_commit$)$ in E: First, observe that V is consistent with $A.writeOrder$. Furthermore, observe that because the checks at lines 23 and 27 of Figure B.5 pass, the entries in $A.writeOrder$ are exactly the entries (rid, tid, m) for which there exists some key such that $lastModification[rid, tid, key] = m$. Meanwhile, from the logic of AddExternalStateEdges, the entries in $lastModification$ are exactly the (rid, tid, key) s.t. transaction (rid, tid) modifies key according to $A.TXL_{(rid,tid)}$ and $(rid, tid) \in Committed$. $lastModification$ maps each such entry (rid, tid, key) to the index of the last operation that writes key in $A.TXL_{(rid,tid)}$. Furthermore, from the logic of AddExternalStateEdges, $(rid, tid) \in Committed$ iff it issues a tx_commit operation according to $A.TXL_{(rid,tid)}$. Thus, the entries of $lastModification$ (which correspond to the entries in the version order) correspond to exactly the PUT operations op s.t. (a) there exists a transaction (rid, tid) s.t. $op \in A.TXL_{(rid,tid)}$, (b) there exists no PUT operation op' to $op.key$ that appears after op in $A.TXL_{(rid,tid)}$, and (c) there exists a tx_commit operation in $A.TXL_t$. Meanwhile, as argued above, the operations in E correspond to exactly the operations in $A.TXL$s, and the order of operations in E is consistent with the order of the corresponding operations in $A.TXL$s. Thus we conclude that the entries in the version order V are exactly the operations (rid, tid, m) s.t. (a) (rid, tid, m) appears in E (b) there exists no $(rid, tid,$ PUT$, key, m', \cdot)$ in E with $m' > m$, and (c) $(rid, tid,$ tx_commit$)$ in E, as required.

We now need to show that $H = (E, V)$ exhibits the required isolation level.

First, observe that:

1. when the isolation level is READ COMMITTED or SERIALIZABILITY, H does not exhibit phenomena G1a and G1b: Phenomena G1a and G1b require that each read of a committed transaction in E should read from an operation in V. As argued above, the entries of E correspond to the entries of $A.TXL$s and the operations in V correspond exactly to the entries in $lastModification$. Thus, we need to show that each GET operation in $A.TXL_t$ s.t. t's last

operation is `tx_commit`, reads from an entry in *lastModification*. This is exactly the check that IsolationLvlVer performs in the case of READ COMMITTED or SERIALIZABILITY in line 27 of Figure B.5.

2. $DSG(H)$ and DG have the same nodes: $DSG(H)$ contains exactly the transactions that commit according to H. Meanwhile, from the construction of H, these transactions are exactly the transactions t s.t. there exists a `tx_commit` operation in $A.TXL_t$. From the logic of AddExternalStateEdges these are exactly the transactions in *Committed* which IsolationLvlVer adds to G. Thus, $DSG(H)$ and DG have the same nodes as required.

3. the edges that AddWriteDependencyEdges adds to DG are exactly the write depend edges of $DSG(H)$: the write depend edges of $DSG(H)$ are exactly the edges between T_1 and T_2 s.t. T_1 installs a version of some key and T_2 installs the next version according to V. On the other side, from the logic of ExtractWriteOrderPerKey and AddWriteDependencyEdges, the white dependency edges of DG are exactly the edges $\langle T_1, T_2 \rangle$ s.t. T_1 installs a version of some key and T_2 installs the next version according to $A.writeOrder$. Because V exactly matches the $A.writeOrder$, we conclude that the write depend edges of $DSG(H)$ are exactly the write dependency edges of DG.

4. the edges that AddReadDependencyEdges adds to DG are exactly the read depend edges of $DSG(H)$ when the isolation level is READ COMMITTED or SERIALIZABILITY: Observe that for these isolation levels, because the history does not exhibit phenomena G1a and G1b, the dictating write of each `GET` of a committed transaction is an operation in V. This implies that the read depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2 \rangle$ for which there exist operations $op_1 \in T_1$ and $op_2 \in T_2$ s.t. op_2 reads from op_1 according to E, $op_1 \in V$ and T_2 commits according to E. Meanwhile, V matches $A.writeOrder$ and the dictating writes of operations in E match $A.TXL$s from construction. Moreover, the committed transactions according to E are exactly those in *Committed*: from the logic of AddExternalStateEdges, *Committed* contains

exactly the committed transactions according to $A.TXL$s and, from the definition of the execution above, these are exactly the transactions that commit according to E. Thus, the read depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2 \rangle$ for which there exist $op_1 \in A.TXL_{T_1}$ and $op_2 \in A.TXL_{T_2}$ s.t. op_2.opcontents $= op_1$, $op_1 \in A.writeOrder$ and $T_2 \in Committed$. From the logic of AddExternalStateEdges and AddReadDependencyEdges, these are exactly the read dependency edges of DG as required.

5. the edges that AddAntiDependencyEdges adds to DG are exactly the anti depend edges of $DSG(H)$: The anti depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2 \rangle$ for which there exists a transaction T_3 and operations $op_1 \in T_1$, $op_2 \in T_2$, $op_3 \in T_3$ s.t. the dictating write of op_1 is op_3 according to E, T_1 commits according to E, and op_3 installs a version of a key and op_2 installs the next version according to V. Meanwhile V exactly matches $A.writeOrder$, and the dictating writes of `GET` operations in E exactly match their dictating writes according to $A.TXL$s. Last, as argued above $Committed$ contains exactly the committed transactions according to E. Thus we conclude that the anti depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2 \rangle$ for which there exists a transaction T_3 and operations $op_1 \in A.TXL_{T_1}$, $op_2 \in A.TXL_{T_2}$, $op_3 \in A.TXL_{T_3}$ s.t. the dictating op_1.opcontents $= op_3$, $T_1 \in Committed$, and op_3 installs a version of a key and op_2 installs the next version according to $A.writeOrder$. These are exactly the anti dependency edges of DG as required.

When the required isolation level is READ UNCOMMITTED, H exhibits the isolation level because it does not exhibit phenomenon G0: First, the results 2 and 3 imply that the subgraph of $DSG(H)$ that contains only write depend edges is exactly DG. Moreover, DG is acyclic because the check at line 11 of Figure B.5 accepts. This implies that $DSG(H)$ does not exhibit phenomenon G0 as required.

When the required isolation level is READ COMMITTED, H exhibits the isolation level because it does not exhibit phenomenon G1: First, result 1 implies that H does not exhibit phenomena G1a and G1b. Moreover, the results 2, 3, and 4 imply that the subgraph of $DSG(H)$ that contains

only write depend and read depend edges is exactly DG. Moreover, DG is acyclic because the check at line 15 of Figure B.5 accepts. This implies that $DSG(H)$ does not exhibit phenomenon G1c, as required.

When the required isolation level is SERIALIZABILITY, H exhibits the isolation level because it does not exhibit phenomena G1 and G2: First, result 1 implies that H does not exhibit phenomena G1a and G1b. Moreover, the results 2, 3, 4, and 5 imply that $DSG(H)$ is DG. Moreover, DG is acyclic because the check at line 20 of Figure B.5 accepts. This implies that $DSG(H)$ is acyclic and, thus, does not exhibit phenomena G1c and G2, as required. □

Sub-lemma 8.4. If Tr is a possible output of Actual(Tr, A, S'), then Tr is a possible output of Operation-wise execution on input Tr by following *RS*.

Proof. Observe that Operation-wise execution is the same as Actual except for the following differences:

- Operation-wise executes the program P whereas Actual executes the annotated program P_a.
- All checks in Actual are discarded in Operation-wise.
- Operation-wise is only presented with *rid*s. The most important consequence of this is that whenever Operation-wise executes a request, it is free to pick which handler to execute from the activated handlers.

First, observe that P_a differs from P only in that it contains annotations (Section B.1.1). Actual skips all these annotations which implies that both executions effectively execute P.

Second, observe that since Actual passes all checks, eliminating these checks from Operation-wise does not affect the flow of execution.

Denote $\text{Actual}_{\text{Tr}}$ the execution of Actual that outputs Tr on input Tr and S'. $\text{Actual}_{\text{Tr}}$ captures both the execution at the server and the execution at the database. To show that Tr is a possible output of Operation-wise, we will show that there exists an execution of Operation-wise on input

Tr and *RS* that is identical to $\text{Actual}_{\text{Tr}}$. We do the proof by induction: we show that if $\text{Actual}_{\text{Tr}}$ and Operation-wise have proceeded in the same way up until the $(i-1)$ step, the next step of $\text{Actual}_{\text{Tr}}$ is a step that Operation-wise can take that will result in the two executions having the same program state and database state after step i.

Induction Base. Because initialization is deterministic, $\text{Actual}_{\text{Tr}}$ and Operation-wise have the same program state prior to executing any operation. Moreover, $\text{Actual}_{\text{Tr}}$ and Operation-wise issue the same operations to the database during initialization. Thus, there exists an execution of these operations in the database of Operation-wise that leads to the database state of $\text{Actual}_{\text{Tr}}$.

Induction Step. Assume that up until the $(i-1)$ step, the two executions have taken identical steps, they have the same program state, and the same database state. Let the i-th operation of *RS* be *rid*.

If this is the first occurrence of *rid* in *RS*, then because *RS* is constructed from S' and the first operation of *rid* in S' is $(rid, 0)$ (S' is a topological sort of G and G contains boundary edges) the ith operation of S' is $(rid, 0)$. Because $\text{Actual}_{\text{Tr}}$ handles $(rid, 0)$ operations in the same way that Operation-wise handles the first occurrence of *rid* in *RS*, the two executions will result in the same state.

Now assume that this is not the first occurrence of *rid* in *RS*. Because *RS* is constructed from the well formed S' by dropping all fields other than *rid*, the corresponding operation in S' is either of the form (rid, hid, i) or (rid, ∞). In either case, $\text{Actual}_{\text{Tr}}$ "resumes" the execution of a handler that is activated. Because the two executions are identical up to step $i-1$, the activated handlers are the same. This implies that the activated handler that $\text{Actual}_{\text{Tr}}$ picks to execute is an activated handler in Operation-wise. Thus, Operation-wise can take the same next step as $\text{Actual}_{\text{Tr}}$ executing the same handler up until its next special operation. Because execution between special operations is deterministic and does not modify the database state, both executions have the same program state and database state up until they execute the next special operation. Moreover, the handling of all special operations other than external state operations is the same in $\text{Actual}_{\text{Tr}}$ and Operation-wise

and such operations don't modify database state. Thus, both executions reach the same program state and database state after step i when the special operation at step i is not a state operation. It remains to show that $\text{Actual}_{\text{Tr}}$ and Operation-wise reach the same program state and database state after step i, when the special operation at step i is an external state operation: Because the two executions have the same program state up until executing the i-th special operation, the parameters of the state operation are the same across executions and, thus, both executions issue the same operation to the database. Because two databases that start from the same state and receive the same operation can execute this operation identically, we conclude that the i-th state operation can be executed by Operation-wise in the same way that it was executed by $\text{Actual}_{\text{Tr}}$. In this case, the database returns the same result under Operation-wise that it returns under $\text{Actual}_{\text{Tr}}$, and the two executions have the same program state and database state after executing the i-th operation as required. □

□

Lemma 9. Given trace Tr and advice A, if $\text{Audit}(\text{Tr}, A)$ accepts, then there exists a well-formed op schedule S that causes $\text{OOOAudit}(\text{Tr}, A, S)$ to accept.

Proof. Use the control flow groupings $A.C$ to construct op schedule S as follows: Initialize S to empty. Then run $\text{Audit}(\text{Tr}, A)$ and

- Every time Audit begins re-executing a control flow group t, add to S entries $(rid, 0)$ for each rid in t
- Every time Audit begins re-executing a handler hid for control flow group t, add to S entries $(rid, hid, 0)$ for all rid in t
- Every time a group t does an operation from inside a handler hid (all requests in the group issue operations together because execution does not diverge), add (rid, hid, opnum) to S for all rid in t where opnum is the running tally of operations for the handler hid

- Every time Audit finishes executing a handler *hid* for requests in group t (all requests finish executing *hid* together), add (rid, hid, ∞) to S for all *rid* in t
- Every time the requests in a group t write their outputs (all requests in the group send responses together because execution does not diverge), add (rid, ∞) to S for all *rid* in t

We now argue that S is well-formed. First, S contains exactly the nodes of G:

- It contains all nodes $(rid, 0)$ and (rid, ∞) s.t. $rid \in$ Tr otherwise the produced outputs are not exactly the outputs in the trace and ReExec rejects in line 62 of Figure B.6.
- S contains nodes $(rid, hid, 0), (rid, hid, \infty)$ for each $(rid, hid) \in A.opcounts$: Notice that S contains nodes $(rid, hid, 0)$ and (rid, hid, ∞) for each (rid, hid) that is executed by ReExec. To show that these nodes are exactly the handler start and handler end nodes of G (lines 39 and 40 of Figure B.2) we need to prove that for each *rid*, the set H of all *hid* s.t. (rid, hid) in $A.opcounts$ and the set H' of all *hid* that are executed by ReExec are equal. We show that $H' \subseteq H$ and that $H \subseteq H'$.

  $H' \subseteq H$: Notice that from the logic of ReExec, H' is exactly the *hid*s that are in *active* during the execution of the group in which *rid* belongs to. We will show that for each *hid* that is added in *active* during the execution of *rid*, $opcounts[(rid, hid)] \neq \emptyset$. *active* is initially empty and entries are added to it in line 12 of Figure B.6 and in ActivateHandlers. In the former case, ReExec rejects if (rid, hid) is not in *opcounts* (line 13 of Figure B.6). In the latter case, *hid* is added to *active* from $activatedHandlers(rid, \cdot, \cdot)$. Notice that *hid* can only be added to $activatedHandlers(rid, \cdot, \cdot)$ at line 26 of Figure B.4 after the check at line 25 of Figure B.4 passes. Thus, for any $hid \in activatedHandlers(rid, \cdot, \cdot)$, $A.opcounts[rid][hid] \neq \emptyset$ as required.

  $H \subseteq H'$: This follows from the fact that ReExec does not reject at line 64 of Figure B.6.
- For each (rid, hid) in $A.opcounts$ it contains all nodes (rid, hid, j) s.t. $j \in [0, A.opcounts[(rid, hid)]]$. This follows from the previous bullet and the fact that for each (rid, hid) the value of j in S prior to the insertion of (rid, hid, ∞) is $A.opcounts[(rid, hid)]$: ReExec does not reject at line 60 of

Figure B.6 and, thus $j \geq A.opcounts[(rid, hid)]$. Moreover, $j \leq A.opcounts[(rid, hid)]$ because otherwise ReExec rejects in line 43 of Figure B.6, in CheckStateOp or CheckHandlerOp.

Moreover S respects program order and activation order (Definition 10) because Audit executes operations according to this order.

Now, we prove that OOOAudit(Tr, A, S) accepts. OOOAudit(Tr, A, S) (Figure B.10) is the same as Audit(Tr, A) (Figure B.6) except the differences that we describe below. For each of them, we show that they do not result in different program state or OOOAudit rejecting.

1. ReExec executes the requests in SIMD style whereas OOOExec round-robins the execution from operation to operation for a group of requests. This does not affect program state; the flow and ordering is the same across both executions. Thus, the produced output is the same.
2. When ReExec executes a group, it picks the next handler to run from *active* whereas OOOExec picks the next handler to run from S. This difference is superficial because S is derived from ReExec.
3. There is a difference in how handler end events are processed. In OOOExec there is an (rid, hid, ∞) case that checks that the next event is a handler exit operation. In ReExec handler exit events are processed in case 2c where the number of operations issued by the handler is checked against $A.opcounts$ (line 60 of Figure B.6). Similar arguments to those made elsewhere (Orochi [146], lemma 8) establish that this difference is superficial.
4. When OOOExec starts executing a handler, it checks that it is in *active*. ReExec does not do this check but picks which handler to run from *active*. The difference does not result in different program state because both executions just require that when a handler starts executing, it must be in *active*.
5. ReExec keeps track of the number of ops that a handler *hid* has executed so far in $idx[hid]$. OOOExec uses the i field in the op schedule entry as the number of ops that the handler has issued so far. The difference does not result in different program state because both i

and $idx[hid]$ correspond to the running counter of operations that the handler issues and thus $i = idx[hid]$ at all times.

6. When a group sends back a response, ReExec checks that the contents of $A.responseEmittedBy$ match re-execution. In OOOExec there is no such check, but there is a (rid, ∞) case. The difference is superficial; both executions reject if $A.responseEmittedBy$ does not match re-execution.

□

C | DEFINITIONS OF ISOLATION LEVELS ACCORDING TO ADYA

In this section we briefly present Adya's definitions for consistency models [16]. We should note that we modify these definitions to make them consistent with our terminology. Additionally, we modify them to make them suitable for transactional key-value stores; for instance, we erase the parts of the definitions that refer to predicates.

In order to define the isolation levels, we need the notion of history:

Definition 11. *History:* A history H captures what happens in the execution of the system. It consists of:

1. An ordered list of operations (TxOp order E). Each such operation can be:
   - $(rid, tid, \texttt{tx\_start})$: transaction start operation for transaction $T_{rid,tid}$
   - $(rid, tid, \texttt{tx\_abort})$ (resp. $(rid, tid, \texttt{tx\_commit})$): transaction abort (resp. commit) operation for transaction $T_{rid,tid}$
   - $(rid, tid, \texttt{PUT}, key, m, v)$: The m-th `PUT` operation of transaction $T_{rid,tid}$ on key that writes value v.
   - $(rid, tid, \texttt{GET}, key, rid', \text{tid}', m)$: `GET` operation of transaction $T_{rid,tid}$ to a data item key that reads the value that was written by the m-th `PUT` operation of transaction $T_{rid',tid'}$) (that is, $key_{rid',tid',m}$)

The TxOp order must obey the following constraints:

(a) It preserves the order of all operations within a transaction including its commit and abort operations

(b) A transaction $T_{rid,tid}$ cannot read version $key_{rid',tid'}$ before it has been produced by $T_{rid',tid'}$. Formally, if an operation $(rid, tid, \texttt{GET}, key, rid', tid', m)$ exists in H, it is preceded by $(rid', tid', \texttt{PUT}, key, m, \cdots)$ in H.

(c) If a transaction modifies a key and later reads it, it will observe its last update to the key. Formally, if an operation $(rid, tid, \texttt{PUT}, key, m, \cdots)$ is followed by an operation $(rid, tid, \texttt{GET}, key, rid', tid', m')$ in H without the interleaving of an operation $(rid, tid, \texttt{PUT}, key, m'', \cdots)$, it should be $rid = rid'$ and $tid = tid'$ and $m = m'$. We call this property *internal consistency*

2. An order all key versions (version order) V created by committed transactions in E, that is, a list of unique tuples (rid, tid, m) s.t. $(rid, tid, m) \in V$ iff (a) $(rid, tid, \texttt{PUT}, key, m, v)$ in E, (b) there exists no $(rid, tid, \texttt{PUT}, key, m', \cdot)$ in E with $m' > m$, and (c $(rid, tid, \texttt{tx\_commit})$ in E.

Adya defines the following types of conflicts between different committed transactions:

- *Read Depends*: A transaction T read depends on transaction T' if T reads an object version that T' writes
- *Anti Depends*: A transaction T anti depends on transaction T' if T' reads a version of an object and T writes its next version
- *Write Depends*: A transaction T write depends on transaction T' if T' writes a version of an object and T writes its next version

Given a history H, the **Direct Serialization Graph (DSG)** arising from H is as follows: Each node in $DSG(H)$ corresponds to a committed top-level transaction in H and directed edges correspond to read, anti or write conflicts. There is a read/anti/write depend edge from the node that corresponds to T to the node that corresponds to T' if T' read/anti/write depends on T.

With the above definitions in mind we define the following phenomena:

- *Phenomenon G0 (Write Cycles).* The history H exhibits phenomenon G0 if $DSG(H)$ contains a directed cycle consisting entirely of write-depend edges.
- *Phenomenon G1a (Aborted Reads).* The history H exhibits phenomenon G1a if it contains an aborted transaction T_1 and a committed transaction T_2 s.t. T_2 has read some object modified by T_1.
- *Phenomenon G1b (Intermediate Reads).* The history H exhibits phenomenon G1b if it contains a committed transaction T_1 that has read a version of an object written by transaction T_2 that was not T_2's final modification of the object.
- *Phenomenon G1c (Circular Information Flow).* The history H exhibits phenomenon G1c if $DSG(H)$ contains a directed cycle formed *without* anti-dependency edges.
- *Phenomenon G1.* The history H exhibits phenomenon G1 if it exhibits phenomenon G1a or G1b or G1c
- *Phenomenon G2 (Anti-depend cycles).* The history H exhibits phenomenon G2 if $DSG(H)$ contains a directed formed from *at least one* anti-dependency edge. Note that G1c and G2 are separate: neither implies the other.

Now we define when a history H exhibits each of the isolation levels we support:

- *Serializability: H* does not exhibit phenomena G1 and G2.
- *Read Committed: H* does not exhibit phenomenon G1
- *Read Uncommitted: H* does not exhibit phenomenon G0

In order for an execution of a key value store to be consistent with Isolation Level I, there should exist a version order s.t. the TxOp order along with this version order define a history H that exhibits isolation level I.

BIBLIOGRAPHY

[1] Babel. https://babeljs.io/.

[2] Bellman. https://github.com/zkcrypto/bellman.

[3] Bellman-bignat. https://github.com/alex-ozdemir/bellman-bignat.

[4] Data breach exposed medical records, including blood test results, of over 100 thousand patients. https://gizmodo.com/data-breach-exposed-medical-records-including-blood-te-1819322884.

[5] keybase. https://keybase.io/.

[6] libaad. https://github.com/alinush/libaad-ccs2019.

[7] Microsoft ION. https://github.com/decentralized-identity/ion.

[8] A pure-Rust implementation of group operations on Ristretto and Curve25519. https://github.com/dalek-cryptography/curve25519-dalek.

[9] Redis. https://redis.io.

[10] Spartan: High-speed zkSNARKs without trusted setup. https://github.com/Microsoft/Spartan.

[11] Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region. https://aws.amazon.com/message/41926/.

[12] wrk. https://github.com/wg/wrk.

[13] Tornado Web Server. https://www.tornadoweb.org/en/stable/, 2020.

[14] Node.js. https://nodejs.org/en/, 2022.

[15] Phoenix Framework. https://phoenixframework.org/, 2022.

[16] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[17] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, 2016.

[18] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *Computer Aided Verification (CAV)*, July 2014.

[19] Jade Alglave and Luc Maranget. Stability in weak memory models. In *Computer Aided Verification (CAV)*, July 2011.

[20] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 51–68, 2018.

[21] Aris Anagnostopoulos, Michael T Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security*, pages 379–393, 2001.

[22] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store *actually* provide? In *USENIX Workshop on Hot Topics in*

System Dependability (HotDep), October 2010. Full version: Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[23] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *SIGMOD*, 2017.

[24] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[25] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 196–207, 1999.

[26] Ahmed Awad and Brad Karp. Execution integrity without implicit trust of system software. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2019.

[27] Ahmed Awad and Brad Karp. Enclave-accelerated replay: Efficient integrity for server applications. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2022.

[28] L Babai. Trading group theory for randomness. In *STOC*, pages 421–429, 1985.

[29] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.

[30] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[31] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for JavaScript/Node.js. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, November 2016.

[32] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[33] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract. In *ITCS*, 2013.

[34] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, 2019.

[35] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, 1994.

[36] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[37] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[38] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, June 2006.

[39] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

[40] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2013.

[41] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, 1991.

[42] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019.

[43] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, October 1988.

[44] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP*, 2013.

[45] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[46] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In *Annual International Cryptology Conference*, pages 681–710. Springer, 2021.

[47] Benedikt Bunz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. ePrint Report 2019/1229, 2019.

[48] Jan Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, 2002.

[49] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[50] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *Journal of the ACM (JACM)*, 65(2):1–41, 2018.

[51] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, 2019.

[52] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2013.

[53] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Report 2020/1522, 2020.

[54] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[55] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2013.

[56] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2), September 2015.

[57] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[58] Intel Corporation. 10th Generation Intel® Core Processor™ Families, 2020.

[59] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *PODC*, pages 73–82, 2017.

[60] Scott A Crosby and Dan S Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *European Symposium on Research in Computer Security*, pages 671–688, 2009.

[61] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse Merkle trees: Caching strategies and secure (non-)membership proofs. Cryptology ePrint Archive, Report 2016/683, 2016.

[62] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *S&P*, 2016.

[63] Amadou Diarra, Sonia Ben Mokhtar, Pierre-Louis Aublin, and Vivien Quéma. Fullreview: Practical accountability in presence of selfish nodes. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 271–280. IEEE, 2014.

[64] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, 33(3):209–225, 2020.

[65] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers

based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques (PDPTA)*, August 1996.

[66] Xianzheng Dou, Peter M. Chen, and Jason Flinn. ShortCut: Accelerating mostly-deterministic code regions. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

[67] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008.

[68] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[69] Arield J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[70] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[71] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *CCS*, 2016.

[72] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[73] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[74] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, 2011.

[75] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software: Practice and Experience*, 34(6):523–547, 2004.

[76] Wojciech Golab, Xiaozhou Li, and Mehul Shah. Analyzing consistency properties for fun and profit. In *PODC*, June 2011.

[77] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *STOC*, 1985.

[78] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, August 2015. Prelim version STOC 2008.

[79] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, 2010.

[80] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *SoCC*, 2014.

[81] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[82] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical account-

ability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

[83] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[84] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: secure applications on an untrusted operating system. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, March 2013.

[85] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. Cryptology ePrint Archive, Report 2021/453, 2021.

[86] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: The lightweight deterministic multi-processor replay of concurrent Java programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, February 2010.

[87] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[88] Intel. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[89] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *Computational Complexity*, 2007.

[90] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194, 2010.

[91] Michelle Keeney, Eileen Kowalski, Dawn M. Cappelli, Andrew P. Moore, Timothy J. Shimeall, and Stephanie R. Rogers. Insider threat study: Computer system sabotage in critical infrastructure sectors. https://apps.dtic.mil/sti/citations/ADA636653, 2005. U.S Secret Service and CERT Coordination Center/SEI.

[92] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for web applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[93] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, page 268–280, nov 2020.

[94] Neal Koblitz and Alfred Menezes. Critical perspectives on provable security: Fifteen years of "another look" papers. *Advances in Mathematics of Communications*, 13(4):517, 2019.

[95] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: A framework for efficient verifiable computation. In *S&P*, 2018.

[96] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021.

[97] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS*, June 2010.

[98] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[99] B. Laurie and E Kasper. Revocation transparency, 2013. www.links.org/files/RevocationTransparency.pdf.

[100] Ben Laurie. Certificate transparency. *Communications of the ACM*, page 40–46, September 2014.

[101] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, 1987.

[102] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[103] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. Cryptology ePrint Archive, Report 2020/1274, 2020.

[104] Jonathan Lee, Kirill Nikitin, and Srinath Setty. Replicated state machines without replicated execution. In *S&P*, 2020.

[105] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge snarks for r1cs. Cryptology ePrint Archive, Report 2021/030, 2021.

[106] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: A concurrent authenticated dictionary with short proofs. Cryptology ePrint Archive, Report 2020/975, 2020.

[107] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[108] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *FOCS*, October 1990.

[109] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, New York, NY, USA, 2015. ACM.

[110] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems*, 29(4), December 2011.

[111] David Mazières. The Stellar consensus protocol: A federated model of Internet-level consensus. Technical report, Stellar Development Foundation, 2016.

[112] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *PODC*, page 108–117, 2002.

[113] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, May 2010.

[114] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, April 2008.

[115] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security*, 2015.

[116] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1988.

[117] Silvio Micali. Computationally sound proofs. 30(4):1253–1298, 2000.

[118] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS*, June 2006.

[119] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Intl. Symp. Computer Architecture (ISCA)*, June 2005.

[120] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, December 1993.

[121] Alina Oprea and Kevin D. Bowers. Authentic time-stamps for archival storage. In *European Symposium on Research in Computer Security*, 2009.

[122] Alex Ozdemir, Riad S. Wahby, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security*, 2020.

[123] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), October 1979.

[124] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *S&P*, May 2013.

[125] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[126] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[127] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage SLAs with CloudProof. 2011.

[128] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *European Symposium on Research in Computer Security*, pages 622–641, 2015.

[129] Drummond Reed, Jason Law, and Daniel Hardman. The technical foundations of Sovrin. Technical report, The Sovrin Foundation, 2016.

[130] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, page 199–212, 2009.

[131] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, August 2004.

[132] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, May 2015.

[133] Koushik Sen, Swaroop Kalasapur, and Tasneem Brutch. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE: Proceedings of the Joint Meeting on Foundations of Software Engineering*, August 2013.

[134] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[135] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, 2020.

[136] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, October 2018.

[137] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, April 2013.

[138] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275, 2020.

[139] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, August 2012.

[140] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, April 1988.

[141] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB linux applications with SGX enclaves. In *Network and Distributed System Security Symposium (NDSS)*, February 2017.

[142] Arnab Sinha and Sharad Malik. Runtime checking of serializability in software transactional memory. In *IPDPS*, pages 1–12. IEEE, 2010.

[143] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *ACM Conference on Programming Design and Implementation (PLDI)*, June 2016.

[144] Emin G un Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[145] William N Sumner, Christian Hammer, and Julian Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *RV*, pages 161–176. Springer, 2011.

[146] Cheng Tan, Lingfan Yu, Joshua B Leners, and Michael Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 546–564. ACM, 2017.

[147] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional {Key-Value} stores verifiably serializable. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 63–80, 2020.

[148] Justin Thaler. Proofs, arguments, and zero-knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.

[149] Justin Thaler. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2022.

[150] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, 2019.

[151] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via Bitcoin. In *S&P*, 2017.

[152] Nirvan Tyagi, Ben Fisch, Joseph Bonneau, and Stefano Tessaro. Client-auditable verifiable registries. Cryptology ePrint Archive, Report 2021/627, 2021.

[153] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2008.

[154] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):3, 2012.

[155] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically securing web 2.0 applications through replicated execution. In *ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[156] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[157] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *S&P*, 2018.

[158] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: From theoretical possibility to near practicality. *Communications of the ACM*, 58(2), January 2015.

[159] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *SIGMOD*, June 2022.

[160] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *Intl. Symp. Computer Architecture (ISCA)*, June 2003.

[161] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. ORDER: Object centRic DEterministic Replay for Java. In *USENIX Annual Technical Conference*, June 2011.

[162] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.

[163] Cristian Zamfir, Gautam Altekar, and Ion Stoica. Automating the debugging of datacenter applications with ADDA. In *Dependable Systems and Networks (DSN)*, June 2013.

[164] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.