# Auditing Outsourced Services

by

Cheng Tan

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2020

---

Professor Michael Walfish

# Acknowledgments

First of all, I want to thank my advisor, Michael Walfish. Without him, this dissertation would not exist, and I would not be who I am today. As a role model, Mike has greatly inspired me with his extreme dedication, brutal honesty, and relentless pursuit of perfection. In the past six years, Mike taught me many things, from doing better research to being a better person. I hope I could be an advisor like him one day.

I want to thank the rest of my committee members: Thomas Wies, Dennis Shasha, Miguel Castro, and Peter Chen. Their excellent works inspired this dissertation. In particular, OROCHI owes a massive debt to Peter's amazing line of work on deterministic record-replay. The basis of the project COBRA—serializability and SAT/SMT solvers—is what I learned from Dennis's Advanced Database class and Thomas's Rigorous Software Development class. I am truly honored to have them on my committee.

I am thankful to Joshua Leners, who helped me as a mentor in my first Ph.D. year. And the project OROCHI was originated from a project that Josh created. I also want to thank Curtis Li, Lingfan Yu, and Changgeng Zhao, who worked with me on OROCHI and COBRA. I remember burning the midnight oil with them for rushing SOSPs and OSDIs. These are my precious memory.

I want to give my special thanks to Zhaoguo Wang and Shuai Mu for always being there and continuously share their invaluable experience with me.

I want to thank my officemate, Sebastian Angel, from whom I learned a lot about research and random stuff. And he brought me into the world of coffee, without which I cannot finish this dissertation.

I am grateful to be a member of the NYU systems group, where I feel at home. I want to thank

Jinyang Li, who served on my qualification committee and gave me sincere advice on research and career planning. I also want to thank Lakshmi Subramanian, Joe Bonneau, Anirudh Sivaraman, and Aurojit Panda for giving me immense help and advice for my job search.

I want to thank all my friends who are now at or was at NYU: Talal Ahmad, Ananth Balashankar, Yu Cao, Varun Chandrasekaran, Qi Chen, Yang Cui, Chaoqiang Deng, Pin Gao, Xiangyu Gao, Zhe Gong, Trinabh Gupta, Zhenyi He, Chien-Chin Huang, Shiva Iyer, Ye Ji, Zhongshi Jiang, Taegyun Kim, Siddharth Krishna, Jing Leng, Jinkun Lin, Lamont Nelson, Nisarg Patel, Zvonimir Pavlinovic, Fabian Ruffy, Srinath Setty, Yan Shvartzshnaider, Mukund Sudarshan, Yixin Tao, Ioanna Tzialla, Ashwin Venkataraman, Riad Wahby, Tao Wang, Minjie Wang, John Westhoff, Yang Zhang, Quanlu Zhang, and Anqi Zhang. Thank you for always being supportive.

Finally, I would like to thank my parents for their unconditional support in my Ph.D. journey.

# Abstract

Outsourcing to the cloud is based on *assuming* that remote servers behave as expected, even under failures, bugs, misconfigurations, operational mistakes, insider threats, and external attacks. Can we instead *verify* their behavior? There have been various attempts at such verification, but these attempts have had to choose: comprehensive guarantees or good performance? This dissertation studies how to get both.

This dissertation focuses on two essential services: outsourced computation and outsourced databases. Verifying them correspondingly introduces two new abstract problems. We call the first problem the *Efficient Server Audit Problem*, which examines how to efficiently verify a concurrent and untrusted server. The second problem is verifying a core correctness contract of black-box databases while scaling to real-world online workloads.

To address the two problems, this dissertation respectively introduces two systems: OROCHI and COBRA. Both systems tolerate arbitrary failures in the service provider, and have good performance: in our experiments, OROCHI's verifier achieves 5.6–10.9× speedup versus simply re-executing inputs, with less than 10% CPU overhead on the server side; COBRA improves over baselines by 10× in verification cost, with modest overhead on clients (less than 5% throughput degradation and about 7% 90-percentile latency increases).

# Contents

# List of Figures

# List of Appendices

# 1 Introduction

How can users *verify* that remote applications (such as web applications and databases) execute as promised?

This question is particularly vital today because companies and individuals outsource their computation and data to the cloud [10, 19]. However, users can legitimately wonder whether cloud providers indeed execute the services faithfully. For one thing, clouds are complicated black boxes, running in different administrative domains from users. Any internal corruption on the cloud—as could happen from bugs [126], misconfigurations [16], operational mistakes [42], insider attacks [28], unexpected failures [56, 115, 135, 219], or adversarial control at any layer of the execution stack—can cause incorrect results.

Beyond that, cloud providers may not be fully trustworthy. They have little incentive to provide unfailingly correct services, as "most of the time" [25] is good enough to defeat spot-checks. Even worse, they might undermine their services for profit. For example, in 2019, a startup company sued Tencent Cloud—the second largest cloud provider in mainland China—for downgrading the accuracy of its machine learning service because another department of Tencent was a business rival to this startup company [43]. Unfortunately, there was no audit mechanism, so the startup company was unable to provide direct evidence of Tencent's misconduct.

In light of the motivation above, this dissertation argues that auditing outsourced services is crucial for users. The abstract problem of auditing outsourced services has been studied before. We call the overall topic *Execution integrity*. Execution integrity ensures that a program runs as written. This topic is separate from—but complementary to—program verification, which is concerned with developing bug-free programs. Prior work studies execution integrity under various

assumptions, system models, and usage scenarios. But these works trade off trust assumptions and performance (see §2.1). This dissertation explores a new design point: comprehensive execution guarantees and good performance for both normal executions and auditing. Specifically, this dissertation has two main goals:

1. *Strong model.* This dissertation makes no assumptions about the failure modes of service providers; they can behave arbitrarily. In particular, we do not assume any software or hardware trusted components in their execution stacks.
2. *Pragmatic orientation.* This dissertation aims at providing verifiability with reasonable performance: clients should experience modest performance overheads for their normal executions, and verification should cost substantially less in computational resources than the original executions.

This dissertation focuses on two essential services: outsourced computing (for example, AWS Lambda, Google Web Hosting, Azure App Service, and Heroku cloud) and outsourced databases (such as Amazon DynamoDB, Amazon Aurora, Azure Cosmos DB, and Google Cloud Datastore). To verify the two services, this dissertation presents two systems, OROCHI and COBRA. OROCHI verifies the execution of a concurrent application on untrusted servers. COBRA verifies an essential database property: serializability, the "gold standard" isolation level [69] and the correctness contract that many applications and programmers implicitly assume [205].

## 1.1 Overview and contributions

This section gives a brief overview of the problems, challenges, and techniques of OROCHI and COBRA as well as our contributions.

**OROCHI: verifying the execution of programs on untrusted servers.** OROCHI is motivated by scenarios where people run programs on remote, rented servers. One example is hosting web applications on the cloud: the deployer of a web application submits their code to a cloud provider, who executes the code on servers over which the deployer has no control.

This dissertation abstracts the underlying problem as the *Efficient Sever Audit problem*: a prin-

cipal uploads a program to an untrusted server; the server is supposed to run the program repeatedly and concurrently on different inputs, and respond with the program's outputs; periodically, a verifier gets the inputs and outputs as ground truth, and must answer the question: were the received outputs truly derived from running the program on the received inputs? Answering this question should be computationally cheaper than naively re-executing the program on all inputs.

This problem is both novel and fundamental. It combines for the first time an untrusted and concurrent server, legacy programs, low server overhead, and efficient verification. To solve the problem, we design and implement OROCHI.

OROCHI includes several techniques. The first enables the verifier to accelerate re-execution. The underlying observation is that similar executions have redundant computation, so the verifier can deduplicate computation. In particular, the verifier batches similar executions that have the same control flow, and re-executes them in a single instruction multiple data (SIMD [41]) style: it handles the instructions with the same opcodes and possibly diverse operands at the same time. Crucially, the re-execution saves CPU cycles by performing *identical* instructions—meaning that instructions have the same opcode and operands—once for the whole batch.

One challenge is that, for accelerating re-execution, the verifier needs to know which executions share the same control flow—a piece of information supplied by the server. Meanwhile, the verifier does not trust the server. To tackle this challenge, the verifier optimistically follows the server's information (called *advice*); at the same time, the verifier checks, at every branch point, whether the batched executions indeed go to the same branch, as alleged. If the batched executions diverge, the verifier rejects.

The second technique in OROCHI solves the problem that, because of batching, the verifier's reads to shared objects (such as databases and key-value stores) are undefined. As re-execution at the verifier does not respect the original execution order, a question to answer when re-executing reads is: what value should be returned for a read, if the corresponding write has not been re-executed yet? The standard way of handling an analogous issue in record-replay systems is to record the return values of reads and feed them back during replay. However, this solution fundamentally trusts the recorder (the server), which conflicts with our setup. Instead, OROCHI solves

this problem by mandating that the server record *operations* (instead of values) into the advice; then, during re-execution, the verifier simulates reads—using the writes in the untrusted advice—and checks the alleged writes when they are regenerated from re-execution.

The third technique in OROCHI checks whether alleged operations on shared objects are consistently ordered with respect to externally observable events, such as the ground truth requests and responses. This is critical because, in the advice, the server can fabricate an operation order that conflicts with the order of external observations, but is arranged to be consistent with the server's bogus outputs. To audit the ordering of intertwined operations and external events, we introduce *consistent ordering verification*, which works as follows. The verifier builds a directed graph that has vertices for operations and external events, and edges for observable orderings, including real-time order, program order, and shared objects' schedule order. Importantly, if this graph is acyclic, then the verifier knows there exists a valid ordering of operations and events.

We have proved (Appendix B) that the techniques together produce a verification protocol such that if it passes, there exists a valid execution schedule to generate the received outputs; otherwise, the outputs do not match any physically *possible* honest execution. As a consequence, an honest server can always pass the verification while a bogus execution always fails verification.

To evaluate the effectiveness of the techniques, we implemented OROCHI for PHP applications. OROCHI's verifier re-executes all requests but in a deduplicated and accelerated manner: the verifier achieves 5.6–10.9× speedup versus simply re-executing inputs, with <10% CPU overhead on the server side. The storage overhead is relatively high: the verifier must keep a copy of the server's persistent state.

**COBRA: verifying serializability of black-box databases.** A new generation of cloud databases has emerged that supports ACID transactions and claims to be *serializable* [79, 206]. Yet, users have no assurance that this contract holds. In this dissertation, we study the problem of verifying serializability of black-box databases with real-world online transactional processing workloads.

COBRA faces two main challenges. First, the problem of verifying serializability of black-box databases has long been known to be NP-complete [79]. Nevertheless, inspired by advanced SAT/SMT solvers and their remarkable achievements in "solving" NP-complete problems in prac-

tice (where often the computationally intractable instances don't arise), we hypothesize that it ought to be possible to verify serializability in many real-world workloads. But, it turns out that, even in real cases, the problem is too hard for SMT solvers alone. To reduce the difficulty of the problem, COBRA introduces new SMT encodings that exploit common patterns in practice (for example, read-modify-write transactions), and leverages the computational power of parallel hardware (GPUs) to accelerate the verification. As a result, COBRA can verify 10,000 transactions in less than 14 seconds for all workloads we experiment with, which improves over natural baselines by at least 10× in the problem size.

Second, to verify serializability continually (for example, for an online database), the verifier must trim transaction history; otherwise, verification would become too costly. The challenge is that the checker seemingly needs to retain all history; this is because serializability does not respect real-time ordering, so future transactions can legitimately read from values that (in a real-time view) have been overwritten. To enable trimming history, COBRA introduces *fence transactions*, which impose coarse-grained synchronization and divide the transaction history into *epochs*. Meanwhile, based on epochs, COBRA runs an algorithm that discards transactions that will no longer be referenced. With these techniques, COBRA can sustainably verify 1,000–2,000 transaction/sec, or 86–173M/day (for comparison, Apple Pay handles 33M/day [5], and Visa handles 150M/day [47]).

## 1.2 Limitations

Both OROCHI and COBRA require the full set of inputs to, and outputs from, the audited service, for example HTTP requests and responses for a web application, and queries and results for a database. Faithfully recording the inputs and outputs is natural for some scenarios (see §3) but not for all. Consider the case where both a cloud database and its clients (say web servers) stay in the same cloud. It is unclear how to collect the database's inputs and outputs without trusting any pieces of the cloud because the communication is internal to the cloud. One solution [65] is to deploy a trace collector between the database and clients that is implemented in trusted hardware (such as Intel SGX [137]); the assumption here is that the collector has a simple implementation

and its correctness is easy to reason about, and the trusted hardware can prevent a (malicious) cloud from tampering with the collector's execution.

The trace requirement also means that the collectors and verifier have to tolerate crashes. But OROCHI and COBRA do not have fault tolerant mechanisms, so we are in effect assuming no crashes. Making the collectors and verifier fault-tolerant is future work (see §5); a natural starting point is transparent state machine replication [48, 104].

Another limitation of OROCHI is that accelerating re-execution requires repeated control flow among normal executions. While this requirement holds for web applications [54, 144], desktop applications [110], map-reduce workloads [213], image processing [103], and blockchain applications, it is unclear whether other classes of applications contain such repetition. If not, OROCHI's verification performance would suffer, which in the worst case is equivalent to simply re-executing all inputs.

Finally, COBRA has several limitations in theory and in practice. First, there is no guarantee that COBRA terminates quickly, as the underlying problem is NP-complete (though it terminates in all our experiments; §4.6). Second, COBRA supports only a key-value API, and thus does not support range queries and other SQL operations, such as "join" and "sum"; if an application requires them, then one must translate these queries and operations to a key-value API. Third, COBRA does not yet support async (event-driven) I/O patterns in clients (only multithreading).

## 1.3 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 surveys the literature on general solutions of execution integrity (§2.1), systems that specifically relate to OROCHI (§2.2), and systems that specifically relate to COBRA (§2.3). Chapter 3 defines the Efficient Server Audit Problem (§3.1), proposes an abstract solution (§3.2), and describes an instantiation system (§3.3) targeting PHP web applications. Chapter 4 introduces the problem of verifying serializability of black-box databases (§4.1 and 4.2), as well as the design (§4.3 and 4.4), implementation (§4.5), and evaluation (§4.6) of COBRA, a system that addresses this problem. Finally, Chapter 5 summarizes this dissertation and discusses OROCHI's and COBRA's limitations, which point to the way to future work.

# 2 | Related Work

We discuss general solutions of execution integrity in Section 2.1. In Section 2.2, we describe systems that specifically relate to OROCHI, including deterministic record-replay systems (§2.2.1) and systems that provide integrity for web applications (§2.2.2). Finally, in Section 2.3, we discuss systems that specifically relate to COBRA, which verify or enforce the correctness of databases.

## 2.1 Execution integrity

Execution integrity—giving some *principal* confidence that an *executor*'s outputs are consistent with an expected *program*—is a broad topic. As mentioned in Section 1.1, this dissertation explores a new variant, the Efficient Server Audit Problem (full definition in §3.1), which combines for the first time: (1) no assumptions about the executor (though our verifier gets a trace of requests/responses), (2) a concurrent executor, and (3) a requirement of scaling to real applications, including legacy ones.

**Replication.** A classic solution to execution integrity is Byzantine replication [91, 121, 146]; the principal needs no verification algorithm but assumes that a super-majority of nodes operates fault-free. With a similar assumption that some nodes must be honest, auditing systems like PeerReview [129] and FullReview [107] require honest nodes in the system to re-execute other nodes' tasks and verify whether the results are faithfully produced. Decentralized platforms like Ethereum [208] also use replication to ensure a consistent status among decentralized nodes, with the assumption that honest nodes own the majority of the system's resources.

**Attestation.** Another classic technique is attestation: proving to the principal that the execu-

tor runs the expected software. This includes TPM-based approaches [93, 131, 160, 161, 171, 180, 184, 193], systems [63, 72, 136, 182, 187] built on SGX hardware [137], and software-based approaches [66, 94, 183, 217] that trust privileged software like hypervisor. But attesting to a (possibly vulnerable) stack does not guarantee the execution integrity of the program atop that stack. Using SGX, we can place the program in its own *enclave*, but it is difficult to rigorously establish that the checks performed by the in-enclave code on the out-enclave code [63, 72, 187] comprehensively detect deviations from expected behavior (though see [192]). In addition, the integrity guarantee provided by hardware is questionable according to a recent attack to SGX [165].

**Probabilistic proofs.** Execution integrity has long been studied by theorists [67, 118, 119, 125, 163], and these ideas have been refined and implemented [75, 99, 170, 186] (see [204] for a survey and [58, 74, 203, 218] for recent developments). This theory makes no assumptions about the executor or the workload. But none of these works handle concurrent executors except for one case [185]. Also, because these works generally represent programs as static circuits in which state operations exhaust a very limited "gate budget", and because the executor's overhead is generally at least six orders of magnitude, they are for now unsuited to legacy applications.

## 2.2 Deterministic record-replay and integrity for the web

Orochi verifies remote program executions by re-executing the program on a trusted verifier, which can be categorized as *deterministic record-replay* [100, 108], discussed in Section 2.2.1. Also, the implemented system orochi targets web applications, and we discuss related systems that provide execution integrity for web applications in Section 2.2.2.

### 2.2.1 Deterministic record-replay

Orochi is the first record-replay system to achieve the following combination: (a) the recorder is untrusted (and the replayer has an input/output trace), (b) replay is accelerated versus re-executing, and (c) there are concurrent accesses to shared objects.

**Untrusted recorder.** In AVM [128], an untrusted hypervisor records alleged network I/O and

non-deterministic events. A replayer checks this log against the ground truth network messages and then re-executes, using VM replay [86, 111]. In Ripley [202], a web server re-executes client-side code to determine whether the output matches what the client claimed. In both cases, the replayer does not trust the recorder, but in neither case is re-execution accelerated. A pioneer in this line of work is DIVA [64], a microprocessor architecture that comprises a DIVA checker (the "trusted replayer") to verify the computation of a core processor (the "untrusted recorder"). The checker runs with fewer resources (in DIVA's case, larger transistors to resist faults such as natural radiation interference); *re*-execution can in principle be cheaper than the original execution because it goes second and can exploit hints from the first run.

**Accelerated replay.** Poirot [144] accelerates the re-execution of web applications. OROCHI imitates Poirot: we borrow the observation that web applications have repeated control flow and the notion of grouping re-execution accordingly, and we follow some of Poirot's implementation and evaluation choices (§3.3.7, §3.4). But there is a crucial distinction. Poirot analyzes patches in application code; its techniques for acceleration (construct templates for each claimed control flow group) and shared objects (replay "reads") fundamentally trust the language runtime and all layers below [144, §2.4].

**Shared objects and concurrency.** We focus on solutions that enable an offline replayer to deterministically re-execute concurrent operations. First, the replayer can be given a thread schedule explicitly [152, 200]. Second, the replayer can be given information to *reconstruct* the thread schedule, for example operation precedence using CREW protocols [95, 112, 148, 151, 214]. Third, the replayer can be given information to *approximately reconstruct* the thread schedule, for example, synchronization precedence or sketches [57, 169, 179].[1] Closest to OROCHI is LEAP [134] (see also [211]), which is in the second category: for each shared Java variable, LEAP logs the sequence of thread accesses. But OROCHI's logs also contain *operands*. Simulate-and-check relates to record-replay speculation [152]: it is reminiscent of the way that the epoch-parallel processors in DoublePlay [200] check the starting conditions of optimistically executing future splices.

[1] DoublePlay [200] and Respec [152] use these techniques but do so online, while searching for a thread schedule to give to an offline replayer.

### 2.2.2 Execution integrity for web applications

Orochi, the implemented system, targets web applications and PHP. There are other systems sharing similar motivations. EVE [139] examines a web application by spot-checking for storage consistency violations but assumes correct application execution. Verena [142] checks the integrity of a remote web application; it doesn't require a trace but does assume a trusted hash server. Verena's techniques are built on authenticated data structures with a restricted API; it does not support general-purpose or legacy web applications. Web tripwires [178] detects in-flight page changes between users and the web application, as could happen because of scripts inserted by client software, malicious ISPs, and attacks like ARP poisoning, which complements systems like orochi. WaRR [61], on the contrary, allows web applications to check users' behaviors, which has an "opposite" threat model that distrusts users.

## 2.3 Checking correctness of databases

Cobra verifies serializability, a correctness contract [101] of a cloud database, which necessitates solving a new technical problem (§4.1): (a) black-box checking (no cooperation from the database, no prior knowledge of the workload, and sticking to a standard key-value API) of (b) serializability, while (c) scaling to real-world online transactional processing workloads.

**Checking serializability of black-box databases.** All prior works that tackle (a) and (b) together do not meet (c). Sinha et al. [190] record the ordering of operations in a modified software transactional memory library to reduce the search space in checking serializability; this work uses a "brute-force" polygraph approach (details in §4.2.3), which runs in exponential time. Biswas and Enea (BE) [83] introduce a serializability checking algorithm that runs in $O(n^c)$ where $n$ is the number of transactions and $c$ is the number of clients (see more details in §4.1 and performance comparison with cobra in §4.6.1). Gretchen [26] uses an off-the-shelf constraint solver to check serializability. All these works can only solve small workloads (<1k transactions), which is far from meeting real-world requirements (for example, Visa handles 1.7k transaction/sec [47]). And none of them supports checking an online database with continuous transactions.

**Checking serializability with internal/extra information.** A line of work [130, 145, 189, 197, 210, 215] checks serializability where the ordering of all conflicting operations are known. But, such *internal* ordering information is unavailable for users of a black-box database. Some of the works [130, 197, 210, 215] infer the ordering by heuristics, hence imposes false positives. Sinha et al. [189] acquire the conflicting operation ordering by recording the accesses to shared objects in software transactional memory (STM), but doing so requires (intrusive) modification to the STM system. Elle [145] selects a workload that makes the write-write ordering manifest; specifically, the database client invokes "append", and writes become appends to a list. This approach is not black box according to our notion: it relies on a non-standard API and a specific workload for testing purposes, rather than monitoring a production workload.

Another body of work examines cloud storage consistency [55, 60, 156, 157]. These works rely on *extra* ordering information obtained through techniques like loosely- or well-synchronized clocks [55, 60, 123, 143, 157], or client-to-client communication [156, 188]. As another example, a gateway that sequences the requests can ensure consistency by enforcing ordering [138, 174, 188, 191], thereby dramatically reducing concurrency.

Different from above approaches, Concerto [62] uses *deferred verification*, allowing it to exploit an offline memory checking algorithm [84] to check online the sequential consistency of a highly concurrent key-value store. Concerto's design achieves orders-of-magnitude performance improvement compared to Merkle tree-based approaches [84, 162], but it also requires modifications of the database. (See elsewhere [113, 153] for related algorithms.)

Some of COBRA's techniques are reminiscent of these works, such as its use of serialization graphs [60, 123]. However, a substantial difference is that COBRA neither modifies the "memory" (the database) to get information about the actual internal schedule nor depends on external synchronization.

**Consistency testing.** Serializability is a particular *isolation* level in a transactional system—the I in ACID transactions. In shared memory systems and systems that offer replication (but do not necessarily support transactions), there is an analogous correctness contract, namely *consistency*. (Confusingly, the "C(onsistency)" in ACID transactions refers to something else [68].) Example

consistency models are linearizability [132], sequential consistency [149], and eventual consistency [173]. Testing adherence to these models is an analogous problem to ours. In both cases, one searches for a schedule that fits the ordering constraints of both the model and the history [123]. As in checking serializability, the computational complexity of checking consistency decreases if a stronger model is targeted (for example, linearizability vs. sequential consistency) [120], or if more ordering information can be (intrusively) acquired (by opening black boxes) [207].

**Detecting application anomalies caused by weak consistency.** Several works [88, 166, 177] detect anomalies for applications deployed on weakly consistent storage. Like COBRA, these works use SAT/SMT solvers on graph-related problems. But the similarities end there: these works analyze application behavior, taking the storage layer as *trusted* input. As a consequence, the technical mechanisms are very different.

**Definitions and interpretations of isolation levels.** COBRA of course uses serialization graphs, which are a common tool for reasoning about isolation levels [53, 80, 167]. However, isolation levels can be interpreted via other means such as excluding anomalies [77] and client-centric observations [102]; it remains an open and intriguing question whether the other definitions would yield a more intuitive and easily-implemented encoding and algorithm than the one in COBRA.

# 3 | The Efficient Server Audit Problem

This chapter studies verifying outsourced computation. To articulate the underlying problem, let's start with a concrete example.

Dana the Deployer works for a company whose employees use an open-source web application built from PHP and a SQL database. The application is critical: it is a project management tool (such as JIRA), a wiki, or a forum. For convenience and performance, Dana wants to run the application on a cloud platform, say AWS [4]. However, Dana has no visibility into AWS. Meanwhile, undetected corrupt execution—as could happen from misconfiguration, errors, compromise, or adversarial control at any layer of the execution stack: the language run-time, the HTTP server, the OS, the hypervisor, the hardware—would be catastrophic for Dana's company. So Dana would like assurance that AWS is executing the actual application as written. How can Dana gain this assurance?

Dana's situation is one example of a fundamental problem, which this dissertation defines and studies: the *Efficient Server Audit Problem*. The general shape of this problem is as follows. A principal supplies a *program* to an untrusted *executor* that is supposed to perform repeated and possibly concurrent executions of the program, on different inputs. The principal later wants to *verify* that the outputs delivered by the executor were produced by running the program. The verification algorithm, or *verifier*, is given an accurate *trace* of the executor's inputs and delivered outputs. In addition, the executor gives the verifier *reports*, but these are untrusted and possibly spurious. The verifier must somehow use the reports to determine whether the outputs in the trace are consistent with having actually executed the program. Furthermore, the verifier must make this determination efficiently; it should take less work than re-executing the program on

every input in the trace.

The requirement of a trace is fundamental: if we are auditing a server's outputs, then we need to know those outputs. Of course, getting a trace may not be feasible in all cases. In Dana's case, the company can place a middlebox at the network border, to capture end-clients' traffic to and from the application. We discuss other scenarios later (§3.3.1).

The Efficient Server Audit Problem is a variant of execution integrity. The novelty in this variant is in combining three characteristics: (1) we make no assumptions about the failure modes of the executor, (2) we allow the executor to be concurrent, and (3) we insist on solutions that scale beyond toy programs and are compatible with (at least some) legacy programs.

The contents of this chapter are organized as follows.

**§3.1 Definition of the Efficient Server Audit Problem.** We first present the problem in theoretical terms. We do this to show the generality and the fundamental challenges.

**§3.2 An abstract solution: ssco.** We exhibit a solution at a theoretical level, so as to highlight the core concepts, techniques, and algorithms. These include:

**§3.2.1 SIMD [41]-on-demand.** The verifier re-executes all requests, in an accelerated way. For a group of requests with the same control flow, the verifier executes a "superposition": instructions with identical operands across requests are performed once, whereas instructions with different operands are executed individually and merged into the superposed execution. This solution assumes that the workload has repeated traversal of similar code paths—which is at least the case for some web applications, as observed by Poirot [144, §5].

**§3.2.3 Simulate-and-check.** How can the verifier re-execute *reads* of persistent or shared state? Because it re-executes requests out of order, it cannot physically re-invoke operations on such state, but neither can it trust reports that are allegedly the originally read values (§3.2.2). Instead, the executor (purportedly) logs each operation's operands; during re-execution, the verifier simulates reads, using the writes in the logs, and checks the logged writes opportunistically.

**§3.2.5 Consistent ordering.** The verifier must ensure that operations can be consistently ordered (§3.2.4). To this end, the verifier builds a directed graph with a node for every external

observation or alleged operation, and checks whether the graph is acyclic. This step incorporates an efficient algorithm for converting a trace into a time precedence graph. This algorithm would accelerate prior work [60, 139] and may be useful elsewhere.

SSCO has other aspects besides, and the unified whole was difficult to get right: our prior attempts had errors that came to light when we tried to prove correctness. This version, however, is proved correct Appendix B.

**§3.3 A built system: OROCHI.** We describe a system that implements SSCO for PHP web applications. This is for the purpose of illustration, as we expect the system to generalize to other web languages, and the theoretical techniques in SSCO to apply in other contexts. OROCHI includes a record-replay system [100, 108] for PHP [92, 144]. The replayer is a modified language runtime that implements SIMD-on-demand execution using *multivalue* types that hold the program state for multiple re-executions. OROCHI also introduces mechanisms, based on a versioned database [34, 92, 122, 194], to adapt simulate-and-check to databases and to deduplicate database queries.

**§3.4 Experimental evaluation of OROCHI.** In experiments with several applications, the verifier can audit 5.6–10.9× faster than simple re-execution; this is a loose lower bound, as the baseline is very pessimistic for OROCHI (§3.4.1). OROCHI imposes overhead of roughly 10% on the web server. OROCHI's reports, per-request, are 3%–11% of the size of a request-response pair. Most significantly, the verifier must keep a copy of the server's persistent state.

SSCO and OROCHI have limitations (also see §1.2 and §3.4.5): first, in SSCO the executor has discretion over scheduling concurrent requests, and it gets additional discretion, in OROCHI, over the return values of non-deterministic PHP built-ins. Second, OROCHI is restricted to applications that do not interact much with other applications; nevertheless, there are suitable application classes, for example LAMP [30]. Third, OROCHI requires minor modifications in some applications, owing to the SSCO model. Finally, the principal can audit an application only after activating OROCHI; if the server was previously running, the verifier has to bootstrap from the pre-OROCHI state.

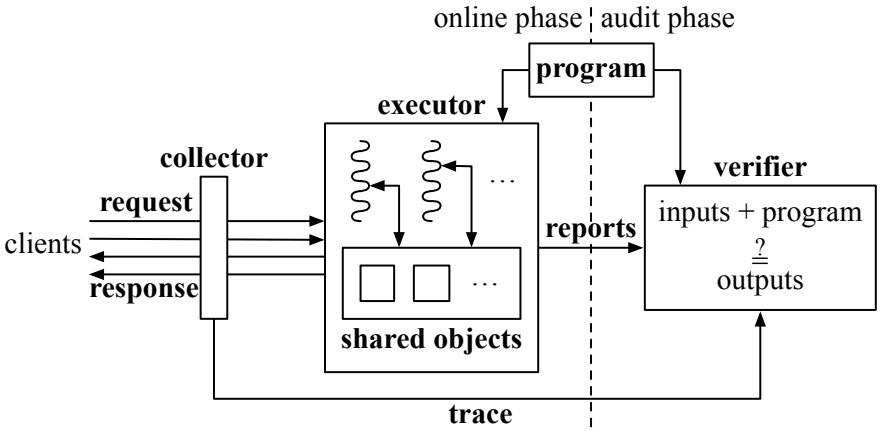



Figure 3.1: The Efficient Server Audit Problem. The objects abstract shared state (databases, key-value stores, memory, etc.). The technical problem is to design the verifier and the reports to enable the verifier, given a trace and a program, to efficiently validate (or invalidate) the contents of responses.

## 3.1 Problem definition

This section defines the Efficient Server Audit Problem. The actors and components are depicted in Figure 3.1.

A *principal* chooses or develops a *program*, and deploys that program on a powerful but untrusted *executor*.

Clients (the outside world) issue *requests* (inputs) to the executor, and receive *responses* (outputs). A response is supposed to be the output of the program, when the corresponding request is the input. But the executor is untrusted, so the response could be anything.

A *collector* captures an ordered list, or *trace*, of requests and responses. We assume that the collector does its job accurately, meaning that the trace exactly records the requests and the (possibly wrong) responses that actually flow into and out of the executor.

The executor maintains *reports* whose purpose is to assist an audit; like the responses, the reports are untrusted.

Periodically, the principal conducts an audit; we often refer to the audit procedure as a *verifier*. The verifier gets a trace (from the accurate collector) and reports (from the untrusted executor). The verifier needs to determine whether executing the program on each input in the trace truly

produces the respective output in the trace.

Two features of our setting makes this determination challenging. First, the verifier is much weaker than the executor, so it cannot simply re-execute all of the requests.

The second challenge arises from concurrency: the executor is permitted to handle multiple requests at the same time (for example, by assigning each to a separate thread), and the invoked program is permitted to issue *operations* to *objects*. An object abstracts state shared among executions, for example a database, key-value store, or memory cells (if shared). We will be more precise about the concurrency model later (§3.2.2). For now, a key point is that, given a trace—in particular, given the ordering of requests and responses in the trace, and given the contents of requests—the number of valid possibilities for the contents of responses could be immense. This is because an executor's responses depend on the contents of shared objects; as usual in concurrent systems, those contents depend on the operation order, which depends on the executor's internal scheduling choices.

Somehow, the reports, though unreliable, will have to help the verifier efficiently tell the difference between valid and invalid traces. In detail, the problem is to design the verifier and the reports to meet these properties:

- *Completeness.* If the executor behaved during the time period of the trace, meaning that it executed the given program under the appropriate concurrency model, then the verifier must accept the given trace.
- *Soundness.* The verifier must reject if the executor misbehaved during the time period of the trace. Specifically, the verifier accepts only if there is some schedule $S$, meaning an interleaving or context-switching among (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 states Soundness precisely.) This property means that the executor can pass the audit only by executing the program on the received requests—or by doing something externally indistinguishable from that.
- *Efficiency.* The verifier must require only a small fraction of the computational resources that

would be required to re-execute each request. Additionally, the executor's overhead must be only a small fraction of its usual costs to serve requests (that is, without capturing reports). Finally, the solution has to work for applications of reasonable scale.

We acknowledge that "small fraction" and "reasonable scale" may seem out of place in a theoretical description. But these characterizations are intended to capture something essential about the class of admissible solutions. As an example, there is a rich theory that studies execution integrity (§2.1), but the solutions (besides not handling concurrency) are so far from scaling to the kinds of servers that run real applications that we must look for something qualitatively different.

## 3.2 A solution: SSCO

This section describes an abstract solution to the Efficient Server Audit Problem, called ssco (a rough abbreviation of the key techniques). ssco assumes that there is similarity among the executions, in particular that there are a relatively small number of control flow paths induced by requests (§3.2.1). ssco also assumes a certain concurrency model (§3.2.2).

**Overview and key techniques.** In ssco, the reports are:

- *Control flow groupings*: For each request, the executor records an opaque tag that purportedly identifies the control flow of the execution; requests that induce the same control flow are supposed to receive the same tag.
- *Operation logs*: For each shared object, the executor maintains an ordered log of all operations (across all requests).
- *Operation counts*: For each request execution, the executor records the total number of object operations that it issued.

The verifier begins the audit by checking that the trace is balanced: every response must be associated with an earlier request, and every request must have a single response or some information that explains why there is none (a network reset by a client, for example). Also, the verifier checks that every request-response pair has a unique *requestID*; a well-behaved executor

ensures this by labeling responses. If these checks pass, we (and the verifier) can refer to request-response pairs by requestID, without ambiguity.

The core of verification is as follows. The verifier re-executes each control flow group in a batch; this happens via *SIMD [41]-on-demand execution* (§3.2.1). During this process, re-executed object operations don't happen directly—they can't, as re-execution follows a different order from the original (§3.2.2). Instead, the operation logs contain a record of reads and writes, and re-execution follows a discipline that we call *simulate-and-check* (§3.2.3): re-executed read operations are fed (or simulated) based on the most recent write entry in the logs, and the verifier checks logged write operations opportunistically. In our context, simulate-and-check makes sense only if alleged operations can be ordered consistent with observed requests and responses (§3.2.4); the verifier determines whether this is so using a technique that we call *consistent ordering verification* (§3.2.5).

At the end, the verifier compares each request's produced output to the request's output in the trace, and accepts if and only if all of them match, across all control flow groups.

The full audit logic is described in Figures 3.3, 3.5, and 3.6, and proved correct in Appendix B.

### 3.2.1 SIMD-on-demand execution

We assume here that requests do not interact with shared objects; we remove that assumption in Section 3.2.2. (As we have just done, we will sometimes use "request" as shorthand for "the execution of the program when that request is input.")

The idea in SIMD-on-demand execution is that, for each control flow group, the verifier conducts a single "superposed" execution that logically executes all requests in that group together, at the same time. Instructions whose operands are different across the separate logical executions are performed separately (we call this *multivalent* execution of an instruction), whereas an instruction executes only once (*univalently*) if its operands are identical across the executions. The concept is depicted in Figure 3.2.

The control flow groupings are structured as a map $C$ from opaque tag to set-of-requestIDs. Of course, the map is part of the untrusted report, so the verifier does not trust it. However, if the map

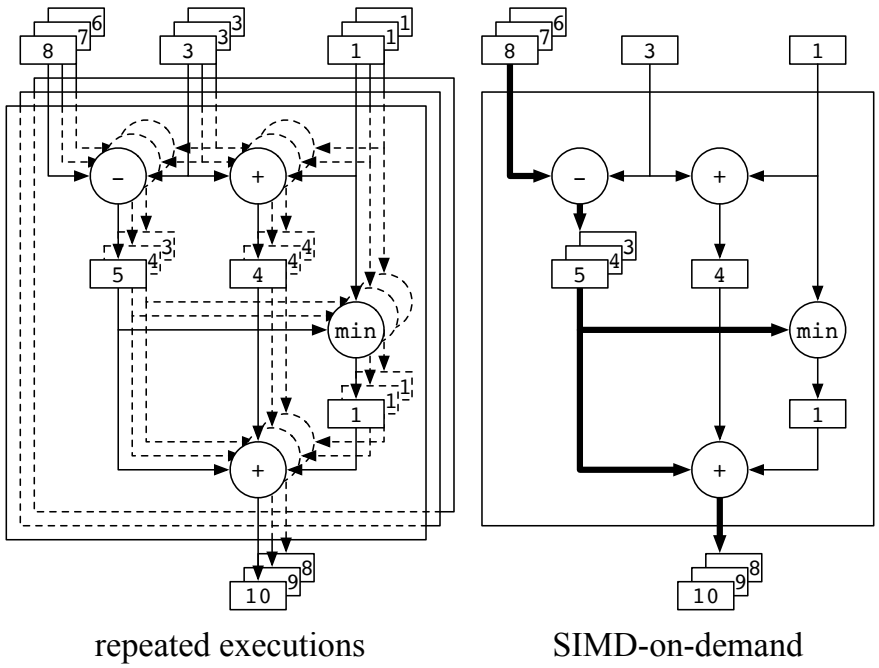



Figure 3.2: Abstract depiction of SIMD-on-demand, for a simple computation. Rectangles represent program variables, circles represent instructions. On the right, thick lines represent explicitly materialized outputs; thin lines represent collapsed outputs.

is incorrect (meaning, two requests in the same control flow group diverge under re-execution), then the verifier rejects. Furthermore, if the map is incomplete (meaning, not including particular requestIDs), then the re-generated responses will not match the outputs in the trace. The verifier can filter out duplicates, but it does not have to do so, since re-execution is idempotent (even with shared objects, below).

Observe that this approach meets the verifier's efficiency requirement (§3.1), if (1) the number of control paths taken is much smaller than the number of requests in the audit, (2) most instructions in a control flow group execute univalently, and (3) it is inexpensive to switch between multivalent and univalent execution, and to decide which to perform. (We say "if" and not "only if" because there may be platforms where, for example, condition (1) alone is sufficient.)

**System preview.** The first two conditions hold in the setting for our built system, OROCHI (§3.3): LAMP web applications. Condition (1) holds because these applications are in a sense routine (they do similar things for different users) and because the programming language is high-level

(for example, string operations or calls like sort() or max() induce the same control flow [144]). Condition (2) holds because the logical outputs have a lot of overlap: different users wind up seeing similar-looking web pages, which implies that the computations that produce these web pages include identical data flows. This commonality was previously observed by Poirot [144], and our experiments confirm it (§3.4.2). Condition (3) is achieved, in OROCHI, by augmenting the language run-time with *multivalue* versions of basic datatypes, which encapsulate the different values of a given operand in the separate executions. Re-execution moves dynamically between a vector, or SIMD, mode (which operates on multivalues) and a scalar mode (which operates on normal program variables).

### 3.2.2 Confronting concurrency and shared objects

As noted earlier, a key question is: how does the verifier re-execute an operation that reads from a shared object? An approach taken elsewhere [92, 144] is to record the values that had been read by each request, and then to supply those values during re-execution. One might guess that, were we to apply this approach to our context where reports are untrusted, the worst thing that could happen is that the verifier would fail to reproduce the observed outputs in the trace—in other words, the executor would be incriminating itself. But the problem is much worse than that: the reported values and the responses could both be bogus. As a result, if the verifier's re-execution dutifully incorporated the purported read values, it could end up reproducing, and thereby validating, a spurious response from a misbehaved executor; this violates Soundness (§3.1).

**Presentation plan.** Below, we define the concurrency model and object semantics, as necessary context. We then cover the core object-handling mechanisms (§3.2.3–§3.2.5). However, that description will be incomplete, in two ways. First, we will not cover every check or justify each line of the algorithms. Second, although we will show with reference to examples why certain alternatives fail, that will be intuition and motivation, only; correctness, meaning Completeness and Soundness (§3.1), is actually established end-to-end, with a chain of logic that does not enumerate or reason about all the ways in which reports and responses could be invalid (Appendix B).

**Concurrency model and object semantics.** In a well-behaved executor, each request induces

**Input** Trace *Tr* **Input** Reports *R* **Global** *OpMap*: (requestID, opnum) → (*i*, seqnum)

```
   Components of the reports R:
// purported groups; §3.2.1
C: CtlFlowTag → Set(requestIDs)
// purported op logs; §3.2.3
OL_i: ℕ+ → (requestID, opnum, optype, opcontents)
// purported op counts; §3.2.3
M: requestID → ℕ


 1: procedure SSCO_AUDIT()
 2:     // Partially validate reports and construct OpMap
 3:     ProcessOpReports()          // defined in Figure 3.5
 4:
 5:     return ReExec()     // line 24
 6:
 7: procedure CHECKOP(rid, opnum, i, optype, oc)
 8:     if (rid, opnum) not in OpMap : REJECT
 9:
10:     î, s ← OpMap[rid, opnum]
11:     ôt, ôc ← (OL_i[s].optype, OL_i[s].opcontents)
12:     if i ≠ î or optype ≠ ôt or oc ≠ ôc :
13:         REJECT
14:     return s
15:
16: procedure SIMOP(i, s, optype, opcontents)
17:     ret ← ⊥
18:     writeop ← walk backward in OL_i from s; stop
19:         when optype=RegisterWrite
20:     if writeop doesn't exist :
21:         REJECT
22:     ret = writeop.opcontents
23:     return ret
24: procedure REEXEC()
25:     Re-execute Tr in groups according to C:
26:
27:         (1) Initialize a group as follows:
28:             Read in inputs for all requests in the group
29:             Allocate structures for each req in the group
30:             // opnum is a per-group running counter
31:             opnum ← 1
32:
33:         (2) During SIMD-on-demand execution:
34:
35:             if execution within the group diverges:
36:                 return REJECT
37:
38:             When the group makes a state operation:
39:                 optype ← the type of state operation
40:                 for all rid in the group:
41:                     i, oc ← op params from execution
42:                     s ← CheckOp(rid, opnum, i,
43:                                 optype, oc) // line 7
44:                     if optype = RegisterRead:
45:                         op result ←
    SimOp(i, s, optype, oc)
46:                 opnum ← opnum + 1
47:
48:         (3) When a request rid finishes:
49:             if opnum < M(rid): return REJECT
50:
51:         (4) Write out the produced outputs
52:
53:     if the outputs from (4) equal the responses in Tr :
54:         return ACCEPT
55:     return REJECT
```

Figure 3.3: The SSCO audit procedure. The supplied trace *Tr* must be balanced (§3.2), which the verifier ensures before invoking SSCO_AUDIT. A rigorous proof of correctness is in Appendix B.

the creation of a separate *thread* that is destroyed after the corresponding response is delivered. A thread runs concurrently with the threads of any other requests whose responses have not yet been delivered. Each thread sequentially performs instructions against an isolated execution context: registers and local memory. As stated earlier, threads perform *operations* on shared objects (§3.1). These operations are blocking, and the objects expose atomic semantics. We assume for simplicity in this section that objects expose a read-write interface; they are thus atomic registers [150]. Later, we will permit more complex interfaces, such as SQL transactions (§3.3.4).

### 3.2.3 Simulate-and-check

The reports in ssco include the (alleged) operations themselves, in terms of their operands. Below, we describe the format and how the verifier uses these operation logs.

**Operation log contents.** Each shared object is labeled with an index $i$. The operation log for object $i$ is denoted $OL_i$, and it has the following form ($\mathbb{N}^+$ denotes the set $\{1, 2, \ldots\}$):

$$OL_i : \mathbb{N}^+ \rightarrow (\text{requestID}, \text{opnum}, \text{optype}, \text{opcontents}).$$

The opnum is per-requestID; a correct executor tracks and increments it as requestID executes. An operation is thus identified with a unique (*rid*, *opnum*) pair. The optype and opcontents depend on the object type. For registers, optype can be RegisterRead (and opcontents are supposed to be empty) or RegisterWrite (and opcontents is the value to write).

**What the verifier does.** The core re-execution logic is contained in ReExec (Figure 3.3, line 24). The verifier feeds re-executed *reads* by identifying the latest write before that read in the log. Of course, the logs might be spurious, so for *write* operations, the verifier opportunistically checks that the operands (produced by re-execution) match the log entries.

In more detail, when re-executing an operation (*rid*, *opnum*), the verifier uses *OpMap* (as defined in Fig. 3.3) to identify the log entry; it then checks that the parameters (generated by program logic) match the logs. Specifically, the verifier checks that the targeted object corresponds to the (unique) log that holds (*rid*, *opnum*) (uniqueness is ensured by checks in Figure 3.5), and

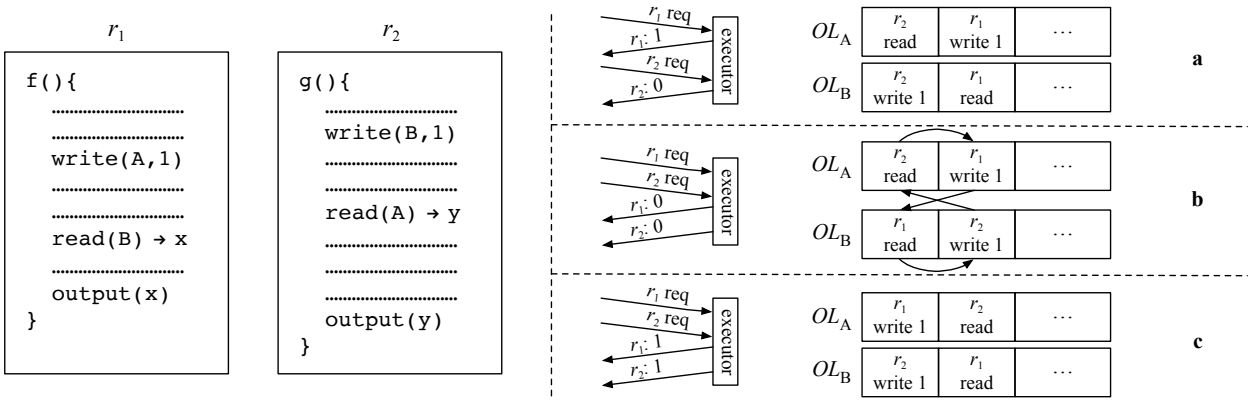



Figure 3.4: Three examples to highlight the verifier's challenge and to motivate consistent ordering verification (§3.2.5). As explained in the text, a correct verifier (meaning Complete and Sound; §3.1) must reject examples **a** and **b**, and accept **c**. In these examples, $r_1$ and $r_2$ are requestIDs in different control flow groups, and their executions invoke different subroutines of the given program. For simplicity, there is only one request per control flow group, and objects are assumed to be initialized to 0. What varies among examples are the timing of requests and responses, the contents of the executor's responses, and the alleged operation logs for objects $A$ and $B$ (denoted $OL_A$, $OL_B$). The opnum component of the log entries is not depicted.

that the produced operands (such as the value to be written) are the same as in the given log entry (lines 39–43, Figure 3.3). If the re-executed operation is a read, the verifier feeds it by identifying the write that precedes (*rid*, *opnum*); this is done in SimOp.

Notice that an operation that reads a given write might re-execute long before the write is validated. The intuition here is that a read's validity is contingent on the validity of all prior write operations in the log. Meanwhile, the audit procedure succeeds only if all checks—including the ones of write operations—succeed, thereby retroactively discharging the assumption underlying every read.

What prevents the executor from justifying a spurious response by inserting into the logs additional operations? Various checks in the algorithm would detect this and other cases. For example, the op count reports $M$ enforce certain invariants, and interlocking checks in the algorithms validate $M$.

### 3.2.4 Simulate-and-check is not enough

To show why simulate-and-check is insufficient by itself, and to illustrate the challenge of augmenting it, this section walks through several simple examples. This will give intuition for the techniques in the next section (§3.2.5).

The examples are depicted in Figure 3.4 and denoted **a**, **b**, **c**. Each of them involves two requests, $r_1$ and $r_2$. Each example consists of a particular trace—or, equivalently, a particular request-response pattern—and particular reports. As a shorthand, we notate the delivered responses with a pair (r1resp, r2resp); for example, the responses in **a** are $(1, 0)$.

A correct verifier must reject **a**, reject **b**, and accept **c**.

To see why, note that in **a**, the executor delivers a response to $r_1$ before $r_2$ arrives. So the executor must have executed $r_1$ and then executed $r_2$. Under that schedule, there is no way to produce the observed output $(1, 0)$; in fact, the only output consistent with the observed events is $(0, 1)$. Thus, accepting **a** would violate Soundness (§3.1).

In **b**, $r_1$ and $r_2$ are concurrent. A well-behaved executor can deliver any of $(0, 1)$, $(1, 0)$, or $(1, 1)$, depending on the schedule that it chooses. Yet, the executor delivered $(0, 0)$, which is consistent with no schedule. So accepting **b** would also violate Soundness.

In **c**, $r_1$ and $r_2$ are again concurrent. This time, the executor delivered $(1, 1)$, which a well-behaved executor can produce, by executing the two writes before either read. Therefore, rejecting **c** would violate Completeness (§3.1).

Now, if the verifier used only simulate-and-check (Figure 3.3), the verifier would accept in all three of the examples. We encourage curious readers to convince themselves of this behavior by inspecting the verifier's logic and the examples. Something to note is that in **a** and **b**, the operation logs and responses are both spurious, but they are arranged to be consistent with each other.

Below are some strawman attempts to augment simulate-and-check, by analyzing all operation logs prior to re-execution.

- What if the verifier (i) creates a global order $O$ of requests that is consistent with the real-time order (in **a**, $r_1$ would be prior to $r_2$ in $O$; in **b** and **c**, either order is acceptable), and (ii) *for each log*, checks that the order of its operations is consistent with $O$? This would rightly reject **a**

($r_1$ is before $r_2$ in $O$ but not in the logs), rightly reject **b** (regardless of the choice of $O$, one of the two logs will violate it), and wrongly reject **c** (for the same reason it would reject **b**). This approach would be tantamount to insisting that entire requests execute atomically (or transactionally)—which is contrary to the concurrency model.

- What if the verifier creates only a partial order $O'$ on requests that is consistent with the real-time order, and then insists that, for each log, the order of operations is consistent with $O'$? That is, operations from concurrent requests can interleave in the logs. This would rightly reject **a** and rightly accept **c**. But it would wrongly accept **b**.
- Now notice that the operations in **b** cannot be ordered: considering log and program order, the operations form a cycle, depicted in Figure 3.4. So what if the verifier (a) creates a directed graph whose nodes are all operations in the log and whose edges are given by log order and program order, and (b) checks that there are no cycles? That would rightly reject **b** and accept **c**. But it would wrongly accept **a**.

The verifier's remaining techniques, described next, can be understood as combining the preceding failed attempts.

### 3.2.5 Consistent ordering verification

At a high level, the verifier *ensures the existence of an implied schedule that is consistent with external observations and alleged operations*. Prior to re-executing, the verifier builds a directed graph $G$ with a node for every *event* (an observed request or response, or an alleged operation); edges represent precedence [150]. The verifier checks whether $G$ is acyclic. If so, then all events can be consistently ordered, and the implied schedule is exactly the ordering implied by $G$'s edges. Note, however, that the verifier does not follow that order when re-executing nor does the verifier consult $G$ again.

Figures 3.5 and 3.6 depict the algorithms. $G$ contains nodes labeled $(rid, opnum)$, one for each alleged operation in the logs. $G$ also has, for each request $rid$ in the trace, nodes $(rid, 0)$ and $(rid, \infty)$, representing the arrival of the request and the departure of the response, respectively. $G$'s edges capture program order via AddProgramEdges and alleged operation order via AddStateEdges.

```
1: Global Trace Tr, Reports R, Graph G, OpMap OpMap
2: procedure ProcessOpReports()
3:
4:     G_Tr ← CreateTimePrecedenceGraph() // Fig 3.6
5:     SplitNodes(G_Tr)
6:     AddProgramEdges()
7:
8:     CheckLogs()              // also builds the OpMap
9:     AddStateEdges()
10:
11:    // standard algorithm [98, Ch. 22]
12:    if CycleDetect(G) : reject
13:
14: procedure SplitNodes(Graph G_Tr)
15:    G.Nodes ← {}, G.Edges ← {}
16:    for each node rid ∈ G_Tr.Nodes :
17:        G.Nodes += { (rid, 0), (rid, ∞) }
18:    for each edge ⟨rid_1, rid_2⟩ ∈ G_Tr.Edges :
19:        G.Edges += ⟨(rid_1, ∞), (rid_2, 0)⟩
20:
21: procedure AddProgramEdges()
22:    for all rid that appear in the events in Tr :
23:        for opnum = 1, ..., R.M(rid) :
24:            G.Nodes += (rid, opnum)
25:            G.Edges += ⟨(rid, opnum-1), (rid, opnum)⟩
26:        G.Edges += ⟨(rid, R.M(rid)), (rid, ∞)⟩
27: procedure CheckLogs()
28:    for log = R.OL_1, ..., R.OL_n :
29:        for j = 1, ..., length(log) :
30:            if log[j].rid does not appear in Tr or
31:                log[j].opnum ≤ 0 or
32:                log[j].opnum > R.M(log[j].rid) or
33:                (log[j].rid, log[j].opnum) is in OpMap :
34:                reject
35:
36:            let curr_op = (log[j].rid, log[j].opnum)
37:            // i is the index such that log = R.OL_i
38:            OpMap[curr_op] ← (i, j)
39:
40:    for all rid that appear in the events in Tr :
41:        for opnum = 1, ..., R.M(rid) :
42:            if (rid, opnum) is not in OpMap : reject
43:
44: procedure AddStateEdges()
45:    // Add edge to G if adjacent log entries are
46:    // from different requests. If they are from
47:    // the same request, check that the
48:    // intra-request opnum increases
49:    for log = R.OL_1, ..., R.OL_n :
50:        for j = 2, ..., length(log) :
51:            let curr_r, curr_op, prev_r, prev_op =
52:              (log[j].rid, log[j].opnum, log[j−1].rid,
53:                  log[j−1].opnum)
54:            if prev_r ≠ curr_r :
55:                G.Edges += ⟨(prev_r, prev_op),
56:                                (curr_r, curr_op)⟩
57:            else if prev_op > curr_op : reject
```

Figure 3.5: ProcessOpReports ensures that events (request arrival, departure of response, and operations) can be consistently ordered. It does this by constructing a graph $G$—the nodes are events; the edges reflect request precedence in $Tr$, program order, and the operation logs—and ensuring that $G$ has no cycles. *OpMap* is constructed here as an index of the operation logs.

```
1:  procedure CreateTimePrecedenceGraph()
2:      // "Latest" requests; "parent(s)" of any new request
3:      Frontier ← {}
4:      G_Tr.Nodes ← {}, G_Tr.Edges ← {}
5:
6:      for each input and output event in Tr, in time order :
7:          if the event is request(rid) :
8:              G_Tr.Nodes += rid
9:              for each r in Frontier :
10:                 G_Tr.Edges += ⟨r, rid⟩
11:         if the event is response(rid) :
12:             // rid enters Frontier, evicting its parents
13:             Frontier −= { r | ⟨r, rid⟩ ∈ G_Tr.Edges }
14:             Frontier += rid
15:     return G_Tr
```

Figure 3.6: Algorithm for explicitly materializing the time-precedence partial order, $<_{Tr}$, in a graph. The algorithm constructs $G_{Tr}$ so that $r_1 <_{Tr} r_2 \iff G_{Tr}$ has a directed path from $r_1$ to $r_2$. *Tr* is assumed to be a (balanced; §3.2) list of request and response events in time order.

**Capturing time precedence.** To be consistent with external observations, $G$ must also capture time precedence. (This is what was missing in the final attempt in §3.2.4.) We say that $r_1$ precedes $r_2$ (notated $r_1 <_{Tr} r_2$) if the trace *Tr* shows that $r_1$ departed from the system before $r_2$ arrived [150]. If $r_1 <_{Tr} r_2$, then the operations issued by $r_1$ must occur in the implied schedule prior to those issued by $r_2$.

Therefore, the verifier needs to construct edges that capture the $<_{Tr}$ partial order, in the sense that $r_1 <_{Tr} r_2 \iff G$ has a directed path from $(r_1, \infty)$ to $(r_2, 0)$. How can the verifier construct these edges from the trace? Prior work [60] gives an *offline* algorithm for this problem that runs in time $O(X \cdot \log X + Z)$, where $X$ is the number of requests, and $Z$ is the minimum number of time-precedence edges needed (perhaps counter-intuitively, more concurrency leads to higher $Z$).

By contrast, our solution runs in time $O(X + Z)$ (see complexity analysis in §B.8), and works in *streaming* fashion. The key algorithm is CreateTimePrecedenceGraph, given in Figure 3.6 and proved correct in Appendix B (Lemma 2). The algorithm tracks a "frontier": the set of latest, mutually concurrent requests. Every new arrival descends from all members of the frontier. Once a request leaves, it evicts all of its parents from the frontier. This algorithm may be of independent interest; for example, it could be used to accelerate prior work [60, 139].

Overall, the algorithms in Figures 3.5 and 3.6 cost $O(X + Y + Z)$ time and $O(Y)$ space (see

analysis in §B.8), with good constants (Fig. 3.10; §3.4.2); here, $Y$ is the number of object operations in the logs.

## 3.3 A built system: OROCHI

The prior two sections described the Efficient Server Audit Problem, and how it can be solved with SSCO. This section applies the model to an example system that we built.

Consider again Dana, who wishes to verify execution of a SQL-backed PHP web application running on AWS. In this context, the *program* is a PHP application (and the separate PHP scripts are subroutines). The *executor* is the entire remote stack, from the hardware to the hypervisor and all the way up to and including the PHP runtime; we often call the executor just the *server*. The *requests* and *responses* are the HTTP requests and responses that flow in and out of the application. The *collector* is a middlebox at the edge of Dana's company, and is placed to inspect and capture end-clients' requests and the responses that they receive. An *object* can be a SQL database, per-client data that persists across requests, or other external state accessed by the application.

We can apply SSCO to this context, if we:

- Develop a record-replay system for PHP in which replay is batched according to SIMD-on-demand (§3.2.1).
- Define a set of object types that (a) abstract PHP state constructs (session data, databases, etc.) and (b) obey the semantics in SSCO (§3.2.2). Each object type requires adapting simulate-and-check (§3.2.3) and, possibly, modifying the application to respect the interfaces of these objects.
- Incorporate the capture (and ideally validation) of certain sources of non-determinism, such as PHP built-ins.

The above items represent the main work of our system, OROCHI. We describe the details in Sections 3.3.3–3.3.7.

### 3.3.1 Applicability of orochi, theory vs. practice

orochi is relevant in scenarios besides Dana's. As an example, Pat the Principal runs a public-facing web application on local hardware and is worried about compromise of the server, but trusts a middlebox in front of the server to collect the trace.

orochi is implemented for PHP-based HTTP applications but in principle generalizes to other web standards. Also, orochi verifies an application's interactions with its *clients*; verifying communication with external services requires additional mechanism (§3.4.5). Ultimately, orochi is geared to applications with few such interactions. This is certainly restrictive, but there is a useful class within scope: the LAMP [30] stack. The canonical LAMP application is a PHP front-end to a database, for example a wiki or bug database.

The model in Sections 3.1 and 3.2 was very general and abstracted away certain considerations that are relevant in orochi's setting. We describe these below:

*Persistent objects.* The verifier needs the server's objects as they were at the beginning of the audited period. If audit periods are contiguous, then the verifier in orochi produces the required state during the previous audit (§3.3.5).

*Server-client collusion.* In Section 3.1, we made no assumptions about the server and clients. Here, however, we assume that the server cannot cause end-clients to issue spurious requests; otherwise, the server might be able to "legally" insert events into history. This assumption fits Dana's situation though is admittedly shakier in Pat's.

*Differences in stack versions.* The verifier's and server's stacks need not be the same. However, it is conceivable that different versions could cause the verifier to erroneously reject a well-behaved server (the inverse error does not arise: validity is defined by the verifier's re-execution). If the verifier wanted to eliminate this risk, it could run a stack with precise functional equivalence to the server's. Another option is to obtain the server-side stack in the event of a divergent re-execution, so as to exonerate the server if warranted.

*Modifications by the network.* Responses modified en route to the collector appear to orochi to be the server's responses; modifications between the collector and end-clients—a real concern in Pat's scenario, given that ISPs have hosted ad-inserting middleboxes [96, 209]—can be addressed

by Web Tripwires (WT) [178], which are complementary to OROCHI.

### 3.3.2 Some basics of PHP

PHP [36] is a high-level language. When a PHP script is run by a web server, the components of HTTP requests are materialized as program variables. For example, if the end-user submits `http://www.site.org/s.php?a=7`, then the web server invokes a PHP runtime that executes `s.php`; within the script `s.php`, `$_GET['a']` evaluates to 7.

The data types are *primitive* (int, double, bool, string); *container* (arrays, objects); *reference*; *class*; *resource* (an abstraction of an external resource, such as a connection to a database system); and *callables* (closures, anonymous functions, callable objects).

The PHP runtime translates each program line to byte code: one or more virtual machine (VM) instructions, together with their operands. (Some PHP implementations, such as HHVM, support JIT, though OROCHI's verifier does not support this mode.) Besides running PHP code, the PHP VM can call built-in functions, written in C/C++.

### 3.3.3 SIMD-on-demand execution in OROCHI

The server and verifier run modified PHP runtimes. The server's maintains an incremental digest for each execution. When the program reaches a branch, this runtime updates the digest based on the type of the branch (jump, switch, or iteration) and the location to which the program jumps. The digest thereby identifies the control flow, and the server records it.

The verifier's PHP runtime is called *acc-PHP*; it performs SIMD-on-demand execution (§3.2.1), as we describe below.

Acc-PHP works at the VM level, though in our examples and description below, we will be loose and often refer to the original source. Acc-PHP broadens the set of PHP types to include *multivalue* versions of the basic types. For example, a multivalue int can be thought of as a vector of ints. A container's cells can hold multivalues; and a container can itself be a multivalue. Analogously, a reference can name a multivalue; and a reference can itself be a multivalue, in which case each of the references in the vector is logically distinct. A variable that is not a multivalue

is called a *univalue*.

All requests in a control flow group invoke the same PHP script *s*. At the beginning of re-executing a control flow group, acc-PHP sets the input variables in *s* to multivalues, based on the inputs in the trace. Roughly speaking, instructions with univalue operands produce univalues, and instructions with multivalue operands produce multivalues. But when acc-PHP produces a multivalue whose components are identical, reflecting a variable that is the same across executions, acc-PHP collapses it down to a univalue; this is crucial to deduplication (§3.4.2). A collapse is all or nothing: every multivalue has cardinality equal to the number of requests being re-executed.

**Primitive types.** When the operands of an instruction or function are primitive multivalues, acc-PHP executes that instruction or function componentwise. Also, if there are mixed multivalue and univalue operands, acc-PHP performs scalar expansion (as in Matlab, etc.): it creates a multivalue, all of whose components are equal to the original univalue. As an example, consider:

```
1   $sum = $_GET['x'] + $_GET['y'];
2   $larger = max ($sum, $_GET['z']);
3   $odd = ($larger % 2) ? "True" : "False";
4   echo $odd;

r1: /prog.php?x=1&y=3&z=10
r2: /prog.php?x=2&y=4&z=10
```

There are two requests: `r1` and `r2`. Each has three inputs: `x`, `y`, and `z`, which are materialized in the program as `$_GET['x']`, etc. Acc-PHP represents these inputs as multivalues: `$_GET['x']` evaluates to $[1, 2]$, and `$_GET['y']` evaluates to $[3, 4]$. In line 1, both operands of `+` are multivalues, and `$sum` receives the elementwise sum: $[4, 6]$. In line 2, `$larger` receives $[10, 10]$, and acc-PHP merges the multivalue to make it a univalue. As a result, lines 3 and 4 execute once, rather than once for each request.

A multivalue can comprise different types. For example, in two requests that took the same code path, a program variable was an int in one request and a float in the other. Our acc-PHP implementation handles an int-and-float mixture. However, if acc-PHP encounters a different

mixture, it retries, by separately re-executing the requests in sequence.

**Containers.** We use the example of a "set" on an object: `$obj->$key = $val`. Acc-PHP handles "gets" similarly, and likewise other containers (arrays, arrays of arrays, etc.).

Assume first that `$obj` is a multivalue. If either of `$key` and `$val` are univalues, acc-PHP performs scalar expansion to create a multivalue for `$key` and `$val`. Then, acc-PHP assigns the *i*th component of `$val` to the property named by the *i*th component of `$key` in the *i*th object in `$obj`.

Now, if `$obj` is a univalue and `$key` is a multivalue, acc-PHP expands the `$obj` into a multivalue, performs scalar expansion on `$val` (if a univalue), and then proceeds as in the preceding paragraph. The reason for the expansion is that in the original executions, the objects were no longer equivalent.

When `$obj` and `$key` are univalues, and `$val` is a multivalue, acc-PHP assigns `$val` to the given object's given property. This is similar to the way that acc-PHP set up `$_GET['a']` as a multivalue in the example above.

**Built-in functions.** For acc-PHP's re-execution to be correct, PHP's built-in functions (§3.3.2) would need to be extended to understand multivalues, perform scalar expansion as needed, etc. But there are thousands of built-in functions.

To avoid modifying them all, acc-PHP does the following. When invoking a built-in function, it checks whether any of the arguments are multivalues (if the function is a built-in method, it also checks whether `$this` is a multivalue). If so, acc-PHP splits the multivalue argument into a set of univalues; assume for ease of exposition that there is only one such multivalue argument. Acc-PHP then clones the environment (argument list, function frame); performs a deep copy of any objects referenced by any of the arguments; and executes the function, once for each univalue. Finally, acc-PHP returns the separate function results as a multivalue and maintains the object copies as multivalues. The reason for the deep copy is that the built-in function could have modified the object differently in the original executions.

**Global variables.** There are two cases to handle. First, if a multi-invoked built-in (as above) modifies a global and if the global is a univalue, acc-PHP dynamically expands it to a multivalue.

Second, a global can be modified by PHP code, implicitly. For example, referencing a nonexistent property from an object causes invocation of a PHP function, known as a magic method [35], which could modify a global. Acc-PHP detects this case, and expands the global into a multivalue.

### 3.3.4 Concurrency and shared objects in orochi

ssco's concurrency model (§3.2.2) fits PHP-based applications, which commonly have concurrent threads, each handling a single end-client request sequentially. orochi supports several objects that obey ssco's required semantics (§3.2.2) and that abstract key PHP programming constructs:

- *Registers*, with atomic semantics [150]. These work well for modeling per-user persistent state, known as "session data." Specifically, PHP applications index per-user state by browser cookie (this is the "name" of the register) and materialize the state in a program variable. Constructing this variable is the "read" operation; a "write" is performed by PHP code, or by the runtime at the end of a request.
- *Key-value stores*, exposing a single-key get/set interface, with linearizable semantics [132]. This models various PHP structures that provide shared memory to requests: the Alternative PHP Cache (APC), etc.
- *SQL databases*, which support single-query statements and multi-query transactions. To make a SQL database behave as one atomic object, we impose two restrictions. First, the database's isolation level must be strict serializability [80, 167].[1] Second, a multi-statement transaction cannot enclose *other* object operations (such as a nested transaction).

The first DB restriction can be met by configuration, as many DBMSes provide strict serializability as an option. However, this isolation level sacrifices some concurrency compared to, say, MySQL's default [45]. The second DB restriction sometimes necessitates minor code changes, depending on the application (§3.4.4).

To adapt simulate-and-check to an object type, orochi first collect an operation log (§3.2.3). To that end, some entity (this step is untrusted) wraps relevant PHP statements, to invoke a recording

[1] Confusingly, our required atomicity is, in the context of ACID databases, not the "A" but the kind of "I" (isolation); see Bailis [68] for an untangling.

library. Second, OROCHI's verifier needs a mechanism for efficiently re-executing operations on the object. We showed the solution for registers in §3.2.3. But that technique would not be efficient for databases or key-value stores: to re-execute a DB "select" query, for example, could require going backward through the entire log.

### 3.3.5 Adapting simulate-and-check to databases

Given a database object $d$—OROCHI handles key-value stores similarly—the verifier performs a *versioned redo* pass over $OL_d$ at the beginning of the audit: it issues every transaction to a *versioned database* [34, 92, 122, 194], setting the version to be the sequence number in $OL_d$. During re-execution, the verifier handles a "write" query (UPDATE, etc.) by checking that the program-generated SQL matches the opcontents field in the corresponding log entry. The verifier handles "read" queries (SELECT, etc.) by issuing the SQL to the versioned DB, specifying the version to be the log sequence number of the current operation. The foregoing corresponds to an additional step in SSCO_AUDIT and further cases in SimOp (Figure 3.3); the augmented algorithms are in Appendix B.

As an optimization, OROCHI applies *read query deduplication*. If two SELECT queries $P$ and $Q$ are lexically identical and if the parts of the DB covered by $P$ and $Q$ do not change between the redo of $P$ and $Q$, then it suffices to issue the query once during re-execution. To exploit this fact, the verifier, during re-execution, clusters all queries in a control flow group and sorts each cluster by version number. Within a cluster, it de-duplicates queries $P$ and $Q$ if the tables that $P$ and $Q$ touch were not modified between $P$'s and $Q$'s versions.

To speed the versioned redo pass, the verifier directs update queries to an *in-memory* versioned database $M$, which acts as a buffer in front of the audit-time versioned database $V$. When the log is fully consumed, the verifier migrates the final state of $M$ to $V$ using a small number of transactions: the verifier dumps each table in $M$ as a single SQL update statement that, when issued to $V$, reproduces the table. The migration could also happen when $M$ reaches a memory limit (although we do not implement this). This would require subsequently re-populating $M$ by reading records from $V$.

### 3.3.6 Non-determinism

orochi includes non-determinism that is not part of the ssco model: non-deterministic PHP built-ins (time, getpid, etc.), non-determinism in a database (e.g., auto increment ids), and whether a given transaction aborts.

Replay systems commonly record non-determinism during online execution and then, during replay, supply the recorded information in response to a non-deterministic call (see §2.2.1 for references). orochi does this too. Specifically, orochi adds a fourth report type (§3.2): non-deterministic information, such as the return values of certain PHP built-in invocations. The server collects these reports by wrapping the relevant PHP statements (as in §3.3.4).

But, because reports are untrusted, orochi's verifier also *checks* the reported non-determinism against expected behavior. For example, the verifier checks that queries about time are monotonically increasing and that the process id is constant within requests. For random numbers, the application could seed a pseudorandom number generator, and the seed would be the non-deterministic report, though we have not implemented this.

Unfortunately, we cannot give rigorous guarantees about the efficacy of these checks, as our definitions and proofs Appendix B do not capture this kind of non-determinism. This is disappointing, but the issue seems fundamental, unless we pull the semantics of PHP into our proofs. Furthermore, this issue exists in all systems that "check" an untrusted lower layer's return values for validity [63, 72, 94, 133, 217].

Beyond that, the server gets discretion over the thread schedule, which is a kind of non-determinism, albeit one that is captured by our definitions and proofs Appendix B. As an example, if the web service performs a lottery, the server could delay responding to a collection of requests, invoke the random number library, choose which request wins, and then arrange the reports and responses accordingly.

### 3.3.7 Implementation details

Figure 3.7 depicts the main components of orochi.

| OROCHI component | Base | LOC written/changed |
|---|---|---|
| Server PHP (§3.3.3) | HHVM [46] | 400 lines of C++ |
| Acc-PHP (§3.3.3–§3.3.6) | HHVM [46] | 13k lines of C++ |
| Record library (§3.3.4, §3.3.6) | N/A | 1.6k lines of PHP |
| DB logging (§3.3.4) | MySQL | 320 lines of C++ |
| In-memory versioned DB (§3.3.5) | SQLite | 1.8k lines of C++ |
| Other audit logic (§3.2, §3.3) | N/A | 2.5k lines of C++/PHP/Bash |
| Rewriting tool (§3.3.7) | N/A | 470 lines of Python, Bash |

Figure 3.7: OROCHI's software components.

A rewrite tool performs required PHP application modifications: inserting wrappers (§3.3.4, §3.3.6), and adding hooks to record control flow digests and maximum operation number. Given some engineering, this rewriting can be fully automatic; our implementation sometimes needs manual help.

To log DB operations (§3.3.4), the server's PHP runtime passes (*rid*, *opnum*) in the comment field of a SQL query; our code in MySQL (v5.6) assigns a unique sequence number to the query (or transaction), necessitating minor synchronization. Each DB connection locally logs its queries in *sub-logs*; later, a stitching daemon merges these sub-logs to create the database operation log.

OROCHI's versioned DB implementation (§3.3.5) borrows Warp's [92] schema, and uses the same query rewriting technique. We implemented OROCHI's audit-time key-value store as a new component (in acc-PHP) to provide a versioned put/get interface.

Acc-PHP has several implementation limitations. One is the limited handling of mixed types, mentioned earlier (§3.3.3); another is that an object that points to itself (such as `$a->b->a`) is not recognized as such, if the object is a multivalue. When acc-PHP encounters such cases, it re-executes requests separately. In addition, acc-PHP runs with a maximum number of requests in a control flow group (3,000 in our implementation); this is because the memory consumed by larger sizes would cause thrashing and slow down re-execution.

In OROCHI, the server must be drained prior to an audit, but this is not fundamental; natural extensions of the algorithms would handle prefixes or suffixes of requests' executions.

| App | audit speedup | server CPU overhead | avg request | reports (per request) baseline | reports (per request) OROCHI | reports (per request) OROCHI ovhd | DB overhead temp | DB overhead permanent |
|---|---|---|---|---|---|---|---|---|
| MediaWiki | 10.9× | 4.7% | 7.1KB | 0.8KB | 1.7KB | 11.4% | 1.0× | 1× |
| phpBB | 5.6× | 8.6% | 5.7KB | 0.1KB | 0.3KB | 2.7% | 1.7× | 1× |
| HotCRP | 6.2× | 5.9% | 3.2KB | 0.0KB | 0.4KB | 10.9% | 1.5× | 1× |

Figure 3.8: OROCHI compared to simple re-execution (§3.4.1). "Audit speedup" is the ratio of audit-time CPU costs, assuming (conservatively) that auditing in simple re-execution is the same cost as serving the legacy application, and (perhaps optimistically) that simple re-execution and OROCHI are given HTTP requests and responses from the trace collector. "Server CPU overhead" is the CPU cost added by OROCHI, conservatively assuming that the baseline imposes no server CPU costs. The reports are compressed (OROCHI's overheads include the CPU cost of compression/decompression; the baseline is not charged for this). "OROCHI ovhd" in those columns is the ratio of (the trace plus OROCHI's reports) to (the trace plus the baseline's reports). "Temp" DB overhead refers to the ratio of the size of the on-disk versioned DB (§3.3.5) to the size of a non-versioned DB.

## 3.4 Evaluation of OROCHI

This section answers the following questions:

- How do OROCHI's verifier speedup and server overhead compare to a baseline of simple re-execution? (§3.4.1)
- What are the sources of acceleration? (§3.4.2)
- What is the "price of verifiability", meaning OROCHI's costs compared to the legacy configuration? (§3.4.3)
- What kinds of web applications work with OROCHI? (§3.4.4)

**Applications and workloads.** We answer the first two questions with experiments, which use three applications: MediaWiki (a wiki used by Wikipedia and others), phpBB (an open source bulletin board), and HotCRP (a conference review application). These applications stress different workloads. Also, MediaWiki and phpBB are in common use, and HotCRP has become a reference point for systems security publications that deal with PHP-based web applications [92, 144, 172, 175, 181, 212]. Indeed, MediaWiki and HotCRP are the two applications evaluated by Poirot [144] (§2.2.1). Our experimental workloads are as follows:

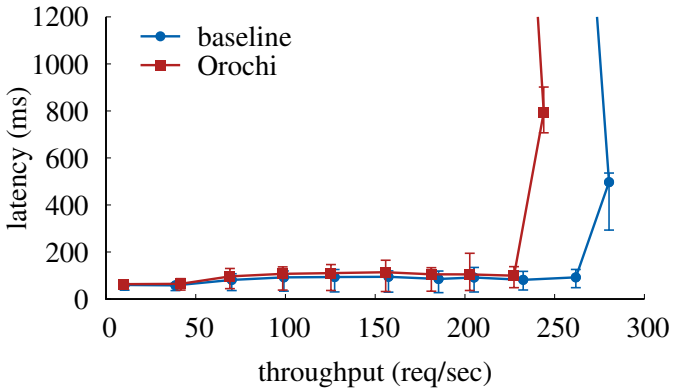



Figure 3.9: OROCHI compared to simple re-execution (§3.4.1). Latency vs. server throughput for phpBB (the other two workloads are similar). Points are 90th (bars are 50th and 99th) percentile latency for a given request rate, generated by a Poisson process. The depicted data are the medians of their respective statistics over 5 runs.

*MediaWiki* (v1.26.2). Our workload is derived from a 2007 Wikipedia trace, which we downsampled to 20,000 requests to 200 pages, while retaining its Zipf distribution ($\beta = 0.53$) [199]. We used a 10 year-old trace because we were unable to find something more recent; we downsampled because the original has billions of requests to millions of pages, which is too large for our testbed (on the other hand, smaller workloads produce fewer batching opportunities so are pessimistic to OROCHI).

*phpBB* (v3.2.0). On September 21, 2017, we pulled posts created over the preceding week from a real-world phpBB instance: CentOS [9]. We chose the most popular topic. There were 63 posts, tens to thousands of views per post, and zero to tens of replies per post. We assume that the ratio of page views from registered users (who log in) to guests (who do not) is 1:40, based on sampling reports from the forum (4–9 registered users and 200–414 guests were online). We create 83 users (the number of distinct users in the posts) to view and reply to the posts. The workload contains 30k requests.

*HotCRP*. We build a workload from 269 papers, 58 reviewers, and 820 reviews, with average review length of 3625 characters; the numbers are from SIGCOMM 2009 [38, 168]. We impose synthetic parameters: one registered author submits one valid paper, with a number of updates distributed uniformly from 1 to 20; each paper gets 3 reviews; each reviewer submits two versions

of each review; and each reviewer views 100 pages. In all, there are 52k requests.

As detailed later (§3.4.4), we made relatively small modifications to these applications. A limitation of our investigation is that all modeled clients use the same browser; however, our preliminary investigation indicates that PHP control flow is insensitive to browser details.

**Setup and measurement.** Our testbed comprises two machines connected to a switch. Each machine has a 3.3GHz Intel i5-6600 (4-core) CPU with 16GB memory and a 250GB SSD, and runs Ubuntu 14.04. One of the machines alternates between the roles of server (running Nginx 1.4.6) and verifier; the other generates load. We measure CPU costs from Linux's `/proc`. We measure throughput and latency at the client.

### 3.4.1 OROCHI VERSUS THE BASELINE

**What is the baseline?** We want to compare OROCHI to a system that audits comprehensively without trusting reports. A possibility is probabilistic proofs [76, 85, 99, 170, 186, 204], but they cannot handle our workloads, so we would have to estimate, and the estimates would yield outlandish speedups for OROCHI (over $10^6\times$). Another option is untrusted full-machine replay, as in AVM [128]. However, AVM's implementation supports only single-core servers, and handling untrusted reports *and* concurrency in VM replay might require research.

Instead, we evaluate against a baseline that is less expensive than both of these approaches, and hence is pessimistic to OROCHI: the legacy application (without OROCHI), which can be seen as a lower bound on hypothetical *simple re-execution*.

We capture this baseline's *audit-time CPU cost* by measuring the legacy server CPU costs; in reality, an audit not designed for acceleration would likely proceed more slowly. We assume this baseline has no *server CPU overhead*; in reality, the baseline would have some overhead. We capture the baseline's *report size* with OROCHI's non-deterministic reports (§3.3.6), because record-replay systems need non-deterministic advice; in reality, the baseline would likely need additional reports to reconstruct the thread schedule. Finally, we assume that the baseline tolerates arbitrary database configurations (unlike OROCHI; §3.3.4), although we assume that the baseline needs to reconstruct the database (as in OROCHI).

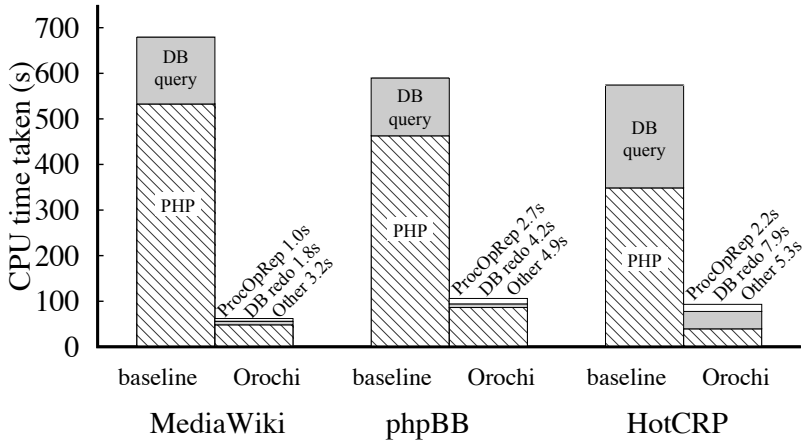



Figure 3.10: Decomposition of audit-time CPU costs. "PHP" (in OROCHI) is the time to perform SIMD-on-demand execution (§3.2.1,§3.3.3) and simulate-and-check (§3.2.3,§3.3.4). "DB query" is the time spent on DB queries during re-execution (§3.3.5). "ProcOpRep" is the time to execute the logic in Figures 3.5 and 3.6. "DB redo" is the time to reconstruct the versioned storage (§3.3.5). "Other" includes miscellaneous costs such as initializing inputs as multivalues, output comparison, etc.

**Comparison.** Figure 3.8 compares OROCHI to the aforementioned baseline. At a high level, OROCHI accelerates the audit compared to the baseline (we delve into this in §3.4.2) but introduces some server CPU cost, with some degradation in throughput, and minor degradation in latency.

The throughput reductions are respectively 13.0%, 11.1% and 17.8% for phpBB, MediaWiki, and HotCRP. The throughput comparison includes the effect of requiring strict serializability (§3.3.4), because the baseline's databases are configured with MySQL's default isolation level (repeatable read).

The report overhead depends on the frequency of object operations (§3.3.4) and non-deterministic calls (§3.3.6). Still, the report size is generally a small fraction of the size of the trace, as is OROCHI's "report overhead" versus the baseline. OROCHI's audit-time DB storage requirement is higher than the baseline's, because of versioning (§3.3.5), but after the audit, OROCHI needs only the "latest" state (for the next audit).

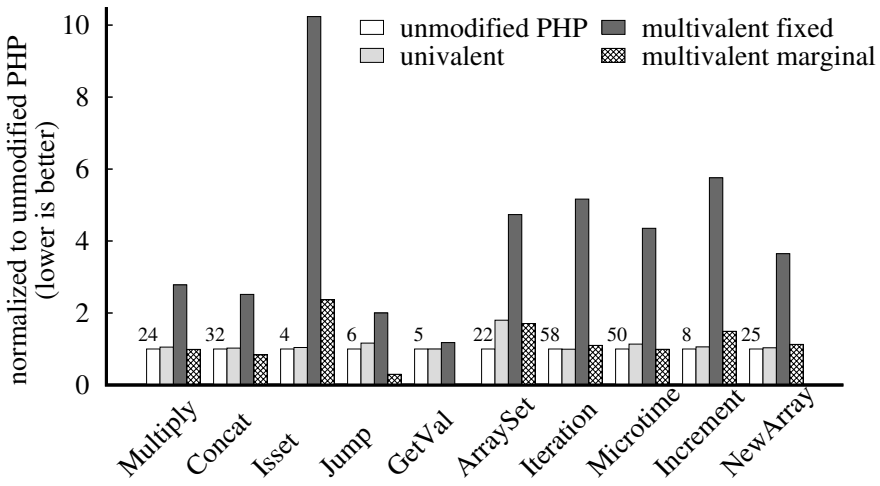



Figure 3.11: Cost of various instructions in unmodified PHP and acc-PHP (§3.3.3). Execution times are normalized clusterwise to unmodified PHP, for which the absolute time is given (in $\mu$s). See text for interpretation.

### 3.4.2 A closer look at acceleration

Figure 3.10 decomposes the audit-time CPU costs. The "DB query" portion illustrates query deduplication (§3.3.5). Without this technique, every DB operation would have to be re-issued during re-execution. (OROCHI's verifier re-issues every register and key-value operation, but these are inexpensive.) Query deduplication is more effective when the workload is read-dominated, as in our MediaWiki experiment.

We now investigate the sources of PHP acceleration; we wish to know the costs and benefits of univalent and multivalent instructions (§3.2.1, §3.3.3). We divide the 100+ PHP byte code instructions into 10 categories (arithmetic, container, control flow, etc.); choose category representatives; and run a microbenchmark that performs $10^7$ invocations of the instruction and computes the average cost. We run each microbenchmark against unmodified PHP, acc-PHP with univalent instructions, and acc-PHP with multivalent instructions; we decompose the latter into marginal cost (the cost of an additional request in the group) and fixed cost (the cost if acc-PHP were maintaining a multivalue with zero requests).

Figure 3.11 depicts the results. The fixed cost of multivalent instructions is high, and the marginal cost is sometimes worse than the unmodified baseline. In general, multivalent execution

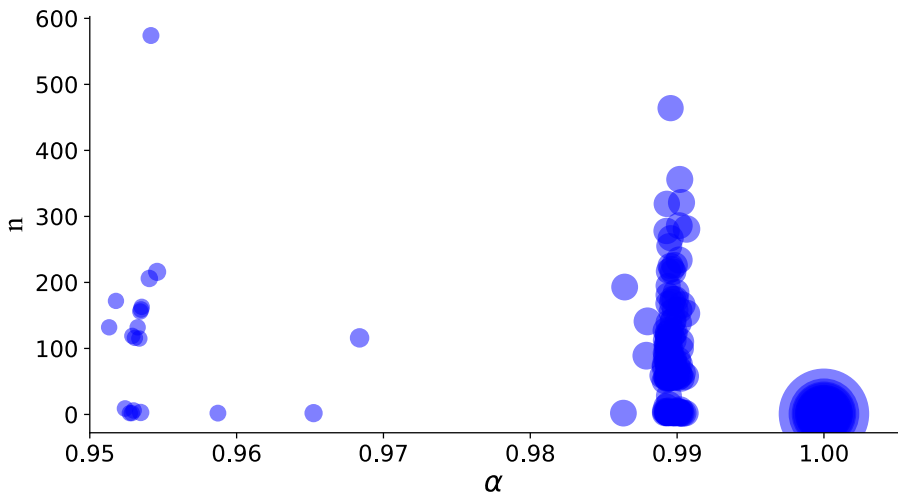



Figure 3.12: Characteristics of control flow groups in the MediaWiki workload. Each bubble is a control flow group; the center of a bubble gives the group's $n$ (number of requests in the group) and $\alpha$ (proportion of univalent instructions); the size of a bubble is proportional to $\ell$ (number of instructions in the group). This workload has 527 total groups (bubbles), 237 groups with $n > 1$, and 200 unique URLs. All groups have $\alpha > 0.95$; only the occupied portion of the $x$-axis is depicted.

is *worse* than simply executing the instruction $n$ times![2] So how does OROCHI accelerate? We hypothesize that (i) many requests share control flow, and (ii) within a shared control flow group, the vast majority of instructions are executed univalently. If this holds, then the gain of SIMD-on-demand execution comes not from the "SIMD" part but rather from the "on demand" part: the opportunistic collapsing of multivalues enables a lot of deduplication.

To confirm the hypothesis, we analyze all of the control flow groups in our workloads. Each group $c$ is assigned a triple $(n_c, \alpha_c, \ell_c)$, where $n_c$ is the number of requests in the group, $\alpha_c$ is the proportion of univalent instructions in that group, and $\ell_c$ is the number of instructions in the group. (Note that if $n_c = 1$, then $\alpha_c = 1.0$.) Figure 3.12 depicts these triples for the MediaWiki workload. There are many groups with high $n_c$, and most groups have very high $\alpha_c$ (the same holds for the other two workloads), confirming our hypothesis. Something else to note is a slight negative correlation between $n_c$ and $\alpha_c$ within a workload, which is not ideal for OROCHI.

[2]One might wonder: would it be better to batch by control flow *and* identical inputs? No; that approach still produces multivalent executions because of shared object reads and non-determinism, and the batch sizes are smaller.

### 3.4.3 The price of verifiability

We now take stock of OROCHI's total overhead by comparing OROCHI to the *legacy configuration*. OROCHI introduces a modest cost to the server: 4.7%–8.6% CPU overhead (Figure 3.8) and temporary storage for trace and reports. But the main price of verifiability is the verifier's resources:

*CPU.* Since the verifier's audit-time CPU costs are between 1/5.6 and 1/10.9 those of the server's costs (per §3.4.1, Figure 3.8), OROCHI requires that the verifier have 9.1%–18.0% of the CPU capacity that the server does.

*Storage.* The verifier has to store the database between audits, so the verifier effectively maintains a copy of the database. During the audit, the verifier also stores the trace, reports, and additional DB state (the versioning information).

*Network.* The verifier receives the trace and reports over the network. Note that in the Dana (§3) and Pat (§3.3.1) scenarios, the principal is already paying (on behalf of clients or the server, respectively) to send requests and responses over the wide area network—which likely swamps the cost of sending the same data to the verifier over a local network.

### 3.4.4 Compatibility

We performed an informal survey of popular PHP applications to understand the effect of OROCHI's two major compatibility restrictions: the non-verification of interactions with other applications (§3.3.1) and the non-nesting of object operations inside DB transactions (§3.3.4).

We sorted GitHub Trending by stars in decreasing order, filtered for PHP applications (filtering out projects that are libraries or plugins), and chose the top 10: Wordpress, Piwik, Cachet, October, Paperwork, Magento2, Pagekit, Lychee, Opencart, and Drupal. We inspected the code (and its configuration and documentation), ran it, and logged object operations. For eight of them, the sole external service is email; the other two (Magento2 and Opencart) additionally interact with a payment server. Also, all but Drupal and October are consistent with the DB requirement.

This study does *not* imply that OROCHI runs with these applications out of the box. It generally takes some adjustment to fit an application to OROCHI, as we outline below.

MediaWiki does not obey the DB requirement. We modified it so that requests read in the relevant APC keys (which we obtain through static inspection plus a dynamic list of needed keys, itself stored in the APC), execute against a local cache of those keys, and flush them back to the APC. This gives up some consistency in the APC, but MediaWiki anyway assumes that the APC is providing loose consistency. We made several other minor modifications to MediaWiki; for example, changing an absolute path (stored in the database) to a relative one. In all, we modified 346 lines of MediaWiki (of 410k total and 74k invoked in our experiments).

We also modified phpBB (270 lines, of 300k total and 44k invoked), to address a SQL parsing difference between the actual database (§3.3.4) and the in-memory one (§3.3.5) and to create more audit-time acceleration opportunities (by reducing the frequency of updates to login times and page view counters). We modify HotCRP (67 lines, of 53k total and 37k invoked), mainly to rewrite `select * from` queries to request individual columns; the original would fetch the begin/end timestamp columns in the versioned DB (§3.3.5, §3.3.7).

### 3.4.5 Discussion and limitations of orochi

Below we summarize orochi and discuss its limitations.

*Guarantees.* orochi is based on ssco, which has provable properties. However, orochi does not provide ssco's idealized Soundness guarantee (§3.1), because of the leeway discussed earlier (§3.3.6). And observable differences in the verifier's and server's stacks (§3.3.1) would make orochi fall short of ssco's idealized Completeness guarantee.

*Performance and price.* Relative to a pessimistic baseline, orochi's verifier accelerates by factors between 5.6–10.9× in our experiments, and server overhead is below 10% (§3.4.1). The CPU costs introduced by orochi are small, compared to what one sometimes sees in secure systems research; one reason is that orochi is not based on cryptography. And while the biggest percentage cost for the verifier is storage (because the verifier has to duplicate it; §3.4.3), storage is generally inexpensive in dollar terms.

*Compatibility and usability.* On the one hand, orochi is limited to a class of applications, as discussed (§3.3.1, §3.4.4). On the other hand, the applications in our experiments—which were

largely chosen by following prior work (discussed early in §3.4)—did not require much modification (§3.4.4). Best of all, OROCHI is fully compatible with today's *infrastructure*: it works with today's end-clients and cloud offerings as-is.

Of course, OROCHI would benefit from extensions. All of the applications we surveyed make requests of an email server (§3.4.4). We could verify those requests—but not the email server itself; that is future work—with a modest addition to OROCHI, namely treating external requests as another kind of response. This would require capturing the requests themselves; that could be done, in Pat's scenario (§3.3.1), by the trace collector or, in Dana's scenario (§3), by redirecting email to a trusted proxy on the verifier.

Another extension is adding a *file* abstraction to our three object types (§3.3.4). This isn't crucial—many applications, including five of the 10 in our survey (§3.4.4), can be configured to use alternatives such as a key-value store—but some deployers might prefer a file system backend. Another extension is filtering large objects from the trace, before it is delivered to the verifier. A possible solution is to leverage browser support for Resource Integrity: the verifier would check that the correct *digest* was supplied to the browser, leaving the actual object check to the browser. Other future work is HTTPS; one option is for the server to record non-deterministic cryptographic input, and the verifier uses it to recover the plaintext stream.

A more fundamental limitation is that if OROCHI's verifier does not have a trace from a period (for example, before OROCHI was deployed on a given server), then OROCHI can verify only by getting the pre-OROCHI collection of objects from the server (requiring a large download) and treating those objects as the true initial state (requiring trust).

# 4 | Verifying Serializability of Black-box Databases

This chapter focuses on auditing another cloud service—cloud databases, which is one of the most popular services, for example, in AWS [6]. Specifically, we aim at verifying *serializability* [79, 167] of black-box databases.

**Why serializabiliy?** First, serializability is the gold-standard isolation level [69] that implies all transactions (a group of operations) appear to execute in a single, sequential order. It is also the one that many applications and programmers implicitly assume: their code would be incorrect if the database provided a weaker contract [141, 205]. Note that serializability is a correctness contract. For example, serializability implies basic integrity: if a returned value does not read from a valid write (which every value should), that will manifest as a non-serializable result. Serializabiliy also implies that the database handles failures robustly: non-tolerated server failures, particularly in the case of a distributed database, are a potential source of non-serializable results.

Second, a new class of cloud databases supporting serializability has emerged, including Amazon DynamoDB and Aurora [2, 3, 201], Azure CosmosDB [7], CockroachDB [11], YugaByte DB [51], and others [18, 20, 23, 24, 97]. This trend started in part because serializability prevents classes of application bugs [32].

However, a user of a cloud database can legitimately wonder whether the database provides the promised contract. Indeed, today's production systems have exhibited various serializability violations [1, 21, 22, 29, 31] (see also §4.6.1). Furthermore, users often have no visibility into a cloud database's implementation. In fact, even when the source code is available [11, 18, 20, 51],

that does not necessarily yield visibility: if the database is hosted by someone else, you can't really be sure of its operation. Meanwhile, any internal corruption—as could happen from misconfiguration, misoperation, compromise, or adversarial control at any layer of the execution stack—can cause a serializability violation. Beyond that, one need not adopt a paranoid stance ("the cloud as malicious adversary") to acknowledge that it is difficult, as a technical matter, to provide serializability *and* geo-distribution *and* geo-replication *and* high performance under various failures [56, 115, 219]. Doing so usually involves a consensus protocol that interacts with an atomic commit protocol [97, 147, 158]—a complex combination, and hence potentially bug-prone.

## 4.1 The underlying problem

The core question of this chapter is: *how can clients verify the serializability of a black-box database?* To be clear, related questions have been addressed before. The novelty in our problem is in combining three aspects:

**(a) Black box, unmodified database.** In our setting, the database does not "know" it's being checked; the input to the verification machinery will be only the inputs to, and outputs from, the database. This matches the cloud context (even when the database is open source, as noted above), and contrasts with work that checks for isolation or consistency anomalies by using "inside information" [87, 130, 166, 189, 197, 210, 216], for example, access to internal scheduling choices. On the client side, we target production workloads that use standard key-value APIs (§4.2), and we do not require prior knowledge of the workloads.

**(b) Serializability.** We focus on serializability, both its strict and non-strict variants, in contrast to weaker isolation levels. The difference between strict and non-strict serializability [80, 167] is that the strict variant has a real-time constraint, which dictates real-time happened-before relationships among transactions, and makes strict serializability computationally simpler to check (by, in effect, reducing the number of possibly-valid execution schedules).

Checking both strict and non-strict serializability has real-world applicability. Many systems today provide strict serializability, including MySQL, Google Spanner, and FaunaDB. At the same

time, an increasing number of production systems provide non-strict serializability, for performance reasons [155]. In PostgreSQL, for example, serializability is implemented not with two-phase locking but with the SSI technique [90], which does not yield strict serializability (see the excellent report by Ports and Grittner [176] for details). Other examples are CockroachDB (its consistency model, while stronger than serializability, is weaker than strict serializability [13]); and YugaByte DB (it provides serializability rather than strict serializability, if there is clock skew [50]).

Most of the intellectual work and effort enters in checking non-strict serializability. That said, even the strict case has aspects of the non-strict case, if the real-time relationships leave the ordering relatively unconstrained. For example, under sufficient concurrency, a workload's non-strict portion—the transactions that are concurrent and have no real-time happened-before relationships—can be large. As a special case of concurrency, under clock drift (see §4.3.5), many transactions would be considered as concurrent, thus makes checking strict serializability computationally expensive (§4.6.1).

**(c) Scalability.** This means, first, scaling to real-world online transactional processing workloads at reasonable cost. It also means incorporating mechanisms that enable a verifier to work incrementally and to keep up with an ever-growing history.

However, aspects (a) and (b) set up a challenge: checking black-box serializability is NP-complete, as Papadimitriou proved 41 years ago [167]. Recent work of Biswas and Enea (BE) [83] lowered the complexity to polynomial time, under natural restrictions (which hold in our context); see also pioneering work by Sinha et al. [190]. However, these two approaches don't meet our goal of scalability. For example, in BE, the number of clients appears in the exponent of the algorithm's running time (§4.6) (e.g., 14 clients means the algorithm is $O(n^{14})$). Furthermore, even if there were a small number of clients, BE does not include mechanisms for handling a continuous and ever-growing history.

Despite the computational complexity, there is cause for hope: one of the remarkable aspects in the field of formal verification has been the use of heuristics to "solve" problems whose general form is intractable. This owes to major advances in solvers (advanced SAT and SMT solvers) [70,

81, 89, 106, 124, 154, 164, 196], coupled with an explosion of computing power. Thus, our guiding intuition is that it ought to be possible to verify serializability in many real-world cases. This chapter describes a system called COBRA, which starts from this intuition, and provides a solution to the problem posed by (a)–(c).

As stated in Section 1.2, COBRA applies to transactional key-value stores (everywhere in this chapter it says "database", this is what we mean). COBRA consists of a third-party, unmodified database that is not assumed to "cooperate"; a set of legacy database clients that COBRA modifies to link to a library; one or more *history collectors* that are assumed to record the actual requests to and responses from the database; and a *verifier* that comprehensively checks serializability, in a way that "keeps up" with the database's (average) load. The database is untrusted while the clients, collectors, and verifier are all in the same trust domain (for example, deployed by the same organization). Section 4.2 further details the setup and gives example scenarios. COBRA solves two main problems:

*1. Efficient witness search (§4.3).* A brute-force way to validate serializability is to demonstrate the existence of a graph $G$ whose nodes are transactions in the history and whose edges meet certain constraints, one of which is acyclicity (§4.2.3). From our starting intuition and the structure of the constraints, we are motivated to use a SAT or SMT solver [49, 71, 106, 195]. But complications arise. To begin with, encoding acyclicity in a SAT instance brings overhead [116, 117, 140] (we see this too; §4.6.1). Instead, COBRA uses a recent SMT solver, MonoSAT [73], that is well-suited to checking graph properties (§4.3.4). However, using MonoSAT alone on the aforementioned brute-force search problem is still too expensive (§4.6.1).

To address this issue, COBRA develops domain-specific pruning techniques and reduces the search problem size. First, COBRA introduces a new encoding that exploits common patterns in real workloads, such as read-modify-write transactions, to efficiently infer ordering relationships from a history (§4.3.1–§4.3.2). (We prove that COBRA's encoding is a valid reduction in Appendix C.1.) Second, COBRA uses parallel hardware (our implementation uses GPUs; §4.5) to compute *all-paths reachability* over a graph whose nodes are transactions and whose edges are known happened-before relationships; then, COBRA is able to efficiently resolve some of the con-

straints, by testing whether a candidate edge would generate a cycle with an existing path.

*2. Scaling to a continuous and ever-growing history (§4.4).* Online cloud databases run in a continuous fashion, where the corresponding history is uninterrupted and grows unboundedly. To support online databases, COBRA verifies in rounds. From round-to-round, the verifier checks serializability on a portion of the history. However, the challenge is that the verifier seemingly needs to involve all history, because serializability does not respect real-time ordering, so future transactions can read from values that (in a real-time view) have been overwritten. To solve this problem, clients issue periodic *fence transactions* (§4.4.2). The epochs impose coarse-grained synchronization, creating a window from which future reads, if they are to be serializable, are permitted to read. This allows the verifier to discard transactions prior to the window.

To be clear, COBRA verifies a given history, not a database implementation per se. Also, COBRA performs ex post facto verification, which cannot prevent databases' misbehavior from happening but nonetheless has real-world applicability (see applicable scenarios in §4.2.1 and more cases in Concerto [62, §1.1]).

We implement COBRA (§4.5) and experiment with it on production databases with various workloads (§4.6). COBRA detects all serializability violations we collect from real systems' bug reports. COBRA's core (single-round) verification improves on baselines by 10× in the problem size it can handle for a given time budget. For example, COBRA finishes checking 10k transactions in 14 seconds, whereas baselines can handle only 1k or less in the same time budget. For an online database with continuous traffic, COBRA achieves a sustainable verification throughput of 2k txn/sec on the workloads that we experiment with (this corresponds to a workload of 170M/day; for comparison, Apple Pay handles 33M txn/day [5], and Visa handles 150M txn/day [47]). COBRA imposes modest overhead (3.7% throughput degradation and 7.0% 90-percentile latency increases for PostgreSQL).

## 4.2 Overview and technical background

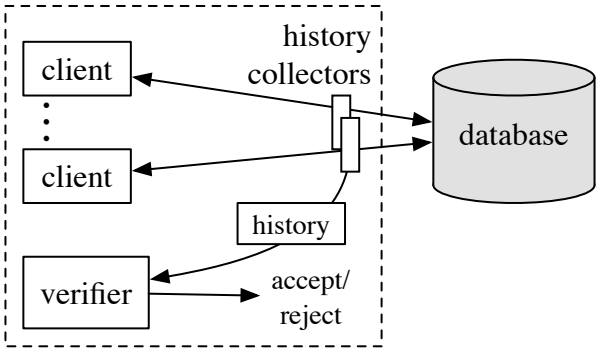



Figure 4.1: COBRA's architecture. The dashed rectangle is a trust domain. The verifier is off the critical path but must keep up on average.

### 4.2.1 Architecture and scenarios.

Figure 4.1 depicts COBRA's high-level architecture. *Clients* issue requests to a *database* (a transactional key-value store) and receive results. The database is untrusted: the results can be arbitrary.

Each client request is one of five operations: start, commit, abort (which refer to *transactions*), and read and write (which refer to *keys*).

A set of *history collectors* sit between clients and the database, and capture the requests that clients issue and the (possibly wrong) results delivered by the database. This capture is a *fragment of a history*. A *history* is a set of operations; it is the union of the fragments from all collectors.

A *verifier* retrieves history fragments from collectors and attempts to verify whether the history is *serializable*; we make this term precise below (§4.2.2). The verifier requires the full history including all requests to, and responses from, the database. As stated in Section 1.2, that assumes no crash in the collectors and the verifier (see also §5). The requirement of all requests and responses is fundamental: if we are auditing a database's outputs, then we need to know those.

The verifier proceeds verification in *rounds*; each round consists of a witness search, the input to which is logically the output of the previous round and new history fragments. The verifier must work against an online and continuously available database; however, the verifier performs its work in the background, off the critical path.

Clients, history collectors, and the verifier are in the same trust domain. This architecture is relevant in real-world scenarios. Consider for example an enterprise web application whose end-

users are geo-distributed employees of the enterprise. Owing to legacy software and systems integration issues, the application servers run on the enterprise's hardware while for performance, fault-tolerance, and durability, the back-end of the web application is a database is operated by a cloud provider [33]. Note that our *clients* are the application servers, as clients of the database. A similarly structured example is online gaming, where the main program runs on company-owned servers while the user data is stored in a cloud database [27].

In these scenarios, the verifier runs on hardware belonging to the trust domain. There are several options, meanwhile, for the collectors. Collectors can be middleboxes situated at the edge of the enterprise or gaming company, allowing them to capture the requests and responses between the database clients and the cloud database. Another option is to run the collector in an untrusted cloud, using a Trusted Execution Environment (TEE), such as Intel's SGX. Recent work has demonstrated such a collector [65], as a TLS proxy that logs inbound/outbound messages, thereby ensuring (via the fact that clients expect the server to present a valid SSL/TLS certificate) that clients' requests and responses are indeed logged.

**Verifier's performance requirement.** The verifier's performance will be reported as capacity (transactions/sec); this capacity must be at least the average offered load seen by the database over some long interval, for example a day. Note that the verifier's capacity need not match the database's *peak* load: because the verifier is off the critical path, it can catch up.

### 4.2.2 Verification problem statement

In the description below, we use Adya's formalism [53], with one difference: *Adya's history* includes write-write ordering for each key (called *version order*, defined below), which comes from the actual execution schedule, a piece of information within the database; in contrast, *cobra's history* (defined below) is collected outside the database so doesn't contain version order.

**Preliminaries.** First, assume that each value written to the database is unique, and each transaction reads and writes a key at most once; thus, any read can be associated with the unique transaction that issued the corresponding write. cobra discharges this assumption with logic in the cobra client library (§4.5) that embeds a unique id in each write and consumes the id

on a read.

A *history* is a set of read and write operations, each associated with a transaction. Each read operation must read from a particular write operation in the history. A *history is serializable* if it is *equivalent* to a *serial schedule* [53]. A serial schedule is a total order of all operations that does not have overlapping transactions. A history and a serial schedule are *equivalent*, if they have the same operations and have the same *reads-from* relationships; that is, for each read, it returns the same value.

An important notion is *version order*, denoted as $\ll$, which represents, for each key, a total order over all of the versions of this key written in a history. Another notion is *conflicts*: two operations (and the transactions that enclose them) are said to *conflict* if they access the same key from different transactions, and at least one of the operations is a write. With a version order, one can identify the ordering of all *conflicting* operations in a history: (i) conflicting writes are ordered by the version order $\ll$; (ii) a conflicting read $r_i$ and write $w_j$ have an ordering determined by the write-write order between $w_j$ and the write ($w_k$) that $r_i$ reads. That is, if $w_j \ll w_k$ or $w_j = w_k$, $r_i$ happens after $w_j$; otherwise, $r_i$ happens before $w_j$.

From a history $H$ and a version order $\ll$, one can construct a *serialization graph* [53] that has a vertex for every transaction in the history and a directed edge or path for every pair of conflicting transactions. An important fact is that the serialization graph is acyclic if and only if the history $H$ is serializable [78, §3.4]. Note that aborted transactions are not included in the serialization graph [53].

**The core problem.** Based on the immediately preceding fact, the question of *whether a history is serializable* can be converted to *whether there exists an acyclic serialization graph from the history and some version order*. So, the core problem is to identify such a serialization graph, or assert that none exists.

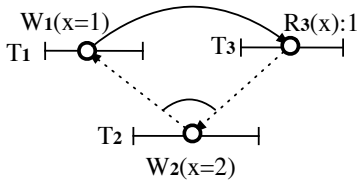
Notice that this question would be straightforward if the database revealed its version order (thus ruling out any other possible version order): one could construct the corresponding serialization graph, and test it for acyclicity. Indeed, this is the problem of testing *conflict* serializability [206].

In our context, where the database is a black box (§4.1) and the version order is unknown, we have to (implicitly) find version orders and test if the serialization graphs of those version orders are acyclic.

### 4.2.3 Starting point: intuition and brute force

This section describes a brute-force solution, which serves as the starting point for COBRA and gives intuition. The approach relies on a data structure called a *polygraph* [167], which captures all possible serialization graphs when the version order is unavailable.

In a polygraph, vertices ($V$) are transactions and edges ($E$) are read-dependencies (cases when a transaction writes a key, and another transaction reads the value written to that key). Note that read-dependencies are evident from the history because values are unique, by assumption (§4.2.2). There is a set, $C$, which we call *constraints*, that captures possible (but unknown) dependencies. Here is an example polygraph:




It has three vertices $V = \{T_1, T_2, T_3\}$, one known edge in $E = \{(T_1, T_3)\}$ from the known read-dependency $W_1(x) \xrightarrow{\text{wr(x)}} R_3(x)$, and one constraint $\langle\, (T_3,\ T_2),\ (T_2,\ T_1)\, \rangle$ which is shown as two dashed arrows connected by an arc. This constraint captures the fact that $T_2$ cannot happen in between $T_1$ and $T_3$, because otherwise $T_3$ should have read $x$ from $T_2$ instead of from $T_1$, which contradicts the fact that $T_3$ reads $x$ from $T_1$. Hence $T_2$ has to happen either after $T_3$ or before $T_1$, but it is unknown which option is the truth.

Formally, a *polygraph* $P = (V,\ E,\ C)$ is a directed graph $(V,\ E)$ together with a set of *bipaths*, $C$; that is, pairs of edges—not necessarily in $E$—of the form $\langle (v,\ u),\ (u,\ w) \rangle$ such that $(w,\ v) \in E$. A bipath of that form can be read as "either $u$ happened after $v$, or else $u$ happened before $w$".

Now, define *the polygraph* $(V,\ E,\ C)$ *associated with a history*, as follows [206]:

- $V$ are all committed transactions in the history
- $E = \{(T_i,\ T_j) \mid T_j \text{ reads from } T_i\}$; that is, $T_i \xrightarrow{\text{wr(x)}} T_j$, for some $x$.
- $C = \{\langle\ (T_j,\ T_k),\ (T_k,\ T_i)\ \rangle \mid (T_i \xrightarrow{\text{wr(x)}} T_j)\ \wedge$
  $(T_k \text{ writes to } x) \wedge T_k \neq T_i \wedge T_k \neq T_j\}$.

The edges in $E$ captures all read-dependencies, which as noted are evident from the history. $C$ captures how uncertainty is encoded into constraints. Specifically, for each read-dependency in the history, all other transactions that write the same key either happen before the given write or else after the given read.

A graph is called *compatible* with a polygraph if the graph has the same nodes and known edges in the polygraph, and the graph chooses one edge out of each constraint; one can think of such a graph as a "solution" to the constraint satisfaction problem posed by the polygraph. Formally, a graph $(V', E')$ is *compatible* with a polygraph $(V, E, C)$ if: $V = V'$, $E \subseteq E'$, and $\forall\langle e_1,\ e_2\rangle \in C,\ (e_1 \in E' \wedge e_2 \notin E') \vee (e_1 \notin E' \wedge e_2 \in E')$.

A crucial fact is: if a graph $G$ is compatible with a given polygraph (based on a history $H$), then $G$ is in fact a serialization graph (§4.2.2) for $H$ with some version order $\ll$ [206]. Furthermore, we have seen that a history $H$ is serializable (§4.2.2) iff there exists a version order $\ll$ such that a serialization graph from $H$ and $\ll$ is acyclic [78]. Putting these facts together yields a brute-force approach for verifying serializability: first, construct a polygraph from a history; second, search for a compatible graph that is acyclic. However, not only does this approach need to consider $|C|$ binary choices ($2^{|C|}$ possibilities) but also $|C|$ is massive: it is a sum of quadratic terms, specifically $\sum_{k \in \mathcal{K}} r_k \cdot (w_k - 1)$, where $\mathcal{K}$ is the set of keys in the history, and each $r_k$ and $w_k$ are the number of reads and writes of key $k$.

## 4.3 Verifying serializability in cobra

Figure 4.2 depicts the verifier and the major components of verification. This section covers one round of verification. As a simplification, assume that the round runs in a vacuum; Section 4.4 discusses how rounds are linked.

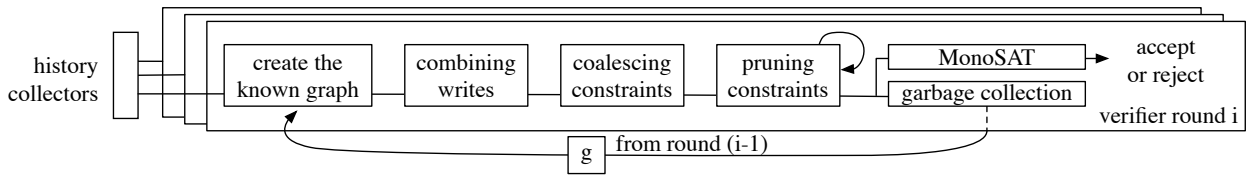



Figure 4.2: The verifier's process, within a round and across rounds.

COBRA uses an SMT solver geared to graph properties, specifically MonoSAT [73] (§4.3.4). Yet, despite MonoSAT's power, encoding the problem as in Section 4.2.3 would generate too much work for it (§4.6.1).

COBRA refines that encoding in several ways. It introduces *write combining* (§4.3.1) and *coalescing* (§4.3.2). These techniques are motivated by common patterns in workloads, and efficiently extract restrictions on the search space that are available in the history. COBRA's verifier also does its own inference (§4.3.3), prior to invoking the solver. This is motivated by observing that (a) having all-pairs reachability information (in the "known edges") yields quick resolution of many constraints, and (b) computing that information is amenable to acceleration on parallel hardware such as GPUs (the computation is iterated matrix multiplication; §4.5).

Figure 4.3 depicts the algorithm that constructs COBRA's encoding and shows how the techniques combine. Note that COBRA relies on a generalized notion of constraints. Whereas previously a constraint was a pair of edges, now a constraint is a pair of *sets of edges*. Meeting a constraint $\langle A, B\rangle$ means including *all* edges in $A$ and excluding *all* in $B$, or vice versa. More formally, we say that a graph $(V', E')$ is *compatible* with a known graph $G = (V, E)$ and generalized constraints $C$ if: $V = V'$, $E \subseteq E'$, and $\forall \langle A, B\rangle \in C, (A \subseteq E' \wedge B \cap E' = \emptyset) \vee (A \cap E' = \emptyset \wedge B \subseteq E')$.

We prove the validity of COBRA's encoding in Appendix C.1. Specifically we prove that *there exists an acyclic graph that is compatible with the constraints constructed by COBRA on a given history if and only if the history is serializable.*

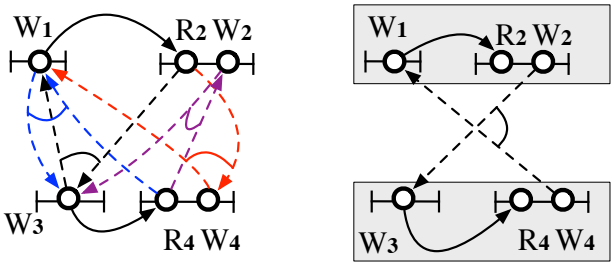
### 4.3.1 Combining writes

COBRA exploits the read-modify-write (RMW) pattern, in which a transaction reads a key and then writes the same key. The pattern is common in real-world scenarios, for example shopping:

```
1: procedure ConstructEncoding(history)
2:     g, readfrom, wwpairs ← CreateKnownGraph(history)
3:     con ← GenConstraints(g, readfrom, wwpairs)
4:     // §4.3.3, executed one or more times
5:     con, g ← Prune(con, g)
6:     return con, g
7:
8: procedure CreateKnownGraph(history)
9:     g ← empty Graph                          // the known graph
10:    wwpairs ← Map {⟨Key, Tx⟩ → Tx} // consecutive writes
11:    // maps a write to its readers
12:    readfrom ← Map {⟨Key, Tx⟩ → Set⟨Tx⟩}
13:    for transaction tx in history :
14:        g.Nodes += tx
15:        for read operation rop in tx :
16:            // "rf_tx" is the tx rop reads from
17:            g.Edges += (rop.rf_tx, tx) // wr-edge
18:            readfrom[⟨rop.key, rop.rf_tx⟩] += tx
19:
20:        // detect read-modify-write transactions
21:        for Keys key that tx both reads and writes :
22:            rop ← the operation in tx that reads key
23:            if wwpairs[⟨key, rop.rf_tx⟩] ≠ null :
24:                REJECT // write overwritten twice, not SER
25:            wwpairs[⟨key, rop.rf_tx⟩] ← tx
26:
27:    return g, readfrom, wwpairs
28:
29: procedure GenConstraints(g, readfrom, wwpairs)
30:    // each key maps to set of chains; each chain is an ordered list
31:    chains ← empty Map {Key → Set⟨List⟩}
32:    for transaction tx in g :
33:        for write wrop in tx :
34:            chains[wrop.key] += [ tx ]      // one-element list
35:
36:    CombineWrites(chains, wwpairs)     // §4.3.1
37:    InferRWEdges(chains, readfrom, g) // infer anti-dependency
38:
39:    con ← empty Set
40:    for ⟨key, chainset⟩ in chains :
41:        for every pair {chain_i, chain_j} in chainset :
42:            con += Coalesce(chain_i, chain_j, key, readfrom) // §4.3.2
43:
44:    return con
45: procedure CombineWrites(chains, wwpairs)
46:    for ⟨key, tx_1, tx_2⟩ in wwpairs :
47:        // By construction of wwpairs, tx_1 is the write immediately
48:        // preceding tx_2 on key. Thus, we can sequence all writes
49:        // prior to tx_1 before all writes after tx_2, as follows:
50:        chain_1 ← the list in chains[key] whose last elem is tx_1
51:        chain_2 ← the list in chains[key] whose first elem is tx_2
52:        chains[key] \= {chain_1, chain_2}
53:        chains[key] += concat(chain_1, chain_2)
54:
55: procedure InferRWEdges(chains, readfrom, g)
56:    for ⟨key, chainset⟩ in chains :
57:        for chain in chainset :
58:            for i in [0, length(chain) − 2] :
59:                for rtx in readfrom[⟨key, chain[i]⟩] :
60:                    if (rtx ≠ chain[i+1]): g.Edges += (rtx, chain[i+1])
61:
62: procedure Coalesce(chain_1, chain_2, key, readfrom)
63:    edge_set_1 ← GenChainToChainEdges(chain_1, chain_2, key, readfrom)
64:    edge_set_2 ← GenChainToChainEdges(chain_2, chain_1, key, readfrom)
65:    return ⟨edge_set_1, edge_set_2⟩
66:
67: procedure GenChainToChainEdges(chain_i, chain_j, key, readfrom)
68:    if readfrom[⟨key, chain_i.tail⟩] = ∅ :
69:        edge_set ← {(chain_i.tail, chain_j.head)}
70:        return edge_set
71:
72:    edge_set ← empty Set
73:    for rtx in readfrom[⟨key, chain_i.tail⟩] :
74:        edge_set += (rtx, chain_j.head)
75:    return edge_set
76:
77: procedure Prune(con, g)
78:    // tr is the transitive closure (reachability of two nodes) of g
79:    tr ← TransitiveClosure(g) // standard algorithm; see [98, Ch.25]
80:    for c = ⟨edge_set_1, edge_set_2⟩ in con :
81:        if ∃(tx_i, tx_j) ∈ edge_set_1 s.t. tx_j ⇝ tx_i in tr :
82:            g.Edges ← g.Edges ∪ edge_set_2
83:            con −= c
84:        else if ∃(tx_i, tx_j) ∈ edge_set_2 s.t. tx_j ⇝ tx_i in tr :
85:            g.Edges ← g.Edges ∪ edge_set_1
86:            con −= c
87:    return con, g
```

Figure 4.3: Cobra's procedure for converting a history into a constraint satisfaction problem (§4.3). After this procedure, cobra feeds the results (a graph of known edges $G$ and set of constraints $C$) to a constraint solver (§4.3.4), which searches for a graph that includes the known edges from $G$, meets the constraints in $C$, and is acyclic. We prove the algorithm's validity in Appendix C.1.

in one transaction, get the number of an item in stock, decrement, and write back the number. COBRA uses RMWs to impose order on writes; this reduces the orderings that the verification procedure would otherwise have to consider. Here is an example:




There are four transactions, all operating on the same key. Two of the transactions are RMW, namely $R_2$, $W_2$ and $R_4$, $W_4$. On the left is the basic polygraph (§4.2.3); it has four constraints (each in a different color), which are derived from considering read-dependencies.

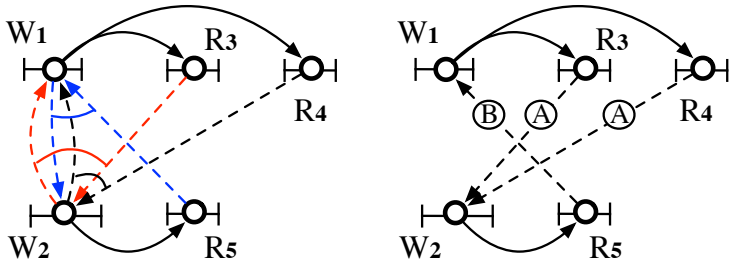
COBRA, however, infers *chains*. A single chain comprises a *sequence of transactions whose write operations are consecutive*; in the figure, a chain is indicated by a shaded area. Notice that the only ordering possibilities exist at the granularity of chains (rather than individual writes); in the example, the two possibilities of course are $[W_1, W_2] \rightarrow [W_3, W_4]$ and $[W_3, W_4] \rightarrow [W_1, W_2]$. This is a reduction in the possibility space; for instance, the original version considers the possibility that $W_3$ is immediately prior to $W_1$ (the upward dashed black arrow), but COBRA "recognizes" the impossibility of that.

To construct chains, COBRA initializes every write as a one-element chain (Figure 4.3, line 34). Then, COBRA consolidates chains: for each RMW transaction $t$ and the transaction $t'$ that contains the prior write, COBRA concatenates the chain containing $t'$ and the chain containing $t$ (lines 25 and lines 46–53).

Note that if a transaction $t$, which is *not* an RMW, reads from a transaction $u$, then $t$ requires an edge (known as an *anti-dependency* in the literature [53]) to $u$'s successor (call it $v$); otherwise, $t$ could appear in the graph downstream of $v$, which would mean $t$ actually read from $v$ (or even from a later write), which does not respect history. COBRA creates the needed edge $t \rightarrow v$ in InferRWEdges (Figure 4.3, line 55). (Note that in the brute-force approach (§4.2.3), analogous edges appear as the first component in a constraint.)

### 4.3.2 Coalescing constraints

This technique exploits the fact that, in many real-world workloads, there are far more reads than writes. At a high level, COBRA combines all reads that read-from the same write. We give an example and then generalize.




three constraints one coalesced constraint

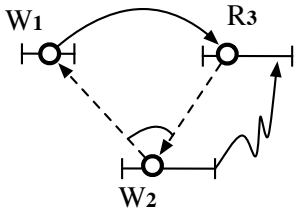
In the above figure, there are five single-operation transactions, to the same key. On the left is the basic polygraph (§4.2.3), which contains three constraints; each is in a different color. Notice that all three constraints involve the question: which write happened first, $W_1$ or $W_2$?

One can represent the possibilities as a constraint $\langle A', B' \rangle$ where $A' = \{(W_1, W_2), (R_3, W_2), (R_4, W_2)\}$ and $B' = \{(W_2, W_1), (R_5, W_1)\}$. In fact, COBRA does not include $(W_1, W_2)$ because there is a known edge $(W_1, R_3)$, which, together with $(R_3, W_2)$ in $A'$, implies the ordering $W_1 \rightarrow R_3 \rightarrow W_2$, so there is no need to include $(W_1, W_2)$. Likewise, COBRA does not include $(W_2, W_1)$ on the basis of the known edge $(W_2, R_5)$. So in the figure COBRA includes the constraint $\langle A, B \rangle = \langle \{(R_3, W_2), (R_4, W_2)\}, \{(R_5, W_1)\} \rangle$.

To construct constraints using the above reductions, COBRA does the following. Whereas the brute-force approach uses all reads and their prior writes (§4.2.3), COBRA considers particular *pairs of writes*, and creates constraints from these writes and their following reads. The particular pairs of writes are the first and last writes from all pairs of chains pertaining to that key. In more detail, given two chains, $chain_i$, $chain_j$, COBRA constructs a constraint $c$ by (i) creating a set of edges $ES_1$ that point from reads of $chain_i$.tail to $chain_j$.head (Figure 4.3, lines 73–74); this is why COBRA does not include the $(W_1, W_2)$ edge above. If there are no such reads, $ES_1$ is $chain_i$.tail $\rightarrow$ $chain_j$.head (Figure 4.3, line 69); (ii) building another edge set $ES_2$ that is the other way around (reads of $chain_j$.tail point to $chain_i$.head, etc.), and (iii) setting $c$ to be $\langle ES_1, ES_2 \rangle$ (Figure 4.3, line 65).

### 4.3.3 Pruning constraints

Our final technique leverages the information that is encoded in paths in the known graph. This technique culls irrelevant possibilities en masse (§4.6.1, Fig. 4.8). The underlying logic of the technique is almost trivial. The interesting aspect here is that the technique is enabled by a design decision to accelerate the computation of reachability on parallel hardware (§4.5 and Figure 4.3, line 79); this can be done since the computation is iterated (Boolean) matrix multiplication. Here is an example:




The constraint is $\langle (R_3, W_2), (W_2, W_1) \rangle$. Having precomputed reachability, cobra knows that the first choice cannot hold, as it creates a cycle with the path $W_2 \rightsquigarrow R_3$; cobra thereby concludes that the second choice holds. Generalizing, if cobra determines that an edge in a constraint generates a cycle, cobra throws away both components of the entire constraint and adds all the other edges to the known graph (Figure 4.3, lines 80–86). In fact, cobra does pruning multiple times, if necessary (§4.5).

### 4.3.4 Solving

The remaining step is to search for an acyclic graph that is compatible with the known graph and constraints, as computed in Figure 4.3. cobra does this by leveraging a constraint solver. However, traditional solvers do not perform well on this task because encoding the acyclicity of a graph as a set of SAT formulas is expensive (a claim by Janota et al. [140], which we also observed, using their acyclicity encodings on Z3 [106]; §4.6.1).

cobra instead leverages MonoSAT, which is a particular kind of SMT solver [81] that includes SAT modulo *monotonic* theories [73]. This solver efficiently encodes and checks graph properties, such as acyclicity.

COBRA represents a verification problem instance (a graph $G$ and constraints $C$) as follows. COBRA creates a Boolean variable $E_{(i,j)}$ for each edge: True means the $i$th node has an edge to the $j$th node; False means there is no such edge. COBRA sets all the edges in $G$ to be True. For the constraints $C$, recall that each constraint $\langle A, B\rangle$ is a pair of sets of edges, and represents a mutually exclusive choice to include either all edges in $A$ or else all edges in $B$. COBRA encodes this in the natural way: $((\forall e_a \in A, e_a) \wedge (\forall e_b \in B, \neg e_b)) \vee ((\forall e_a \in A, \neg e_a) \wedge (\forall e_b \in B, e_b))$. Finally, COBRA enforces the acyclicity of the searched-for compatible graph (whose candidate edges are given by the known edges and the constrained edge variables) by invoking a primitive provided by the solver.

**COBRA vs. MonoSAT.** One might ask: if COBRA's encoding makes MonoSAT faster, why use MonoSAT? Can we take the domain knowledge further? Indeed, in the limiting case, COBRA could re-implement the solver! However, MonoSAT, as an SMT solver, leverages many prior optimizations. One way to understand COBRA's decomposition of function is that COBRA's preprocessing exploits some of the structure created by the problem of verifying serializability, whereas the solver is exploiting residual structure common to many graph problems.

### 4.3.5 On strict serializability

COBRA's verifier checks strict serializability [80, 167] by adding a new type of edges, time-ordering edges, which are derived from timestamps associated to operations (described below). With these time-ordering edges in the known graph, the verifier performs the serializability checking algorithm that we have seen (Figure 4.3).

To get time-ordering edges, the verifier needs operations' timestamps. For those databases which provide internal timestamps (for example, Google Spanner), the verifier uses the database's timestamps for time-ordering edges. But for others which do not provide internal timestamps (for example, CockroachDB and YugaByte DB), COBRA's collectors tag each operation with a (local) timestamp. During verification, the verifier adds a time-ordering edge if one transaction's commit operation has a smaller timestamp than another transaction's begin operation. To capture these edges efficiently, COBRA borrows the algorithm CreateTimePrecedenceGraph from OROCHI

(Figure 3.6), which materializes the time precedence partial order among operations.

However, clock drift among collectors becomes a challenge: under clock drift, the timestamps from different collectors can be skewed, thus it is unsafe to infer real-time happened-before relationships simply by comparing timestamps. To tackle this problem, COBRA introduces *clock drift threshold* (100ms [17] by default), a parameter that indicates the maximum clock difference among collectors. COBRA's verifier creates a time-ordering edge only when one transaction's commit timestamp is earlier than the other transaction's begin timestamp by at least a clock drift threshold. That means all transactions within a clock drift threshold are concurrent, which makes checking strict serializability computationally expensive and requires COBRA's techniques (§4.3.1–§4.3.3) to accelerate verification (see §4.6.1).

## 4.4 Garbage collection and scaling

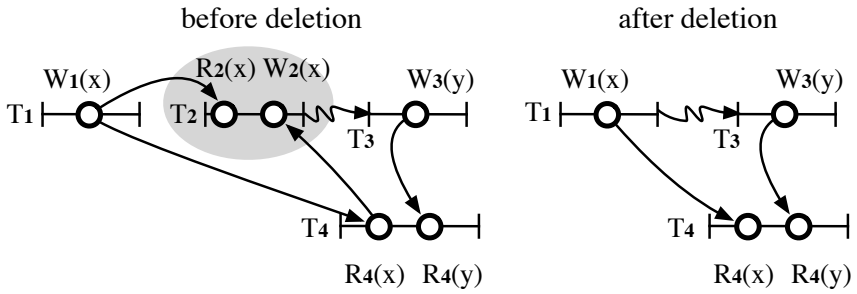
COBRA verifies in rounds. There are two motivations for rounds. First, new history is continually produced, of course. Second, there are limits on the maximum problem size (number of transactions) that the verifier can handle within reasonable time (§4.6.2); breaking the task into rounds keeps each solving task manageable.

In the first round, a verifier starts with nothing and creates a graph from CreateKnownGraph, then does verification. After that, the verifier receives more client histories; it reuses the graph from the last round (the $g$ in ConstructEncoding, Figure 4.3, line 6), and adds new nodes and edges to it from the new history fragments received (Figure 4.2).

The technical problem is to keep the input to verification bounded. So the question COBRA must answer is: which transactions can be deleted safely from history? Below, we describe the challenge (§4.4.1), the core mechanism of fence transactions (§4.4.2), and how the verifier deletes safely (§4.4.3). Due to space restrictions, we only describe the general rules and insights. A complete specification and correctness proof are in Appendix C.2.

### 4.4.1 The challenge

The core challenge is that past transactions can be relevant to future verifications, even when those transactions' writes have been overwritten. Here is an example:




Suppose a verifier saw three transactions ($T_1$, $T_2$, $T_3$) and wanted to remove $T_2$ (the shaded transaction) from consideration in future verification rounds. Later, the verifier observes a new transaction $T_4$ that violates serializability by reading from $T_1$ and $T_3$. To see the violation, notice that $T_2$ is logically subsequent to $T_4$, which generates a cycle ($T_4 \xrightarrow{\text{rw}} T_2 \rightsquigarrow T_3 \xrightarrow{\text{wr}} T_4$). Yet, if we remove $T_2$, there is no cycle. Hence, removing $T_2$ is not safe: future verifications would fail to detect certain kinds of serializability violations.

Note that this does not require malicious or exotic behavior from the database. For example, consider an underlying database that uses multi-version values and is geo-replicated: a client can retrieve a stale version from a local replica.

### 4.4.2 Epochs and fence transactions

cobra addresses this challenge by creating *epochs* that impose a coarse-grained ordering on transactions; the verifier can then discard information from older epochs. To avoid confusion, note that epochs are a separate notion from rounds: a verification round includes multiple epochs.

To memorialize epoch boundaries in history, clients issue *fence transactions*. A fence transaction is a transaction that reads-and-writes a single key named "fence" (a dedicated key that is used by fence transactions only). Each client issues fence transactions periodically (for example, every 20 transactions).

What prevents the database from defeating the point of epochs by placing all of the fence

transactions at the beginning of a notional serial schedule? COBRA leverages a property that many serializable databases provide: preserved *session-order*. That is, the serialization order obeys the execution order at each client-database session (for example, a JDBC connection). Many production databases (for example, PostgreSQL, Azure Cosmos DB, and Google Cloud Datastore) provide this property; for those which do not, COBRA requires clients to build the session-order explicitly, for example, by mandating that all transactions from the same client read-modify-write the same (per-client) key. As a consequence, the epoch ordering intertwines with the workload.

Meanwhile, the verifier adds "client-order edges" to the set of known edges in $g$ (the verifier knows the client order from the history collector). The verifier also assigns an *epoch number* to each transaction. To do so, the verifier traverses the known graph ($g$), locates all the fence transactions, chains them into a list based on the RMW relation (§4.3), and assigns their position in the list as their epoch numbers. Then, the verifier scans the graph again, and for each normal transaction on a client that is between fences with epoch $i$ and epoch $j$ ($j > i$), the verifier assigns epoch number $j - 1$.

During the scan, assume the largest epoch number that has been seen or surpassed by every client is $epoch_{agree}$. Then we have the following guarantee.

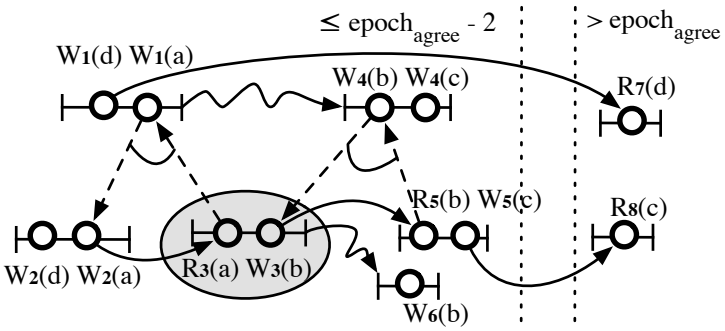
**Guarantee**. For any transaction $T_i$ whose epoch $\leq (epoch_{agree} - 2)$, and for any transaction (including future ones) $T_j$ whose epoch $\geq epoch_{agree}$, the known graph $g$ contains a path $T_i \rightsquigarrow T_j$.

To see why the guarantee holds, consider the problem in three parts. First, for the fence transaction with epoch number $epoch_{agree}$ (denoted as $F_{ea}$), $g$ must have a path $F_{ea} \rightsquigarrow T_j$. Second, for the fence transaction with epoch number $(epoch_{agree} - 1)$ (denoted as $F_{ea-1}$), $g$ must have a path as $T_i \rightsquigarrow F_{ea-1}$. Third, $F_{ea-1} \rightarrow F_{ea}$ in $g$.

The guarantee suggests that no future transaction (with epoch $\geq epoch_{agree}$) can be a direct predecessor of such $T_i$, otherwise a cycle will appear in $g$. We can extend this property to use in garbage collection. In particular, if all predecessors of $T_i$ have epoch number $\leq (epoch_{agree} - 2)$, we call $T_i$ a *frozen* transaction, as no future transaction can precede it.

### 4.4.3 Safe garbage collection

cobra's garbage collection algorithm targets frozen transactions—as they are guaranteed not to be subsequent (in the notional serial schedule) to any future transaction. The verifier needs to keep those frozen transactions that have the most recent writes to some key (because they might be read by future transactions). If there are multiple writes to the same key and the verifier cannot distinguish the most recent, the verifier keeps them all. Meanwhile, if a future transaction reads from a deleted transaction (which, owing to fence transactions, will manifest as a serializability violation), the verifier detects this and rejects the history. One might think this approach is enough. However, consider:




The shaded transaction ($T_3$; transaction ids indicated by operation subscripts) is frozen and is not the most recent write to any key. But with the two future transactions ($T_7$ and $T_8$), deleting the shaded transaction results in failing to detect cycles in $g$.

To see why, consider operations on key $c$: $W_4(c)$, $W_5(c)$, and $R_8(c)$. By the epoch guarantee (§4.4.2), both $T_4$ and $T_5$ happen before $T_8$. Plus, $R_8(c)$ reads from $W_5(c)$, hence $W_4(c)$ must happen before $W_5(c)$ (otherwise, $R_8(c)$ should have read from $W_4(c)$). As a consequence, the constraint $\langle (T_5, T_4),\ (T_4, T_3) \rangle$ is solved: $T_4 \longrightarrow T_3$ is chosen. Similarly, because of $R_7(d)$, the other constraint is solved and $T_3 \longrightarrow T_1$. With these two solved constraints, there is a cycle ($T_1 \rightsquigarrow T_4 \longrightarrow T_3 \longrightarrow T_1$). Yet, if the verifier deletes $T_3$, the cycle would be undetected.

What's going on here is that the future transactions "finalized" some constraints from the past, causing cycles whereas in the past the constraints were "chosen" in a different way. To prevent cases like this, cobra's verifier keeps transactions that are involved in any potentially cyclic

| COBRA component | LOC written/changed |
|---|---|
| COBRA client library | |
| history recording | 620 lines of Java |
| database adapters | 900 lines of Java |
| COBRA verifier | |
| data structures and algorithms | 2k lines of Java |
| GPU optimizations | 550 lines of CUDA/C++ |
| history parser and others | 1.2k lines of Java |

Figure 4.4: Components of COBRA implementation.

constraints, which works as follow. During garbage collection, the verifier clones the known graph, adds all edges in constraints as actual edges, and checks whether a candidate transaction belongs to cycles that contain most-recent-writes. If so, the verifier keeps it and otherwise garbage collects it.

This approach is safe (a deleted transaction will never generate a cycle with future transactions) because a candidate transaction is a frozen transaction and does not contain any most-recent-writes; hence it cannot be read by future transactions. In addition, it has no cycles with most-recent-writes in the cloned graph, which means that the candidate transaction has no predecessors that could be read by future transactions. Thus, this candidate transaction will not be affected by future transactions and is safe to delete.

## 4.5 Implementation

The components of COBRA's implementation are listed in Fig. 4.4. Our implementation includes a client library and a verifier. COBRA's client library wraps other database libraries: JDBC, Google Datastore library, and RocksJava. It enforces the assumption of uniquely written values (§4.2.2). , by adding a unique id to a client's writes, and stripping them out of reads. It also issues fence transactions (§4.4.2). Finally, in our current implementation, we simulate history collection (§4.2.1) by collecting histories in this library; future work is to move this function to a proxy.

The verifier iterates the pruning logic within a round, stopping when it finds nothing more to prune or when it reaches a configurable max number of iterations (to bound the verifier's work);

a better implementation would stop when the cost of the marginal pruning iteration exceeds the improvement in the solver's running time brought by this iteration.

Another aspect of pruning is GPU acceleration. Recall that pruning works by computing the transitive closure of the known edges (Fig. 4.3, line 79). COBRA uses the standard algorithm: repeated squaring of the Boolean adjacency matrix [98, Ch.25] as long as the matrix keeps changing, up to $\log |V|$ matrix multiplications. ($\log |V|$ is the worst case and occurs when two nodes are connected by a ($\geq |V|/2+1$)-step path; at least in our experiments, this case does not arise much.) The execution platform is cuBLAS [14] (a dense linear algebra library on GPUs) and cuSPARSE [15] (a sparse linear algebra library on GPUs), which contain matrix multiplication routines.

COBRA includes several optimizations. It invokes a specialized routine for triangular matrix multiplication. (COBRA first tests the graph for acyclicity, and then indexes the vertices according to a topological sort, creating a triangular matrix.) COBRA also exploits sparse matrix multiplication (cuSPARSE), and moves to ordinary (dense) matrix multiplication when the density of the matrix exceeds a threshold (chosen to be $\geq 5\%$ of the matrix elements are non-zero, the empirical cross-over point that we observed).

Whenever COBRA's verifier detects a serializability violation, it creates a certificate with problematic transactions. The problematic transactions are either a cycle in the known graph detected by COBRA's algorithms, or a unsatisfiable core (a set of unsatisfiable clauses that translates to problematic transactions) produced by the SMT solver.

## 4.6 Experimental evaluation

We answer three questions:

- What are the verifier's costs and limits, and how do these compare to baselines?
- What is the verifier's end-to-end, round-to-round *sustainable capacity*? This determines the offered load (on the actual database) that the verifier can support.
- How much runtime overhead (in terms of throughput and latency) does COBRA impose for clients? And what are COBRA's storage and network overheads?

**Benchmarks and workloads.** We use four benchmarks:

- *TPC-C* [44] is a standard. A warehouse has 10 districts with 30k customers. There are five types of transactions (frequencies in parentheses): new order (45%), payment (43%), order status (4%), delivery (4%), and stock level (4%). In our experiments, we use one warehouse, and clients issue transactions based on the frequencies.
- *C-Twitter* [8] is a simple clone of Twitter, according to Twitter's own description [8]. It allows users to tweet a new post, follow/unfollow other users, and show a timeline (the latest tweets from followed users). Our experiments include one thousand users. Each user tweets 140-word posts and follows/unfollows other users based on a Zipfian distribution ($\alpha = 100$).
- *C-RUBiS* [40, 59] simulates bidding systems like eBay [40]. Users can register accounts, register items, bid for items, and comment on items. We start the market with 20k users and 200k items.
- *BlindW* is a microbenchmark to demonstrate COBRA's performance in extreme scenarios. It creates 10k keys, and runs *read-only* and *write-only* transactions, each of which has eight operations. This benchmark has two variants: (1) *BlindW-RM* represents a read-mostly workload that contains 90% read-only transactions; and (2) *BlindW-RW* represents a read-write workload, evenly divided between read-only and write-only transactions. (Of course, there are more challenging workloads: > 50% writes. We do not experiment with them because COBRA targets common online transaction processing workloads (OLTP) in which reads are the majority.)

**Databases and setup.** We evaluate COBRA on Google Cloud Datastore [23], PostgreSQL [37, 176], and RocksDB [39, 109]. In our experimental setup, clients interact with Google Cloud Datastore through the wide-area Internet, and connect to a PostgreSQL server through a 1Gbps network.

Database clients run on two machines with a 3.3GHz Intel i5-6600 (4-core) CPU, 16GB memory, a 250GB SSD, and Ubuntu 16.04. For PostgreSQL, a database instance runs on a machine with a 3.8GHz Intel Xeon E5-1630 (8-core) CPU, 32GB memory, a 1TB disk, and Ubuntu 16.04. For RocksDB, the same machine hosts the client threads and RocksDB threads, which all run in the same process. We use a *p3.2xlarge* Amazon EC2 instance as the verifier, with an NVIDIA Tesla V100 GPU, a 8-core CPU, and 64GB memory.

### 4.6.1 One-shot verification

In this section, we consider "one-shot verification" a verifier gets a history and decides whether that history is serializable. In our setup, clients record history fragments and store them as files; a verifier reads them from the local file system. In this section, the database is RocksDB (PostgreSQL and Google Cloud Datastore give similar results).

**Baselines.** We have four baselines:

- **A serializability-checking algorithm:** To the best of our knowledge, the algorithm of Biswas and Enea [83] is the most efficient algorithm to check serializability. In our experiments, we use their Rust implementation [82].
- **SAT solver:** We use the same solving baseline that Biswas and Enea use for their own comparisons [83]: encoding serializability verification into SAT formulas, and feeding this encoding to MiniSAT [114], a popular SAT solver.
- **cobra, subtracted:** We implement the original polygraph (§4.2.3), directly encode the constraints (without the techniques of §4.3), and feed them to the MonoSAT SMT solver [73].
- **SMT solver:** An alternative use of SMT, and a natural baseline, is a linear arithmetic encoding: each node is assigned a distinct integer index, with read-from relationships creating inequality constraints, and writes inducing additional constraints (for a total of $O(|V|^2)$ constraints, as in §4.2.3). The solver is then asked to map nodes to integers, subject to those constraints [117, 140]. We use Z3 [106] as the solver.

**Verification runtime vs. number of transactions.** We compare cobra to other baselines, on the various workloads. There are 24 clients. We vary the number of transactions in the workload, and measure the verification time. Figure 4.5 depicts the results on the BlindW-RW benchmark. On all five benchmarks, cobra does better than MonoSAT and Z3, which do better than MiniSAT and the serializability-checking algorithm. As a special case, there is, for TPC-C, an alternative that beats MonoSAT and Z3 and has the same performance as cobra. Namely, add edges that are inferred from RMW operations in history to a candidate graph (without constraints, and so missing a lot of dependency information), topologically sort it, and check whether the result

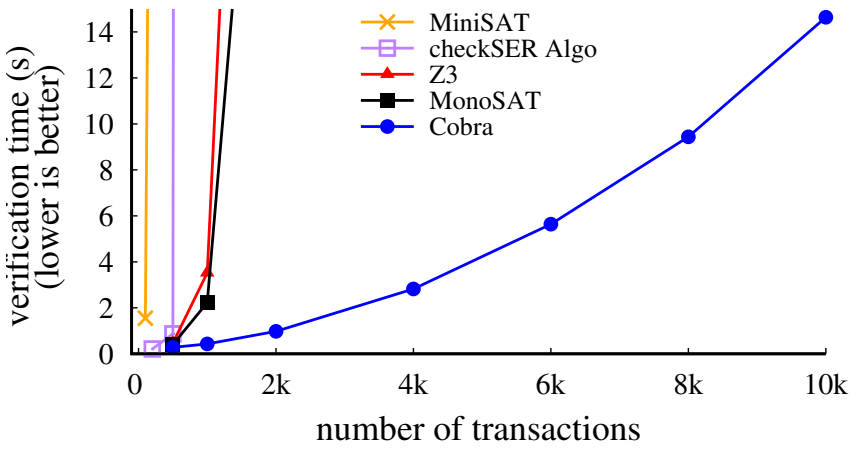



Figure 4.5: Cobra's running time is shorter than other baselines' on the BlindW-RW workload. The same holds on the other benchmarks (not depicted). Verification runtime grows superlinearly.

| Violation | Database | #Txns | Time |
|---|---|---|---|
| G2-anomaly [22] | YugaByteDB 1.3.1.0 | 37.2k | 66.3s |
| Disappearing writes [1] | YugaByteDB 1.1.10.0 | 2.8k | 5.0s |
| G2-anomaly [21] | CockroachDB-beta 20160829 | 446 | 1.0s |
| Read uncommitted [31] | CockroachDB 2.1 | 20⋆ | 1.0s |
| Read skew [29] | FaunaDB 2.5.4 | 8.2k | 11.4s |

Figure 4.6: Serializability violations that COBRA checks. "Violation" describes the phenomenon that clients experience. "Database" is the database (with version number) that causes the violation. "#Txns" is the size of the violation history. "Time" is the runtime for COBRA to detect the violation. ⋆ The bug report only contains a small fragment of the history.

matches history; if not, repeat. This process has even worse order complexity than the one in §4.2.3, but it works for TPC-C because that workload has *only* RMW transactions, and thus the candidate graph *is* (luckily) a serialization graph.

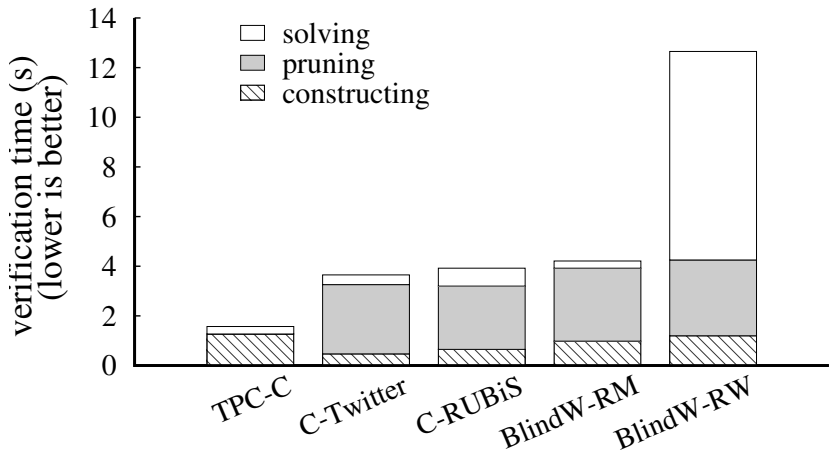



Figure 4.7: Decomposition of COBRA runtime, on 10k-transaction workloads. In benchmarks with RMWs only (the left one), there is no constraint, so COBRA doesn't do pruning; in benchmarks with many reads and RMWs (the middle three), the dominant component is pruning not solving, because COBRA's own logic can identify concrete dependencies; in benchmarks with many blind writes (the last one), solving is a much larger contributor because COBRA is not able to eliminate as many constraints.

**Detecting serializability violations.** We investigate COBRA's performance on an unsatisfiable instance: does it trigger an exhaustive search, at least on the real-world workloads we found? We evaluate COBRA on five real-world workloads that are known to have serializability violations (we downloaded the given database's reported histories from the and fed them to COBRA). COBRA detects them in reasonable time. Figure 4.6 shows the results.

**Decomposition of COBRA's verification runtime.** We measure the wall clock time of COBRA's verification on our setup, broken into three stages: *constructing*, which includes creating the graph of known edges, combining writes, and creating constraints (§4.3.1–§4.3.2); *pruning* (§4.3.3), which includes the time taken by the GPU; and *solving* (§4.3.4), which includes the time spent within MonoSAT. We experiment with all benchmarks, with 10k transactions. Figure 4.7 depicts the results.

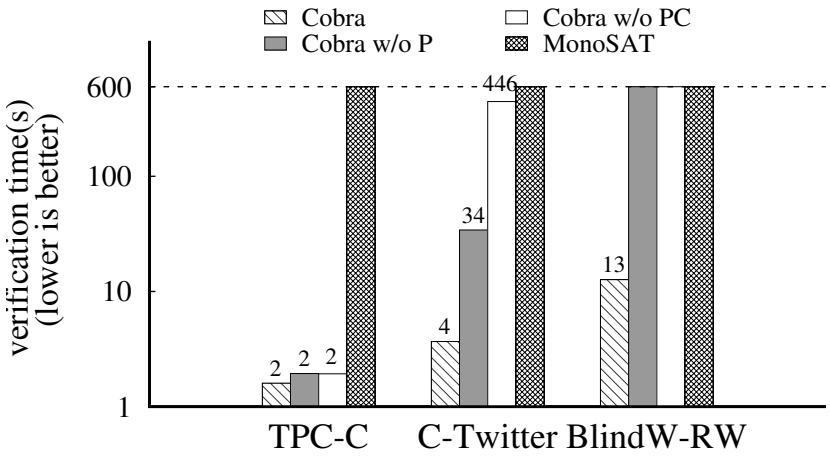



Figure 4.8: Differential analysis on different workloads, log-scale, with runtime above bars. On TPC-C, combining writes exploits the RMW pattern and solves all the constraints. On C-Twitter, each of COBRA's components contributes meaningfully. On BlindW-RW, pruning is essential.

**Differential analysis.** We experiment with four variants: COBRA itself; COBRA without pruning (§4.3.3); COBRA without pruning and coalescing (§4.3.2), which is equivalent to MonoSAT plus write combining (§4.3.1); and the MonoSAT baseline. We experiment with three benchmarks, with 10k transactions. Figure 4.8 depicts the results.

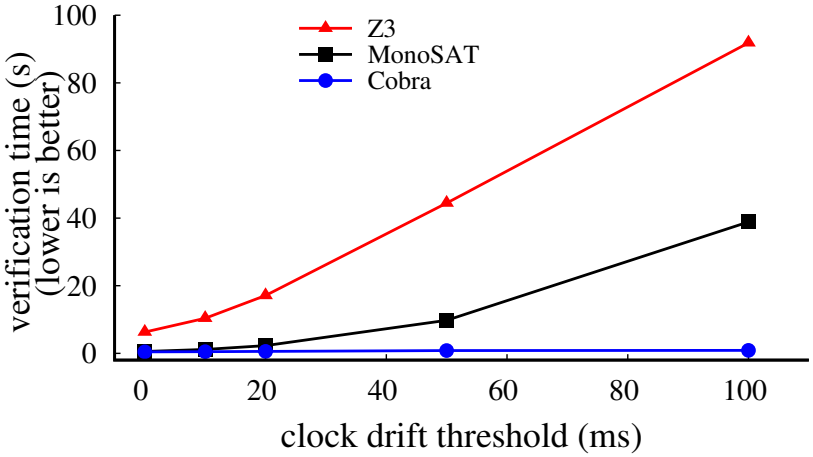



Figure 4.9: COBRA's running time is shorter than other baselines' on checking strict serializability under clock drift. The workload is 2,000 transactions of BlindW-RW. The maximum clock drift threshold (100 ms) follows this report [17]; similar thresholds can be found elsewhere [12, 52].

**Checking strict serializability under clock drift.** As we mentioned (§4.1, §4.3.5), clock drift happens in reality and adds complexity to checking strict serializability. To measure this effect, we experiment with COBRA and the two baselines MonoSAT (with original polygraph encoding) and Z3 (with the linear SMT encoding), under different clock drifts, on the same workload. The workload has eight clients running BlindW-RW on 1k keys for one second with a throughput of 2k transaction/sec. To make checking strict serializability easier, clients avoid issuing transactions all at the same time: they spread the workload evenly by issuing 20 transactions every 10ms. Figure 4.9 shows the results; COBRA outperforms other two baselines by 45× and 107× in verification time under the default clock drift (100ms).

### 4.6.2 Scaling

We want to know: what offered load (to the database) can COBRA support on an ongoing basis? To answer this question, we must quantify COBRA's *verification capacity*, in txns/second. This depends on the characteristics of the workload, the number of transactions one round (§4.4) verifies ($\#tx_r$), and the average time for one round of verification ($t_r$). Note that the variable here is $\#tx_r$; $t_r$ is a function of that choice. So the *verification capacity* for a particular workload is defined as: $\max_{\#tx_r}(\#tx_r/t_r)$.

To investigate this quantity, we run our benchmarks on RocksDB with 24 concurrent clients, a fence transaction every 20 transactions. We generate a 100k-transaction history ahead of time. For that same history, we vary $\#tx_r$, plot $\#tx_r/t_r$, and choose the optimum.

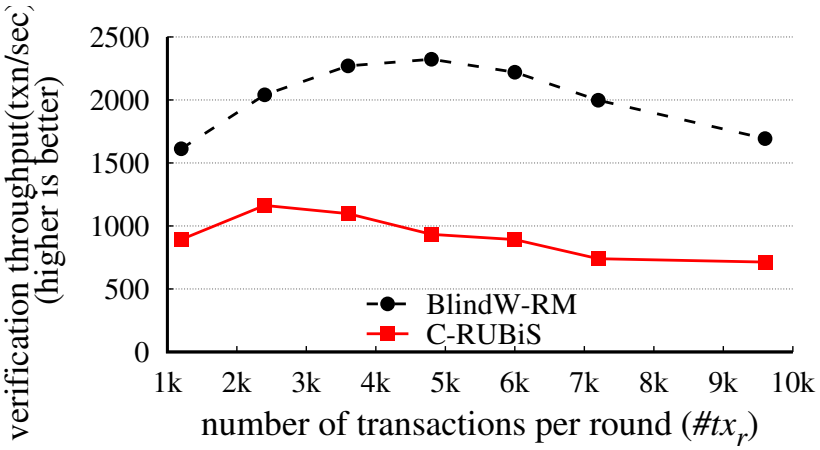



Figure 4.10: Verification throughput vs. round size ($\#tx_r$). The verification capacity for BlindW-RM (the dashed line) is 2.3k txn/sec when $\#tx_r$ is 5k; the capacity for C-RUBiS (the solid line) is 1.2k txn/sec when $\#tx_r$ is 2.5k.

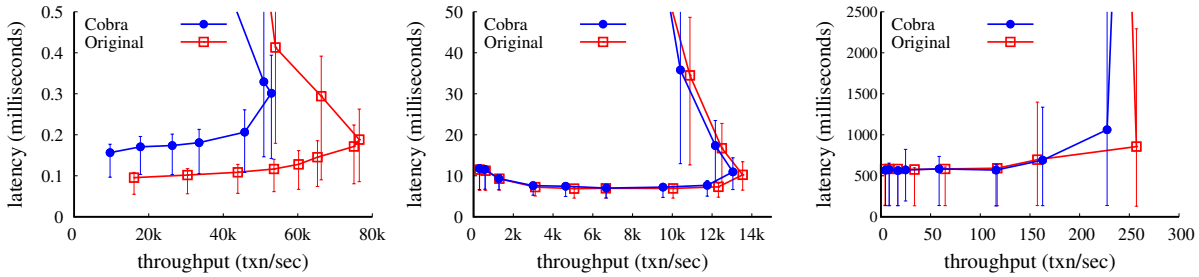
Figure 4.10 depicts the results. When $\#tx_r$ is smaller, COBRA does not have enough transactions for garbage collection, hence wastes cycles on redundantly analyzing transactions from prior rounds; when $\#tx_r$ is larger, COBRA suffers from a problem size that is too large (recall that verification time increases superlinearly; §4.6.1). For different workloads, the optimal choices of $\#tx_r$ are different.

In workload BlindW-RW, COBRA runs out of GPU memory. The reason is that due to many blind writes in this workload, COBRA is unable to garbage collect enough transactions and fit the remaining history into the GPU memory. Our future work is to investigate this case and design a more efficient (in terms of deleting more transactions) garbage collection algorithm.

### 4.6.3 COBRA ONLINE OVERHEADS

The baseline in this section is the legacy system; that is, clients use the unmodified database library (for example, JDBC), with no recording of history.

**Latency-versus-throughput.** We evaluate COBRA's client-side throughput and latency in the three setups, tuning the number of clients (up to 256) to saturate the databases. Figure 4.11 depicts the results.




(a) RocksDB (b) PostgreSQL (c) Google Datastore

Figure 4.11: Throughput and latency, for C-Twitter benchmark. For our RocksDB setup, 90th percentile latency increases by 2×, with 50% throughput penalty, an artifact of history collection (disk bandwidth contention between clients and the DB). COBRA imposes minor overhead for our PostgreSQL. For Google Datastore, the throughput penalty reflects a ceiling (a maximum number of operations per second) imposed by the cloud service and the extra operations caused by fence transactions.

**Network cost and history size.** We evaluate the network traffic on the client side by tracking the number of bytes sent over the NIC. We measure the history size by summing sizes of the history files. Figure 4.12 summarizes.

| workload | network overhead | | history size |
|---|---|---|---|
| | traffic | percentage | |
| BlindW-RW | 227.4 KB | 7.28% | 245.5 KB |
| C-Twitter | 292.9 KB | 4.46% | 200.7 KB |
| C-RUBiS | 107.5 KB | 4.53% | 148.9 KB |
| TPC-C | 78.2 KB | 2.17% | 1380.8 KB |

Figure 4.12: Network and storage overheads per 1k transactions. The network overheads comes from fence transactions and the metadata (for example, transaction ids and write ids) added by COBRA's client library.

# 5 | Summary and Next Steps

This dissertation studied how to audit two essential services: outsourced computing and outsourced databases. We proposed two systems, orochi and cobra, that verify the respective services, without making any assumptions about remote servers, while having good performance.

In particular, Chapter 3 defined a general problem of execution integrity for concurrent servers (§3.1); exhibited an abstract solution, ssco, based on new kinds of replay (§3.2); and described a system, orochi, that instantiates ssco for web applications and runs on today's cloud infrastructure (§3.3–§3.4). Chapter 4 introduced another system, cobra. It is the first system that tackles the problem of (a) black-box checking of (b) serializability (§4.3), while (c) scaling to real-world online transactional processing workloads (§4.4).

**Next steps.** Orochi and cobra have limitations (see also §1.2). These limitations suggest next steps for both systems. We elaborate below.

First, orochi and cobra perform ex post facto verifications, which cannot prevent a service from misbehaving. An interesting problem is to recover from misbehavior. To recover, when detecting a misbehavior, orochi and cobra can generate a report about the misbehavior, ask users to provide a candidate correct behavior, then roll back the service to a point before the misbehavior, and replay the following requests. A similar topic is patch-based auditing, proposed in Poirot [144] (see also [159, 198]); here, one replays prior requests against patched code to see if the responses are now different. The above proposed recovery procedure can address this patch-based auditing as well, by patching code before the replay; and in addition, it can audit the effect of a patch at any layer, not just in PHP code (as in Poirot).

Second, both orochi and cobra require all requests to, and responses from, the audited ser-

vice. This assumes the verifier and the collectors do not crash. To achieve fault tolerance of the verifier and the collectors, one readily applicable approach is to deploy transparent state machine replication (such as VMware vSphere Replication [48] or Crane [104]). But this approach is costly. Instead, it might be possible to tolerate faults by storing the verifier's state and collectors' logs in the cloud as well, which requires a "double verification"—one for verifying the integrity and freshness of the state and logs, and the other for verifying the actual service. Moreover, this approach could help mitigate another limitation, namely that the verifier has to store all service's persistent state (for example, the database). Now, by offloading state to the cloud, the verifier could have much smaller storage, at the cost of verifying state when fetching it from the cloud.

Third, OROCHI and COBRA audit the service comprehensively. Though accelerated, the audit is still expensive (for example, requiring one tenth of the server's computational resources). To further lower the verification cost, one approach is to sacrifice the comprehensiveness of the audit and spot-check the service with a probabilistic guarantee. This approach of course contradicts the goals of this dissertation (§1), but it might be attractive to users who have limited local resources (for example, mobile devices) but still want to gain some correctness guarantees. However, the challenge here is that spot-checking might fail to provide any guarantee, if tampering to the service's state hasn't been detected. In particular, not checking a single malicious request (for example, adding a malicious user as an administrator in the database) may cause accepting all future incorrect requests (for example, all administrative operations issued by this malicious user).

Finally, so far, OROCHI and COBRA require strong consistency models: OROCHI requires shared objects to be linearizable or strict serializable, and COBRA only checks serializability. However, for performance reasons, many applications use weak consistency models (for example, Sequential Consistency and Casual Consistency) and weak isolation levels (for example, Snapshot Isolation and Read Committed). It will greatly expand OROCHI's and COBRA's applicability to support weak consistency models.

**Closing words.** Despite the limitations, OROCHI and COBRA open a new design point for audit systems—comprehensive guarantees together with good performance. In addition, the two systems introduced many new techniques and algorithms—SIMD-on-demand execution, simulate-

and-check, consistent ordering verification, an efficient SMT-encoding for polygraphs, a GPU-based transitive closure algorithm, and so on. Some of these may be of independent interest. Finally, we hope that this dissertation can motivate service providers to release verifiable versions of today's cloud services, so that future users don't have to *assume*, but can *be sure*, that their services perform as expected.

# Appendices

# A | A Note About the Contents

Chapter 3 revises a prior work:

- Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web. Symposium on Operating Systems Principles (SOSP), October 2017.

Appendix B revises the extended version of the publication above:

- Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web (extended version). arXiv:1709.08501, April 2018.

Chapter 4 revises a prior work:

- Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. Symposium on Operating Systems Design and Implementation (OSDI), November 2020.

Appendix C revises the extended version of the publication above:

- Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making Transactional Key-Value Stores Verifiably Serializable (extended version). arXiv:1912.09018, July 2020.

# B | Correctness Proof of Orochi's Audit Algorithm

This appendix states the definition of correctness and then proves that Orochi's audit algorithm (and associated report collection process) meets that definition. The algorithm is defined in Section 3.2 and extended with additional object types in Sections 3.3.4–3.3.5. For completeness, the algorithm is included in this appendix in Figure B.1, where it is called SSCO_AUDIT2.

## B.1 Definition of correctness

The two correctness properties are Completeness and Soundness. We have described these properties (§3.1) and now make them more precise.

Model. We presume the setting of Section 3.1 and the concurrency model of Section 3.2.2. We use those definitions. Additionally, in this appendix, we will sometimes use the word "request" as a shorthand for "the execution of the program, on input given by the request"; there are examples of this use in the next several sentences.

Each request consists of a sequence of instructions. A special instruction allows a request to invoke an *operation* on a shared object, which we often call a *state item*. A request can execute any number of such state operations over the lifetime of request, but it can issue only one at a time. Our algorithm handles three kinds of state items (§3.2.2, §3.3.4): atomic registers [150]; linearizable key-value stores exposing single-key get/set operations; and strictly serializable databases, with the restrictions stated in Section 3.3.4.

It will be convenient to have a notation for events (requests and responses) in the trace. We represent such events as a tuple:

$$(\text{RESPONSE} \mid \text{REQUEST}, \textit{rid}, [\text{contents}])$$

A trace is a timestamped or ordered list of such tuples (the exact timing does not matter, only the relative order of events). We assume that traces are balanced: every response is associated to a request, and every request has exactly one response. In practice, the verifier can ensure this property prior to beginning the audit.

**Definition 1** (Completeness)**.** *A report collection and audit procedure are defined to be* Complete *if the following holds. If the executor executes the program (under the model above) and the given report collection procedure, then the given audit procedure (applied to the resulting trace and reports) accepts.*

To define Soundness in our context requires several additional notions.

*Request schedule.* A request schedule models the thread context switch schedule, and is an ordered list of requestIDs. For example:

$$\text{req 1, req 23, req 1, req 14, req 5, req 1, } \ldots$$

Notice that requestIDs are permitted to repeat in the schedule.

*Operationwise execution.* Consider a (physically impossible) model where, instead of requests arriving and departing, the executor has access to all requestIDs in a trace and their inputs. Operationwise execution means executing the program against all requestIDs by following a request schedule. Specifically, for each request id (rid) in the request schedule, in order:

- If it is the rid's first appearance, the executor reads in that request's input and allocates the structures needed to run the program on that input.
- Otherwise, the executor runs the request up to and including the request's next interaction with the outside world. This will either be an operation on a state object, or the delivered

output (as the response).

After each such event, the request is held, until the executor reschedules it. If a request is scheduled after it has delivered its response, the executor immediately yields and chooses the next rid in the request schedule.

*Request precedence.* A trace *Tr* induces a partial order on requests. We say that a request $r_1$ precedes another request $r_2$ in a trace if the trace shows that (the execution of) $r_1$ must have ended before (the execution of) $r_2$ began. We notate this relation as $<_{Tr}$. That is, $r_1 <_{Tr} r_2$ if the event (RESPONSE, $r_1$, $\cdot$) occurs in *Tr* before (REQUEST, $r_2$, $\cdot$).

*Real-time consistency.* A request schedule *S* of the kind specified above is *real-time consistent* with $<_{Tr}$ if for any $r_1, r_2$ that appear in *S*, $r_1 <_{Tr} r_2 \implies$ all instances of $r_1$ in *S* are sequenced before all instances of $r_2$ in *S*.

Now we can define Soundness:

**Definition 2** (Soundness)**.** *A report collection and audit procedure are defined to be* Sound *if the following holds. If the given audit procedure accepts a trace Tr and reports R, then there exists some request schedule, S, such that:*

*(a) The outputs produced by operationwise execution according to S (on the inputs in Tr) are exactly the responses in the trace Tr, and*

*(b) S is real-time consistent with* $<_{Tr}$.

COMMENTS ON THE SOUNDNESS DEFINITION. The Soundness definition is bulky, and it may be unintuitive. But it is capturing something natural: it says that a trace *Tr* and reports can pass the given audit process only if the *Tr* is consistent with having actually executed the requests in *Tr*, according to a physical schedule at the executor.

Here is a potentially more intuitive description. The verifier accepts only if the following holds: there exists a schedule *S* such that if we feed requests to a well-behaved executor, in the exact order that they arrived, and if the executor physically context-switches among them according to *S*, then the executor will emit the precise responses that are in the trace, in the precise order (relative to the requests and the other responses) that they appear in the trace.

One might wonder why do we not simply phrase the definition as, "If the audit procedure accepts, then it means that the executor executed the program." The reason is that making this determination in our model is impossible: the trace collector, and the verifier, have no visibility to look "inside" the executor. For all they can tell, the executor is passing their checks by doing something radically different from executing the program. The key point, however, is that if the verifier accepts, it means that what the executor actually did and what it is supposed to be doing are completely indistinguishable from outside the executor.

The definition does contain some leeway for an untrusted executor: any physical schedule that it chooses will result in the verifier accepting (provided the executor actually executes the requests according to that schedule). But that leeway seems unavoidable: in this execution and concurrency model, even a well-behaved executor has complete discretion over the schedule.

Model vs. proofs. Our pseudocode and proofs assume the model stated at the outset of this section, but with one simplification. Specifically, the pseudocode and the proofs treat a database transaction as a series of statements, with no code interspersed (see, for example, line 29 in Figure B.1). We impose the simplification to avoid some tedium in the proofs. However, the model itself reflects the implementation (§3.3), which does permit code (for example, PHP) to execute in between SQL statements that are part of the same transaction. We address the complexity in Section B.7.

Model vs. reality. A major difference between our model and web applications is that we are assuming that state operations and scheduling are the sole sources of non-determinism. In reality, there is another source of non-determinism: the return values of some functions (those that query the time, etc.). To take this into account would complicate the definitions and proofs; for example, Soundness and Operationwise execution would need to explicitly refer to the reports that hold the return values of non-deterministic functions (§3.3.6). To avoid this complexity, our claims here ignore this source of non-determinism.

**Input** Trace *Tr* **Input** Reports *R* **Global** *OpMap*: (requestID, opnum) → (*i*, seqnum) **Global** *kv*, *db* // versioned storage

Components of the reports *R*:
*C*: CtlFlowTag → Set(requestIDs) // alleged groups; §3.2.1
$OL_i$: $\mathbb{N}^+$ → (requestID, opnum, optype, opcontents) // §3.2.3
*M*: requestID → $\mathbb{N}$ // op counts; §3.2.3

The form of the opcontents depends on the optype:

| optype | opcontents | optype | opcontents |
|---|---|---|---|
| RegisterRead | empty | KvSet | key and value to write |
| RegisterWrite | value to write | DBOp | SQL statement(s), whether succeeds |
| KvGet | key to read | | |

```
 1: procedure SSCO_AUDIT2()
 2:     // Partially validate reports (§3.2.5) and construct OpMap
 3:     ProcessOpReports()          // defined in Figure 3.5
 4:
 5:     // OL_{i_kv} is op log for versioned key-value store (§3.3.5)
 6:     kv.Build(OL_{i_kv})
 7:     // OL_{i_db} is op log for versioned database (§3.3.5)
 8:     db.Build(OL_{i_db})
 9:
10:     return ReExec2 ()     // line 31
11:
12: procedure CheckOp(rid, opnum, i, optype, opcontents)
13:     if (rid, opnum) not in OpMap : REJECT
14:     î, s ← OpMap[rid, opnum]
15:     ôt, ôc ← (OL_i[s].optype, OL_i[s].opcontents)
16:     if i ≠ î or optype ≠ ôt or opcontents ≠ ôc : REJECT
17:     return s
18:
19: procedure SimOp(i, s, optype, opcontents)
20:     ret ← ⊥
21:     if optype = RegisterRead :
22:         writeop ← walk backward in OL_i from s; stop when
23:             optype=RegisterWrite
24:         if writeop doesn't exist : REJECT
25:         ret = writeop.opcontents
26:     else if optype = KvGet :
27:         ret = kv.get(opcontents.key, s)
28:     else if optype = DBOp :
29:         ret = db.do_trans(opcontents.transaction, s)
30:     return ret
31: procedure ReExec2()
32:     Re-execute Tr in groups according to C:
33:
34:         (1) Initialize a group as follows:
35:             Read in inputs for all requests in the group
36:             Allocate program structures for each request in the group
37:             opnum ← 1     // opnum is a per-group running counter
38:
39:         (2) During SIMD-on-demand execution (§3.2.1):
40:
41:             if execution within the group diverges: return REJECT
42:
43:             When the group makes a state operation:
44:                 optype ← the type of state operation
45:                 for all rid in the group:
46:                     i, oc ← state op parameters from execution
47:                     s ← CheckOp(rid, opnum, i, optype, oc) // ln 12
48:                     if optype ∈ {RegisterRead, KvGet, DBOp}:
49:                         state op result ← SimOp(i, s, optype, oc) // ln 19
50:                 opnum ← opnum + 1
51:
52:         (3) When a request rid finishes:
53:             if opnum < M(rid): return REJECT
54:
55:         (4) Write out the produced outputs
56:
57:     if the outputs from (4) are exactly the responses in Tr :
58:         return ACCEPT
59:     return REJECT
```

Figure B.1: SSCO audit procedure. This is a refinement of Figure 3.3. This one includes additional objects: a versioned database and key-value store, as used by OROCHI (§3.3.5). As in Figure 3.3, the Trace *Tr* is assumed to be balanced.

## B.2 Proof outline and preliminaries

Our objective is to prove that SSCO_AUDIT2 (Figure B.1), together with the corresponding report collection process (§3.2–§3.3), meets Completeness (Definition 1) and Soundness (Definition 2). Below, we outline the main idea of the proof. However, this is just a rough description; the proofs themselves go in a different order (as befits the technique).

- A central element in the proofs is a variant of SSCO_AUDIT2, called OOOAudit (§B.4). This variant relies on out-of-order, simulated execution, which we call OOOExec (in contrast to ReExec2, Figure B.1). OOOExec follows some supplied op schedule $S$ (a list of requestIDs), and executes one request at a time up through the request's next op (rather than the grouped execution; §3.2.1). $S$ is required to respect program order. Thus, while requests can be arbitrarily interleaved in $S$, each executes sequentially.
- We establish that OOOExec (following some schedule $S$) is equivalent to ReExec2 (following some control flow reports $C$); the argument for this is fairly natural and happens at the end (§B.6).

The proof focuses on OOOAudit. The rough argument for the Soundness of OOOAudit is as follows (§B.5). Assume that OOOAudit accepts on some schedule $S$.

- We establish that OOOAudit is indifferent to the schedule (§B.4): following $S_1$ is equivalent to following any other schedule $S_2$, provided both respect program order.
- Let schedule $S'$ be an ordering (a topological sort) of the graph $G$. An ordering exists because $G$ has no cycles (otherwise, OOOAudit would not have accepted $S$). We establish that $G$, and hence $S'$, respects the partial order given by externally observable events (§B.3). By the previous bullet, OOOAudit accepts when following $S'$.
- We establish that this simulated execution (that is, OOOAudit following $S'$) is equivalent to physical execution according to $S'$. The idea underlying the argument is that the checks in the simulated execution (which pass) ensure that the op parameters in the logs, and the ops in the graph, match physical execution. That in turn means that simulating certain operations (such

as reads) by consulting the logs produces the same result as in the physical execution.

- Meanwhile, the final check of OOOAudit (which, again, passes) establishes that the produced outputs (produced by the simulated execution) equal the responses in the trace. This bullet and the previous together imply that physical execution according to $S'$ produces the responses in the trace. Combining with the second bullet, we get that $S'$ is a schedule of the kind required by the soundness definition.

The argument for Completeness of OOOAudit uses similar reasoning to the argument for Soundness (§B.5). The essence of the argument is as follows. If the executor operates correctly, then (1) the reports supplied to the verifier constitute a precise record of online execution, and (2) OOOAudit, when supplied $S'$, reproduces that same online execution. As a result, the contents that OOOAudit expects to see in the reports are in fact the actual contents of the reports (both reflect online execution). Therefore, the checks in OOOAudit pass. That implies that OOOAudit accepts any schedule that respects program order.

Conventions and notation. Per Section 3.2 and Figure B.1, the reports $R$ have components $C, M, OL_1, \ldots, OL_n$, where $n$ is the number of state items. For convenience, we will use this notation, rather than $R.M$, $R.OL_i$, etc. Note that for a DB log, the opcontents is the SQL statement(s) in a transaction. For example, if a DB is labeled with $i = 3$, then $OL_3[j]$.opcontents contains the SQL statements in the $j^{\text{th}}$ DB transaction.

A ubiquitous element in the proofs is the graph $G$ that is constructed by ProcessOpReports. $G$ depends on $Tr$ and $R$, but we will not notate this dependence explicitly. Likewise, when notating directed paths in $G$, we leave $G$ implicit; specifically, the notation $p \rightsquigarrow q$ means that there is directed path from node $p$ to node $q$ in graph $G$.

## B.3 Ordering requests and operations

Our first lemma performs a bit of advance housekeeping; it relates the graph $G$, the Op Count reports ($M$), and the operation log reports ($OL$). (The lemma is phrased in terms of *OpMap*, rather

than operation log reports, but in a successful run, *OpMap* contains one entry for each log entry, per Figure 3.5, line 38.) The lemma says that, if ProcessOpReports succeeds, then $G$ "matches" *OpMap*, in the sense that every operation label in $G$ has an entry in *OpMap*, and every entry in *OpMap* appears in $G$. This in turn means that there is a correspondence between the nodes of $G$ and the operations in the logs.

Note that if ProcessOpReports succeeds, it does not mean that $G$ or the operation logs are "correct"; overall, they could be nonsense. For example, $M(rid)$ could be too big or too small for a given *rid*, meaning that there are spurious or insufficient entries in the logs (and $G$). Or the operations of a given *rid* could be in the wrong operation log (meaning that the reports include a wrong claim about which state item a given operation targets). Or the logged contents of a write operation could be spurious. There are many other cases besides. All of them will be detected during re-execution. Importantly, we do not need to enumerate the forms of incorrectness. Rather, we let the proofs establish end-to-end correctness.

**Lemma 1** (Report consistency)**.** *If ProcessOpReports accepts, then the domain of OpMap (which is all entries in the log files) is exactly the set*

$$T = \{(rid, j) \mid rid \text{ is in the trace and } 1 \leq j \leq M(rid)\},$$

*and*

$$G.Nodes = T \cup \{(rid, 0), (rid, \infty) \mid rid \text{ is in the trace}\}.$$

*Proof.* Take $(rid, j)$ in $T$. By definition of $T$, *rid* is in the trace and $1 \leq j \leq M(rid)$. The final check in CheckLogs (which passed, per the premise) considers this element, and ensures that it is indeed in *OpMap*. Now consider the domain of *OpMap*. An element $(rid, j)$ can be inserted into *OpMap* (Figure 3.5, line 38) only if *rid* is in the trace, $j > 0$, and $j \leq M(rid)$ (lines 30–32).

The second part of the lemma is immediate from the logic in AddProgramEdges (Figure 3.5).

□

The remaining lemmas in this section establish that the order in $G$ is consistent with externally observable events, as well as the claimed ordering in the operation logs. The first step is to prove

that the graph $G_{Tr}$ produced by CreateTimePrecedenceGraph (Figure 3.6) explicitly materializes the $<_{Tr}$ relation. We use $r_1 \prec r_2$ to denote that there is a directed path from $r_1$ to $r_2$ in $G_{Tr}$.

**Lemma 2** (Correctness of CreateTimePrecedenceGraph)**.** *For all $r_1, r_2$ in Tr, $r_1 <_{Tr} r_2 \iff r_1 \prec r_2$.*

*Proof.* We begin with the $\implies$ direction. Take $r_1, r_2$ with $r_1 <_{Tr} r_2$. Consider a sequence $T$

$$r_1 <_{Tr} s_1 <_{Tr} \cdots <_{Tr} s_n <_{Tr} r_2$$

that is "tight" in that one cannot insert further elements that obey $<_{Tr}$ between the members of this sequence. (At least one such sequence must exist.) We claim that there is a directed path:

$$r_1 \prec s_1 \prec \cdots \prec s_n \prec r_2.$$

Now, if for all adjacent elements $t, u$ in sequence $T$, there is an edge $\langle t, u \rangle$ in $G_{Tr}$, then the claim holds.

Assume toward a contradiction that there are adjacent $t, u$ without such an edge. Then, at the time that CreateTimePrecedenceGraph (Figure 3.6) processes the event (REQUEST, $u$, $\cdot$), request $t$ must have been already evicted from Frontier (if $t$ had not been evicted, then line 10 would have created the edge $\langle t, u \rangle$). This eviction must have been caused by some request $v$. But this implies that (RESPONSE, $t$, $\cdot$) precedes (REQUEST, $v$, $\cdot$) in the trace.[1] Furthermore, (RESPONSE, $v$, $\cdot$) precedes (REQUEST, $u$, $\cdot$) (because the eviction occurred before CreateTimePrecedenceGraph handled (REQUEST, $u$, $\cdot$)). Summarizing, there is a request $v$ for which:

$$t <_{Tr} v <_{Tr} u,$$

which contradicts the assumption that $t$ and $u$ were adjacent in sequence $T$.

For the $\impliedby$ direction, consider $r_1, r_2$ with $r_1 \prec r_2$. If $r_1 \not<_{Tr} r_2$, then the directed path

[1] The detailed justification is that the eviction could have happened only if there is an edge $\langle t, v \rangle$ (by line 13); such an edge can exist only if $t$ was in the Frontier when CreateTimePrecedenceGraph processed (REQUEST, $v$, $\cdot$) (by line 10); and $t$ entered the Frontier after CreateTimePrecedenceGraph processed (RESPONSE, $t$, $\cdot$) (by line 14).

includes an edge $e = \langle s_1, s_2 \rangle$ for which $s_1 \not<_{Tr} s_2$; this follows from the fact that $<_{Tr}$ is transitive. Now, consider the point in CreateTimePrecedenceGraph at which $e$ was added (line 10). At that point, $s_1$ was in Frontier, which implies that (RESPONSE, $s_1$, $\cdot$) was observed already in the scan. This implies that (RESPONSE, $s_1$, $\cdot$) precedes (REQUEST, $s_2$, $\cdot$) in the trace, which means $s_1 <_{Tr} s_2$: contradiction. □

**Lemma 3** ($G$ obeys request precedence)**.** *At the end of ProcessOpReports,* $r_1 <_{Tr} r_2 \iff (r_1, \infty) \rightsquigarrow (r_2, 0)$.

*Proof.* $\implies$ : This follows from the proof of the prior lemma, the application of SplitNodes (Figure 3.5) to $G_{Tr}$, and the fact that for each $s_i$ in $T$, $(s_i, 0) \rightsquigarrow (s_i, \infty)$ (which itself follows from the program edges, added in Figure 3.5, lines 25–26).

$\impliedby$ : Consider $r_1, r_2$ with $(r_1, \infty) \rightsquigarrow (r_2, 0)$. If $r_1 \not<_{Tr} r_2$, then the directed path includes an edge $e = \langle (s_1, \infty), (s_2, 0) \rangle$ for which $s_1 \not<_{Tr} s_2$; this follows from the fact that $<_{Tr}$ is transitive. But if $e$ is an edge in $G$, then $\langle s_1, s_2 \rangle$ is an edge in $G_{Tr}$, which implies, by application of Lemma 2, that $s_1 <_{Tr} s_2$: contradiction. □

**Lemma 4** ($G$ obeys log precedence)**.** *If ProcessOpReports accepts, then for all operation logs* $OL_i$,

$$1 \le j < k \le \text{length}(OL_i) \implies$$
$$(OL_i[j].\text{rid},\ OL_i[j].\text{opnum}) \rightsquigarrow (OL_i[k].\text{rid},\ OL_i[k].\text{opnum}).$$

*Proof.* Fix $OL_i, j$; induct over $k$. Base case: $k = j + 1$. If $OL_i[j].\text{rid} = OL_i[j{+}1].\text{rid}$, then the check in Figure 3.5, line 57 and the existing program edges together ensure that $(OL_i[j].\text{rid},\ OL_i[j].\text{opnum}) \rightsquigarrow (OL_i[j{+}1].\text{rid},\ OL_i[j{+}1].\text{opnum})$. If on the other hand $OL_i[j].\text{rid} \neq OL_i[j{+}1].\text{rid}$, then line 56 in AddStateEdges inserts an edge between the two nodes.

Inductive step: consider $k + 1$. Reasoning identical to the base case gives $(OL_i[k].\text{rid},\ OL_i[k].\text{opnum}) \rightsquigarrow (OL_i[k{+}1].\text{rid},\ OL_i[k{+}1].\text{opnum})$. And the induction hypothesis gives $(OL_i[j].\text{rid},\ OL_i[j].\text{opnum}) \rightsquigarrow (OL_i[k].\text{rid},\ OL_i[k].\text{opnum})$. Combining the two paths establishes the result. □

```
1:  Global Trace Tr, Reports R     // includes OL_i
2:
3:  // S is an op schedule (§B.4)
4:  procedure OOOExec(S)
5:      for each (rid, opnum) in S :
6:          if opnum = 0 :
7:              Read in inputs from request rid in Tr
8:              Allocate program structures for a thread to run rid
9:
10:         else if opnum = ∞ : // check that the thread produces output
11:             Run rid's allocated thread until the next event.
12:             If the event is a state operation or silent exit:
13:                 return REJECT
14:             Write out the produced output
15:
16:         else
17:             Run rid up to, but not including the next event; if the
18:             event is not a state operation, return REJECT
19:
20:             i, optype, oc ← state op parameters from execution
21:             s ← CheckOp(rid, opnum, i, optype, oc)
22:             if optype ∈ {RegisterRead, KvGet, DBOp} :
23:                 state op result ← SimOp(i, s, optype, oc)
24:
25:     if all produced outputs exactly match the responses in Tr :
26:         return ACCEPT
27:     return REJECT
```

Figure B.2: Definition of OOOExec, a variant of ReExec2 (Figure B.1) that executes according to an op schedule (§B.4). A central concept in the correctness proofs is OOOAudit, which is the same as SSCO_AUDIT2 (Figure B.1), except that ReExec2 is replaced with OOOExec.

## B.4 Op schedules and OOOAudit

A lot of the analysis in the proof is with respect to a hypothetical audit procedure, which we call OOOAudit, that is a variant of SSCO_AUDIT2. OOOAudit performs out-of-order execution of requests but not in a grouped way (as in §3.2.1); the specifics are given shortly. OOOAudit relies on an augmented notion of a request schedule, defined below, in which requests are annotated with a per-request op number (or infinity). These annotations are not algorithmically necessary; they are a convenience for the proofs.

**Definition 3** (Op schedule)**.** *An op schedule is a map:*

$$S\colon \mathbb{N} \to \mathit{RequestId} \times (\mathbb{N} \cup \{\infty\}).$$

*For example,*

$$(1,0), (23,0), (1,1), (23,1), (23,2), (23,\infty), (1,2), \ldots$$

**Definition 4** (Well-formed op schedule)**.** *An op schedule S is* well-formed *(with respect to a trace Tr and set of reports R) if (a) S is a permutation of the nodes of the graph G that is constructed by ProcessOpReports, and (b) S respects program order.*

**Definition 5** (OOOAudit)**.** *Define a procedure* OOOAudit*(Trace Tr, Reports R, OpSched S) that is the same as* SSCO_AUDIT2*(Trace Tr, Reports R), except that*

*ReExec2*() *(Figure B.1, line 31)*

*is replaced with*

*OOOExec*(*S*) *(Figure B.2)*

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

$$\mathit{OOOAudit}(\mathit{Tr}, R, S_1) = \mathit{OOOAudit}(\mathit{Tr}, R, S_2).$$

*Proof.* Consider both invocations, one with $S_1$ and one with $S_2$. The schedule ($S_1$ versus $S_2$) does not affect OOOAudit until the line that invokes OOOExec. So ProcessOpReports fails in both executions, or neither. Assume that these procedures succeed, so that both executions reach OOOExec. We need to show that $\text{OOOExec}(S_1) = \text{OOOExec}(S_2)$.

$S_1$ and $S_2$ have the same operations, because they are both constructed from the same graph $G$.

Meanwhile, OOOExec re-executes each request (meaning each rid) in isolation. To see this, notice that none of the lines of OOOExec modifies state that is shared across requests (*OpMap*, *kv*, etc.). Therefore, the program state (contents of PHP variables, current instruction, etc.) of a re-executed request rid evolves deterministically from operation to operation, and hence the handling of each operation for each rid is deterministic. This holds regardless of where the operations of an rid appear in an op schedule, or how the operations are interleaved.

Now, if OOOExec($S_1$) accepts, then all checks pass, and all produced outputs match the responses in the trace. The preceding paragraph implies that OOOExec($S_2$) would encounter the same checks (and pass them), and produce the same outputs. On the other hand, if OOOExec($S_1$) rejects, then there was a discrepancy in one of the checks or in the produced output. OOOExec($S_2$) either observes the same discrepancy, or else it rejected earlier than this, where "early" is with reference to the sequencing in $S_2$. Summarizing, OOOExec($S_1$) and OOOExec($S_2$) deliver the same accept or reject decision, as was to be shown. □

## B.5 Soundness and completeness of OOOAudit

**Lemma 6** (OOOAudit Soundness)**.** *If there exists a well-formed op schedule $S$ for which OOOAudit($Tr, R, S$) accepts, then there exists a request schedule $S''$ with properties (a) and (b) from Definition 2 (Soundness).*

*Proof.* If OOOAudit($Tr, R, S$) accepts, then there are no cycles in the graph $G$ produced by ProcessOpReports (OOOAudit calls into ProcessOpReports, and ProcessOpReports—specifically lines 12–12 in Figure 3.5—would reject if there were a cycle). This means that $G$ can be sorted topologically. Let the op schedule $S'$ be such a topological sort. Define the request schedule $S''$ to be the same as $S'$ but with the opnum component discarded.

$S'$ is well-formed: it contains the operations of $G$, and it respects program order because there are edges of $G$ between every two state operations in the same request. By Lemma 5, OOOAudit($Tr, R, S'$) returns accept.

*Property (b).* Observe that no (*rid*, *opnum*) appears twice in $S'$; this follows from the construc-

tion of $S'$ and $G$. Thus, one can label each $(rid, opnum)$ in $S'$ with its sequence number in $S'$; call that labeling *Seq*. Also, note that for nodes $n_1, n_2$ in $G$, if $n_1 \rightsquigarrow n_2$, then $Seq(n_1) < Seq(n_2)$; this is immediate from the construction of $S'$ as a topological sort of $G$, and below we refer to this fact as "$\rightsquigarrow$ implies $<$".

Now, assume to the contrary that $S''$ does not meet property (b) in Definition 2. Then there exist $r_1, r_2$ with $r_1 <_{Tr} r_2$ and at least one appearance of $r_2$ occurring in $S''$ before at least one appearance of $r_1$. In that case, $S'$ must contain $(r_1, i), (r_2, j)$ such that $Seq(r_2, j) < Seq(r_1, i)$. Thus, we have:

$$
\begin{aligned}
Seq(r_2, j) &< Seq(r_1, i) && \text{[from contrary hypothesis]} \\
&\leq Seq(r_1, \infty) && [\rightsquigarrow \text{ implies } <] \\
&< Seq(r_2, 0) && [\text{Lemma 3}; \rightsquigarrow \text{ implies } <] \\
&\leq Seq(r_2, j) && [\rightsquigarrow \text{ implies } <]
\end{aligned}
$$

which is a contradiction.

*Property (a).* We establish this property by arguing, first, that re-executing (according to the op schedule $S'$) is the same as a physical (online) execution, in which the request scheduling is given by $S'$. This is the longest (and most tedious) step in the proof of soundness. Second, we argue that such a physical execution is equivalent to the earlier notion of operationwise execution (§B.1). To make these arguments, we define two variants of the audit immediately below, and then prove the two equivalences in sub-lemmas:

*Actual.* Define a variant of OOOAudit($Tr, R, S$) called *Actual*(Trace *Tr*, Reports *R*, OpSched *S*). In Actual, there is a physical state object $i$ for each operation log $OL_i$. Execution in Actual proceeds identically to execution in OOOAudit, except that state operations are concretely executed, instead of simulated. Specifically, Actual invokes CheckOp but then, instead of simulating certain operations (Figure B.2, lines 22–23), it performs all operations against the corresponding physical state object.

*Operationwise.* Define *Operationwise*(Trace *Tr*, RequestSched *RS*) to be the same as Actual,

except that:

(i) All checks, including CheckOp, are discarded (notice that the signature of Operationwise does not include the reports that would enable checks).

(ii) Operationwise is not presented with opnums (notice that the schedule argument *RS* is a request schedule, not an op schedule). Instead, Operationwise simulates opnums: when an rid first appears in *RS*, Operationwise does what Actual does when the opnum is 0 (it reads in the inputs, allocates program structures, etc.). Subsequent appearances of rid cause execution through the next operation or the ultimate output.

**Sub-lemma 6a.** *OOOAudit*$(Tr, R, S')$ *and Actual*$(Tr, R, S')$ *produce the same outputs.*

*Proof.* We will argue that every schedule step preserves program state in the two runs. Specifically, we claim that for each $(rid, opnum)$, both runs have the same state at line 24 (Figure B.2). We induct over the state operations in $S'$, turning to the inductive step first.

*Inductive step.* Consider a state operation in $S'$; it has the form $(rid, j)$, where $j \in \{1, 2, \ldots, M(rid)\}$. The induction hypothesis is that for all entries before $(rid, j)$ in $S'$, the two runs have the same state in line 24 (Figure B.2).

If $j = 1$, note that execution proceeds deterministically from thread creation, so lines 20–22 execute the same in the two runs. If $j > 1$, then execution of operation $(rid, j-1)$ was earlier in $S'$ and execution of *rid* "left off" at line 24. The induction hypothesis implies that, at that point, the state in the two runs was the same. From that point, through the current operation's lines 20–22, execution is deterministic. In both cases ($j = 1$ and $j > 1$), CheckOp (line 21) passes in Actual; this is because it passes in OOOAudit (which we know because, as established at the beginning of the proof of the overall lemma, OOOAudit$(Tr, R, S')$ accepts), and Actual and OOOAudit invoke CheckOp with the same parameters.

It remains to show that, if $optype \in \{\text{RegisterRead}, \text{KvGet}, \text{DBOp}\}$, then both runs read the same value in line 23. To this end, we establish below a correspondence between the history of operations in Actual and OOOAudit. Let $\hat{i}, \hat{s} \leftarrow OpMap[rid, j]$. Because CheckOp passes in both Actual and OOOAudit (with the same parameters in both runs), $i = \hat{i}$, and thus both Actual and OOOAudit will interact with the same logical object (Actual does so via physical operations;

OOOAudit consults the corresponding log). We refer to this object as $i$ below.

*Claim. Define $Q$ as $(OL_i[1], \ldots, OL_i[\hat{s}-1])$; if $\hat{s} = 1$, $Q$ is defined to be empty. Then $Q$ describes the operations, in order, that Actual issues against physical state object $i$, prior to $(rid, j)$.*

*Proof.* We will move back and forth between referring to operations by their location in a log ($OL_i[k]$) and by $(rid, opnum)$ (the domain of *OpMap*). There is no ambiguity because CheckLogs (Figure 3.5) ensures a bijection between log entries and $(rid, opnum)$ pairs.

Each of the elements of $Q$, meaning each $(OL_i[k].\text{rid}, OL_i[k].\text{opnum})$, $1 \le k \le \hat{s}-1$, appears in $S'$ before the current operation (in an order that respects the log sequence numbers $1, \ldots, \hat{s}-1$); this follows from Lemma 4 (and the fact that $S'$ is a topological sort). Furthermore, in OOOAudit, these are the *only* operations in $S'$ (before the current operation) that interact with $OL_i$. To establish this, assume to the contrary that there is an additional operation $(rid', j')$ that appears in $S'$ before the current operation, with $OpMap[rid', j'] = (i, t)$, for some $t$. If $t \le \hat{s}$, that violates the aforementioned bijection; if $t > \hat{s}$, that violates Lemma 4.

Now, consider the execution in Actual. If the history of operations to the corresponding physical state object does not match $Q$, then there is an operation in the relevant prefix of $S'$ for which the two runs diverge. Consider the *earliest* such divergence (that is, Actual and OOOAudit are tracking each other up to this operation); call that earliest diverging operation $(rid^*, j^*)$.

Consider what happens when $(rid^*, j^*)$ executes. Both runs produce $i^*$, $optype^*$, $oc^*$, the state op parameters yielded by execution (Figure B.2, line 20). These three parameters are the same across Actual and OOOExec, by application of the induction hypothesis (again using reasoning as we did above: for operation $(rid^*, j^*-1)$, program state was the same in line 24, etc.). Now, consider CheckOp (line 21). Both runs obtain $i', s' \leftarrow OpMap[rid^*, j^*]$, and $ot', oc' \leftarrow OL_{i^*}[s'].\text{optype}, OL_{i^*}[s'].\text{opcontents}$ (Figure B.1, lines 14–15). But CheckOp passes in OOOExec (as argued earlier), and hence, $i^* = i'$, $optype^* = ot'$, $oc^* = oc'$. This means that the state operation issued by Actual corresponds precisely to what the log dictates (same logical object, same operation type, same parameters, etc.).

Thus, if the two runs diverge, it must be in the sequence number. In OOOExec, $(rid^*, j^*)$ causes operation $s'$ (ordinally) to log $OL_{i'}$. If the operation in Actual would be operation number

$s^*$ (ordinally) to object $i'$, where $s^* \neq s'$, then there was an earlier divergence—either Actual did not issue an operation to object $i'$ when OOOExec issued an operation to $OL_{i'}$, or Actual issued an operation to object $i'$ when OOOExec did not issue an operation to $OL_{i'}$. But $(rid^*, j^*)$ was the earliest divergence, so we have a contradiction. □

Now we must establish that the operations actually return the same values in Actual and OOOExec (in Figure B.2, line 23). We begin with RegisterRead. For such operations, Actual returns the current value of the register; by register semantics, this value is that of the most recent "write" in time. Because $Q$ precisely reflects the history of operations in Actual (per the Claim), this most recent write is the RegisterWrite operation in $Q$ with the highest log sequence number. The contents of this operation is precisely what OOOAudit "reads" into program variables in SimOp (see Figure B.1, line 23). Thus, OOOAudit and Actual read the same value into program variables (Figure B.2, line 23).

Now let us consider what happens if *optype* is KvGet or DBOp. OOOAudit invokes either *kv*.get(key, $s$) or *db*.do_trans(transaction, $s$) (Figure B.1, lines 27 and 29). Each of these calls is equivalent to:

- Constructing state by replaying in sequence $OL_i[1], \ldots, OL_i[\hat{s}-1]$ (specifically, the opcontents field of these log entries), and then
- Issuing the operation given by $OL_i[\hat{s}]$.opcontents.

This equivalence is intuitively what *db* and *kv* provide, and we impose this equivalence as the required specification (see §B.7 for implementation considerations). Meanwhile, by the earlier Claim, the history of operations to object $i$ in Actual before the current operation is $OL_i[1], \ldots, OL_i[\hat{s}-1]$. Moreover, the current operation in Actual is given by *optype* and *oc* (Figure B.2, line 20), which respectively equal $OL_i[\hat{s}]$.optype and $OL_i[\hat{s}]$.opcontents; this follows from the fact that CheckOp passes for operation $(rid, j)$ in both executions. Therefore, Actual and OOOExec "see" equivalent histories and an equivalent current operation, for the state object in question. They therefore return the same result (Figure B.2, line 23).

*Base case.* The first state operation in $S'$ has the form $(rid, 1)$. The reasoning proceeds identi-

cally to the inductive step, for $j = 1$. Here in the base case, $Q$ is always empty,[2] but this does not affect the logic. □

**Sub-lemma 6b.** *Execution of program logic and state operations proceeds identically in Actual$(Tr, R, S')$ and Operationwise$(Tr, S'')$. In particular, they produce the same outputs.*

*Proof.* Actual$(Tr, R, S')$ passes all checks, so eliminating them does not affect the flow of execution. Furthermore, aside from the case opnum=0, the opnum component in $S'$ does not influence the flow of execution in Actual; the component only induces checks, which aren't run in Operationwise. For the opnum=0 case, notice that for each $rid$, $(rid, 0)$ always appears in $S'$ before $(rid, j)$, for any $j > 0$ or $j = \infty$ (this is because $S'$ is a topological sort of $G$). Thus, the treatment by Operationwise$(Tr, S'')$ of the first occurrence of $rid$—namely that it is as if Actual is encountering $(rid, 0)$—means that Operationwise and Actual execute this case identically. □

To conclude, recall from the outset of the proof that OOOAudit$(Tr, R, S')$ accepts, which implies that it produces as outputs precisely the responses in the trace. Sub-lemmas 6a and 6b then imply that Operationwise$(Tr, S'')$ produces those outputs too. Meanwhile, Operationwise$(Tr, S'')$ has the precise form of operationwise execution (defined in Section B.1), which completes the argument. □

**Lemma 7** (OOOAudit Completeness)**.** *If the executor executes the given program (under the concurrency model given earlier) and the given report collection procedure, producing trace Tr and reports R, then for any well-formed op schedule S, OOOAudit$(Tr, R, S)$ accepts.*

*Proof.* Consider ProcessOpReports and OOOExec in turn.

**Sub-lemma 7a.** *ProcessOpReports passes.*

*Proof.* If the executor is well-behaved, then CheckLogs passes; this is because a well-behaved executor correctly sets $M$ and places each operation in exactly one log. Under those conditions, the checks in CheckLogs pass.

[2]We know that $Q$ is empty, as follows. Let $\hat{i}, \hat{s} \leftarrow OpMap[rid, 1]$. If $\hat{s} > 1$, then operation $OL_{\hat{i}}[1]$ would have appeared earlier in $S'$, by Lemma 4. Therefore, $\hat{s} = 1$, which, by definition of $Q$, makes $Q$ empty.

Now we need to show that CycleDetect (Figure 3.5, line 12) passes, i.e., we need to show that there are no cycles. If the executor is well-behaved, then there is a total ordering that defines when all log entries were written in the actual online execution; this is because entries are part of the "emissions" from a sequentially consistent execution. Furthermore, we can define in this total ordering "request begin" (which happens at the instant a request begins executing) and "request end" (which happens at the instant a request finishes executing). Notate these events as $(rid, 0)$ and $(rid, \infty)$, respectively. By sequential consistency, the $(rid, 0)$ event must precede all other $(rid, \cdot)$ events in the total ordering, and likewise $(rid, \infty)$ must succeed all other $(rid, \cdot)$ in the total ordering. Also, in the actual execution, if one request began after another ended, a well-behaved executor must have executed all operations for the former after all operations for the latter, so the total ordering respects that property too.

Now, in ProcessOpReports (Figure 3.5), an edge can be added to $G$ only in four cases (lines 56, 25, 26, and 10):

- An edge $(n_1, n_2)$ can be added to indicate that operation $n_1$ occurred before operation $n_2$, in the same log.
- An edge $(n_1, n_2)$ can be added to indicate that operation $n_1$ preceded operation $n_2$ in the same request.
- Edges for $\langle (rid, 0), (rid, 1) \rangle$ and $\langle (rid, m), (rid, \infty) \rangle$ are added, where $m$ is the purported maximum opnum for *rid*.
- If an edge of the form $\langle (r_1, \infty), (r_2, 0) \rangle$ is added, then $r_2$ began after $r_1$ ended.

Observe that in all four cases, an edge $\langle n_1, n_2 \rangle$ is added to the audit-time graph only if operation $n_1$ preceded operation $n_2$ in the total ordering given by the online execution. In other words, the graph edges are always consistent with the total ordering. Thus, if there is a cycle $n_1 \rightsquigarrow \cdots \rightsquigarrow n_1$ in $G$, it means that $n_1$ preceded itself in the total ordering, which contradicts the notion of total ordering. So there are no cycles. □

As established immediately above, $G$ has no cycles. It can therefore be topologically sorted.

**Sub-lemma 7b.** *Define op schedule $S'$ to be a topological sort of graph $G$. Then, the invocation*

*OOOExec*$(S')$*:*

*(i) reproduces the program state of online execution, and*

*(ii) passes all checks*

*Proof.* Induct on the sequence $S'$.

Base case: The first operation in $S'$ has no ancestors in $G$. It is thus the first occurrence of its request and has the form $(rid, 0)$. OOOExec$(S')$ handles this by reading in input and allocating program structures deterministically; this is the same behavior as in the online execution.

Inductive step: assume that the claim holds for the first $\ell - 1$ operations in $S'$. Denote the op with sequence $\ell$ as $(rid, j)$. We reason by cases.

*Case I:* $j = 0$. Same reasoning as the base case.

*Case II:* $j = \infty$. Recall that $M$ is the Op Count report (§B.2). Consider the operation $(rid, M(rid))$; it appears in $S'$ prior to $(rid, j)$ and as the most recent entry for *rid*. This follows from the logic in ProcessOpReports and its callees. The induction hypothesis implies that program state in OOOExec$(S')$ is identical to the original online execution at $(rid, M(rid))$. This means that OOOExec$(S')$ will take the same next step that the original took, in terms of state operation versus exit versus output (because the original followed the program code, just as OOOExec$(S')$ is doing). Now, if that step were something other than an output, that would imply that $M(rid)$ was unfaithful to the online execution, contradicting the premise of a well-behaved executor. So the next interaction is indeed an output (in both executions), meaning that the check in OOOExec$(S')$ in the opnum=$\infty$ case passes (Figure B.2, line 12). And the produced output is the same; in other words, the output produced by OOOExec$(S')$ is the same as what was produced online, which is what is in the trace. Thus, the output sameness check passes.

*Case III:* $j = 1$. By the induction hypothesis, OOOExec$(S')$ and the online execution had the same program state at $(rid, 0)$. This implies that OOOExec$(S')$ will take the same next step that the original took (in terms of state operation versus exit versus output). If that step is an output or exit rather than a state operation, that would imply that the executor inserted a spurious operation in the logs, contradicting the premise of a well-behaved executor. So the step is indeed an operation (in both executions). Being well-behaved, the executor recorded that operation as $(rid, 1)$ in the

appropriate operation log, and this is the operation in question. Furthermore, the contents of the log entry (meaning the fields optype and opcontents) are faithful to the execution. Because of the determinism in passing from $(rid, 0)$ to $(rid, 1)$, the same program state is reproduced during OOOExec($S'$), implying that all checks in CheckOp pass.

RegisterWrite and KvSet operations do not affect program state. Our remaining task under this case is to show that if the op has optype of RegisterRead, KvGet, or DBOp, then OOOExec($S'$) produces the same value that the online execution did. To this end, let $(i, s) = OpMap[rid, 1]$, and consider the first $s-1$ operations to $OL_i$ in the original execution. These operations have been recorded as $OL_i[1], \ldots, OL_i[s-1]$, because the executor, being well-behaved, is tracking operations correctly. Thus, these log entries give the precise history to this state object (in the original execution) at the time of operation number $s$. (Note that this log could be any kind of log: register, key-value, etc.) Call the $s-1$ ops collectively $Q$. At this point, we can pick up the reasoning in the inductive step of Sub-lemma 6a after the Claim, only replacing "Actual" with "online execution".

*Case IV:* $1 < j \leq M(rid)$. $S'$ respects program order, so we can invoke the induction hypothesis on $(rid, j-1)$ to conclude that program state after executing $(rid, j-1)$ is the same in OOOExec($S'$) as it was when that operation was executed online. At this point, the same reasoning as in Case III applies, substituting $(rid, j)$ for $(rid, 1)$ and $(rid, j-1)$ for $(rid, 0)$. □

Sub-lemmas 7a and 7b imply that OOOAudit($Tr, R, S'$) accepts. Applying Lemma 5 to $S$ and $S'$ completes the proof. □

## B.6 Equivalence of OOOAudit and ssco_audit2

Having established the Completeness and Soundness of OOOAudit, it remains to connect the grouped executions (§3.2.1) to those of OOOAudit.

**Lemma 8.** *Given trace Tr and reports R, if* ssco_audit2($Tr, R$) *accepts, then there is a well-formed op schedule S that causes* OOOAudit($Tr, R, S$) *to accept.*

*Proof.* Recall that $C$ denotes the control flow grouping within $R$ (§B.2). One can construct $S$ as follows: Initialize $S$ to empty. Then run SSCO_AUDIT2($Tr$, $R$) (Figure B.1), and every time SSCO_AUDIT2 begins auditing a control flow group $t$, add to $S$ entries $(rid, 0)$ for each $rid$ in the set $C(t)$. Whenever a group issues an operation (which, because the grouped execution does not diverge, the group does together), add $(rid, j)$ to $S$ for each $rid$ in $C(t)$, where $j$ is the running tally of opnums. When the requests write their output (which, again, they do together), add $(rid, \infty)$ to $S$ for each $rid$ in $C(t)$.

*Claim.* For each $rid$, the value of $j$ in $S$ prior to the $(rid, \infty)$ insertion is equal to $M(rid)$. Proof: SSCO_AUDIT2 accepted so passes the line that checks whether a request issues at least $M(rid)$ operations (Figure B.1, line 53), implying that $j \geq M(rid)$. But $(rid, j)$ is in *OpMap* (otherwise line 13 would have rejected), and so, by Lemma 1, $j \leq M(rid)$. So this Claim is established. □

By the Claim, by the fact that $j$ increments (starting from 0), and by the fact that SSCO_AUDIT2's acceptance implies that all trace responses are produced (so all requests are executed), the constructed op schedule $S$ has all nodes from $G$. $S$ also respects program order. It is thus well-formed.

Meanwhile, executing OOOAudit($Tr$, $R$, $S$) would precisely replicate what happens in SSCO_AUDIT2($Tr$, $R$) because the only difference in execution is that the latter interleaves at the instruction and operand level, which does not affect program state; the flow and ordering is otherwise the same. This means that program state is also the same across the two algorithms at the time of issued operations, and hence the produced output is the same.

The checks are also the same. There is a superficial difference in how the "end state" is handled, but observe that both executions reject if a request $rid$ attempts to issue more than $M(rid)$ operations (in that case, the corresponding operation is not in *OpMap*, so CheckOp rejects, specifically line 13, Figure B.1) or if a request attempts to exit, having issued fewer than $M(rid)$ operations (this happens in ReExec2 with an explicit check in Figure B.1, line 53, and in OOOExec because if an operation produces output before the opnum=$\infty$ case, the algorithm rejects in Figure B.2, lines 17–18).

Therefore, if all checks pass in SSCO_AUDIT2($Tr$, $R$), so do all checks in OOOAudit($Tr$, $R$, $S$), and OOOAudit($Tr$, $R$, $S$) accepts. □

Combining Lemma 8 with Lemma 6, we obtain SSCO_AUDIT2's soundness:

**Theorem 9** (SSCO_AUDIT2 soundness)**.** *Given trace Tr and reports R, if* SSCO_AUDIT2$(Tr, R)$ *accepts, then there exists a request schedule with properties (a) and (b) from Definition 2 (Soundness).*

**Theorem 10** (SSCO_AUDIT2 completeness)**.** *If the executor executes the given program (under the concurrency model given earlier) and the given report collection procedure, producing trace Tr and reports R, then* SSCO_AUDIT2$(Tr, R)$ *accepts.*

*Proof.* Use $C$ (the control flow grouping reports) to construct the following op schedule $S$: take each control flow group that SSCO_AUDIT2 would execute, and insert each request's operations in layers: first all of the opnum=0 entries appear for each *rid* in the control flow group, then all of the opnum=1 entries, etc., up through $M(rid)$ for each *rid* in the control flow group, and then all of the $(rid, \infty)$ entries, again for each *rid* in the control flow group. Note that $M(rid)$ must be constant for all rids in a control flow group because otherwise $M$ is wrong or else the control flow grouping is not valid, either of which contradicts the executor being well-behaved.

$S$ respects program order, by construction. $S$ also includes all nodes from $G$. This follows because the executor is well-behaved, implying that $C$ includes all requestIDs in the trace. Meanwhile, $S$ includes $(rid, 0), (rid, 1), \ldots, (rid, M(rid)), (rid, \infty)$ for each of these rids, and those are exactly the nodes of $G$. Thus, $S$ is well-formed.

Lemma 7 implies that OOOAudit$(Tr, R, S)$ accepts. Compare the executions in SSCO_AUDIT2$(Tr, R)$ and OOOAudit$(Tr, R, S)$; the executions have the same logic, except for three differences:

(i) ReExec2 has an explicit check about whether a request issues fewer than $M(rid)$ operations (Figure B.1, line 53), whereas OOOExec has a separate opnum=$\infty$ case (Figure B.2, line 10).

(ii) ReExec2 executes a group from operation $j-1$ to operation $j$ in SIMD-style (§3.2.1) whereas OOOExec round-robins the execution from $j-1$ to $j$, for a group of requests.

(iii) ReExec2 rejects if execution diverges.

Difference (i) was handled in the proof of Lemma 8: the difference is superficial, in that both executions are requiring a request *rid* to issue exactly $M(rid)$ operations.

Difference (ii) does not result in different program state across the two executions. This is because any implementation of SIMD-on-demand (for example, OROCHI's acc-PHP; §3.3.3) is supposed to ensure that the SIMD-style execution is identical to executing each request in the group individually (as is done in OOOAudit($Tr, R, S$)), and so the results of all instructions (including op values, etc.) are the same between SSCO_AUDIT2($Tr, R$) and OOOAudit($Tr, R, S$).

For difference (iii), we have to argue that there is no divergence across requests within a control flow group in SSCO_AUDIT2($Tr, R$). Assume otherwise. Say that the divergence happens between $(rid, j)$ and $(rid, j+1)$, for one or more rids (in some control flow group), and consider the execution of all requests in the group up to $(rid, j)$ in SSCO_AUDIT2($Tr, R$). The program state produced by this execution is equivalent to the program state at the same point in $S$ when executing OOOAudit($Tr, R, S$), because that is the whole point to the SIMD-style execution.

Consider now OOOAudit($Tr, R, S'$), where $S'$ is a topological sort of $G$ (we know that a topological sort exists because there are no cycles in $G$, and we know that there are no cycles in $G$ using the same reasoning as in Sub-lemma 7a). This execution results in the identical program state for each request as OOOAudit($Tr, R, S$), as argued in the proof of Lemma 5. But by Sub-lemma 7b, OOOAudit($Tr, R, S'$) reproduces the original online execution. This implies that if execution diverges during SSCO_AUDIT2($Tr, R$) for two requests in some control flow grouping, then the two requests had different executions during the original online execution. But if they did and if the executor placed them in the same control flow group, the executor is not well-behaved, in contradiction to the premise. □

## B.7 Details of versioned storage

Key-value stores. Recall the requirement referenced in the proof of Sub-lemma 6a: letting $i^*$ identify the key-value store object and its operation log, *invoking* $kv$.get($k, s$) *must be equivalent to creating a snapshot of a key-value store by replaying the operations* $OL_{i^*}[1], \ldots, OL_{i^*}[s-1]$, *and then invoking "get(k)" on that snapshot.*

To meet this requirement, OROCHI (§3.3) implements $kv$ as a map from keys to (seq,value) pairs. The invocation $kv$.Build($OL_{i^*}$) (Figure B.1, line 6) constructs this map from all of the KvSet

operations in $OL_{i^*}$. During re-execution, $kv$.get$(k, s)$ (Figure B.1, line 27) performs a lookup on key $k$ to get a list of (seq,value) pairs, and then performs a search to identify, of these pairs, the one with the highest seq less than $s$ (or false if there is no such pair); $kv$.get returns the corresponding value.

Summarizing, $kv$.get$(k, s)$ returns, of all of the entries in $OL_{i^*}$, the KvSet to key $k$ with highest sequence less than $s$. Meanwhile, if one were to replay $OL_i[1], \ldots, OL_i[s-1]$ to an abstract key-value store and then issue a "get($k$)", one would get the most recent write—which is the same as the highest sequenced one in the set $OL_i[1], \ldots, OL_i[s-1]$. Thus, the implementation matches the requirement.

Databases. Transactions create some complexity. On the one hand, the pseudocode (Figures B.1 and B.2) and proofs treat multiple SQL statements in a transaction as if they are a single operation. On the other hand, in the implementation (and in the model given at the outset; §B.1), code can execute between the individual SQL statements of a transaction.

We briefly describe how to adapt the pseudocode and proofs to the actual system. Our point of reference will be Figure B.1. As a bookkeeping detail, the system maintains a per-query unique *timestamp*. This identifier is not in the operation logs; it's constructed by the verifier. When building the versioned database (Figure B.1, line 8), the verifier assigns each query the timestamp $ts = s \cdot \text{MAXQ} + q$, where $s$ is the enclosing transaction's sequence number in the operation log, MAXQ is the maximum queries allowed in one transaction (10000 in our implementation), and $q$ is the query number within the transaction. Another detail is that, for the database operation log, each entry's opcontents field is structured as an array of queries.

In the pseudocode, we alter lines 46–49 (in Figure B.1). For DBOps, CheckOp and SimOp need to happen in a loop, interleaved with PHP execution. Instead of checking the entire transaction at once, these functions check the individual queries within the transaction. Specifically, using a query's timestamp, CheckOp and SimOp check whether each query produced by program execution is the same as the corresponding query in the operation log, and simulate the queries against versioned storage.

The proofs can regard program state as proceeding deterministically from query to query, in

analogy with how the proofs currently regard program state proceeding deterministically from op to op. This is valid because, per the concurrency and atomic object model, there are no state operations interleaved with the enclosing transaction (§3.3.4, §B.1).

For the system and the proofs to make sense, the versioned database implementation has to meet the following requirement, which is analogous to that given for key-value stores earlier. Let $i^*$ identify the database object and its operation log.

- For timestamp *ts*, let $s = \lfloor ts/\text{MAXQ} \rfloor$ and let $q = ts \bmod \text{MAXQ}$; for convenience, let *queries* = $OL_i[s]$.opcontents.queries.
- The values returned by invoking *db*.do_query(*sql*, *ts*) must be equivalent to:
  - Creating a snapshot of a database by replaying the transactions $OL_{i^*}[1], \ldots, OL_{i^*}[s-1]$ followed by the queries $queries[1], \ldots, queries[q-1]$, and then
  - Issuing the query *sql*.

To meet this requirement, OROCHI (§3.3) implements *db* atop a traditional SQL database, in a manner similar to WARP [92]. Specifically, the database used for the application is augmented with two columns: *start_ts* indicates when a given row was updated to its current value, and *end_ts* indicates when this row is updated to the next value. The invocation *db*.Build($OL_{i^*}$) (Figure B.1, line 8) inserts rows with the relevant column values, using all of the queries in $OL_{i^*}$. During re-execution, *db*.do_query(*sql*, *ts*) obtains its results by passing *sql* to the underlying storage, augmented with the condition $start\_ts \leq ts < end\_ts$.

One can show that this approach meets the requirement above, but the details are tedious.

## B.8 Efficiency of ProcessOpReports (time, space)

In this section, we analyze the time and space needed to execute ProcessOpReports (Figure 3.5) and the space needed to hold *OpMap*. Let $X$ be the total number of requests, $Y$ be the total number of state operations, and $Z$ be the cardinality of the minimum set of edges needed to represent the $<_{Tr}$ relation. Roughly speaking, the more concurrency there is, the higher $Z$ is. For intuition, if there are always $P$ concurrent requests, which arrive in $X/P$ epochs (so all requests in an epoch

are concurrent with each other but succeed all of the requests in the prior epoch), then $Z \approx X \cdot P/2$ (every two adjacent epochs is a bipartite graph with all nodes on one side connecting to all nodes on the other).

**Lemma 11.** *The time and space complexity of ProcessOpReports are both $O(X + Y + Z)$. The space complexity of OpMap is $O(Y)$.*

*Proof.* We begin with time complexity. The graph $G$ is maintained as an adjacency list, so we assume that inserting a node or edge is a constant-time operation. ProcessOpReports first constructs $R.M$, at a cost of $O(X)$; this is not depicted.

After that, ProcessOpReports comprises six procedures: CreateTimePrecedenceGraph, SplitNodes, AddProgramEdges, CheckLogs, AddStateEdges, and CycleDetect.

To analyze CreateTimePrecedenceGraph, notice that, when handling a request's arrival, the algorithm iterates over *Frontier*, the number of iterations being equal to the number of edges connecting this edge to its predecessors. Similarly, when handling the request's arrival, the algorithm iterates over those same edges. So the total number of iterations has the same order complexity as the number of edges added; this is exactly $Z$, because CreateTimePrecedenceGraph adds the optimal number of edges (shown in the next claim). This implies that CreateTimePrecedenceGraph runs in time $O(X + Z)$.

SplitNodes performs a linear pass over the nodes and edges of $G_{Tr}$ so runs in time $O(X + Z)$.

AddProgramEdges and CheckLogs each perform at least one iteration for each state operation and each request, so these are both $O(X + Y)$. AddStateEdges iterates over every state operation in the logs, so it is $O(Y)$.

The dominant cost is CycleDetect. This is done with a standard depth-first search [98, Ch. 22], which is $O(V + E)$, where $V$ is the number of vertices and $E$ is the number of edges in the graph $G$. In our context, $V = 2 \cdot X + Y$, because each state op has a vertex, and we have the $(\cdot, 0)$ and $(\cdot, \infty)$ vertices for each *rid*. To upper-bound $E$, let us analyze each vertex type. The edges into $(rid, 0)$ and out of $(rid, \infty)$ are "split" from the original $Z$ edges that CreateTimePrecedenceGraph added to $G_{Tr}$; additionally, the out-edges from the $(rid, 0)$ vertices and the in-edges to the $(rid, \infty)$ vertices add an additional $2X$ edges total. An op vertex can have 4 edges at most: 2 in-edges and

2 out-edges, because in the worst case there is one in-edge imposed by program order and one in-edge imposed by log order, and likewise for out-edges. So an upper-bound on the number of edges is $2 \cdot X + 4 \cdot Y + Z$ (which is loose, as there cannot be more than $Y$ "log" edges). Summing, $O(V + E) = O(X + Y + Z)$, as claimed.

*Space complexity.* The trace $Tr$ and reports are $O(X)$ and $O(Y)$, respectively; $R.M$ is $O(X)$. The space of the graph $G$ is proportional to the sum of vertices and edges, which as established above is $O(X + Y + Z)$. Finally, *OpMap* is $O(Y)$ because there is one entry for each state operation. □

**Lemma 12.** *CreateTimePrecedenceGraph adds the minimum number of edges sufficient to capture the $<_{Tr}$ relation.*

*Proof.* The argument is very similar to Theorem 5 in the full version of Anderson et al. [60]; we rehearse it here.

Define the set of edges OPT as the minimum-sized set of edges in $G_{Tr}$ such that for all requests $r_1, r_2$: $r_1 <_{Tr} r_2 \iff$ there is a directed path in OPT from $r_1$ to $r_2$. We want to establish that the set of edges added by CreateTimePrecedenceGraph, call it $E$, is a subset of OPT.

If not, then there is an edge $e \in E$ but $e \notin$ OPT; label the vertices of $e$ as $r_1$ and $r_2$. Because $e \in E$, Lemma 2 implies that $r_1 <_{Tr} r_2$. But this implies, by definition of OPT, that there is a directed path from $r_1$ to $r_2$ in OPT. Yet, $e \notin$ OPT, which implies that there is at least one other request $r_3$ such that there are directed paths in OPT from $r_1$ to $r_3$ and from $r_3$ to $r_2$. This in turn means, again by definition of OPT, that

$$r_1 <_{Tr} r_3 <_{Tr} r_2.$$

However, if this is the case, then $r_3$ would have evicted $r_1$ (or an intermediate request) from the frontier by the time that $r_2$ arrived. Which implies that $r_2$ could not have been connected to $r_1$ in $E$. This is a contradiction. □

We established that $E$, which captures the relation $<_{Tr}$ (per Lemma 2), is a subset of OPT. Yet, OPT is the smallest set of edges needed to capture the relation. Therefore, $E$ and OPT are equal.

# C | Correctness Proof of Cobra's Audit Algorithm

## C.1 The validity of cobra's encoding

Recall the crucial fact in Section 4.2.3: an acyclic graph that is compatible with a polygraph constructed from a history exists iff that history is serializable [167]. In this section, we establish the analogous statement for cobra's encoding. We do this by following the contours of Papadimitriou's proof of the baseline statement [167]. However cobra's algorithm requires that we attend to additional details, complicating the argument somewhat.

### C.1.1 Definitions and preliminaries

In this section, we define the terms used in our main argument (§C.1.2): *history*, *serial schedule*, *cobra polygraph*, and *chains*.

**History and serial schedule.** The description of histories and serial schedules below restates what is in section 4.2.2.

A *history* is a set of read and write operations, each of which belongs to a transaction.[1] Each write operation in the history has a key and a value as its arguments; each read operation has a key as argument, and a value as its result. The result of a read operation is the same as the value argument of a particular write operation; we say that this read operation *reads from* this write

[1]The term "history" [167] was originally defined on a fork-join parallel program schema. We have adjusted the definition to fit our setup (§4.2).

operation. We assume each value is unique and can be associated to the corresponding write; in practice, this is guaranteed by COBRA's client library described in Section 4.5. We also say that a transaction $tx_i$ *reads (a key k) from* another transaction $tx_j$ if: $tx_i$ contains a read *rop*, *rop* reads from write *wrop* on $k$, and $tx_j$ contains the write *wrop*.

A *serial schedule* is a total order of all operations in a history such that no transactions overlap. A history is *equivalent* to a serial schedule if: they have the same operations, and executing the operations in the schedule order on a single-copy set of data results in the same read results as in the history. So a read reading-from a write indicates that this write is the read's *most recent write* (to this key) in a serial schedule.

**Definition 1** (Serializable history)**.** *A* serializable history *is a history that is equivalent to a serial schedule.*

**COBRA polygraph.** In the following, we define a COBRA polygraph; this is a helper notion for the known graph ($g$ in the definition below) and generalized constraints (*con* in the definition below) mentioned in Section 4.3.

**Definition 2** (COBRA polygraph)**.** *Given a history h, a* COBRA polygraph $Q(h) = (g, con)$ *where g and con are generated by* CONSTRUCTENCODING *from Figure 4.3.*

We call a directed graph $\hat{g}$ *compatible* with a COBRA polygraph $Q(h) = (g, con)$, if $\hat{g}$ has the same nodes as $g$, includes the edges from $g$, and selects one edge set from each constraint in *con*.

**Definition 3** (Acyclic COBRA polygraph)**.** *A COBRA polygraph $Q(h)$ is* acyclic *if there exists an acyclic graph that is compatible with $Q(h)$.*

**Chains.** When constructing a COBRA polygraph from a history, function COMBINEWRITES in COBRA's algorithm (Figure 4.3) produces *chains*. One chain is an ordered list of transactions, associated to a key $k$, that (supposedly) contains a sequence of consecutive writes (defined below in Definition 5) on key $k$. In the following, we will first define what is a sequence of consecutive writes and then prove that a chain is indeed such a sequence.

**Definition 4** (Successive write)**.** *In a history, a transaction $tx_i$ is a* successive write *of another transaction $tx_j$ on a key $k$, if (1) both $tx_i$ and $tx_j$ write to $k$ and (2) $tx_i$ reads $k$ from $tx_j$.*

**Definition 5** (A sequence of consecutive writes)**.** A sequence of consecutive writes *on a key $k$ of length $n$ is a list of transactions $[tx_1, \ldots, tx_n]$ for which $tx_i$ is a successive write of $tx_{i-1}$ on $k$, for $1 < i \leq n$.*

Although the overall problem of detecting serializability is NP-complete [167], there are *local* malformations, which immediately indicate that a history is not serializable. We capture two of them in the following definition:

**Definition 6** (An easily rejectable history)**.** An easily rejectable history *$h$ is a history that either (1) contains a transaction that has multiple successive writes on one key, or (2) has a cyclic known graph $g$ of $Q(h)$.*

An easily rejectable history is not serializable. First, if a history has condition (1) in the above definition, there exist at least two transactions that are successive writes of the same transaction (say $tx_i$) on some key $k$. And, these two successive writes cannot be ordered in a serial schedule, because whichever is scheduled later would read $k$ from the other rather than from $tx_i$. Second, if there is a cycle in the known graph, this cycle must include multiple transactions (because there are no self-loops, since we assume that transactions never read keys after writing to them). The members of this cycle cannot be ordered in a serial schedule.

**Lemma 7.** *COBRA rejects easily rejectable histories.*

*Proof.* COBRA (the algorithm in Figure 4.3 and the constraint solver) detects and rejects easily rejectable histories as follows. (1) If a transaction has multiple successive writes on the same key in $h$, COBRA's algorithm explicitly detects this case. The algorithm checks, for transactions reading and writing the same key (line 21), whether multiple of them read this key from the same transaction (line 23). If so, the transaction being read has multiple successive writes, hence the algorithm rejects (line 24). (2) If the known graph has a cycle, COBRA detects and rejects this history when checking acyclicity in the constraint solver. □

On the other hand, if a history is not easily rejectable, we want to argue that each chain produced by the algorithm is a sequence of consecutive writes.

**Claim 8.** *If COBRA's algorithm makes it to line 35 (immediately before CombineWrites), then from this line on, any transaction writing to a key $k$ appears in exactly one chain on $k$.*

*Proof.* Prior to line 35, COBRA's algorithm loops over all the write operations (line 32–33), creating a chain for each one (line 34). As in the literature [167, 206], we assume that each transaction writes to a key only once. Thus, any *tx* writing to a key $k$ has exactly one write operation to $k$ and hence appears in exactly one chain on $k$ in line 35.

Next, we argue that CombineWrites preserves this invariant. This suffices to prove the claim, because after line 35, only CombineWrites updates chains (variable *chains* in the algorithm).

The invariant is preserved by CombineWrites because each of its loop iterations splices two chains on the same key into a new chain (line 53) and deletes the two old chains (line 52). From the perspective of a transaction involved in a splicing operation, its old chain on key $k$ has been destroyed, and it has joined a new one on key $k$, meaning that the number of chains it belongs to on key $k$ is unchanged: the number remains 1. □

One clarifying fact is that a transaction can appear in multiple chains on different keys, because a transaction can write to multiple keys.

**Claim 9.** *If COBRA's algorithm does not reject in line 24, then after CreateKnownGraph, for any two distinct entries $ent_1$ and $ent_2$ (in the form of $\langle key, tx_i, tx_j \rangle$) in wwpairs: if $ent_1$.key = $ent_2$.key, then $ent_1.tx_i \neq ent_2.tx_i$ and $ent_1.tx_j \neq ent_2.tx_j$.*

*Proof.* First, we prove $ent_1.\text{tx}_i \neq ent_2.\text{tx}_i$. In COBRA's algorithm, line 25 is the only point where new entries are inserted into *wwpairs*. Because of the check in line 23–24, the algorithm guarantees that a new entry will not be inserted into *wwpairs* if an existing entry has the same $\langle key, tx_i \rangle$. Also, existing entries are never modified. Thus, there can never be two entries in *wwpairs* indexed by the same $\langle key, tx_i \rangle$.

Second, we prove $ent_1.\text{tx}_j \neq ent_2.\text{tx}_j$. As in the literature [167, 206], we assume that one transaction reads a key at most once.[2] As a consequence, the body of the loop in line 21, including line 25, is executed at most once for each (key,tx) pair. Therefore, there cannot be two entries in *wwpairs* that match $\langle key, \_, tx \rangle$. □

**Claim 10.** *In one iteration of* CombineWrites *(line 46), for* $ent_i = \langle key, tx_1, tx_2 \rangle$ *retrieved from wwpairs, there exist* $chain_1$ *and* $chain_2$*, such that* $tx_1$ *is the tail of* $chain_1$ *and* $tx_2$ *is the head of* $chain_2$*.*

*Proof.* Invoking Claim 8, denote the chain on *key* that $tx_1$ is in as $chain_i$; similarly, denote $tx_2$'s chain as $chain_j$.

Assume to the contrary that $tx_1$ is not the tail of $chain_i$. Then there is a transaction $tx'$ next to $tx_1$ in $chain_i$. But the only way for two transactions ($tx_1$ and $tx'$) to appear adjacent in a chain is through the concatenation in line 53, and that requires an entry $ent_j = \langle key, tx_1, tx' \rangle$ in *wwpairs*. Because $tx'$ is already in $chain_i$ when the current iteration happens, $ent_j$ must have been retrieved in some prior iteration. Since $ent_i$ and $ent_j$ appear in different iterations, they are two distinct entries in *wwpairs*. Yet, both of them are indexed by $\langle key, tx_1 \rangle$, which is impossible, by Claim 9.

Now assume to the contrary that $tx_2$ is not the head of $chain_j$. Then $tx_2$ has an immediate predecessor $tx'$ in $chain_j$. In order to have $tx'$ and $tx_2$ appear adjacent in $chain_j$, there must be an entry $ent_k = \langle key, tx', tx_2 \rangle$ in *wwpairs*. Because $tx'$ is already in $chain_j$ when the current iteration happens, $ent_k$ must have been retrieved in an earlier iteration. So, $ent_k = \langle key, tx', tx_2 \rangle$ and $ent_i = \langle key, tx_1, tx_2 \rangle$ are distinct entries in *wwpairs*, which is impossible, by Claim 9. □

**Lemma 11.** *If* $h$ *is not easily rejectable, every chain is a sequence of consecutive writes after* CombineWrites*.*

*Proof.* Because $h$ is not easily rejectable, it doesn't contain any transaction that has multiple successive writes. Hence, COBRA does not reject in line 24 and can make it to CombineWrites.

At the beginning (immediately before CombineWrites), all chains are single-element lists (line 34). By Definition 5, each chain is a sequence of consecutive writes with only one transaction.

[2] In our implementation, this assumption is guaranteed by COBRA's client library (§4.5).

Assume that, before loop iteration $t$, each chain is a sequence of consecutive writes. We show that after iteration $t$ (before iteration $t+1$), chains are still sequences of consecutive writes.

If $t \leq \text{size}(wwpairs)$, then in line 46, COBRA's algorithm gets an entry $\langle key, tx_1, tx_2 \rangle$ from $wwpairs$, where $tx_2$ is $tx_1$'s successive write on $key$. Also, we assume one transaction does not read from itself (as in the literature [167, 206]), and since $tx_2$ reads from $tx_1$, $tx_1 \neq tx_2$. Then, the algorithm references the chains that they are in: $chain_1$ and $chain_2$.

First, we argue that $chain_1$ and $chain_2$ are distinct chains. By Claim 8, no transaction can appear in two chains on the same key, so $chain_1$ and $chain_2$ are either distinct chains or the same chain. Assume they are the same chain ($chain_1 = chain_2$). If $chain_1$ ($= chain_2$) is a single-element chain, then $tx_1$ (in $chain_1$) is $tx_2$ (in $chain_2$), a contradiction to $tx_1 \neq tx_2$.

Consider the case that $chain_1$ ($= chain_2$) contains multiple transactions. Because $tx_2$ reads from $tx_1$, there is an edge $tx_1 \rightarrow tx_2$ (generated from line 17) in the known graph of $Q(h)$. Similarly, because $chain_1$ is a sequence of consecutive writes (the induction hypothesis), any transaction $tx$ in $chain_1$ reads from its immediate prior transaction, hence there is an edge from this prior transaction to $tx$. Since every pair of adjacent transactions in $chain_1$ has such an edge, the head of $chain_1$ has a path to the tail of $chain_1$. Finally, by Claim 10, $tx_2$ is the head of $chain_2$ and $tx_1$ is the tail of $chain_1$, as well as $chain_1 = chain_2$, there is a path $tx_2 \rightsquigarrow tx_1$. Thus, there is a cycle ($tx_1 \rightarrow tx_2 \rightsquigarrow tx_1$) in the known graph, so $h$ is easily rejectable, a contradiction.

Second, we argue that the concatenation of $chain_1$ and $chain_2$, denoted as $chain_{1+2}$, is a sequence of consecutive writes. Say the lengths of $chain_1$ and $chain_2$ are $n$ and $m$ respectively. Since $chain_1$ and $chain_2$ are distinct sequences of consecutive writes, all transactions in $chain_{1+2}$ are distinct and $chain_{1+2}[i]$ reads from $chain_{1+2}[i-1]$ for $i \in \{2, \ldots, n+m\} \setminus \{n+1\}$. For $i = n+1$, the preceding also holds, because $tx_1$ is $chain_1$'s tail ($= chain_{1+2}[n]$), $tx_2$ is $chain_2$'s head ($= chain_{1+2}[n+1]$), and $tx_2$ is the successive write of $tx_1$ ($tx_2$ reads from $tx_1$). Thus, $chain_{1+2}$ is a sequence of consecutive writes, according to Definition 5.

If $t > \text{size}(wwpairs)$ and the loop ends, then chains don't change. As they are sequences of consecutive writes after the final step (when $t = \text{size}(wwpairs)$), they still are after CombineWrites.

□

In the following, when we refer to chains, we mean the state of chains after executing CombineWrites.

### C.1.2 The main argument

This section's two theorems (Theorem 12 and 17) together prove the validity of cobra's encoding.

**Theorem 12.** *If a history $h$ is serializable, then $Q(h)$ is acyclic.*

*Proof.* Because $h$ is serializable, there exists a serial schedule $\hat{s}$ that $h$ is equivalent to.

**Claim 13.** *For a transaction rtx that reads from a transaction wtx in h, rtx appears after wtx in $\hat{s}$.*

*Proof.* This follows from the definitions given at the start of the section: if *rtx* reads from *wtx* in *h*, then there is a read operation *rop* in *rtx* that reads from a write operation *wrop* in *wtx*. Thus, as stated earlier and by definition of matching, *rop* appears later than *wrop* in $\hat{s}$. Furthermore, by definition of serial schedule, transactions don't overlap in $\hat{s}$. Therefore, all of *rtx* appears after all of *wtx* in $\hat{s}$. □

**Claim 14.** *For any pair of transactions (rtx, wtx) where rtx reads a key k from wtx in h, no transaction wtx′ that writes to k can appear between wtx and rtx in $\hat{s}$.*

*Proof.* Assume to the contrary that there exists *wtx′* that appears in between *wtx* and *rtx* in $\hat{s}$. By Claim 13, *rtx* appears after *wtx* in $\hat{s}$. Therefore, *wtx′* appears in $\hat{s}$ before *rtx* and after *wtx*. Thus, in $\hat{s}$, *rtx* does not return the value of *k* written by *wtx*. But in *h*, *rtx* returns the value of *k* written by *wtx*. Thus, $\hat{s}$ and *h* are not equivalent, a contradiction. □

In the following, we use $head_k$ and $tail_k$ as shorthands to represent, respectively, the head transaction and the tail transaction of $chain_k$. And, we denote that $tx_i$ appears before $tx_j$ in $\hat{s}$ as $tx_i <_{\hat{s}} tx_j$.

**Claim 15.** *For any pair of chains $(chain_i, chain_j)$ on the same key k, if $head_i <_{\hat{s}} head_j$, then (1) $tail_i <_{\hat{s}} head_j$ and (2) for any transaction rtx that reads k from $tail_i$, $rtx <_{\hat{s}} head_j$.*

*Proof.* First, we prove $tail_i <_{\hat{s}} head_j$. If $head_j \in chain_j$ then $head_j \notin chain_i$, by Claim 8. If $chain_i$ has only one transaction (meaning $head_i = tail_i$), then $tail_i = head_i <_{\hat{s}} head_j$.

Next, if $chain_i$ is a multi-transaction chain, it can be written as

$$tx_1, \cdots tx_p, tx_{p+1}, \cdots tx_n.$$

By Lemma 11, $chain_i$ is a sequence of consecutive writes on $k$, so each transaction reads $k$ from its prior transaction in $chain_i$. Then, by Claim 13, $tx_p <_{\hat{s}} tx_{p+1}$, for $1 \leq p < n$. Now, assume to the contrary that $head_j <_{\hat{s}} tail_i$ $(= tx_n)$. Then, by the given, $tx_1 (= head_i) <_{\hat{s}} head_j <_{\hat{s}} tx_n$. Thus, for some $1 \leq p < n$, we have $tx_p <_{\hat{s}} head_j <_{\hat{s}} tx_{p+1}$. But this is a contradiction, because $tx_{p+1}$ reads $k$ from $tx_p$, and thus by Claim 14, $head_j$ cannot appear between them in $\hat{s}$.

Second, we prove that any transaction $rtx$ that reads $k$ from $tail_i$ appears before $head_j$ in $\hat{s}$. Assume to the contrary that $head_j <_{\hat{s}} rtx$. We have from the first half of the claim that $tail_i <_{\hat{s}} head_j$. Thus, $head_j$ appears between $tail_i$ and $rtx$ in $\hat{s}$, which is again a contradiction, by Claim 14.

□

Now we prove that $Q(h)$ is acyclic by constructing a compatible graph $\hat{g}$ and proving $\hat{g}$ is acyclic. We have the following fact from function Coalesce.

**Fact 16.** *In Coalesce, each constraint $\langle A, B \rangle$ is generated from a pair of chains $(chain_1, chain_2)$ on the same key $k$ in line 62. All edges in edge set $A$ point to $head_2$, and all edges in $B$ point to $head_1$. This is because all edges in $A$ have the form either $(rtx, head_2)$ or $(tail_1, head_2)$; see lines 69 and 73–74. Similarly by swapping $chain_1$ and $chain_2$ (line 63 and 64), edges in $B$ point to $head_1$.*

We construct graph $\hat{g}$ as follows: first, let $\hat{g}$ be the known graph of $Q(h)$. Then, for each constraint $\langle A, B \rangle$ in $Q(h)$, and letting $head_1$ and $head_2$ be defined as in Fact 16, add $A$ to $\hat{g}$ if $head_1 <_{\hat{s}} head_2$, and otherwise add $B$ to $\hat{g}$. This process results in a directed graph $\hat{g}$.

Next, we show that all edges in $\hat{g}$ are a subset of the total ordering in $\hat{s}$; this implies $\hat{g}$ is acyclic.

First, the edges in the known graph (line 17 and 60) are a subset of the total ordering given by $\hat{s}$. Each edge added in line 17 represents that the destination vertex reads from the source vertex in $h$. By Claim 13, this ordering holds in $\hat{s}$. As for the edges in line 60, they are added to capture

the fact that a read operation (in transaction *rtx*) that reads from a write (in transaction $chain[i]$) is sequenced before the next write on the same key (in transaction $chain[i+1]$), an ordering that also holds in $\hat{s}$. (This is known as an *anti-dependency* in the literature [53].) If this ordering doesn't hold in $\hat{s}$, then $chain[i+1] <_{\hat{s}} rtx$, and thus $chain[i] <_{\hat{s}} chain[i+1] <_{\hat{s}} rtx$, which contradicts Claim 14.

Second, consider the edges in $\hat{g}$ that come from constraints. Take a constraint $\langle A, B \rangle$ generated from chains $(chain_1,\ chain_2)$ on the same key. If $head_1 <_{\hat{s}} head_2$, then by Fact 16 and construction of $\hat{g}$, all added edges have the form $(tail_1, head_2)$ or $(rtx, head_2)$, where *rtx* reads from $tail_1$. By Claim 15, the source vertex of these edges appears prior to $head_2$ in $\hat{s}$; thus, these edges respect the ordering in $\hat{s}$. When $head_2 <_{\hat{s}} head_1$, the foregoing argument works the same, with appropriate relabeling. Hence, all constraint edges chosen in $\hat{g}$ are a subset of the total ordering given by $\hat{s}$. This completes the proof. □

**Theorem 17.** *If $Q(h)$ is acyclic, then the history $h$ is serializable.*

*Proof.* Given that $Q(h)$ is acyclic, COBRA accepts $h$. Hence, by Lemma 7, $h$ is not easily rejectable. And, by Lemma 11, each chain (after CombineWrites) is a sequence of consecutive writes.

Because $Q(h)$ is acyclic, there must exist an acyclic graph $q$ that is compatible with $Q(h)$.

**Claim 18.** *If $tx_i$ appears before $tx_j$ in a chain $chain_k$, then graph $q$ has $tx_i \rightsquigarrow tx_j$.*

*Proof.* Because $chain_k$ is a sequence of consecutive writes, a transaction *tx* in $chain_k$ reads from its immediate predecessor in $chain_k$, hence there is an edge in the known graph (generated by line 17) from the predecessor to *tx*. Because every pair of adjacent transactions in $chain_k$ has such an edge and $tx_i$ appears before $tx_j$ in $chain_k$, $tx_i \rightsquigarrow tx_j$ in $Q(h)$'s known graph. As $q$ is compatible with $Q(h)$, such a path from $tx_i$ to $tx_j$ also exists in $q$. □

**Claim 19.** *For any chain $chain_i$ (on a key $k$) and any transaction $wtx_j \notin chain_i$ that writes to $k$, graph $q$ has either: (1) paths from $tail_i$ and transactions that read keyfrom $tail_i$ (if any) to $wtx_j$, or (2) paths from $wtx_j$ to all the transactions in $chain_i$.*

*Proof.* Call the chain that $wtx_j$ is in $chain_j$. By Claim 8, $chain_j$ exists and $chain_j \neq chain_i$.

For $chain_i$ and $chain_j$, $Q(h)$ has a constraint $\langle A, B\rangle$ that is generated from them (line 29). This is because $chain_i$ and $chain_j$ touch the same key $k$, and COBRA's algorithm creates one constraint for every pair of chains on the same key (line 41). (We assume $chain_i$ is the first argument of function COALESCE and $chain_j$ is the second.)

First, we argue that the edges in edge set $A$ establish $tail_i \rightsquigarrow head_j$ and $rtx \rightsquigarrow head_j$ ($rtx$ reads $k$ from $tail_i$) in the known graph; and $B$ establishes $tail_j \rightsquigarrow head_i$. Consider edge set $A$. There are two cases: (i) there are reads $rtx$ reading from $tail_i$, and (ii) there is no such read. In case (i), the algorithm adds $rtx \rightarrow head_j$ for every $rtx$ reading-from $tail_i$ (line 73–74). And $rtx \rightarrow head_j$ together with the edge $tail_i \rightarrow rtx$ (added in line 17) establish $tail_i \rightsquigarrow head_j$. In case (ii), COBRA's algorithm adds an edge $tail_i \rightarrow head_j$ to $A$ (line 69), and there is no $rtx$ in this case. Similarly, by switching $i$ and $j$ in the above reasoning (except we don't care about the reads in this case), edges in $B$ establish $tail_j \rightsquigarrow head_i$.

Second, because $q$ is compatible with $Q(h)$, it either (1) contains $A$:

$$\begin{aligned} tail_i / rtx &\rightsquigarrow head_j && \text{[proved in the first half]} \\ &\rightsquigarrow wtx_j && \text{[Claim 18; } wtx_j \in chain_j\text{]} \end{aligned}$$

or else (2) contains $B$:

$$\begin{aligned} wtx_j &\rightsquigarrow tail_j && \text{[Claim 18; } wtx_j \in chain_j\text{]} \\ &\rightsquigarrow head_i && \text{[proved in the first half]} \\ &\rightsquigarrow tx && \text{[Claim 18; } tx \in chain_i\text{]} \end{aligned}$$

The argument still holds if $wtx_j = head_j$ in case (1): remove the second step in (1). Likewise, if $wtx_j = tail_j$ in case (2), remove the first step in (2). □

**Claim 20.** *For any pair of transactions $(wtx, rtx)$ where $rtx$ reads a key $k$ from $wtx$ and any other transaction $wtx'$ that writes to $k$, graph $q$ has either $wtx' \rightsquigarrow wtx$ or $rtx \rightsquigarrow wtx'$.*

*Proof.* By Claim 8, $wtx$ must appear in some chain $chain_i$ on $k$. Each of the three transactions ($wtx$, $rtx$, and $wtx'$) has two possibilities relative to $chain_i$:

1. $wtx$ is either the tail or non-tail of $chain_i$.
2. $rtx$ is either in $chain_i$ or not.
3. $wtx'$ is either in $chain_i$ or not.

In the following, we enumerate all combinations of the above possibilities and prove the claim in all cases.

- $wtx = tail_i$.

  Then, $rtx$ is not in $chain_i$. (If $rtx$ is in $chain_i$, its enclosing transaction would have to be subsequent to $wtx$ in $chain_i$, which is a contradiction, since $wtx$ is last in the chain.)

  - $wtx' \in chain_i$.

    Because $wtx$ is the tail, $wtx'$ appears before $wtx$ in $chain_i$. Thus, $wtx' \rightsquigarrow wtx$ in $q$ (Claim 18).

  - $wtx' \notin chain_i$.

    By invoking Claim 19 for $chain_i$ and $wtx'$, $q$ either has (1) paths from each read ($rtx$ is one of them) reading-from $tail_i$ ($= wtx$) to $wtx'$, therefore $rtx \rightsquigarrow wtx'$. Or else $q$ has (2) paths from $wtx'$ to every transaction in $chain_i$, and $wtx \in chain_i$, thus $wtx' \rightsquigarrow wtx$.

- $wtx \neq tail_i \wedge wtx \in chain_i$.

  - $rtx \in chain_i$.

    Because $chain_i$ is a sequence of consecutive writes on $k$ (Lemma 11) and $rtx$ reads $k$ from $wtx$, $rtx$ is the successive write of $wtx$. Therefore, $rtx$ appears immediately after $wtx$ in $chain_i$.

    - $wtx' \in chain_i$.

      Because $rtx$ appears immediately after $wtx$ in $chain_i$, $wtx'$ either appears before $wtx$ or after $rtx$. By Claim 18, there is either $wtx' \rightsquigarrow wtx$ or $rtx \rightsquigarrow wtx'$ in $q$.

    - $wtx' \notin chain_i$.

      By invoking Claim 19 for $chain_i$ and $wtx'$, $q$ either has (1) $tail_i \rightsquigarrow wtx'$, together with $rtx \rightsquigarrow tail_i$ (or $rtx = tail_i$) by Claim 18, therefore $rtx \rightsquigarrow wtx'$. Or else $q$ has (2) $wtx' \rightsquigarrow wtx$ ($wtx'$ has a path to every transaction in $chain_i$, and $wtx \in chain_i$).

  - $rtx \notin chain_i$.

If $rtx \notin chain_i$, because of InferRWEdges (line 55), $rtx$ has an edge (in the known graph, hence in $q$) to the transaction that immediately follows $wtx$ in $chain_i$, denoted as $wtx^*$ (and $wtx^*$ must exist because $wtx$ is not the tail of the chain).

- $wtx' \in chain_i$.

  Because $wtx^*$ appears immediately after $wtx$ in $chain_i$, $wtx'$ either appears before $wtx$ or after $wtx^*$. By Claim 18, $q$ has either $wtx' \rightsquigarrow wtx$ or $wtx^* \rightsquigarrow wtx'$ which, together with edge $rtx \longrightarrow wtx^*$ from InferRWEdges, means $rtx \rightsquigarrow wtx'$.

- $wtx' \notin chain_i$.

  By invoking Claim 19 for $chain_i$ and $wtx'$, $q$ has either (1) $tail_i \rightsquigarrow wtx'$ which, together with $rtx \longrightarrow wtx^*$ (from InferRWEdges) and $wtx^* \rightsquigarrow tail_i$ (Claim 18), means $rtx \rightsquigarrow wtx'$. Or else $q$ has (2) $wtx' \rightsquigarrow wtx$ ($wtx'$ has a path to every transaction in $chain_i$, and $wtx \in chain_i$).

□

By topologically sorting $q$, we get a serial schedule $\hat{s}$. Next, we prove $h$ is equivalent to $\hat{s}$, hence $h$ is serializable (Definition 1).

Since $h$ and $\hat{s}$ have the same set of transactions (because $q$ has the same transactions as the known graph of $Q(h)$, and thus also the same as $h$), we need to prove only that for every read that reads from a write in $h$, the write is the most recent write to that read in $\hat{s}$.

First, for every pair of transactions ($wtx$, $rtx$) such that $rtx$ reads a key $k$ from $wtx$ in $h$, $q$ has an edge $wtx \longrightarrow rtx$ (added to the known graph in line 17); thus $rtx$ appears after $wtx$ in $\hat{s}$ (a topological sort of $q$). Second, by invoking Claim 20 for ($wtx$, $rtx$), any other transaction writing to $k$ is either "topologically prior" to $wtx$ or "topologically subsequent" to $rtx$. This ensures that, the most recent write of $rtx$'s read (to $k$) belongs to $wtx$ in $\hat{s}$, hence $rtx$ reads the value of $k$ written by $wtx$ in $\hat{s}$ as it does in $h$. This completes the proof. □

## C.2 Garbage collection correctness proof

Section 4.3 and Appendix C.1 describe how COBRA checks serializabilty for a fixed set of transactions. This section introduces how COBRA supports an online database which has an ever-growing history, by verification in rounds.

### C.2.1 Verification in rounds

To handle a continuous and ever-growing history, COBRA verifies in rounds. In each round, COBRA's verifier fetches new transactions and checks serializability incrementally on the transactions received. Figure C.1 depicts the algorithm of verification in rounds.

In the following, we define terms used in the context of verification in rounds: *complete history*, *continuation*, *strong session serializable*, and *extended known graph*.

**Complete history and continuation.** A *complete history* is a prerequisite of checking serializability. If a history is incomplete and some of the transactions are unknown, it is impossible to decide whether this history is serializable.

**Definition 21** (Complete history)**.** *A history $h$ is a* complete history *if all read operations in $h$ read from write operations in $h$.*

In the beginning of each round, COBRA's verifier receives a set of new transactions. We call such newly received transactions a *continuation* [127] if they read either from the transactions in prior rounds or from other newly received transactions, defined below.

**Definition 22** (Continuation)**.** *A* continuation *$r$ of a complete history $h$ is a set of transactions in which all the read operations read from transactions in either $h$ or $r$.*

We denote the combination of a complete history $h$ and its continuation $r$ as $h \circ r$. By Definition 21, $h \circ r$ is also a complete history.

In the following, we assume that the verifier receives a complete history, and the transactions received in each round are a continuation of the history from the previous rounds. However, in

```
1:  procedure VerifySerializability()
2:      g ← empty graph
3:      wwpairs ← empty map {⟨Key, Tx⟩ → Tx}
4:      readfrom ← empty map {⟨Key, Tx⟩ → Set⟨Tx⟩}
5:      while True :
6:          h ← fetch a continuation from history collectors
7:          g, readfrom, wwpairs ←
8:              CreateKnownGraph2(g, readfrom, wwpairs, h) // ln 14
9:
10:         g, con ← EncodeAndSolve(g, readfrom, wwpairs) // ln 35
11:
12:         g, readfrom, wwpairs ← GarbageCollection(g, con) // ln 44
13:
14: procedure CreateKnownGraph2(g, readfrom, wwpairs, h)
15:     for transaction tx in h :
16:         g.Nodes += tx
17:         for read operation rop in tx :
18:             // if read from deleted transactions, reject
19:             if rop.read_from_tx not in g: REJECT
20:             g.Edges += (rop.read_from_tx, tx)
21:             readfrom[⟨rop.key, rop.read_from_tx⟩] += tx
22:         for all Keys key that are both read and written by tx :
23:             rop ← the operation in tx that reads key
24:             if wwpairs[⟨key, rop.read_from_tx⟩] ≠ null :
25:                 REJECT
26:             wwpairs[⟨key, rop.read_from_tx⟩] ← tx
27:
28:     for each session s :          // add SO-edges
29:         list_s ← an ordered list of transactions issued through s in g
30:         for i in [0, length(list_s) − 2] :
31:             g.Edges += (list_s[i], list_s[i + 1])
32:
33:     return g, readfrom, wwpairs
34:
35: procedure EncodeAndSolve(g, readfrom, wwpairs)
36:     con ← GenConstraints(g, readfrom, wwpairs) // Fig 4.3, ln 29
37:     con, g ← Prune(con, g)                     // Fig 4.3, ln 77
38:
39:     encode (con, g); use MonoSAT to solve      // §4.3.4
40:     if MonoSAT outputs unsat: REJECT
41:
42:     return g, con
43:
44: procedure GarbageCollection(g, con)
45:     epoch_agree ← AssignEpoch(g)          // ln 56
46:     SetFrozen(g, epoch_agree)             // ln 81
47:     SetFrontier(g, epoch_agree)           // ln 87
48:     SetRemovable(g, con)                  // ln 97
49:     return SafeDeletion2(g, readfrom, wwpairs) // ln 104
50:
51: procedure GenPolySCCs(g, con)
52:     g′ ← g
53:     g′.Edges ← g′.Edges ∪ {all edges in con}
54:     psccs ← CalcStronglyConnectedComponents(g′) //[98, Ch22]
55:     return psccs
56: procedure AssignEpoch(g)
57:     epoch_num ← 0
58:     topo_tx ← TopologicalSort(g) // see [98, Ch22]
59:     for tx in topo_tx :          // assign epoch to Wfences
60:         if tx writes to key "EPOCH" :
61:             tx.epoch ← epoch_num
62:             epoch_num ← epoch_num + 1
63:     for tx in topo_tx :          // assign epoch to Rfences
64:         if tx reads but not writes key "EPOCH" :
65:             tx.epoch ← tx.read_from_tx.epoch
66:
67:     epoch_agree ← inf
68:     for each session s :         // assign epoch to normal transactions
69:         list_s ← an ordered list of transactions issued through s in g
70:         rlist_s ← reversed ordered list of list_s
71:         cur_epoch ← inf
72:         for tx in rlist_s :
73:             if tx touches key "EPOCH" :
74:                 if cur_epoch = inf :
75:                     epoch_agree ← min(epoch_agree, tx.epoch)
76:                 cur_epoch ← tx.epoch
77:             else:
78:                 tx.epoch ← (cur_epoch = inf ? inf : cur_epoch − 1)
79:     return epoch_agree
80:
81: procedure SetFrozen(g, epoch_agree)
82:     fepoch ← epoch_agree − 2
83:     for tx in g :
84:         if tx.epoch ≤ fepoch and all tx's predecessors have ≤ fepoch :
85:             tx.frozen ← True
86:
87: procedure SetFrontier(g, epoch_agree)
88:     fepoch ← epoch_agree − 2
89:     frt ← ∅
90:     for Key key in g :
91:         frt += { tx_i ∈ g | tx_i.epoch ≤ fepoch ∧ tx_i writes key
92:             ∧(∄tx_j ∈ g, s.t. tx_i ⇝ tx_j
93:                     ∧tx_j.epoch ≤ fepoch ∧ tx_j writes key) }
94:     for tx in frt :
95:         tx.frontier ← True
96:
97: procedure SetRemovable(g, con)
98:     psccs ← GenPolySCCs(g, con)          // ln 51
99:     for pscc in psccs :
100:        if ∀tx ∈ pscc, tx.frozen = True ∧ tx.frontier = False :
101:            for tx in pscc :
102:                tx.removable ← True
103:
104: procedure SafeDeletion2(g, readfrom, wwpairs)
105:     for tx in g :
106:         if tx.removable = True and tx doesn't touch key "EPOCH" :
107:             g.Nodes −= tx
108:             g.Edges −= {edges with tx as one endpoint}
109:             readfrom −= {tuples containing tx}
110:             wwpairs −= {tuples containing tx}
111:     return g, readfrom, wwpairs
```

Figure C.1: Cobra's algorithm for verification in rounds.

practice, the received transactions may not be a continuation because, for example, the verifier may receive history fragments from some collectors later than others. To have a complete history for COBRA's verification algorithm, COBRA's verifier preprocesses the received history fragments, filters out the transactions whose predecessors are unknown, and saves them for future rounds.

**Strong session serializable.** As mentioned in Section 4.4.2, COBRA targets databases whose serialization order respects the session-order, that is the transaction issuing order in each session. If a history satisfies serializability (Definition 1) and the corresponding serial schedule preserves the session-order, we call this history *strong session serializable* [105], defined below.

**Definition 23** (Strong session serializable history)**.** *A* strong session serializable history *is a history that is equivalent to a serial schedule $\hat{s}$, such that $\hat{s}$ preserves the transaction issuing order for all sessions.*

Notice that COBRA requires that queries in each session are blocking (§4.2). So, for one session, its transaction issuing order is the order seen by its history collector (one session connects to one collector). The verifier also knows such ordering by referring to the history fragments.

**Extended known graph.** In the following, we define a helper notion *extended known graph* which is the data structure passing between rounds. An extended known graph is a tuple ($g$, *readfrom*, *wwpairs*) generated by CREATEKNOWNGRAPH2 (Figure C.1, line 8), which contains the known graph $g$ and two relationships extracted from the history: reading-from and consecutive-write (*readfrom* and *wwpairs*).

In the following, we use $g_e(h)$ to represent an extended known graph generated from a history $h$; and we use $g_e(g_e^i, r)$ to represent an extended known graph generated from (i) the previous round's extended known graph $g_e^i$ and (ii) a continuation $r$, namely:

$$g_e(g_e^i, r) = \text{CREATEKNOWNGRAPH2}(g_e^i.g, g_e^i.readfrom, g_e^i.wwpairs, r)$$

In fact, $g_e(h)$ is a shortened form of $g_e(\emptyset, h)$.

**Fact 24.** *For a complete history $h$ and its continuation $r$, $g_e(h \circ r) = g_e(g_e(h), r)$. An extended known graph has three components: the known graph $g$, readfrom, and wwpairs. Regardless of processing $h$*

*and $r$ together in $g_e(h \circ r)$ or separately in $g_e(g_e(h), r)$, the verifier creates the three components in the same way because the information carried by transactions are the same (in Figure C.1, the known graph's vertices are added in line 16; the known graph's edges are added in line 20 and 31; readfrom is updated in line 21; and wwpairs is updated in line 26).*

To verify an ever-growing history, COBRA's verifier needs to delete transactions. Next, we define COBRA's deletion on an extended known graph.

**Definition 25** (Deletion from an extended known graph)**.** *A* deletion *of a transaction $tx_i$ from an extended known graph $g_e(h)$ is to (1) delete the vertex $tx_i$ and edges containing $tx_i$ from the known graph $g$ in $g_e(h)$; and (2) delete tuples that include $tx_i$ from readfrom and wwpairs.*

We use $g_e(h) \ominus tx_i$ to denote deleting $tx_i$ from an extended known graph $g_e(h)$. Note that deletions happen on the extended known graph (a data structure in the verifier), instead of the history (inputs to the verifier). Hence deleting a transaction from the extended known graph does not contradict the assumption that the history is complete.

### C.2.2 Proof outline

Our objective is to prove that COBRA's verification in round (Figure C.1) checks serializability of a continuous history. That is, COBRA's algorithm of deleting transactions doesn't affect the acyclicity of polygraphs. This section previews the key notions and idea of the proof.

The proof uses two types of polygraphs: (original) *polygraphs* and *COBRA polygraphs* (both defined in §C.2.3). COBRA polygraphs are what COBRA's verifier materializes and uses to find transactions to delete, whereas polygraphs are used for proof purposes only. The reason why we use polygraphs is that they have simple and fixed size constraints (§4.2.3), which significantly simplifies the proof.

The rough argument of the proof is as follows: (1) based on COBRA polygraphs, we define what transactions are to be deleted by the verifier; (2) we prove that deleting such transactions does not affect the acyclicity of polygraphs; (3) because a polygraph is acyclic if only if the corresponding COBRA polygraph is acyclic (see Lemma 26 below), such deletions do not affect the acyclicity of COBRA polygraphs either, which is what COBRA checks.

For point (1), what transactions are safe to delete, we need three key notions, a rough description below:

- *Frozen transactions* (§C.2.4) are transactions that have paths to future transactions in the known graph. The paths are based on fence transactions (defined in §C.2.4, see also §4.4.2).
- *Frontier* (§C.2.5) is a set of transactions containing the "oldest" writes to each key that future transactions can read from.
- *Poly-strongly connected components* (short as P-SCC, §C.2.5) are strongly connected components defined on a COBRA polygraph (details in Definition 31). Each component is a set of transactions that could have cycles including edges from constraints.

We claim that a transaction is safe to delete if it (a) is frozen, (b) doesn't not belong to the frontier, and (c) the P-SCC it is in only contains frozen transactions that do not belong to the frontier (see also Definition 36).

As for point (2), the safe deletion argument, we establish that if a transaction is frozen and does not belong to the frontier, it cannot generate a cycle with future transactions in the known graph (§C.2.4 and §C.2.5); and if all transactions in the same P-SCC are frozen and do not belong to the frontier, they cannot have cycles with future transactions through edges in constraints (§C.2.6). Thus, such transactions can be safely deleted without having to worry cycles with future transaction through either the known graph or the constraints (§C.2.7).

### C.2.3 Polygraph and cobra polygraph

Notice that an extended known graph contains all information from a history. So both polygraphs (§4.2.3) and COBRA polygraphs (Definition 2) can be built from extended known graphs, instead of histories.

Specifically, a polygraph $(V, E, C)$ can be built from an extended known graph $g_e(h)$ as:

- $V$ are all vertices in $g_e(h).g$.
- $E = \{(tx_i, tx_j) \mid \langle key, tx_i, tx_j \rangle \in g_e(h).readfrom\}$; that is, $tx_i \xrightarrow{\text{wr(x)}} tx_j$, for some $x$.
- $C = \{\langle (tx_j, tx_k), (tx_k, tx_i) \rangle \mid (tx_i \xrightarrow{\text{wr(x)}} tx_j) \wedge$
  $(tx_k \text{ writes to } x) \wedge tx_k \neq tx_i \wedge tx_k \neq tx_j\}$.

We denote the polygraph generated from an extended known graph $g_e(h)$ as $P(g_e(h))$.

Since constructing an extended known graph is part of COBRA's algorithm, it is natural to construct a COBRA polygraph from an extended known graph, which works as follow: assign COBRA polygraph's known graph to be $g_e(h).g$ and generate constraints by invoking

$$\textsc{GenConstraints}(g_e(h).g,\ g_e(h).\mathit{readfrom},\ g_e(h).\mathit{wwpairs}).$$

We denote the COBRA polygraph generated from extended known graph $g_e(h)$ as $Q(g_e(h))$.

**Lemma 26.** *Given a complete history $h$ and its extended known graph $g_e(h)$, the following expressions are equivalent:*

*(1) history $h$ is serializable.*

*(2) polygraph $P(g_e(h))$ is acyclic.*

*(3) COBRA polygraph $Q(g_e(h))$ is acyclic.*

*Proof.* Papadimitriou [167, Lemma 2] proves (1) $\iff$ (2) . Theorem 12 and Theorem 17 in Appendix C.1 prove (1) $\iff$ (3). □

Note that, in order to check strong session serializability, the verifier adds transactions' session-order to polygraph and COBRA polygraph by inserting edges for transactions in the same session (Figure C.1, line 28–31). We call such edges *session order edges* (short as SO-edges). These SO-edges establish Lemma 26 for strong session serializability, which can be proved by adding session order requirements to both the serial schedules and the (original and COBRA) polygraphs in the Papadimitriou's proof [167, Lemma 2] and proofs in Appendix C.1.

Everywhere in the following it says "serializable" we mean "strong session serializable".

### C.2.4 Fence transactions, epoch, and frozen transactions

As stated in Section 4.4.1, the challenge of deleting transactions is that serializability does not respect real-time order across sessions and it is unclear which transactions can be safely deleted. To address this challenge, COBRA uses *fence transactions* to split the history into *epochs*; based on

epochs, the verifier can mark transactions as *frozen* which indicates that future transactions can never be predecessors of these transactions.

**Fence transactions.** As described in Section 4.4.2, *fence transactions* (defined below) are predefined transactions that are periodically issued by clients. They read and write a predefined key called the *epoch key* (the string "EPOCH" below).

```
1   begin_tx()
2   epoch = read("EPOCH")
3   if epoch > local_epoch:
4     local_epoch = epoch
5   else:
6     local_epoch = local_epoch + 1
7     write("EPOCH", local_epoch)
8   commit_tx()
```

Based on the value read from the epoch key, a fence transaction is either a *write fence transaction* (short as Wfence) or a *read fence transaction* (short as Rfence): Wfences read-modify-write the epoch key, and Rfences only read the epoch key. In a history, we define that the fence transactions are *well-formed* as follows.

**Definition 27** (Well-formed fence transactions)**.** *In a history, fence transactions are* well-formed *when (1) all write fence transactions are a sequence of consecutive writes to the epoch key; and (2) all read fence transactions read from write fence transactions.*

**Claim 13.** *For a history $h$ that is complete and not easily rejectable, fence transactions in $h$ are well-formed.*

*Proof.* Because the epoch key is predefined and reserved for fence transactions, Wfences are the only transactions that write this key. Given that Wfences read-modify-write the epoch key, a Wfence read from either another Wfence or the (abstract) initial transaction if the epoch key hasn't been created.

Next, we prove that all Wfences are a sequence of consecutive writes. Given that $h$ is not easily rejectable, by Definition 6, there is no cycle in the known graph. Hence, if we start from any Wfence and repeatedly find the predecessor of current Wfence on the epoch key (the predecessor is known because Wfences also read the epoch key), we will eventually reach the initial transaction (because the number of Wfences in $h$ is finite). Thus, all Wfences and the initial transaction are connected and form a tree (the root is the initial transaction). Also, because $h$ is not easily rejectable, no write transaction has two successive writes on the same key. So there is no Wfence that has two children in this tree, which means that the tree is actually a list. And each node in this list reads the epoch key from its preceding node and all of them write the epoch key. By Definition 5, this list of Wfences is a sequence of consecutive writes.

Finally, because $h$ is a complete history, all Rfences read from transactions in $h$. Plus, only Wfences update the epoch key, hence Rfences read from Wfences in $h$. □

**Epochs.** Well-formed fence transactions cluster normal transactions (transactions that are not fence transactions) into *epochs*. Epochs are generated as follows (AssignEpoch in Figure C.1, line 56). First, the verifier scans the Wfences list and assigns each Wfence with an epoch number which is this Wfence's index in the list (Figure C.1, line 59–62). Second, the verifier assigns epoch numbers to Rfences which are the epoch numbers from the Wfences they read from (Figure C.1, line 63–65). Finally, the verifier assigns epoch numbers to normal transactions which are the epoch number of their successive fence transactions in the same session minus one (Figure C.1, line 78).

When assigning epochs, the verifier keeps track of the largest epoch number that all sessions have exceeded, denoted as $epoch_{agree}$ (Figure C.1, line 75). In other words, every session has issued at least one fence transaction that has epoch number $\geq epoch_{\text{agree}}$.

One clarifying fact is that the epoch number assigned to each transaction is not the value (an integer) of the epoch key in the database; epoch numbers come from indices of the Wfence sequence. The verifier doesn't need the actual values in the epoch key to assign epochs.

In the following, we denote a transaction with an epoch number $t$ as a transaction with $epoch[t]$.

**Lemma 28.** *If a history $h$ is not easily rejectable, then a fence transaction with $epoch[t]$ has a path to any fence transaction with $epoch[> t]$ in the known graph of $g_e(h)$.*

*Proof.* First, we prove that a fence transaction with $epoch[t]$ has a path to another fence transaction with $epoch[t+1]$. Given history $h$ is not easily rejectable, by Claim 13, fence transactions are well-formed, which means all the Wfences are a sequence of consecutive writes (by Definition 27). Because the Wfence with $epoch[t]$ ($t$ is its position in the sequence) and the Wfence with $epoch[t+1]$ are adjacent in the sequence, there is an edge of reading-from dependency from the Wfence with $epoch[t]$ to the Wfence with $epoch[t+1]$.

Now consider a Rfence with $epoch[t]$ which reads the epoch key from the Wfence with $epoch[t]$. Because COBRA's algorithm adds anti-dependency edges which point from one write transaction's succeeding read transactions to its successive write on the same key (Figure 4.3, line 60), there is an edge from the Rfence with $epoch[t]$ to the Wfence with $epoch[t+1]$. Plus, all Rfences with $epoch[t+1]$ read from the Wfence with $epoch[t+1]$, hence fence transactions with $epoch[t]$ have paths to fence transactions with $epoch[t+1]$.

By induction, for any fence transaction with $epoch[t+\Delta]$ ($\Delta > 1$), a fence transaction with $epoch[t]$ has a path to it. □

**Claim 14.** *For a history $h$ and any future continuation $r$, if $h \circ r$ is not easily rejectable, then any transaction with $epoch[\leq epoch_{agree} - 2]$ has a path in $g_e(h \circ r)$ to any transaction in $r$.*

*Proof.* Take any normal transaction $tx_i$ with $epoch[t]$ ($t \leq epoch_{\text{agree}} - 2$) and call $tx_i$'s session $S_i$. Because the epoch of a normal transaction equals the epoch number of its successive fence transactions in the same session minus one (Figure C.1, line 78), there is a fence transaction $tx_f$ in $S_i$ with $epoch[t+1]$ ($t+1 \leq epoch_{\text{agree}} - 1$). By Lemma 28, $tx_f$ has a path to any fence transaction with $epoch[epoch_{\text{agree}}]$. And, by the definition of $epoch_{\text{agree}}$, each session has at least one fence transaction with $epoch[\geq epoch_{\text{agree}}]$ in $h$. Thus, there is always a path from $tx_i$—through $tx_f$ and the last fence transactions of a session in $h$—to transactions in $r$.

If $tx_i$ is a fence transaction and has $epoch[\leq epoch_{\text{agree}} - 2]$, by Lemma 28, $tx_i$ has a path to the last fence transaction in every session, which has $epoch[\geq epoch_{\text{agree}}]$. Thus, $tx_i$ has a path to any future transaction in $r$.

□

**Frozen transaction.** With epochs, COBRA can define *frozen transactions*, which are "old" enough such that no future transactions can be scheduled prior to these transactions in any valid serial schedule. Intuitively, if a transaction is frozen, this transaction can never be involved in any cycles containing future transactions.

**Definition 29** (Frozen transaction)**.** *For a history $h$ that is not easily rejectable, a* frozen transaction *is a transaction that has $epoch[\leq epoch_{agree} - 2]$ and all its predecessors in $g_e(h)$ also have $epoch[\leq epoch_{agree} - 2]$.*

### C.2.5 Removable transactions

To tell what transactions can be safely deleted, we need to introduce two more notions: *frontier* and *poly-strongly connected component*. In addition, we will prove that *pruning* (§4.3.3), as an optimization, does not affect the acyclicity of a polygraph; hence it is safe for the verifier to do pruning in each round.

**Frontier.** As stated in Section 4.4, the verifier needs to retain most-recent-writes to keys in frozen transactions because future transactions may read from them. For a history $h$, we call the transactions with such writes as the *frontier* of $h$, defined below.

**Definition 30** (Frontier)**.** *For a history $h$ that is not easily rejectable, the* frontier *of $h$ is the set of transactions that (a) have $epoch[\leq epoch_{agree} - 2]$ and (b) contain at least one write to some key $x$, such that no successor of this transaction in $g_e(h)$ with $epoch[\leq epoch_{agree} - 2]$ writes $x$.*

**Poly-strongly connected component.** As described in Section 4.4.3, the verifier needs to capture the possible cycles that are generated from constraints. To achieve this, we define *poly-strongly connected components* (short as P-SCC) which are sets of transactions that may have cycles because of constraint edges. Intuitively, if two transactions appears in one P-SCC, it is possible (but not certain) for them to form a cycle; but if two transactions do not belong to the same P-SCC, they do not form a cycle.

**Definition 31** (Poly-strongly connected component)**.** *Given a history $h$ and its* COBRA *polygraph $Q(g_e(h))$, the* poly-strongly connected components *are the strongly connected components of a directed graph that is the known graph of $Q(g_e(h))$ with all edges in the constraints added to it.*

**Lemma 32.** *In a history $h$ that is not easily rejectable, for any two transactions $tx_i$ and $tx_j$ writing the same key $x$, if $tx_i \not\rightsquigarrow tx_j$ and $tx_j \not\rightsquigarrow tx_i$ in $g_e(h)$, then $tx_i$, $tx_j$, and the transactions reading $x$ from them (if any) are in the same P-SCC.*

*Proof.* By Claim 8, each of $tx_i$ and $tx_j$ appears and only appears in one chain (say $chain_i$ and $chain_j$, respectively). Because $tx_i \not\rightsquigarrow tx_j$ and $tx_j \not\rightsquigarrow tx_i$, $chain_i \neq chain_j$. COBRA's algorithm generates a constraint for every pair of chains on the same key (Figure 4.3, line 41), so there is a constraint $\langle A, B\rangle$ for $chain_i$ and $chain_j$, which includes $tx_i$ and $tx_j$.

Consider this constraint $\langle A, B\rangle$. One of the two edge sets ($A$ and $B$) contains edges that establish a path from the tail of $chain_i$ to the head of $chain_j$—either a direct edge (Figure 4.3, line 69), or through a read transaction that reads from the tail of $chain_i$ (Figure 4.3, line 74). Similarly, the other edge set establishes a path from the tail of $chain_j$ to the head of $chain_i$. In addition, by Lemma 11, in each chain, there is a path from its head to its tail through the reading-from edges in the known graph (Figure 4.3, line 17). Thus, there is a cycle involving all transactions of these two chains. By Definition 31, all the transactions in these two chains—including $tx_i$, $tx_j$, and the transaction reading $x$ from them—are in one P-SCC. □

Note that though P-SCCs are defined over COBRA polygraphs, the "membership" of P-SCCs doesn't have to be discussed in the context of a COBRA polygraph. It is determined by the history whether two transactions belong to the same P-SCC (one could imagine running an algorithm of constructing P-SCCs in the background).

**Pruning.** COBRA's verifier does pruning in every round (PRUNE; Figure C.1, line 37). Pruning changes the extended known graph $g_e(h)$ by adding edges that can be inferred from node reachability (§4.3.3). In the following, we prove that pruning has no effect on the acyclicity of polygraphs. Further, we extend the notion *easily rejectable history* with pruning.

**Fact 33.** *Pruning doesn't affect the acyclicity of a polygraph (or a* COBRA *polygraph). For a constraint*

*to be pruned, since the constraint is a binary choice and COBRA knows the fact that choosing one option will generate cycles (Figure 4.3, line 82, 85), COBRA can safely discard this option and choose the other option. Thus, pruning only stop searching on cyclic compatible graphs of the polygraph (or the COBRA polygraph), hence has no effect on searching for acyclic compatible graphs.*

**(Extended) easily rejectable history.** In order to involve the changes imposed by pruning and rule out more local malformations that are not serializable, we extend the definition of an easily rejectable history (Definition 6) as follows.

**Definition 34** (An (extended) easily rejectable history)**.** An (extended) easily rejectable history *h is a history that either (1) contains a transaction that has multiple successive writes on one key, or (2) has a cyclic extended known graph* $g_e(h)$ *after pruning.*

**Corollary 35.** *COBRA rejects (extended) easily rejectable histories.*

*Proof.* By Lemma 7, COBRA rejects a history when (1) it contains a transaction that has multiple successive writes one one key; (2) If the extended known graph $g_e(h)$ has a cycle, COBRA detects and rejects this history when checking acyclicity in the constraint solver. □

**Removable transactions.** *Removable transactions* (defined below) are the transactions that can be safely deleted by COBRA's verifier. In COBRA's algorithm, they are deleted from the extended known graph at the end of each round (Figure C.1, line 106).

**Definition 36** (Removable transaction)**.** *For a history that is not easily rejectable, a transaction tx is* removable *if:*

- *tx is a frozen transaction.*
- *tx does not belong to the frontier of h.*
- *tx is in a P-SCC that only contains frozen transactions, and no transaction in this P-SCC belong to the frontier.*

From the definition above, we know that if one transaction is removable, then all transactions in the same P-SCC are also removable. Also, if one transaction is removable, we can conclude that no future transactions read from it, proved below.

**Claim 15.** *For a history $h$ and any its continuation $r$ such that $h \circ r$ is not easily rejectable, no transaction in $r$ can read from a removable transaction of $h$.*

*Proof.* Assume to the contrary that there exists a transaction $tx_k \in r$ reading a key $x$ from $tx_i$ which is a removable transaction. By Definition 36, $tx_i$ is not in the frontier of $h$; and by Definition 30, for all keys written by $tx_i$ ($x$ included) there must exist some successors with $epoch[\leq epoch_{\text{agree}} - 2]$ that writes these keys. Call the successor that writes $x$, $tx_j$.

Now, consider transactions ($tx_i$, $tx_j$, $tx_k$). In a polygraph, they form a constraint: $tx_i$ and $tx_j$ both writes to key $x$; and $tx_k$ reads $x$ from $tx_i$. The constraint is $\langle tx_k \longrightarrow tx_j, tx_j \longrightarrow tx_i \rangle$. Because $tx_j$ is a successor of $tx_i$, after pruning, the edge $tx_k \longrightarrow tx_j$ should be added to $g_e(h \circ r)$. Meanwhile, $tx_j$ has $epoch[\leq epoch_{\text{agree}} - 2]$; by Claim 14, $tx_j$ has a path to any transaction in $r$, $tx_k$ included. Therefore, there is a cycle $tx_k \longrightarrow tx_j \rightsquigarrow tx_k$ in $g_e(h \circ r)$, a contradiction to a not easily rejectable history $h \circ r$. □

### C.2.6 Solved constraints and unsolved constraints

In the original polygraph construction (§4.2.3), constraints are generated for every reading-from pair (a read reading from a write) and another write to the same key. However, some of them are "useless"—with or without them doesn't affect the acyclicity of a polygraph. In this section, we classify constraints into two types, *solved constraints* and *unsolved constraints*, which simplifies proofs of the main argument in the next section.

As stated in Section 4.2.3, a constraint in a polygraph involves three transactions: two write transactions ($tx_{\text{w1}}$, $tx_{\text{w2}}$) writing to the same key and one read transaction ($tx_{\text{r}}$) reading this key from $tx_{\text{w1}}$. And, this constraint ($\langle tx_{\text{r}} \longrightarrow tx_{\text{w2}}, tx_{\text{w2}} \longrightarrow tx_{\text{w1}} \rangle$) has two ordering options, either (1) $tx_{\text{w2}}$ appears before both $tx_{\text{w1}}$ and $tx_{\text{r}}$, or (2) $tx_{\text{w2}}$ appears after them. We call a constraint as a *solved constraint* when the known graph has already captured one of the options.

**Definition 37** (Solved constraint)**.** *For a history $h$ that is not easily rejectable, a constraint $\langle tx_r \longrightarrow tx_{w2}, tx_{w2} \longrightarrow tx_{w1} \rangle$ is a* solved constraint*, when the known graph of $g_e(h)$ has either $tx_r \rightsquigarrow tx_{w2}$ or $tx_{w2} \rightsquigarrow tx_{w1}$.*

**Fact 38.** *Eliminating solved constraints from a polygraph doesn't affect its acyclicity because the ordering of the three transactions in a solved constraint has already existed in the known graph.*

For those constraints that are not solved constraints, we call them *unsolved constraints*. Notice that both solved constraints and unsolved constraints are defined on polygraphs (not COBRA polygraphs). With solved and unsolved constraints, we conclude the following Lemmas.

**Lemma 39.** *In a not easily rejectable history $h$, for any unsolved constraint $\langle tx_r \rightarrow tx_{w2}, tx_{w2} \rightarrow tx_{w1} \rangle$, all three transactions ($tx_{w1}$, $tx_{w2}$, and $tx_r$) are in the same P-SCC.*

*Proof.* Consider the relative position of $tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$ in $g_e(h)$. First, we know that $tx_{\mathrm{w2}} \not\rightsquigarrow tx_{\mathrm{w1}}$ because otherwise, by Definition 37, the constraint is a solved constraint. Second, we prove $tx_{\mathrm{w1}} \not\rightsquigarrow tx_{\mathrm{w2}}$. Assume $tx_{\mathrm{w1}} \rightsquigarrow tx_{\mathrm{w2}}$; then, after pruning, $g_e(h)$ has an edge $tx_{\mathrm{r}} \rightarrow tx_{\mathrm{w2}}$ which contradicts that the constraint is an unsolved constraint. Therefore, $g_e(h)$ has no path between $tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$. By Lemma 32, $tx_{\mathrm{w1}}$, $tx_{\mathrm{w2}}$, and $tx_{\mathrm{r}}$ are in the same P-SCC. □

**Lemma 40.** *Given a history $h$ and a continuation $r$ such that $h \circ r$ is not easily rejectable, there is no unsolved constraint that includes both a removable transaction in $h$ and a transaction in $r$.*

*Proof.* Consider a constraint $\langle tx_{\mathrm{r}} \rightarrow tx_{\mathrm{w2}}, tx_{\mathrm{w2}} \rightarrow tx_{\mathrm{w1}} \rangle$ ($tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$ write to the same key $x$; $tx_{\mathrm{r}}$ reads $x$ from $tx_{\mathrm{w1}}$) where one of the three transactions is removable and another one is a transaction in $r$.

In the following, by enumerating all combinations of possibilities, we prove such a constraint is always a solved constraint. Therefore, no unsolved constraint includes both removable transactions and transactions in $r$.

- The removable transaction is $tx_{\mathrm{r}}$.

  Because $tx_{\mathrm{r}}$ is removable, it is a frozen transaction. By Definition 29, as a predecessor of $tx_{\mathrm{r}}$ ($tx_{\mathrm{r}}$ reads from $tx_{\mathrm{w1}}$), $tx_{\mathrm{w1}}$ has $epoch[\leq epoch_{\mathrm{agree}} - 2]$. Hence, the remaining transaction $tx_{\mathrm{w2}}$ must be the transaction in $r$. By Claim 14, $tx_{\mathrm{r}}$ have a path to any transactions in $r$ including $tx_{\mathrm{w2}}$. Thus, this constraint is a solved constraint.
- The removable transaction is $tx_{\mathrm{w1}}$.

Because $tx_{\mathrm{r}}$ reads $x$ from $tx_{\mathrm{w1}}$, by Claim 15, $tx_{\mathrm{r}}$ cannot be a transaction in $r$. Hence, the transaction in $r$ must be $tx_{\mathrm{w2}}$.

Because $tx_{\mathrm{w1}}$ is not part of the frontier of $h$, by Definition 30, there exists some successor $tx_s$ with $epoch[\leq epoch_{\mathrm{agree}} - 2]$ which also writes $x$. Because $tx_{\mathrm{w1}}$ and $tx_s$ writes $x$ and $tx_{\mathrm{r}}$ reads from $tx_{\mathrm{w1}}$, they forms a constraint. Consider this constraint $\langle tx_s \longrightarrow tx_{\mathrm{w1}}, tx_{\mathrm{r}} \longrightarrow tx_s \rangle$: because $tx_s$ is a successor of $tx_{\mathrm{w1}}$, after pruning, $g_e(h)$ has an edge $tx_{\mathrm{r}} \longrightarrow tx_s$.

Finally, because $tx_s$ has $epoch[\leq epoch_{\mathrm{agree}} - 2]$ and $tx_{\mathrm{w2}} \in r$, by Claim 14, $tx_s \rightsquigarrow tx_{\mathrm{w2}}$; hence $tx_{\mathrm{r}} \longrightarrow tx_s \rightsquigarrow tx_{\mathrm{w2}}$. By Definition 37, the constraint $\langle tx_{\mathrm{r}} \longrightarrow tx_{\mathrm{w2}}, tx_{\mathrm{w2}} \longrightarrow tx_{\mathrm{w1}} \rangle$ is a solved constraint.

- The removable transaction is $tx_{\mathrm{w2}}$.

  Because there must be one transaction in $r$, both $tx_{\mathrm{r}} \in h$ and $tx_{\mathrm{w1}} \in h$ are impossible. Also, because $h$ is a complete history, it is impossible to have $tx_{\mathrm{r}} \in h$ but the transaction it reads $tx_{\mathrm{w1}} \in r$. Hence, there are two possibilities:

  - $tx_{\mathrm{w1}} \in r \wedge tx_{\mathrm{r}} \in r$.

    By Claim 14, $tx_{\mathrm{w2}}$ has paths to transactions in $r$ including $tx_{\mathrm{w1}}$ and $tx_{\mathrm{r}}$. Hence, the constraint is a solved constraint.

  - $tx_{\mathrm{w1}} \in h \wedge tx_{\mathrm{r}} \in r$.

    Now, consider the relative position of $tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$. Because a transaction $tx_{\mathrm{r}} \in r$ reads from $tx_{\mathrm{w1}}$, by Claim 15, $tx_{\mathrm{w1}}$ is not a removable transaction. Further, by Definition 36, $tx_{\mathrm{w2}}$ belongs to a P-SCC in which all transactions are removable transactions, so $tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$ are not in the same P-SCC. Hence, by Lemma 32, either $tx_{\mathrm{w1}} \rightsquigarrow tx_{\mathrm{w2}}$ or $tx_{\mathrm{w2}} \rightsquigarrow tx_{\mathrm{w1}}$.

    Next, we prove $tx_{\mathrm{w1}} \rightsquigarrow tx_{\mathrm{w2}}$ is impossible. Assume $tx_{\mathrm{w1}} \rightsquigarrow tx_{\mathrm{w2}}$. For the constraint $\langle tx_{\mathrm{r}} \longrightarrow tx_{\mathrm{w2}}, tx_{\mathrm{w2}} \longrightarrow tx_{\mathrm{w1}} \rangle$, after pruning, $g_e(h \circ r)$ has the edge $tx_{\mathrm{r}} \longrightarrow tx_{\mathrm{w2}}$. Meanwhile, $tx_{\mathrm{w2}}$ has $epoch[\leq epoch_{\mathrm{agree}} - 2]$, by Claim 14, $tx_{\mathrm{w2}} \rightsquigarrow tx_{\mathrm{r}}$. Thus, $g_e(h \circ r)$ is cyclic, a contradiction to that $h \circ r$ is not easily rejectable.

    Above all, $tx_{\mathrm{w2}} \rightsquigarrow tx_{\mathrm{w1}}$; hence, $\langle tx_{\mathrm{r}} \longrightarrow tx_{\mathrm{w2}}, tx_{\mathrm{w2}} \longrightarrow tx_{\mathrm{w1}} \rangle$ is a solved constraint.

□

### C.2.7 The main argument

In this section, Theorem 45 proves the correctness of COBRA's garbage collection algorithm.

**Claim 16.** *For a history $h$ and a continuation $r$ such that $h \circ r$ is not easily rejectable, for a removable transaction $tx_i$, there are no edges in $g_e(h \circ r)$ between $tx_i$ and transactions in $r$.*

*Proof.* According to COBRA's algorithm (Figure C.1), it adds four types of edges (specified below, denoted by ①–④) during processing transactions in $r$, and none of them includes $tx_i$.

Because the known graph of $g_e(h \circ r)$ is acyclic, by Claim 15, no transactions in $r$ read from $tx_i$, so no reading-from edges (①, Figure C.1, line 20) or anti-dependency edges (②, Figure 4.3, line 60) with $tx_i$ are added to $g$. Also, because $tx_i$ has $epoch[epoch_{\text{agree}} - 2]$, there must be a fence transaction that comes after $tx_i$ from the same session, hence there is no session order edge from $tx_i$ to transactions in $r$ (③, Figure C.1, line 31). Finally, by Lemma 40, no unsolved constraints include both $tx_i$ and any transactions in $r$. Thus pruning, which adds edges because of resolving constraints (④, Figure 4.3, line 82, 85), doesn't add edges with $tx_i$ to $g$. □

**Lemma 41.** *Given a history $h$ and a continuation $r$ such that $h \circ r$ is not easily rejectable, for any removable transaction $tx_i$, $g_e(g_e(h) \ominus tx_i, r) = g_e(h \circ r) \ominus tx_i$*

*Proof.* First, we prove $g_e(g_e(h) \ominus tx_i, r) = g_e(g_e(h), r) \ominus tx_i$, meaning that the final extended known graph remains the same whether COBRA's algorithm deletes $tx_i$ before or after processing $r$. To do so, we prove that COBRA's algorithm updates all three components in an extended known graph—*readfrom*, *wwpairs*, and the known graph $g$—the same way, with or without $tx_i$.

- For *readfrom*, because $tx_i$ is removable, by Claim 15, no transactions in $r$ can read from it. Therefore, with or without $tx_i$, pairs of transactions added to *readfrom* remain the same.
- For *wwpairs*, again by Claim 15, there are no read-modify-write transactions in $r$ that read from $tx_i$, hence consecutive writes added to *wwpairs* are the same< with or without $tx_i$.
- For the known graph $g$, the vertices added (transactions in $r$) are the same with and without $tx_i$; by Claim 16, the edges added are also the same.

Above all, $g_e(g_e(h) \ominus tx_i, r)$ and $g_e(g_e(h), r) \ominus tx_i$ are the same, and by Fact 24, we prove the lemma: $g_e(g_e(h) \ominus tx_i, r) = g_e(g_e(h), r) \ominus tx_i = g_e(h \circ r) \ominus tx_i$. □

**Claim 17.** *Given a not easily rejectable history $h$, a continuation $r$, and a removable transaction $tx_i$, after deleting $tx_i$ from $g_e(h)$, COBRA still rejects $h \circ r$ if it is easily rejectable.*

*Proof.* By Definition 34, $h \circ r$ is easily rejectable either because (1) it contains a write transaction having multiple successive writes, or because (2) it has cycles in $g_e(h \circ r)$.

For (1), if $tx_i$ contains neither the write that has multiple successive writes (call it $w_{msw}$ writing to some key $x$) nor its successive writes, these writes are in $g_e(h \circ r) \ominus tx_i$, and by Corollary 35 COBRA rejects.

If $tx_i$ contains $w_{msw}$, given that $h$ is not easily rejectable, there is at least one successive write (say in $tx_j$) in $r$. For the removable transaction $tx_i$ and $tx_j$ in $r$, we can reuse the proof of Claim 15, and there exists a transaction $tx_k$ that is a successor of $tx_i$ and has a cycle with $tx_j$. Hence, $g_e(h \circ r) \ominus tx_i$ is cyclic and COBRA rejects.

If $tx_i$ contains one of the successive writes, the transaction containing $w_{msw}$ (call it $tx_a$) is $tx_i$'s predecessor in $g_e(h)$. Again, $tx_a$ has at least one successive write (say in $tx_j$) in $r$. Because $tx_i$ is removable, by Definition 29, $tx_a$ is a frozen transaction, and $tx_a$ does not have the most recent write to $x$. Again, by reusing the proof of Claim 15, there is a cycle between $tx_a$ and $tx_j$ in $g_e(h \circ r) \ominus tx_i$. Thus, COBRA rejects.

For (2), we prove that if $g_e(h \circ r)$ is cyclic, then $g_e(g_e(h) \ominus tx_i, r)$ is also cyclic, hence COBRA rejects. By Lemma 41, it is identical to prove a cyclic $g_e(h \circ r) \ominus tx_i$. Assume to the contrary that $g_e(h \circ r) \ominus tx_i$ is acyclic. Then, because $g_e(h \circ r)$ is cyclic while $g_e(h)$ and $g_e(h \circ r) \ominus tx_i$ are acyclic, there exists a cycle including $tx_i$ and a transaction $tx_j \in r$.

Consider the path $tx_j \rightsquigarrow tx_i$ in the cycle. The path cannot be an edge ($tx_j \longrightarrow tx_i$) because, by reusing the proof of Claim 16, we can prove none of the four types of edge in $g_e(h \circ r)$ is possible. So the path must be $tx_j \rightsquigarrow tx_s \longrightarrow tx_i$ where $tx_s \in h$. Because $tx_i$ is removable, $tx_s$ has $epoch[\leq epoch_{\text{agree}} - 2]$, and by Claim 14, $tx_s \rightsquigarrow tx_j$. Thus, $g_e(h \circ r) \ominus tx_i$ has a cycle between $tx_s$ and $tx_j$, and COBRA rejects.

□

**Theorem 42.** *Given a history $h$ that is serializable and a continuation $r$, for a set of removable transactions $d$, there is: $P(g_e(h \circ r) \ominus d)$ is acyclic $\Leftrightarrow$ $P(g_e(h \circ r))$ is acyclic.*

*Proof.* If $h \circ r$ is easily rejectable, by Corollary 35 and Claim 17, COBRA rejects; both $P(g_e(h \circ r) \ominus d)$ and $P(g_e(h \circ r))$ are cyclic.

Now, we consider the case where $h \circ r$ is not easily rejectable.

"$\Rightarrow$". In $P(g_e(h \circ r))$, for one transaction $tx_i \in d$, by Lemma 39, all transactions in unsolved constraints that involves $tx_i$ are in the same P-SCC, and by Definition 36 they are all removable. Call all such P-SCCs for different transactions in $d$, $\bigcup pscc$. Then, we can partition $h \circ r$ into $\bigcup pscc$ and others (call them $\overline{\bigcup pscc}$), and there are no unsolved constraints including transactions from both $\bigcup pscc$ and $\overline{\bigcup pscc}$.

Next, we prove that $P(g_e(h \circ r))$ is acyclic, by constructing an acyclic compatible graph $\hat{g}$. By Fact 38, we can safely ignore solved constraints. Consider the transactions in $\overline{\bigcup pscc}$, the unsolved constraints are the same in both $P(g_e(h \circ r))$ and $P(g_e(h \circ r) \ominus d)$; given that $P(g_e(h \circ r) \ominus d)$ is acyclic, there exists constraint choices that makes $\hat{g}$ acyclic for $\overline{\bigcup pscc}$ part of the polygraph.

Now, consider transactions in $\bigcup pscc$. Because all transactions in $\bigcup pscc$ are removable, they are in $h$. Plus, $h$ is serializable, so there exists constraint choices for unsolved constraints in $\bigcup pscc$ such that $\hat{g}$ has no cycle among the transactions of $\bigcup pscc$. Finally, because there is no unsolved constraint between $\bigcup pscc$ and $\overline{\bigcup pscc}$, and $g_e(h \circ r)$'s known graph is acyclic (because $h \circ r$ is not easily rejectable), $\hat{g}$ is acyclic.

"$\Leftarrow$". Because $P(g_e(h \circ r))$ is acyclic, there exists an acyclic compatible graph $\hat{g}$. We can construct a compatible graph $\hat{g'}$ for $P(g_e(h \circ r) \ominus d)$ by choosing all constraints according to $\hat{g}$—choose the edges in constraints that appear in $\hat{g}$. Given that the known graph in $P(g_e(h \circ r) \ominus d)$ is a subgraph of $P(g_e(h \circ r))$'s, $\hat{g'}$ is a subgraph of $\hat{g}$. Hence, $\hat{g'}$ is acyclic, and $P(g_e(h \circ r) \ominus d)$ is acyclic. □

**Lemma 43.** *Given a serializable history $h$ and a continuation $r$, for a set of removable transactions $d$, there is: $P(g_e(h \circ r) \ominus d)$ is acyclic $\iff$ $Q(g_e(h \circ r) \ominus d)$ is acyclic.*

*Proof.* "$\Rightarrow$". By Lemma 42, $P(g_e(h \circ r) \ominus d)$ is acyclic $\Rightarrow$ $P(g_e(h \circ r))$ is acyclic. Further, by Lemma 26, $Q(g_e(h \circ r))$ is acyclic. Because removing nodes, edges, and constraints from a COBRA polygraph does not adding cycles, $Q(g_e(h \circ r) \ominus d)$ is acyclic.

"$\Leftarrow$". Assume to the contrary that $Q(g_e(h \circ r) \ominus d)$ is acyclic but $P(g_e(h \circ r) \ominus d)$ is cyclic. Because

$Q(g_e(h \circ r) \ominus d)$ is acyclic, there exists an acyclic compatible graph $\hat{g}$. By topological sorting $\hat{g}$, we have a sequential list $\hat{s}$ containing transactions in $h \circ r \setminus d$. Now, consider a history $h'$ that includes all transactions in $h \circ r \setminus d$ while removing all read operations reading from transactions in $d$. By Definition 21, $h'$ is a complete history, and by Theorem 17, $\hat{s}$ is a sequential schedule.

Now, consider $\hat{s}$ and $P(g_e(h \circ r) \ominus d)$. We build a new polygraph *polyg* by adding all happened-before relationships in $\hat{s}$ as edges to $P(g_e(h \circ r) \ominus d)$. Because the known graph of $P(g_e(h \circ r) \ominus d)$ is a subgraph of the acyclic graph $\hat{g}$, and $\hat{s}$ is a topological sort of $\hat{g}$, *polyg* has an acyclic known graph that has a total order for nodes (from $\hat{s}$). But *polyg* is cyclic because $P(g_e(h \circ r) \ominus d)$ is cyclic. Thus, there must exist at least one constraint in *polyg*, such that choose either option will generate a cycle in the known graph.

Consider this constraint $\langle tx_{\mathrm{r}} \rightarrow tx_{\mathrm{w2}}, tx_{\mathrm{w2}} \rightarrow tx_{\mathrm{w1}} \rangle$ ($tx_{\mathrm{w1}}$ and $tx_{\mathrm{w2}}$ write to the same key $x$; $tx_{\mathrm{r}}$ reads $x$ from $tx_{\mathrm{w1}}$). Choosing either $tx_{\mathrm{r}} \rightarrow tx_{\mathrm{w2}}$ or $tx_{\mathrm{w2}} \rightarrow tx_{\mathrm{w1}}$ generates a cycle; that is, the known graph of *polyg* has both $tx_{\mathrm{w2}} \rightsquigarrow tx_{\mathrm{r}}$ and $tx_{\mathrm{w1}} \rightsquigarrow tx_{\mathrm{w2}}$. Therefore, the sequential ordering of these three transactions in $\hat{s}$ is $[tx_{\mathrm{w1}}, tx_{\mathrm{w2}}, tx_{\mathrm{r}}]$ which contradicts the fact that $tx_{\mathrm{r}}$ reads from $tx_{\mathrm{w1}}$.

□

In the following, we use $h_i$ to represent the transactions fetched in $i_{\mathrm{th}}$ round. The first round's history $h_1$ is a complete history itself; for the $i_{\mathrm{th}}$ round ($i \geq 2$), $h_i$ is a continuation of the prior history $h_1 \circ \cdots \circ h_{i-1}$. Note that $h_1 \circ \cdots \circ h_i \circ h_i$ is not easily rejectable implies that any complete sub-history of it is also not easily rejectable.

**Lemma 44.** *Given that history $h_1 \circ \cdots \circ h_i \circ h_{i+1}$ is not easily rejectable, if a transaction is removable in $h_1 \circ \cdots \circ h_i$, then it remains removable in $h_1 \circ \cdots \circ h_i \circ h_{i+1}$.*

*Proof.* Call this removable transaction $tx_a$ and the P-SCC it is in during round $i$ as $pscc_b$.

Because COBRA's algorithm does not delete fence transactions (Figure C.1, line 106), the epoch numbers for normal transactions in round $i$ remain the same in round $i+1$. Hence, the $epoch_{\mathrm{agree}}$ in round $i+1$ is greater than or equal to the one in round $i$. Thus, $tx_a$ remains to be a frozen transaction. In addition, $tx_a$ was not in the frontier of $h_1 \circ \cdots \circ h_i$, and it will not be with a new continuation $h_{i+1}$.

Because history $h_1 \circ \cdots \circ h_i \circ h_{i+1}$ is not easily rejectable, there are no cycles in the known graph of $g_e(h_1 \circ \cdots \circ h_i \circ h_{i+1})$. Also, by Lemma 39, transactions in $pscc_b$ do not have unsolved constraints with any transaction in $h_{i+1}$. Thus, in round $i+1$, the new P-SCC that $tx_a$ is in only includes transactions from old $pscc_b$, which were removable transactions for round $i$. By reusing the argument in the immediately above paragraph, we know transactions in $pscc_b$ remain to be frozen and do not belong to the frontier in round $i+1$.

Above all, by Definition 36, $tx_i$ is removable in round $i+1$. □

In the following, we denote the transactions deleted in the $i_{\text{th}}$ round as $d_i$. The verifier deletes $d_i$ at the end of round $i$.

**Theorem 45.** *COBRA's algorithm runs for n rounds (before deleting $d_n$) and doesn't reject if and only if history $h_1 \circ h_2 \cdots \circ h_n$ is serializable.*

*Proof.* We prove by induction.

The base case: for the first round, COBRA's algorithm only gets the history $h_1$. By Theorem 17 and Theorem 12 in Appendix C.1, COBRA doesn't reject if and only if $h_1$ is serializable.

The inductive case: for round $i$, assume the induction hypothesis that COBRA's algorithm doesn't reject for the last $i-1$ rounds and history $h_1 \circ h_2 \cdots \circ h_{i-1}$ is serializable. In round $i$, COBRA' algorithm receives $h_i$ from collectors, and gets the extended known graph from round $i-1$, which is $g_e(h_1 \circ \cdots \circ h_{i-1}) \ominus (d_0 \cup \cdots \cup d_{i-1})$. By Fact 33, pruning doesn't affect the acyclicity of polygraphs and COBRA polygraphs, so we can ignore pruning in the following proof. By Lemma 44, the deleted transactions in previous rounds $d_0 \cup \cdots \cup d_{i-1}$ are removable in round $i$.

Next, we prove that COBRA's algorithm doesn't reject (the COBRA polygraph is acyclic) if and only if $h1 \circ \cdots \circ h_i$ is serializable.

$Q(g_e(g_e(h_1 \circ \cdots \circ h_{i-1}) \ominus (d_0 \cup \cdots \cup d_{i-1}), h_i))$ is acyclic

$\iff$ $Q(g_e(h_1 \circ \cdots \circ h_i) \ominus (d_0 \cup \cdots \cup d_{i-1}))$ is acyclic

[Lemma 41]

$\iff$ $P(g_e(h_1 \circ \cdots \circ h_i) \ominus (d_0 \cup \cdots \cup d_{i-1}))$ is acyclic

[Lemma 43]

$\iff$ $P(g_e(h_1 \circ \cdots \circ h_i))$ is acyclic

[Theorem 42]

$\iff$ $h_1 \circ \cdots \circ h_i$ is serializable

[Lemma 26]

□

# Bibliography


[1] Acknowledged inserts can be present in reads for tens of seconds, then disappear. https://github.com/YugaByte/yugabyte-db/issues/824.

[2] Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[3] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[4] Amazon Web Services (AWS). https://aws.amazon.com/.

[5] Apple Pay transaction volume and new user growth outpacing PayPal, tim cook says. https://9to5mac.com/2019/07/30/apple-pay-transactions-users-paypal/.

[6] AWS partner's data reveals most popular Amazon Cloud products of 2018. https://awsinsider.net/articles/2019/01/23/top-aws-products.aspx.

[7] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.

[8] Big data in real time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter.

[9] CentOS forum. https://www.centos.org/forums/.

[10] Cloud adoption statistics for 2020. https://hostingtribunal.com/blog/cloud-adoption-statistics/.

[11] CockroachDB: Distributed SQL. https://www.cockroachlabs.com.

[12] CockroachDB: What happens when node clocks are not properly synchronized? https://www.cockroachlabs.com/docs/stable/operational-faqs.html#what-happens-when-node-clocks-are-not-properly-synchronized.

[13] CockroachDB's consistency model. https://www.cockroachlabs.com/blog/consistency-model/.

[14] cuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[15] cuSPARSE: Sparse Linear Algebra on GPUs. https://developer.nvidia.com/cusparse.

[16] 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.

[17] Executive summary: Computer network time synchronization. https://www.eecis.udel.edu/~mills/exec.html.

[18] FaunaDB. https://fauna.com.

[19] Forecast number of personal cloud storage consumers/users worldwide from 2014 to 2020 (in millions). https://www.statista.com/statistics/499558/worldwide-personal-cloud-storage-users/.

[20] FoundationDB. https://www.foundationdb.org.

[21] G2: anti-dependency cycles. https://github.com/cockroachdb/cockroach/issues/10030.

[22] G2-item anomaly with master kills. https://github.com/YugaByte/yugabyte-db/issues/2125.

[23] Google Cloud Datastore. https://cloud.google.com/datastore/.

[24] Google Cloud Spanner. https://cloud.google.com/spanner/.

[25] Google terms of services. https://policies.google.com/terms?hl=en.

[26] Gretchen: Offline serializability verification, in Clojure. https://github.com/aphyr/gretchen.

[27] How Halo5 implemented social gameplay using Azure Cosmos DB. https://azure.microsoft.com/en-us/blog/how-halo-5-guardians-implemented-social-gameplay-using-azure-documentdb/.

[28] Insider threats as the main security threat in 2017. https://www.tripwire.com/state-of-security/security-data-protection/insider-threats-main-security-threat-2017/.

[29] Jepsen: FaunaDB 2.5.4. http://jepsen.io/analyses/faunadb-2.5.4.

[30] LAMP (software bundle). https://en.wikipedia.org/wiki/LAMP_(software_bundle).

[31] Lessons learned from 2+ years of nightly Jepsen tests. https://www.cockroachlabs.com/blog/jepsen-tests-lessons/.

[32] NewSQL database systems are failing to guarantee consistency, and I blame Spanner. http://dbmsmusings.blogspot.com/2018/09/newsql-database-systems-are-failing-to.html.

[33] Norwegian electronics giant scales for sales, sets record with cloud-based transaction processing. https://customers.microsoft.com/en-us/story/elkjop-retailers-azure.

[34] Oracle flashback technology. http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html.

[35] PHP magic methods. http://php.net/manual/en/language.oop5.magic.php.

[36] PHP manual. http://php.net/manual/en/index.php.

[37] PostgreSQL. https://www.postgresql.org/.

[38] The process of ACM SIGCOMM 2009. http://www.sigcomm.org/conference-planning/the-process-of-acm-sigcomm-2009.

[39] RocksDB. https://rocksdb.org/.

[40] RUBiS. https://rubis.ow2.org/.

[41] SIMD. https://en.wikipedia.org/wiki/SIMD.

[42] Summary of the Amazon S3 service disruption in the northern virginia (US-EAST-1) region. https://aws.amazon.com/message/41926/.

[43] Tencent Cloud denies technical attack against rival. http://www.szdaily.com/content/2019-01/08/content_21332774.htm.

[44] TPC-C. http://www.tpc.org/tpcc/.

[45] Transaction Isolation Levels. https://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-isolation-levels.html.

[46] A virtual machine designed for executing programs written in Hack and PHP. https://github.com/facebook/hhvm.

[47] Visa: Small business retail. https://usa.visa.com/run-your-business/small-business-tools/retail.html.

[48] VMware vSphere Replication. https://www.vmware.com/products/vsphere/replication.html.

[49] The Yices SMT solver. http://yices.csl.sri.com/.

[50] YugaByte DB 1.3.1, undercounting counter. http://jepsen.io/analyses/yugabyte-db-1.3.1.

[51] YugaByte DB: Home. https://www.yugabyte.com.

[52] YugaByte DB source code. https://github.com/yugabyte/yugabyte-db/blob/3b90e8560b8d8bc81fba6ba9b9f2833e83e2244e/src/yb/util/physical_time.cc#L36.

[53] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[54] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2014.

[55] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *USENIX Workshop on Hot Topics in System Dependability (HotDep)*, Dec. 2008.

[56] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[57] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[58] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[59] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *IEEE Workshop on Workload Characterization*, Nov. 2002.

[60] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store *actually* provide? In *USENIX Workshop on Hot Topics in System Depend-*

*ability (HotDep)*, Oct. 2010. Full version: Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[61] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *Dependable Systems and Networks (DSN)*, June 2011.

[62] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: a high concurrency key-value store with integrity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 2017.

[63] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[64] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 1999.

[65] A. Awad and B. Karp. Execution integrity without implicit trust of system software. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2019.

[66] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[67] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *ACM Symposium on the Theory of Computing (STOC)*, May 1991.

[68] P. Bailis. Linearizability versus Serializability. http://www.bailis.org/blog/linearizability-versus-serializability/, Sept. 2014.

[69] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: virtues and limitations. *Proceedings of the VLDB Endowment (PVLDB)*, Sept. 2014.

[70] T. Balyo, M. J. Heule, and M. Jarvisalo. SAT competition 2016: Recent developments. In *AAAI Conference on Artificial Intelligence (AAAI)*, Feb. 2017.

[71] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification (CAV)*, July 2011.

[72] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[73] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In *AAAI Conference on Artificial Intelligence (AAAI)*, Jan. 2015.

[74] E. Ben-Sasson, I. Ben-Tov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, and M. Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 2017.

[75] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR International Cryptology Conference (CRYPTO)*, Aug. 2013.

[76] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, Aug. 2014.

[77] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 1995.

[78] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[79] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[80] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[81] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[82] R. Biswas and C. Enea. dbcop: source code. https://zenodo.org/record/3367334.

[83] R. Biswas and C. Enea. On the complexity of checking transactional consistency. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2019.

[84] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3), Sept. 1994.

[85] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[86] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[87] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2017.

[88] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Static serializability analysis for causal consistency. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2018.

[89] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Computer Aided Verification (CAV)*, July 2008.

[90] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[91] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[92] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[93] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM Virtual Execution Environments (VEE)*, Mar. 2013.

[94] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. 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)*, Mar. 2008.

[95] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2013.

[96] J. Cheng. NebuAd, ISPs sued over DPI snooping, ad-targeting program. *Ars Technica*, Nov. 2008. https://arstechnica.com/tech-policy/2008/11/nebuad-isps-sued-over-dpi-snooping-ad-targeting-program/.

[97] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), June 2013.

[98] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. The MIT Press, Cambridge, MA, 2009.

[99] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science (ITCS)*, Jan. 2012.

[100] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. D. 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.

[101] N. Crooks. *A client-centric approach to transactional datastores.* PhD thesis, The University of Texas at Austin, 2020.

[102] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: a client-centric specification of database isolation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, July 2017.

[103] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. SPEED: Accelerating enclave applications via secure deduplication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, July 2019.

[104] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos made transparent. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[105] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *International Conference on Data Engineering (ICDE)*, Mar. 2004.

[106] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 2008.

[107] A. Diarra, S. B. Mokhtar, P.-L. Aublin, and V. Quéma. FullReview: Practical accountability in presence of selfish nodes. In *IEEE International Symposium on Reliable Distributed Systems*, 2014.

[108] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques (PDPTA)*, Aug. 1996.

[109] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in RocksDB. In *The Conference on Innovative Data Systems Research (CIDR)*, Jan. 2017.

[110] X. Dou, P. M. Chen, and J. Flinn. Shortcut: accelerating mostly-deterministic code regions. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2019.

[111] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[112] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *ACM Virtual Execution Environments (VEE)*, Mar. 2008.

[113] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography Conference*, Mar. 2009.

[114] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 2003.

[115] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Conference on File and Storage Technologies (FAST)*, Feb. 2017.

[116] M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *European Conference on Artificial Intelligence (ECAI)*, 2014.

[117] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: acyclicity. In *European Workshop on Logics in Artificial Intelligence*, 2014.

[118] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *IACR International Cryptology Conference (CRYPTO)*, Aug. 2010.

[119] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2013.

[120] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[121] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[122] A. Goel, K. Farhadi, K. Po, and W. Feng. Reconstructing system state for intrusion analysis. *ACM SIGOPS Operating Systems Review*, 42(3):21–28, Apr. 2008.

[123] W. Golab, X. Li, and M. Shah. Analyzing consistency properties for fun and profit. In *ACM Symposium on Principles of Distributed Computing (PODC)*, June 2011.

[124] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *The Design, Automation, and Test in Europe (DATE) Conference*, Mar. 2002.

[125] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, Aug. 2015. Prelim version STOC 2008.

[126] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *The ACM Symposium on Cloud Computing (SoCC)*, Nov. 2014.

[127] T. Hadzilacos and N. Yannakakis. Deleting completed transactions. *Journal of Computer and System Sciences*, 38(2), 1989.

[128] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[129] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

[130] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *The International Conference on Software Engineering (ICSE)*, May 2008.

[131] 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)*, Oct. 2014.

[132] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[133] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: secure applications on an untrusted operating system. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013.

[134] J. Huang, P. Liu, and C. Zhang. LEAP: The lightweight deterministic multi-processor replay of concurrent Java programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, Feb. 2010.

[135] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[136] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[137] Intel. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2017.

[138] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *International Conference on Data Engineering (ICDE)*, Apr. 2013.

[139] S. Jana and V. Shmatikov. EVE: Verifying correct execution of cloud-hosted web applications. In *USENIX HotCloud Workshop*, June 2011.

[140] M. Janota, R. Grigore, and V. Manquinho. On the quest for an acyclic graph. *arXiv preprint arXiv:1708.01745*, 2017.

[141] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *International Conference on Very Large Databases (VLDB)*, 2007.

[142] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *IEEE Symposium on Security and Privacy*, May 2016.

[143] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *IEEE Symposium on Security and Privacy*, May 2015.

[144] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[145] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[146] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[147] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[148] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, June 2010.

[149] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, C-28(9), Sept. 1979.

[150] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[151] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, 1987.

[152] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. 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)*, Mar. 2010.

[153] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2006.

[154] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *AAAI Conference on Artificial Intelligence (AAAI)*, Feb. 2016.

[155] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 2017.

[156] Q. Liu, G. Wang, and J. Wu. Consistency as a service: Auditing cloud consistency. *IEEE Transactions on Network and Service Management*, 11(1), Mar. 2014.

[157] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[158] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment (PVLDB)*, 6(9), July 2013.

[159] M. Maurer and D. Brumley. Tachyon: tandem execution for efficient live patch testing. *USENIX Security*, pages 617–630, Aug. 2012.

[160] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, May 2010.

[161] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, Apr. 2008.

[162] R. C. Merkle. A digital signature based on a conventional encryption function. In *IACR International Cryptology Conference (CRYPTO)*, Aug. 1987.

[163] S. Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[164] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, June 2001.

[165] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *IEEE Symposium on Security and Privacy*, 2020.

[166] K. Nagar and S. Jagannathan. Automated detection of serializability violations under weak consistency. *arXiv preprint arXiv:1806.08416*, 2018.

[167] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), Oct. 1979.

[168] D. Papagiannaki and L. Rizzio. The ACM SIGCOMM 2009 Technical Program Committee Process. *ACM CCR*, 39(3):43–48, July 2009.

[169] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[170] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[171] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[172] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*, May 2009.

[173] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

[174] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX Annual Technical Conference*, June 2011.

[175] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[176] D. R. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment (PVLDB)*, Aug. 2012.

[177] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan. CLOTHO: directed test generation for weakly consistent database systems. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2019.

[178] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with Web Tripwires. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[179] M. Ronsse and K. D. Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[180] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, Aug. 2004.

[181] D. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[182] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, May 2015.

[183] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[184] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[185] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[186] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[187] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-tcb linux applications with sgx enclaves. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[188] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *ACM Workshop on Cloud Computing Security*, Oct. 2010.

[189] A. Sinha and S. Malik. Runtime checking of serializability in software transactional memory. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Apr. 2010.

[190] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, 2011.

[191] R. Sinha and M. Christodorescu. VeritasDB: High throughput key-value store with integrity. *IACR Cryptology ePrint Archive*, 2018.

[192] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2016.

[193] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[194] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, Sept. 1986.

[195] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, June 2009.

[196] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In *Computer Aided Verification (CAV)*, July 2002.

[197] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *International Conference on Runtime Verification*, Sept. 2011.

[198] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[199] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[200] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):3, 2012.

[201] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora : Design considerations for high throughput cloud-native relational databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 2017.

[202] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing web 2.0 applications through replicated execution. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[203] R. S. Wahby, Y. Ji, A. J. Blumberg, abhi shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[204] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM (CACM)*, Feb. 2015.

[205] T. Warszawski and P. Bailis. ACIDRain: Concurrency-related attacks on database-backed web applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 2017.

[206] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[207] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17:164–182, 1993.

[208] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[209] R. Wray. BT drops plan to use Phorm targeted ad service after outcry over privacy. *The Guardian*, July 2009. https://www.theguardian.com/business/2009/jul/06/btgroup-privacy-and-the-net.

[210] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Notices*, 40(6), 2005.

[211] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object centRic DEterministic Replay for Java. In *USENIX Annual Technical Conference*, June 2011.

[212] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[213] E. Yoon and P. Liu. Practical verification of MapReduce computation integrity via partial re-execution. *arXiv preprint arXiv:2002.09560*, 2020.

[214] C. Zamfir, G. Altekar, and I. Stoica. Automating the debugging of datacenter applications with ADDA. In *Dependable Systems and Networks (DSN)*, June 2013.

[215] K. Zellag and B. Kemme. How consistent is your cloud application? In *The ACM Symposium on Cloud Computing (SoCC)*, Oct. 2012.

[216] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *Proceedings of the VLDB Endowment (PVLDB)*, Feb. 2014.

[217] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[218] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, May 2017.

[219] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.