Efficient Auditing of Event-driven Web Applications

Ioanna Tzialla[⋆†], Jeffery Wang[⋆], Jingyi Zhu[‡], Aurojit Panda[⋆], Michael Walfish[⋆]

[⋆]NYU [†]Google [‡]ETH Zurich

Abstract

When a deployer of a web application puts that application on a server (on-prem or cloud), how can they be sure that the application is executing as intended? This paper studies how the deployer can *efficiently check* that the execution is faithful. We seek mechanisms that: (i) work with web applications that are built with modern event-driven web frameworks, (ii) impose tolerable computation and communication overheads on the web server, and (iii) are complete and sound. We exhibit such a mechanism, based on a new record-replay algorithm. We have implemented our algorithm in Karousos, a system that audits Node.js web applications.

CCS Concepts • **Security and privacy** → **Web application security**; • **Computer systems organization** → **Reliability**.

Keywords Execution Integrity, Web applications, JavaScript

ACM Reference Format:
Ioanna Tzialla, Jeffery Wang, Jingyi Zhu, Aurojit Panda, and Michael Walfish . 2024. Efficient Auditing of Event-driven Web Applications. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 53 pages. https://doi.org/10.1145/3627703.3650089

1 Introduction

Consider a developer, Cam, who writes or trusts source code for a web application (for example, written in Node.js). On Cam's machine, the code runs a certain way; call this version of the application the "golden master". This does not mean that the application has no bugs, only that it's the version Cam wishes to execute. Cam now deploys that code on-premise (on-prem) or on a remote cloud, and wants to be sure that the code executes the same way as the golden master would. Unfortunately, Cam cannot simply assume such *faithful execution.* The issues include differences in any layer of the stack below the application: language run-time, operating system, hypervisor, and hardware. Such differences could result from different versions. Differences could also result from bugs [47], misconfiguration, operational error, or—in the case of a remote server—insider [54] and co-tenant attacks [75]. So how can Cam get assurance that the application is executing as the golden master would?

The general topic here is *execution integrity*: giving some principal (such as Cam) confidence that running a given program (such as Cam's application) on given inputs (in Cam's case, requests to a web server) truly produces the alleged outputs (in Cam's case, responses from the web server). Note that this is complementary to *program verification*, which is about ensuring that source code meets a specification; here, we take the code as a given, and want to make sure that it is actually executed. Solutions to execution integrity include Byzantine fault-tolerant replication [25], AVMs [48], and probabilistic proofs [88, Ch.19] (zero-knowledge proofs and so on). However, these works are not geared to Cam's question: they make assumptions about the server's failure modes, don't scale to legacy web applications, or require the principal to do too much work (see §2.2 and §7).

Attestation using TPMs [26, 49, 62, 63, 73, 76, 79, 84] and enclaves [13, 16, 20, 52, 77, 81, 83] guarantees that a precise software stack was running at a given instant. However, as we argue in the next section (§2.2), placing an entire stack in an enclave does not solve Cam's problem. Cam may wish to use Platform as a Service (PaaS) or serverless deployment models, in which case not only is the remote stack different from Cam's but also Cam has no visibility into the remote stack. Moreover, the state of a running remote stack can be corrupted over time, even if initially attested-to.

One work, Orochi [87], proposed a different approach based on validating outputs, given observed inputs. In Orochi, a *verifier* (a machine under Cam's control) performs a comprehensive audit. Orochi requires a *collector*, situated logically in front of the server, that captures a ground-truth *trace* of exactly the inputs to, and outputs from, the server. One option is to run the collector on-prem and proxy all traffic through it; another is to use attestation. Section 2.2 further discusses the requirement of a collector.

Given this setup, the verifier re-executes from the inputs in the (trusted) trace, checking that the re-executed outputs match the outputs in the trace. Crucial to this process is (untrusted) *advice* that the verifier receives from the server, which enables the verifier to accelerate re-execution versus naive replay, by re-executing requests in batches. The advice

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650089

also helps the verifier make sense of concurrent executions. This is a harder problem than standard record-replay [29, 30, 33] (§7), in part because the advice can be adversarial. In fact, approaches to advice validation that *seem* right can be misled into wrongly accepting a bogus execution at the server (see §4.3 and [87, §3.4–3.5] for examples).

Our work is inspired by Orochi. However, Orochi has a restricted execution model, which limits applicability to a small subset of web applications. First, each client request in Orochi must be handled within a single execution context, as in PHP. This rules out web applications that use event-driven frameworks such as Tornado [4], Node.js [5], and Phoenix [6]. Without taking a position in the eternal events-versus-threads debate, we note that most modern web application frameworks are written in the event-driven style. Second, Orochi assumes that little state is shared between execution contexts; if more state were shared, Orochi's protocols would require the server to send an impractically large quantity of advice to the verifier. Third, external state in Orochi, such as a transactional key-value store, is assumed to meet the strong condition of strict serializability [72]; yet, many external data stores default to weaker isolation levels and may not even offer strict serializability [18].

Addressing these restrictions introduces new technical problems. Defining and solving them is the work of this paper, which we do in the context of a system called *Karousos*. Karousos borrows the Orochi setting (collector, untrusted server, verifier). Karousos makes the following contributions:

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

Karousos shifts the tradeoff curve and identifies a point on the shifted curve, with several interlocking ideas. First, Karousos's verifier batches together requests that induce the same trees of events (§4.1), regardless of the original order of the corresponding handlers. Second, Karousos introduces a notion of *R-ordered* (§4.2): two dependent operations during execution (for example, a write of a program variable followed by a read of that variable) are *R*-ordered if they are guaranteed to be *re*-executed in that same order. The server then logs only operations that are not *R*-ordered. Third, for unlogged operations, the verifier consults a *version history* that it constructs while re-executing (§4.2).

Ensuring that re-executions are sensible. Without further mechanism, a misbehaving server could make the verifier accept executions (as embodied in traces) that are inconsistent with executions of the original code (§4.3, §4.4). Karousos handles server misbehavior by requiring that requests are served, program variables are accessed, and transactions are executed, all in an order consistent with each other—and with the program. Note that this is much more easily said than done, because the verifier must check consistency across three sources of ordering, only one of which (the trace) is trusted; the others must be validated with specific kinds of crosschecks or through the course of re-execution.

Karousos's techniques include, for program variables, reconstructing an alleged partial order of variable accesses while executing (§4.3). For transactional state stores (§4.4), the principal correctness conditions surround isolation levels (serializability, read-committed, and so on). But the verifier cannot simply use existing algorithms for testing isolation, such as Adya's [7], because there is no trustworthy source of internal transaction history. Instead, the Karousos verifier runs Adya's algorithms against an alleged history, thereby contingently justifying that history, and then ensures that the contingent history is consistent with the rest of the execution.

Although one might think that existing frameworks [29, 30, 33, 68–70] could substitute for the techniques of Karousos, our experience has been that first-cut "solutions" subtly fail: an adversarial server can mislead the verifier or the verifier cannot validate even an honest server's execution. Regardless, because of the context—arbitrary server behavior while the verifier is computationally weaker than the server—any proposal in this context carries the burden of proof. Specifically, any proposal requires a rigorous proof of both Completeness (the verifier accepts executions that are faithful to the original code) and Soundness (the verifier rejects unfaithful ones, regardless of server misbehavior).

Proof of correctness. We supply such proofs for Karousos (Appendix C).

Implementation. Our implementation of Karousos supports web applications that are written in Node.js and use MySQL as a transactional key-value store. As we explain later (§5), developers wanting to use our implementation need to annotate portions of their code. Our implementation supports a core of JavaScript, disallowing certain other constructs.

We have evaluated Karousos on a popular wiki application [2] and two model web applications (§6). For the wiki application, Karousos's server has processing overhead of 1.2–2.8× that of an unmodified server; we believe this is a reasonable price to pay for execution integrity. By contrast, probabilistic proofs (succinct arguments, zero-knowledge proofs, and so on) [17, 37, 38, 43, 44, 53, 64] are, as of this writing, the only other approach to execution integrity that does not trust server hardware or assume a fault-free fraction of replicas; but probabilistic proofs impose server overhead on the order of $10^6\times$, despite recent progress (and marketing); see [88, Ch.19] for a survey of implementations.

For the wiki application, the Karousos verifier is between 19%–34% faster than an implementation of Orochi for Node.js,

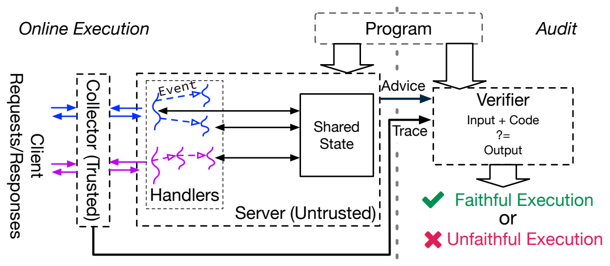


Figure 1. The problem: efficiently auditing an untrusted server.

and between 1.8×–16.6× faster than an alternative that sequentially re-executes. Also, the advice produced by the Karousos for the wiki application are of reasonable size (24 to 146 MB for 600 requests), and Karousos’s logs are no larger, and sometimes 50% smaller than, those in our Orochi implementation.

These results are encouraging, but Karousos has clear limitations. First, the implemented system requires developers to manually annotate the program (§5) and update the annotations with new code releases; this burden could be lifted by fully automating annotation using a static analyzer, for example, one built on escape analysis [51, 94]. Second, JavaScript workers are disallowed; this is not fundamental. More fundamentally, timers are disallowed, range queries on transactional state are not supported, and snapshot isolation is not supported. Addressing these restrictions would require extending our algorithms and proofs; we leave this to future work. Finally, at the level of architecture, Karousos verifies only a single web application, not multiple interacting server-side applications.

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

2 Setup and background

2.1 Problem: comprehensive server audit

Here, we define our problem abstractly, to showcase the challenge while avoiding distracting details. Later (§5), we will translate it to event-driven web applications. Our presentation is inspired by, and has some textual debts to, Orochi [87, §2].

Figure 1 depicts the problem. Some *principal* (like Cam) deploys a program P on an untrusted *server* (for example, running on a cloud platform).[1] Clients make *requests* to the server. Requests can be concurrent with each other, and P can be a concurrent program. A *response* is allegedly the result of invoking P against the corresponding request.

The principal has access to a *trace* of the actual requests and (possibly unfaithful) responses. The trace is provided by a *collector* that is assumed to work correctly (we delve into collection in the next section). We can think of the trace as the ground truth record of what enters and leaves the server.

[1]We are leaving unspecified whether P’s deployment includes a specific stack chosen by the principal; the problem statement is relevant either way.

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

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

- *Completeness.* If the server behaved properly during the time period of the trace (which includes collecting advice honestly), then the verifier must accept the given trace.
- *Soundness.* The verifier must reject if the server misbehaved. Specifically, the verifier accepts only if there is some schedule S of (possibly concurrent) executions, such that: (a) executing the given program against the inputs in the trace, while following S, reproduces exactly the respective outputs in the trace, and (b) S is consistent with the ordering in the trace. (Appendix C.2 states Soundness precisely.) This property means that the server can pass the audit only by behaving in a way that is, to external observations, indistinguishable from actually executing the program on the received requests.
- *Efficiency.* This means several things in our context. (a) The verifier, being computationally weaker than the server, needs to perform less computation than the server; in particular, the work of the verifier should be computationally less costly than naively re-executing each request in the trace one-by-one. (b) The advice sent from the server to the verifier needs to be kept small. (c) Advice collection should not significantly impact the server’s response latency. We are willing to tolerate some computational overhead at the server, as we expect auditability to cost something.

2.2 Execution integrity

Though it is a crucial property, execution integrity can be counter-intuitive; for example, it is sometimes confused with the orthogonal concern of program correctness. So in this section we aim to clarify some aspects of the model, and answer natural questions. Before continuing, we want to be clear: the audit setup does not presume a bug-free application. Instead, given that the program in question, when run on a known stack, behaves a certain way (which might be buggy), we want to guarantee that: either the untrusted server ran that same way or else the verifier complains.

Enclaves and attestation. Although we aren't taking a stand against enclaves [13, 16, 20, 31, 40, 45, 52, 77, 81, 83] in general—indeed, we suggest a limited use of them below—our view is that putting the entire stack in an enclave is insufficient to provide a strong notion of execution integrity.

First, the stack in question might not be defined. Cam (as the verifier) might have one stack while the cloud provider has another. Indeed, in PaaS and serverless models, the cloud provider can and does optimize their platform to their hardware infrastructure, and doesn't release the stack. (AWS, for example, patches both the operating system executing Lambda functions [10] and the Node.JS runtime executing within [11], but neither are available for download.)

Second, stacks are messy. Attestation ensures that the *initial* state of the stack is approved. But the attested-to stack is not formally validated; it has vulnerabilities and thus attack surface. In particular, it is exposed to all of the traffic directed to all layers of the stack (IP, TCP, etc.), as well as co-located applications. For example, adversarial traffic aimed at TCP could subvert the OS, which enclaves do not prevent.[2]

An audit-based verifier, by contrast, is far less likely to be subverted, because the verifier receives a small subset of all traffic that the server does: only the application-level requests and responses (for example, the contents of HTTP requests). This is the trace, which is "strained out" by the collector. The verifier then delivers those requests directly to the re-executing application. Thus, relative to placing the entire stack in an enclave, the audit setup has a narrower attack surface, smaller TCB, and fewer assumptions.

Collection. A comprehensive audit solution requires ground-truth outputs and inputs. This is the role of the collector. One option for the collector is to run a TLS endpoint in an attested-to enclave [87, §7]. As argued in DOG [15], this configuration resists the sort of attacks-on-the-stack that we alluded to above, because only the TLS implementation, not the stack itself, is trusted. Notice, however, that the fundamental point applies beyond attestation and enclaves. The fundamental point is that trusting a collector (to deliver a ground-truth trace) is a smaller assumption than trusting the entire stack (to execute faithfully): the collector does far less. Indeed, in contrast to the behemoth of code in a modern OS, the collector could be implemented with only a thin supervisory layer, or even hardware, deployed as a bump-in-the-wire that has no stack of its own [87, §1,§4.1§7].

Other approaches to execution integrity. Comprehensive auditing is *verifier-efficient*: the verifier does less work than naively re-executing. Two other strands of work share this property (see also §7): (1) Byzantine fault-tolerant replication [25], which requires $> \frac{2}{3}$ of the replicas to be fault-free. (2) Probabilistic proofs [17, 37, 38, 43, 44, 53, 64] place no trust in the server (and would also need a trace). These theoretical constructs have seen mushrooming implementations (see [92][88, Ch.19] for surveys). However, probabilistic proofs would impose immense overhead (factors of $10^6\times$ are common); they are not close to handling realistic web applications.

[2]Providing attestation certificates in server responses [24, 60, 65, 85, 97] does not solve the problem, since it is not clear how the principal can enforce the use or checks of such certificates.

2.3 Our starting point: Orochi

Orochi [87] is a comprehensive server audit that, as stated in the introduction, re-executes all requests in a trace, checking that the produced responses match the outputs in the trace.

Orochi addresses the challenge of a computationally limited verifier by exploiting an aspect of web applications: many executions follow the same code paths [55, 87]. The Orochi server is supposed to track control flow, and then specify (in the advice) *control flow groups*, meaning which requests have the same control flow as each other. The verifier then re-executes a single control flow group as a batch, using *SIMD-on-demand*. If an instruction has the same operands across a batch, the verifier re-executes that instruction only once, and otherwise executes the opcode for each request in the batch. This technique is facilitated by a datatype called a multivalue, which collapses when all of the entries in the multivalue are identical, and expands into a vector when needed.

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

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

Despite this technique, the server could arrange responses and advice to cause the verifier to accept bogus executions [87, §3.4]. Consequently, another technique in Orochi is *consistent ordering verification*; the verifier builds a graph that includes every operation, request arrival, and response delivery, with edges indicating ordering (time-order between requests, program order between operations, operation order from the logs). The verifier then insists that the graph is acyclic.

Orochi's techniques are provably Complete and Sound (§2.1). However, Orochi makes simplifying assumptions. First, although the server is concurrent, requests are handled mostly in isolation, in straight-line fashion (with the unrealistic assumption that when a response is delivered, the request has no further effect). This rules out many web application architectures and all event-driven frameworks. Second, Orochi would produce unacceptably verbose logs (contra Efficiency; §2.1) in a setting where a lot of state is shared between discrete execution units (for example, program variables that are accessed by multiple event handlers). Third, external state such as transactional storage must be strongly consistent, and must be accessed synchronously; this too rules out many deployment scenarios.

3 Execution model

We define an execution model, *KEM*, for unmodified concurrent web applications. In subsequent sections we use KEM to describe our core algorithm; the proofs presented in the Appendix also build on KEM. KEM is intended to capture the semantics of Node.js programs; it does not model the behavior of transactional state (for that, see §4.4). Furthermore, KEM models a runtime that can have multiple concurrent threads executing at a time. This is more general than the Node.js runtime (and indeed other JavaScript runtimes) which is single-threaded, allowing us to minimize assumptions about the runtime.

KEM models the state of a program as a set of variables, a set of zero-or-more pending *events* and a set of zero-or-more *event handlers* (defined in the following paragraphs). Program code can read or update any in-scope variable. However, similar to JavaScript, functions and closures capture variables by reference. Consequently, all variables in scope when a function or closure is defined are in scope for the body of the function; even local variables might be accessed from multiple functions. As a result, a variable might be concurrently accessed and updated by multiple concurrent threads. KEM assumes all accesses are sequentially consistent [57]. This assumption is justified; indeed, all extant JavaScript code assumes sequential consistency.

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

As in JavaScript, KEM *event handlers* are closures. Program code can add or remove handlers by calling *register* or *unregister*. Both functions take as input an event type and a closure, which we sometimes refer to, loosely, as the *function* associated with the event. Event handlers in KEM can perform computation, modify in-scope variables, emit events, and register or unregister handlers. We refer to handlers that are associated with "new user request" events as *request handlers*. KEM assumes that event handlers run to completion and that a handler's execution is not interrupted when it emits an event. We use the term *handler activation* to refer to the act of the runtime's dispatch loop calling an event handler. Each such activation creates a unique handler. Two separate handlers can, however, have the same code (for example, if a given function is activated twice).

Programs in KEM begin execution by calling a designated initialization function. This models the fact that JavaScript programs, including Node.js programs, generally have static initialization code outside of function bodies, for example as part of object declarations. We assume that the initialization function is deterministic.

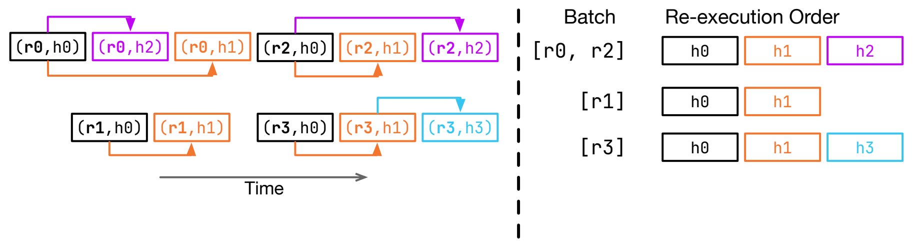
Activation partial order. The execution model described above induces a partial order on handler activations, A. Given handler activations h_0 and h_1, define the relation $\text{activator}(h_1) = h_0$ if and only if h_0 emitted the event e that led to h_1's activation or h_0 issued the I/O request or transactional request whose completion resulted in h_1's activation. This definition implies that any handler activation h without an activator (i.e., $\nexists h'$ s.t. $\text{activator}(h) = h'$) must have been run in response to a user request. For analytical convenience, we treat the initialization function's execution as a handler activation I, and use I as the activator for all user request activations. Observe that given our execution model and this definition, any handler $h \neq I$ must have a unique $\mathit{activator}(h) \neq h$. We use the *activator* relation to define the partial order A as the transitive closure of the activator relation. That is, we say $(h, h') \in A$ if $\text{activator}(h') = h$ or there exists $h = h_0, h_1, h_2, \ldots, h_n = h'$ such that $1 \le i \le n$, $\text{activator}(h_i) = h_{i-1}$. We sometimes write $(h, h') \in A$ as $h \prec_A h'$.

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

Related work. KEM extends $\lambda_{\mathcal{JS}}$ [46] which provides a semantic model for JavaScript. While $\lambda_{\mathcal{JS}}$ does not model events, prior work [61] shows extensions that model several event-driven frameworks. Similarly, KEM extends $\lambda_{\mathcal{JS}}$ by adding constructs for registering and unregistering event handlers (or listeners) and for emitting events. Unlike these works, KEM does not make assumptions about the order in which event handlers are executed nor about the number of concurrently executing event handlers. Thus, our algorithms, which are designed to check execution integrity for all KEM executions, can be extended to other languages and event-driven frameworks. Furthermore, this generality means that Karousos can be used even with future Node.js runtimes that adopt different event dispatch loops or use multiple threads.

```
function h0(arg) {          function h2(arg) {
  // handler for request      // handler for e2
  emit(e1, arg);              // …
  if (arg % 2 == 0)         }
     emit(e2, arg);
}
                            function h3(arg) {
function h1(arg) {            // handler for e3
  // handler for e1           // …
  if (arg % 7 == 0)         }
     emit(e3, arg)
}
```

(a) Application pseudocode



(b) Request execution timeline.

(c) Re-execution batches and ordering.

Figure 2. Grouped re-execution in Karousos: (a) Pseudocode for a simple application; (b) An example execution trace. A directed arrow between handlers h_0 and h_1 indicates that $h_0 \prec_A h_1$, i.e., the arrows represent the activation partial order; (c) Re-execution groups and the order in which groups are re-executed. Observe that requests r_0 and r_2 are batched together for replay despite executing handlers h_1 and h_2 in different orders.

4 Auditing event-driven servers

As in Orochi [87] (§2.3), the Karousos server collects advice that tells the verifier how to re-execute groups of requests simultaneously, which the verifier does using the SIMD-on-demand technique (§2.3). Karousos must address a key question: how should it group code to be re-executed? There is an essential trade-off: the more batching that is permitted (and hence the more opportunity for re-execution efficiency), the more there can be reordering in re-execution (relative to the original execution). However, the more reordering, the more the server has to collect advice to facilitate faithful replay; for example, if a read of a program variable is re-executed before the dictating write for that read, then the re-executed read would have to be somehow fed from advice.

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

Karousos aims for the midpoint of this trade-off: we want to enable a lot of reordering (to expose batching opportunities) while controlling the burden of logging. In the remainder of this section, we describe the choice of batching granularity (§4.1) and how Karousos facilitates faithful replay of operations on program variables (§4.2), assuming an honest server. We then describe how Karousos defends against an untrusted server (§4.3). Section 4.4 extends the design to transactional state.

Completeness and Soundness. In the following subsections, we will not discuss Completeness and Soundness explicitly, but we do discuss these properties right now. Note that all of the mechanisms of Karousos are relevant to both properties, in that the mechanisms must be designed so that (a) an honest verifier can replay an execution and (b) when the advice or observed outputs (in the trace) are wrong, the verifier can detect that fact. Intuitively, the reason that Karousos's mechanisms provide Soundness is that its algorithms insist (and reject otherwise) that there is some physically plausible ordering of events (at an honest server) that could have been produced by the actual program on the observed inputs. And, the reason that those same mechanisms provide Completeness is that they give information (for example, about scheduling) that allows the verifier to replay an execution.

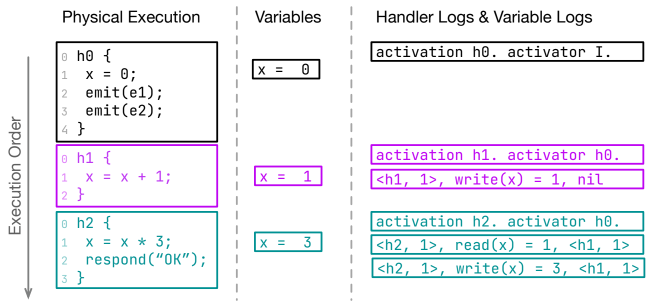
The mechanisms in Karousos, taken individually, do not obviously reflect the underlying complexity, which is substantial. This complexity derives from the need to handle arbitrary combinations of adversarial advice. The complexity does need to show somewhere; it appears in the proofs themselves (Appx C).

4.1 Batched re-execution in Karousos

In Karousos, a re-execution group comprises requests that have the same tree of handlers—that is, the same A relation (§3)—and the same in-handler control flows, meaning that corresponding handlers in different requests follow the same branches. Re-execution respects the A relation and program order within a handler but does not respect temporal order. Specifically, later requests can be re-executed before, or simultaneously with, temporally earlier ones. For example, in Figure 2, r_2 is later than r_0 and r_1, yet r_2 is re-executed together with r_0 and before r_1. Similarly, handlers within a request, if not ordered by A, can be reordered during re-execution; for example, (r_0, h_1) and (r_0, h_2) in Figure 2.

Section 5 describes how the server tracks the A relation and the control flow within a handler. Having done so, the server places in the advice a tag for each request in the trace (§2.1), where requests with the same tags allegedly belong in the same re-execution group.

Of course, the verifier does not trust that the server is honest about the claimed grouping. However, the verifier expects the server to include in its advice a description of the activation partial order A. Specifically, the advice is supposed to include, for each request, a *handler log*, with entries for each emit, register, and unregister (§3). An entry specifies the alleged activator (the handler), the alleged event, and (for register and unregister) the allegedly registered/unregistered function. When re-executing an emit, the verifier "trusts"

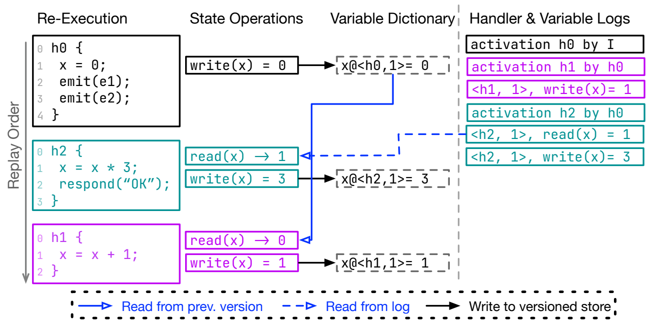


```
0 h0 {
1  x = 0;
2  emit(e1);
3  emit(e2);
4 }
0 h1 {
1  x = x + 1;
2 }
0 h2 {
1  x = x * 3;
2  respond("OK");
3 }
```

```
x =  0
x =  1
x =  3
```

```
activation h0. activator I.
activation h1. activator h0.
<h1, 1>, write(x) = 1, nil
activation h2. activator h0.
<h2, 1>, read(x) = 1, <h1, 1>
<h2, 1>, write(x) = 3, <h1, 1>
```

Figure 3. For program variables, Karousos only logs reads and writes that are not R-ordered. Execution points in the figure are depicted as a <handler, line number> tuple, so `<h2, 1>` means line 1 in handler h_2. Although the figure shows the full value logged on a read, in fact when logging reads in a variable log, the server records only the locations of the read and of the dictating write. When logging writes, the server records the value being written, and the locations of the write and the write that is being overwritten.



```
0 h0 {
1  x = 0;
2  emit(e1);
3  emit(e2);
4 }
0 h2 {
1  x = x * 3;
2  respond("OK");
3 }
0 h1 {
1  x = x + 1;
2 }
```

```
write(x) = 0
read(x) → 1
write(x) = 3
read(x) → 0
write(x) = 1
```

```
x@<h0,1>= 0
x@<h2,1>= 3
x@<h1,1>= 1
```

```
activation h0 by I
activation h1 by h0
<h1, 1>, write(x)= 1
activation h2 by h0
<h2, 1>, read(x) = 1
<h2, 1>, write(x)= 3
```

Figure 4. Re-execution in Karousos: During re-execution Karousos maintains a dictionary for each variable (the figure shows the dictionary for x) that contains previous values. Any logged reads return the logged value, while any unlogged reads use the most recent value (as defined by $\prec_R$) from the variable's dictionary. The figure depicts an abridged version of the logs from Figure 3. The notation `x@<h0, 1>` represents, for example, the value of `x` after line 1 of handler h_0 was executed.

the handler log, which implicitly indicates which functions are registered for the given event (all functions that have been registered but not unregistered before the emit). When re-executing register and unregister operations, the verifier checks that these operations are exactly the ones that appear in the log, thereby vindicating the "trust" placed in the handler log when re-executing emits. There are additional checks, for example that all emit entries in the handler log correspond to events that materialize during re-execution. Appendix C contains details, including the necessary bookkeeping.

4.2 Trusted recorder, out-of-order replay

How does Karousos faithfully re-execute reads of program variables? As we saw (§4.1), requests can be re-executed in the opposite order from what happened originally, which means that a read can be re-executed before the corresponding write. This section assumes that the server is honest.

What the (honest) Karousos server does. As a strawman, the server could include in its advice the values of all read operations, which the verifier could use to feed each re-executed read operation [51, 55, 68, 87] (see also §7). However, in logging every read or write of a program variable, this solution conflicts with the goal of controlling the log size (§2.1).

In contrast, the Karousos server decides dynamically whether to log a given operation. Karousos introduces the concept of *R-ordered*: two operations are R-ordered if one is guaranteed to be re-executed before the other under any possible grouping during re-execution. We say that they are *R-concurrent* if they are not R-ordered. More formally, we define a partial order R over operations. We say that $(o, o') \in R$ or $o \prec_R o'$ if (a) o was executed as a part of handler activation h, o' was executed as a part of handler activation h', and $(h, h') \in A$; or (b) o and o' were both executed as part of handler activation h and o was executed before o'. Observe that R can be regarded as the union of A and the program order, and that R is formalizing the constraints on re-execution that were stated in Section 4.1.

With this definition, we can now say what the Karousos server puts in the advice. Each variable notionally has a *variable log*. Then, as depicted in Figure 3, the server logs reads of program variables that are not R-ordered with respect to the dictating write. The server also logs writes of program variables that are not R-ordered with respect to the preceding write; this helps validate executions from untrusted servers (§4.3). In both cases, the server logs the function location and value written by the dictating or preceding write.

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

We say "in principle" because Karousos must solve a problem: feeding the re-executed read with the correct write operation. To illustrate the challenge, consider the naive solution of simply applying re-executed writes to a reconstructed copy of the variable, and feeding non-logged reads from that variable. In the re-execution depicted in Figure 4, this naive solution would cause h_1 to incorrectly read `x=3` (the most recently re-executed write) rather than `x=0`, which is the value faithful to the original execution (Figure 3).

Thus, the Karousos re-executor keeps, for each variable, all values written during the re-execution, indexed by the identifier of the handler and the line within the handler. We call this versioned variable the *variable's dictionary*. Figure 4 depicts the technique. The re-executor knows that if a read is unlogged, then originally that read must have observed a write that was prior according to R. To find that dictating write, the re-executor looks for the latest write in the variable's dictionary, where "latest" refers to the R relation. One can think of this as starting at the current handler, looking for the last write (if any) to the given variable by the current handler, and then repeating this step for each successive ancestor in the A tree until one encounters a write to the variable.

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

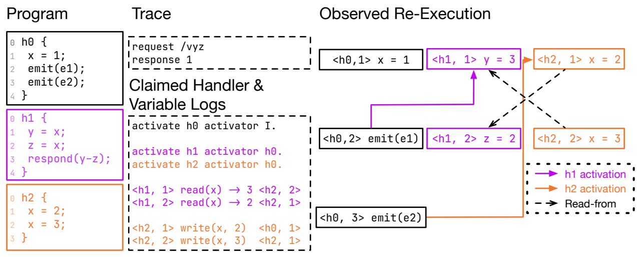
Discussion. Recall our goal of conserving log space (§2.1). First, and most important, the server places in the variable logs only what is necessary, given the possible reorderings that can happen from batched re-execution.

Second, we have designed the batching scheme so that logging is infrequently needed. In particular, looking at a tree of handlers where each handler touches state, a common pattern is that only the reads are concurrent with each other: consider, for example, an execution with one or more writes in a handler h, followed by a set of n reads, each in a handler h'_i, where h activates each $h'_1, \ldots, h'_n$. In this example, there is no logging required because each read is R-ordered: during the original execution, each read observes a write from h, which is an ancestor of the given h'_i. Notice that the preceding holds regardless of whether the h'_i are re-ordered during re-execution. Overall, this leads to good batching opportunities while controlling logging (§6.2–§6.3).

4.3 Untrusted recorder, out-of-order replay

This section relaxes the assumption of a well-behaved server. To motivate the relevant mechanisms in Karousos, we will consider several attacks. However, the soundness of the protocol (§2.1) is not based on reasoning about each thing that can go wrong but instead on an end-to-end proof (Appendix C.3.2).

Absent further mechanism, an adversarial server could put arbitrary values in a variable log, thereby causing re-executed reads to deviate from program execution. Thus, the verifier checks that the values of re-executed writes match



```
h0 {
 x = 1;
 emit(e1);
 emit(e2);
}
h1 {
 y = x;
 z = x;
 respond(y-z);
}
h2 {
 x = 2;
 x = 3;
}
```

```
request /vyz
response 1
```

```
activate h0 activator I.

activate h1 activator h0.
activate h2 activator h0.

<h1, 1> read(x) → 3 <h2, 2>
<h1, 2> read(x) → 2 <h2, 1>

<h2, 1> write(x, 2)  <h0, 1>
<h2, 2> write(x, 3)  <h2, 1>
```

Figure 5. An example of what the Karousos verifier could observe when re-executing the given program based on a dishonest server's advice. Karousos allows out-of-order re-execution by design, and can thus observe the execution shown. However, this execution is physically impossible: based on the execution model (§3) and the possible interleavings, the response should never be positive. So, the verifier ought to reject it.

alleged writes in the variable log. This is essentially Orochi's simulate-and-check (§2.3), adapted to Karousos's log structure.

At this point, one might conclude that the worst the server can do is incriminate itself by failing to justify an execution. But in fact, by creating both bad advice and bogus outputs, the server could fool the verifier into accepting impossible executions, with semantically invalid responses. Figure 5 depicts a small example. Other misbehavior is possible too, for example, the server could arrange for the verifier to wrongly validate "reads from the future", which would enable the server to rationalize an allegedly-read but wrong value, if a later request writes that value to a shared variable.

To ensure that the executions reproduced by the verifier are physically possible and consistent with external observations (meaning the trusted trace; §2.1), the verifier has a postprocessing phase, where it creates an execution graph G covering its entire audit. The graph establishes an alleged ordering among operations, and the verifier checks that it is acyclic. This technique creates a cycle in the example: $(h1, 2) \longrightarrow (h2, 2) \longrightarrow (h1, 1)$. This technique is inspired by other systems, including Orochi [87] (§2.3); see also Section 7. Like Orochi, the Karousos verifier includes edges for time precedence (referring to the ordering of requests in the trace; §2.1) and program order of operations.

The novel aspects in Karousos are as follows. First, Karousos includes edges that reflect the alleged activation partial order, A, based on the handler logs (§4.1). Second, the Karousos verifier embeds in G the alleged operation history of all variables. (Section 5 describes an optimization whereby our implementation tracks the history of fewer variables.) Notice that the history of accesses to a variable in the original execution should be a write, followed by zero or more reads, followed by a write, followed again by zero or more reads, and so on. The verifier reconstructs this partial order from a combination of re-executing and the variable logs.

Specifically, for each variable and each write w to that variable, the verifier maintains during re-execution a list of `read_observers`: all the reads r that observe w in re-execution, inferred from the variable log (if r was allegedly not R-ordered with w), or the versioned variable (if r was not present in the variable log). The verifier also maintains for each write w a `write_observer`: the write w' that succeeds w. Because w' and w might not be R-ordered—and thus the `write_observer` of w might not be inferrable from re-execution—write-write pairs are logged, as stated in Section 4.2. Now, after re-executing, the verifier uses these lists to embed edges in the graph G: WR (read-from edges, using `read_observers`), WW edges (write-write, using each write's `write_observer`), and RW (anti-dependency edges, connecting a given write's `read_observers` to that write's `write_observer`). Intuitively, these edges encode the history type mentioned in the prior paragraph.

Provided G has no cycles (and together with the verifier's other checks), the entire execution (of all requests in the audit) is well-ordered and physically possible, thus meeting the requirement of Soundness (§2.1).

4.4 Transactional state

Model We consider a transactional key-value store (KV store) that provides one of the following isolation levels: serializability, read committed, or read uncommitted [18]. Snapshot isolation is future work (§1). Each request issues operations to the KV store: `tx_start`, `tx_commit`, `tx_abort`, `PUT`, or `GET`. A transaction might be split across multiple handlers, but we assume that if multiple handlers issue operations on the same transaction, these handlers are not concurrent; in practice, the principal can efficiently check that the program meets this restriction before outsourcing the program.

Adya's isolation testing We build on Adya's algorithms [7]. For transactional KV stores, Adya's algorithms take as input the *history* of execution that comprises: (a) the *event order* at the KV store, which in this paper we call *TxOp order* to avoid confusion. This is a partial order of all operations in the KV store that preserves the order of operations within each transaction and includes the dictating write for each read, and (b) a *version order*: for each key, a total order of all committed values.

To test for an isolation level, these algorithms construct a graph H from the history. This is distinct from the graph G from earlier (§4.3), though both encode kinds of operation orders. The nodes of H correspond to the committed transactions in the TxOp order. H contains a *read-depend* edge $\langle T_1, T_2 \rangle$ if some operation in transaction T_2 reads from an operation in transaction T_1. It contains a *write-depend* edge $\langle T_1, T_2 \rangle$ if transaction T_1 writes some version of a key and transaction T_2 installs the next version. It contains an *anti-depend* edge $\langle T_1, T_2 \rangle$ if transaction T_1 reads some version of a key and transaction T_2 installs the next version.

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

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

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

First, similar to Section 4.3, the verifier ensures that all operations in the transaction logs are produced during re-execution. Second, the verifier ensures that the transaction logs are well-formed; specifically the verifier checks, by comprehensively inspecting the transaction logs, that transactions observe their own writes. Third, the verifier ensures consistency between the transaction logs and write order by checking that the operations in the write order are the last operations of committed transactions in the transaction logs.

Finally, the verifier needs to check that the transaction logs correspond to a legal KV store execution history that is consistent with the rest of the advice. Consider a server that claims that request r_1 issues the following operations, where k is a key in the KV store and x is a program variable: $op_1 = \texttt{GET}(k)$; $op_2 = \text{write}(x, 1)$, and request r_2 issues: $op_3 = \text{read}(x)$; $op_4 = \texttt{PUT}(k, 1)$. Additionally, the server claims that the dictating write of op_3 is op_2 and that op_1 reads from op_4. But op_3 reading from op_2 implies that op_2 originally preceded op_3, which implies that op_1 precedes op_4. Thus, the server is claiming, preposterously, that op_1 read from an operation that, according to the rest of the advice, was executed after it. To detect these types of misbehaviors, the verifier expands the graph G (§4.3) with nodes for external state operations, and adds write-read edges from PUTs to the corresponding GETs.[3]

5 Implementation

This section describes how the design in Section 4 is instantiated in a built system for auditing Node.js applications that optionally use MySQL as a transactional KV store.

Our system uses a transpiler to reduce the amount of effort that the principal needs to expend when using Karousos. Given an input program, this transpiler generates two programs: an instrumented version of the server that generates re-execution advice and can be deployed in an untrusted environment; and a verifier. We implemented our transpiler by extending the Babel [1] JavaScript transpiler, via Babel's plugin mechanism. Our transpiler supports a core subset of Node.js, however we currently do not support some features, including JavaScript workers and timers, and monkey patching.

The transpiler does not fully automate the process of using Karousos. Most significantly, the implementation of Karousos includes a substantial performance optimization that requires developer input. Namely, the developer annotates the variables that might be accessed by R-concurrent operations; such a variable is called a *loggable* variable. Now, if a variable is *not* annotated, that tells Karousos to assume every operation on the variable is R-ordered, allowing the server to skip the corresponding checks of R-concurrent accesses and the verifier not to track that variable's versions (§4.2). We note that marking a variable that has no R-concurrent operations loggable impacts performance but has no effect on Karousos's Soundness or Completeness (§2.1). Conversely, not annotating a loggable variable does not impact Karousos's Soundness (all unfaithful executions will be rejected) but compromises Completeness (some faithful executions might not be accepted).

Beyond that, the developer must change the application to use Karousos-provided versions of the Knex and Express libraries. The Karousos versions of these libraries are augmented to aid in advice generation; Express is augmented to annotate request handlers (§3) while Knex is augmented to collect TxOp order (§4.4). One can in principle extend the Karousos transpiler to automate these tasks. Below, we describe implementations of some of Karousos's mechanisms.

Identifying batches (§4.1). Recall that the Karousos server has to group requests (§4.1) with the same A relation and the same control flow within the handlers. To encode the A relation in a way that is invariant across requests, the server assigns an identifier to each function (functionID), and computes a *handlerID* as a digest of the functionID, the event that activates the handler, and the activator's handlerID. Notice that a handlerID is unique only within a request, and that if two requests have the same set of handler IDs, they have the same handler tree. To encode control flow within a handler, the server (as in Orochi [87, §4.3] and EAR [16, §3.1]) computes a *control flow digest*, updating it according to which branches are taken by the handler (the transpiler instruments the code that the server executes to enable tracking of branches during runtime). Then, the server computes the top-level *tag* of a request (§4.1) as a digest of all handler IDs and their corresponding control flow digests.

Accelerated re-execution (§2.3, §4.1). Karousos borrows *SIMD-on-demand* (§2.3) from Orochi [87] but implements it differently. Whereas Orochi modified a PHP runtime to expose *multivalue* (§2.3) versions of primitive types, we use the transpiler to turn program variables into multivalues.

Testing A, computing the activator relation (§3, §4.2). The Karousos server needs an efficient check of whether two handlers are ordered by A; similarly, the verifier needs to efficiently compute a handler's activator, when interrogating the variable dictionary (§4.2). For these purposes, the implemented server assigns a *label* to each handler so that two handlers are ordered by A iff the label of the one is a prefix of the other. In contrast to handlerIDs, labels do not correspond across requests; handler labels encode only enough information to check the A relation and compute activator(). Mechanically, a handler's label is computed at runtime as `parent_label/num` where `num` is the number of children of the parent that have executed so far.

Non-determinism. Node.js programs often use non-deterministic operations, which Karousos handles as other record-replay systems do [29, 30, 33]: the server records the result of each non-deterministic operation in the advice, and, during re-execution, the verifier supplies the recorded information in response to the operation. Karousos does not currently give soundness guarantees about non-deterministic operations, but prior works show how to implement basic checks of well-formedness [13, 20, 27, 50, 87, 96].

Transactional state (§4.4). Karousos uses MySQL as a transactional KV store by requiring individual queries to SELECT or UPDATE only a single row, specified by the row's primary key. This maps naturally to the abstract PUT-GET interface from Section 4.4. The server generates the transaction log (§4.4) by logging operations when they are executed by the application. The server captures the dictating PUT of each GET operation by storing each row's last writer in the row itself.

Our implementation obtains the write order (§4.4) by repurposing MySQL's binary log, or binlog. Repurposing the binlog required some effort, since it is designed for a different purpose (state replication), is in a format that is not well-documented, and contains extraneous information.

[3]It would be wrong to augment G with write-write edges or read-write edges between external state operations as Karousos does for program variables (§4.3). Program variables are sequentially consistent, whereas external state operations are more weakly ordered even in valid executions. These types of edges would thus constrain TxOp order artificially, causing the verifier to mark such executions as invalid, undermining Completeness (§2.1).

LOC, challenge, and limitation. Our implementation comprises 16,600 lines of JavaScript. In addition to the transpiler (5,700 lines), the implementation includes a library of helper functions used by the transpiled server and verifier (10,900 lines) and a program that periodically processes the MySQL binlog (100 lines of JavaScript).

Maintaining the activation partial order was a significant source of implementation overheads, and is also a significant source of runtime overheads (§6.1). It requires endowing each handler activation with knowledge of its activator's ID, and passing that information to all functions called by the handler. Meanwhile, many JavaScript functions are implemented in native code, and the transpiled code cannot change their call signatures or semantics. Our transpiler adopts a variety of strategies for this purpose some of which are reminiscent of techniques in Jardis [19] (for example, it saves the activator's id in a global variable that can be later retrieved).

6 Evaluation

We evaluate Karousos by answering the following questions:

1. What is the overhead of collecting advice (§6.1)?
2. What speedup does Karousos get from batching requests during verification, and what is the impact of Karousos's techniques (§6.2)?
3. What is the size of the advice that the server sends to the verifier, and what impact does Karousos's techniques have (§6.3)?

Applications. We evaluate Karousos with two model applications that we developed, which are designed to exercise and evaluate Karousos's algorithms, and can thus exhibit pathological behavior. We also evaluate with a real-world application: Wiki.js[2]. Details follow.

Message of the day: We created an application (executes ~1.6k LOC, including libraries) with which users can get or set a "message of the day" (MOTD). When setting the MOTD, a user specifies whether the message should be displayed every day or only on a particular day. Messages and metadata are stored in a local hashmap rather than in a transactional store.

Stack dump logging: We created an application (executes ~9k LOC, including libraries) to track stack dumps. Users can submit stack dumps, count how many times a stack dump has been reported, and get a list of unique dumps. Stack dumps, and the number of times they have been reported, are stored in a table (in the transactional store) indexed by the stack dump's digest. When a dump is submitted, the application first checks if a concurrent request has reported the same dump, in which case it returns a retry error (to avoid deadlocks). Next, the application checks if the dump is unique, if so it is added to the table. Otherwise, the number of times the dump has been submitted is incremented. To respond to list requests, the application issues a query for each digest in a particular variable that it maintains; the variable contains all digests stored in the table. The application thereby exercises the transactional key-value store interface (§4.4). Our figures call this application **stacks**.

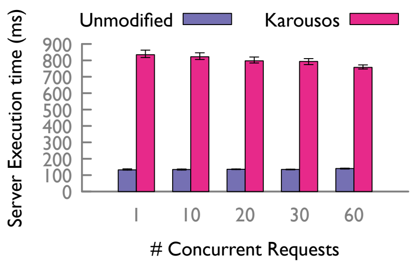
Wiki.js [2]: We modified the code to use only the Node.js features that our implementation supports and to add annotations (§5). These modifications required changing 200 lines of code (in a project with ~19k lines of code). The majority of these changes are simple: we merely needed to identify and annotate shared variables.

Baselines. We use three baselines:

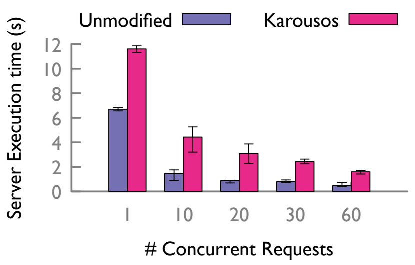
1. An *unmodified server*, to evaluate the overhead added by Karousos.
2. A *sequential re-executor*, which is the application server, modified to re-execute from the trusted trace. This helps evaluate the Karousos verifier, and is pessimistic for Karousos: any verifier that uses re-execution and does not batch requests would in addition need to consult some sort of advice (which this baseline does not do), and would thus be at least as slow as this baseline.
3. *Orochi-JS*, which helps evaluate the Karousos verifier, compared to Orochi. We cannot directly run Orochi, since its implementation is bound to PHP [87, §4]. Instead, we implement Orochi's algorithms using the Karousos codebase. Specifically, requests are placed in a re-executed batch only if they induce the identical sequence of handlers, not merely a topologically equivalent tree (§2.3, §4.1). Also, all accesses to (loggable) variables are logged, rather than only the R-concurrent accesses (§2.3, §4.2).

Workloads. For Wiki.js we use a mixed workload consisting of 25% page creations, 15% comment creations, and 60% render requests. The ratios are loosely derived from a Wikipedia trace [89]. For the other applications, we use three types of workloads: (a) read-heavy with 90% read requests and 10% write requests (90% reads in the figures); (b) write-heavy with 90% write requests and 10% read requests (90% writes in the figures); and (c) mixed with 50% write requests and 50% read requests. Across all workloads, write requests to the stack dump application are split so that 10% of them report a new stack dump and the remaining 90% report a previously reported one. Our experiments vary the number of concurrent requests from 1 to 60, and use 600 requests. Unless otherwise specified, graphs show the median from 10 experiments, and errors bars show 5th and 95th percentile values.

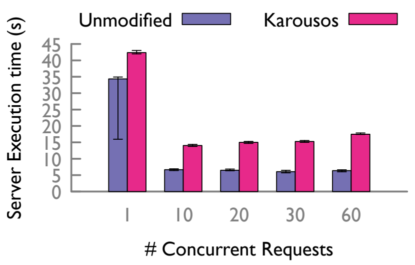
Testbed. All experiments are run on servers equipped with a 3.7GHz Intel(R) Xeon(R) E5-1630 v3 (4-core) CPU with 32GB RAM and 1TB SSD, running Ubuntu 16.04. We run the server and verifier using the Node.js v12.20.0 runtime, and use MySQL 8.0.19 as the transactional store. The application and MySQL are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



(a) MOTD (90% writes)



(b) Stacks (90% reads)



(c) Wiki.js (mixed)

Figure 6. A Karousos server compared to an unmodified server, in terms of processing time, for 480 requests. We show results for workloads with the largest overheads; overheads are otherwise lower (see text).

6.1 Advice collection overheads

We measure the time taken to serve a request trace while varying the number of concurrent requests, for Karousos and the unmodified server. Of the 600-request workload, each experiment uses the first 120 requests to warm up the application; we report time taken to serve the remaining 480 requests.

Figure 6 depicts the results. In the MOTD application, Karousos's overheads for the server depend on the type of workload and the number of concurrent requests. Workload has a stronger effect. The more writes, the worse Karousos's overhead. Specifically, for the write-heavy workload, the total execution time at the Karousos server is 5.4–6.3× larger than the baseline. Not depicted are the mixed workload, where Karousos's overhead is 3.4–3.7× larger, and the read-heavy workload, where Karousos's overhead is 2.5–2.7× larger (see Appx B). The reason is that recording a write access to a variable is more expensive on average: an R-concurrent write induces one or two logged values, whereas an R-concurrent read induces zero or one logged values (§4.2).

In the stack dump application, the execution time both for the unmodified server and for the Karousos server decreases as concurrency increases. This is an artifact of the application, in which increasing concurrency leads to more conflicts, which leads to retry errors (as described earlier), and thus less useful work. Furthermore, overheads in this application are higher for workloads with more reads. Consequently, in the read-heavy workload (depicted) Karousos overheads are 1.7–3.5×, in the mixed workload (not depicted) they are 1.4–3.6×, and in the write-heavy workload (not depicted) they are 1.2–2× (see Appx B). For read requests, the bottleneck is tracking the activation partial order (§3, §4.1, §5), a burden that rises with the degree of concurrency. For write requests, by contrast, write transactions are the bottleneck for both the Karousos and unmodified servers. Consequently, Karousos's overheads have a smaller effect on the application's processing time for write-heavy workloads.

For Wiki.js, the Karousos server's response latency is 1.2–2.8× higher than the baseline. Similar to read requests in the stack dump application, overheads in this application increase with the number of concurrent requests because tracking activation order becomes more expensive. However, in Wiki.js, each request has a smaller number of activations (each request causes fewer transactions), so we see a smaller increase in overheads as we increase concurrency.

6.2 Verification performance

We compare Karousos's verifier to the re-execution and Orochi-JS baselines, using the 600-request workload, and measuring total time to verify a trace.

Figure 7 depicts the results. In the MOTD application, Karousos is worse (∼ 22×) than sequential execution for the write-heavy workload. For the mixed workload (depicted in Appx B), Karousos is ∼ 4.3× more expensive; for the read-heavy workload, Karousos is 30% faster than sequential execution. The reason is that the bottleneck for re-executing any request, whether batched or otherwise, is accessing the hashmap. Meanwhile, accesses to the hashmap are not deduplicated. The write-heavy workload has higher verification time when using Karousos because the number of writes dictate the size of the value dictionary (§4.2) and thus the verifier's heap size (which in turn dictates allocation and memory management overheads).

In this application, Karousos has no benefit over Orochi-JS; the reason is interesting. Because there is only one handler, all handler executions are user request activations; thus, all are *R*-concurrent with each other, as children of *I* (§3, §4.2). Indeed, these requests, though not necessarily physically concurrent, can be re-executed in any order. Now, because all are *R*-concurrent, Karousos logs all of the accesess (as Orochi-JS does). Batching is also the same because, with no tree of handlers, Karousos and Orochi-JS group identically.

In the stack dump application, Karousos outperforms sequential execution when there is no concurrency. With concurrency, the comparison is equivocal because advice size increases, and this effect competes with the benefit from more batching opportunities (discussed immediately below). In all workloads for this application, Karousos outperforms Orochi-JS. By analyzing the workloads, we find that the higher the number of concurrent handlers activated by requests in the workload, the larger the speedup for Karousos relative to Orochi-JS. This is because a larger number of concurrently activated handlers increases the likelihood that

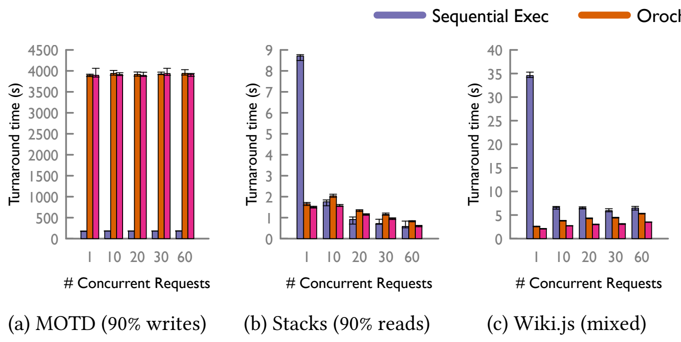


Figure 7. Karousos verification time vs baselines, on 600-request workload.

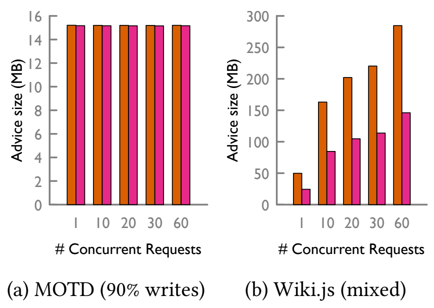


Figure 8. Size of advice, on 600-request workload.

handlers are re-ordered (§4.1), decreasing the grouping opportunities for Orochi-JS, thus demonstrating the importance of Karousos's design decision.

For Wiki.js, the Karousos verifier outperforms both baselines. The Karousos verifier's re-execution time increases with the number of concurrent requests (because more concurrency results in more advice that needs to be processed by the verifier), but not as dramatically as in stack dump because Wiki.js activates fewer handlers. Also, as the number of concurrent requests increases, so do the speedups of Karousos compared to Orochi-JS: with no concurrent requests the Karousos verifier is 19% faster than the Orochi-JS verifier while for 60 concurrent requests it is 34% faster.

The speedup of Karousos, relative to both baselines, also improves as we increase the number of requests being verified (not depicted). That is because, the more requests, the more opportunities for batching.

6.3 Advice size

The experimental configuration is the same as in the prior section. Figure 8 depicts the size of the advice sent by a Karousos server to a Karousos verifier. In the MOTD application, advice size does not vary with the number of concurrent requests, and is the same for Karousos and Orochi-JS. That is because nearly all of the advice ($\sim$ 95%) is the variable log (§4.2) of the hashmap. Meanwhile, every request is logged in both configurations because all accesses are R-concurrent, as explained in Section 6.2.

For Wiki.js, advice size under Karousos increases with the number of concurrent requests, because more accesses are logged, and because some of the logged objects (for example, an object that pools connections to the transactional store) increase in size with the degree of concurrency. The majority of the advice is variable logs (65% of total advice for no concurrent requests, and 95% of total advice for 60 concurrent requests). Karousos has smaller advice size than Orochi-JS, by 50%, because Karousos logs less (§4.2), thus demonstrating the effectiveness of Karousos's logging technique.

For the stack dump application, we do not report how advice size depends on the number of concurrent requests. This is because, for this application, a larger number of concurrent requests does not result in the execution of more concurrent handlers: as mentioned earlier, the application returns a retry error if one concurrent request reports the same stack dump as another, which actually leads to fewer concurrent handlers. Empirically, we observed that the advice size for this application was similar under Karousos and Orochi-JS, because although Karousos improves the size of the variable logs, the size of the handler log remains the same in both configurations. However, much of this application's state is in transactional storage, and variable logs are a relatively small component of the overall advice.

7 Other related work

We have discussed related works (§3, §4.4, §5) including those aiming at Karousos's goal of verifier-efficient execution integrity (§2.2, §2.3). Here we focus on other related techniques, primarily record-replay. Record-replay is a vast area, with several excellent surveys [29, 30, 33]. Karousos is the first to support the combination of: (a) an untrusted recorder, (b) accelerated replay, (c) executions with concurrency, and (d) controlling the size of advice supplied to the replayer.

Record-replay for execution integrity. AVM [48], Ripley [91], and Dickerson et al. [32] meet (a) but not (b). In AVM, an untrusted hypervisor records an execution while a trusted replayer uses something akin to our trace, together with VM replay [23, 36], to validate the execution. AVM's performance would be similar to the evaluated baseline (see Fig. 7). In Ripley [91], a web server re-executes client-side code. In Dickerson et al. [32], miners in a blockchain network execute transactions in parallel while validators re-execute the transactions in each block deterministically and concurrently. In DIVA [14], a trusted checker accelerates re-execution of an untrusted uniprocessor core; this meets (a) and (b) but not (c).

Record-replay with a trusted recorder. None of the works that we cite in the remainder of this section are designed for an untrusted server (characteristic (a)); any proposed use of them for execution integrity would require proof (§1).

Several works aim at (b). In Poirot [55] and Shortcut [34], the recorder captures hints that the replayer uses to re-execute (in Poirot's case, in a batch, as in Karousos), but the hints necessitate trust in the recorder.

Many works re-execute concurrent executions while controlling the size of advice supplied to the replayer; they are geared to (c) and (d). Below, we cover techniques relating to Karousos's variable logs (§4.2); for other record-replay work that targets concurrency, see JaRec [39], Respec [59], DoublePlay [90], and citations therein.

In Netzer [71], implemented in hardware by FDR [93] (see also [74]), when a data race occurs, the recorder logs the conflicting operations; the goal is to log a minimal set of such races. The replayer synchronizes these data races to reproduce the original order. In Bugnet [67], implemented in software by PinSEL [66] (see also [22, 74]), the recorder applies memory store operations to main memory and a *shadow memory*; on a load, if the main and shadow values disagree, the recorder infers that the memory was concurrently modified (for example by DMA), and logs the load. This technique also appears in Jalangi [78], which re-executes JavaScript and shares some of Karousos's approaches to handling calls to native code (§5).

These techniques are reminiscent of how Karousos decides whether to log an access to a program variable. However, they handle only *physical* concurrency, not R-concurrency (§4.2). Notice that R-concurrency is strictly harder: two accesses that are not physically concurrent (and hence are reconstructible with a traditional re-execution) could nevertheless be R-concurrent, and hence need logging (§4.2).

A third approach is to log enough information for the replayer to reconstruct a thread schedule equivalent to the original. In CREW systems [28, 35, 56, 58, 95], the recorder logs, for read operations, a current version number; for write operations, the recorder logs the number of readers before this write. The replayer then blocks a given write until all prior readers execute their read. As in Karousos, this approach has the freedom to re-order concurrent reads, with the "number of readers" (collected online) playing the same role as the anti-dependency edges in Karousos (§4.3). However, CREW reproduces a schedule equivalent to the original physical one, and thus cannot handle R-concurrency.

LEAP [51] has a similar log structure to Karousos's variable logs (§4.2), and thus would be amenable to out-of-order re-execution. However, LEAP is not designed to control the size of the logs. ORDER [94] improves on LEAP; it statically analyzes a program to determine which accesses need logging. Karousos could borrow these techniques to automatically identify which variables are loggable (§5).

Other related techniques. Prior work uses server-side logging in the JavaScript context to optimize page load times [68, 69], debug web applications [70], and facilitate archives [41]. These works are built on the Scout [68] framework, which comprehensively tracks server-side data flow. This work, impressively, shows that such tracking can have negligible overhead. However, the logs in question are large. This is acceptable in that context; for example, optimization generally happens during testing. Karousos's logging, by contrast, happens during online use, so is aimed at keeping communication overhead low, so logs selectively. More fundamentally, data flow tracking at the server is tantamount to an untrusted recorder: any proposed use of this mechanism requires rigorous proof.

JARDIS [19] is a time-travel debugger for JavaScript; it allows (among other things) stepping from the execution of a callback backward to the handler that registered that callback. To do so, JARDIS wraps each handler, to pass in information about where it was registered, enabling the debugger to "walk up the activation stack." This is similar to Karousos's use of handler labels (§5).

Karousos's techniques for ensuring well-ordered executions (§4.3–§4.4) relate to memory checking and consistency checking: typically there is a dependency graph that the checker wants to be acyclic [7–9, 12, 21, 42, 72, 80, 82, 86].

8 Summary and discussion

Karousos introduces several new techniques to record-replay systems, including formalizing (with the definition of R-ordered) the kind of reordering that can exist in batched re-execution systems. The evaluation results show that auditability in this context has a price, primarily in server overhead. Although it is higher than we might like, it is not exorbitant, and now we know what the price actually is. Besides, we expect auditability to cost something. The results also show that Karousos's individual techniques balance re-execution throughput and the size of advice, and that in many applications and workloads (though not pathological ones) Karousos benefits over naive baselines and an implementation of Orochi for Node.js. Our implementation requires work from the developer to apply Karousos; nonetheless, Karousos substantially expands the frontier of comprehensive server auditing.

Acknowledgments

This work was improved by comments by Brad Karp, Jinyang Li, Cheng Tan, our shepherd Liuba Shrira, and anonymous reviewers of Eurosys 2024, OSDI 2023, and NSDI 2023. This work was supported by NSF under awards CNS-2145471 and CNS-1514422, AFOSR under award FA9550-18-1-0421, DARPA under Agreement No. HR00112020022, and a gift from Google. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government.

References

[1] Babel. https://babeljs.io/.
[2] Wiki.js. https://github.com/Requarks/wiki.

[3] wrk. https://github.com/wg/wrk.

[4] Tornado Web Server. https://www.tornadoweb.org/en/stable/, 2020.

[5] Node.js. https://nodejs.org/en/, 2022.

[6] Phoenix Framework. https://phoenixframework.org/, 2022.

[7] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *Computer Aided Verification (CAV)*, July 2014.

[9] Jade Alglave and Luc Maranget. Stability in weak memory models. In *Computer Aided Verification (CAV)*, July 2011.

[10] Amazon Web Services. AWS Lambda FAQs. https://aws.amazon.com/lambda/faqs/.

[11] Amazon Web Services. Deploy Node.js Lambda functions with .zip file archives. https://docs.aws.amazon.com/lambda/latest/dg/nodejs-package.html.

[12] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store *actually* provide? In *USENIX Workshop on Hot Topics in System Dependability (HotDep)*, October 2010. Full version: Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[14] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 196–207, 1999.

[15] Ahmed Awad and Brad Karp. Execution integrity without implicit trust of system software. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2019.

[16] Ahmed Awad and Brad Karp. Enclave-accelerated replay: Efficient integrity for server applications. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2022.

[17] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *ACM Symposium on the Theory of Computing (STOC)*, May 1991.

[18] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[19] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for JavaScript/Node.js. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, November 2016.

[20] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[21] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[22] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, June 2006.

[23] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[24] Xavier Carpent, Gene Tsudik, and Norrathep Rattanavipanon. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In *DATE*, pages 1191–1194, 2018.

[25] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[26] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2013.

[27] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[28] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2013.

[29] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2), September 2015.

[30] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[31] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *HotNets*, pages 38–45, 2022.

[32] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, 33(3):209–225, 2020.

[33] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques (PDPTA)*, August 1996.

[34] Xianzheng Dou, Peter M. Chen, and Jason Flinn. ShortCut: Accelerating mostly-deterministic code regions. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

[35] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008.

[36] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[37] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *IACR International Cryptology Conference (CRYPTO)*, August 2010.

[38] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2013.

[39] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software: Practice and Experience*, 34(6):523–547, 2004.

[40] Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia. Treaty: Secure distributed transactions. In *DSN*. IEEE, 2022.

[41] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V Madhyastha. Jawa: Web archival in the era of JavaScript. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[42] Wojciech Golab, Xiaozhou Li, and Mehul Shah. Analyzing consistency properties for fun and profit. In *PODC*, June 2011.

[43] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, August 2015. Prelim version STOC 2008.

[44] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[45] Jinyu Gu, Xinyue Wu, Bojun Zhu, Yubin Xia, Binyu Zang, Haibing Guan, and Haibo Chen. Enclavisor: A hardware-software co-design for enclaves on untrusted cloud. *IEEE Transactions on Computers*, 70(10):1598–1611, 2021.

[46] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, 2010.

[47] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *SoCC*, 2014.

[48] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[49] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[50] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: secure applications on an untrusted operating system. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, March 2013.

[51] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: The lightweight deterministic multi-processor replay of concurrent Java programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, February 2010.

[52] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[53] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity (CCC)*, 2007.

[54] Michelle Keeney, Eileen Kowalski, Dawn M. Cappelli, Andrew P. Moore, Timothy J. Shimeall, and Stephanie R. Rogers. Insider threat study: Computer system sabotage in critical infrastructure sectors. https://apps.dtic.mil/sti/citations/ADA636653, 2005. U.S Secret Service and CERT Coordination Center/SEI.

[55] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for web applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[56] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS*, June 2010.

[57] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[58] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, 1987.

[59] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[60] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. VButton: Practical Attestation of User-Driven Operations in Mobile Apps. In *MobiSys*, 2018.

[61] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, New York, NY, USA, 2015. ACM.

[62] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, May 2010.

[63] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, April 2008.

[64] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[65] Thomas Moyer, Kevin Butler, Joshua Schiffman, Patrick McDaniel, and Trent Jaeger. Scalable Web Content Attestation. *IEEE Transactions on Computers*, 61(5):686–699, 2012.

[66] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS*, June 2006.

[67] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Intl. Symp. Computer Architecture (ISCA)*, June 2005.

[68] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.

[69] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[70] Ravi Netravali and James Mickens. Reverb: Speculative debugging for web applications. In *SoCC*, pages 428–440, 2019.

[71] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, December 1993.

[72] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), October 1979.

[73] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[74] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[75] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, page 199–212, 2009.

[76] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, August 2004.

[77] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, May 2015.

[78] Koushik Sen, Swaroop Kalasapur, and Tasneem Brutch. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE: Proceedings of the Joint Meeting on Foundations of Software Engineering*, August 2013.

[79] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[80] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming*

Languages and Systems (TOPLAS), 10(2):282–312, April 1988.

[81] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB linux applications with SGX enclaves. In *Network and Distributed System Security Symposium (NDSS)*, February 2017.

[82] Arnab Sinha and Sharad Malik. Runtime checking of serializability in software transactional memory. In *IPDPS*, pages 1–12. IEEE, 2010.

[83] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *ACM Conference on Programming Design and Implementation (PLDI)*, June 2016.

[84] Emin G un Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[85] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the Scalability of Platform Attestation. In *STC*, 2008.

[86] William N Sumner, Christian Hammer, and Julian Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *RV*, pages 161–176. Springer, 2011.

[87] Cheng Tan, Lingfan Yu, Joshua B Leners, and Michael Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 546–564. ACM, 2017.

[88] Justin Thaler. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2022.

[89] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[90] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):3, 2012.

[91] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically securing web 2.0 applications through replicated execution. In *ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[92] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM (CACM)*, 58(2):74–84, February 2015.

[93] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *Intl. Symp. Computer Architecture (ISCA)*, June 2003.

[94] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. ORDER: Object centRic DEterministic Replay for Java. In *USENIX Annual Technical Conference*, June 2011.

[95] Cristian Zamfir, Gautam Altekar, and Ion Stoica. Automating the debugging of datacenter applications with ADDA. In *Dependable Systems and Networks (DSN)*, June 2013.

[96] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[97] Tianwei Zhang and Ruby B. Lee. CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing. In *ISCA*, 2015.

A Artifact Appendix

A.1 Abstract

The primary purpose of the artifact is to allow reproduction of the results in Figures 6, 7, and 8 of the paper. We also provide instructions on how to reproduce the results in figures 9, 10, 11, and 12 of Appendix B. All runtime estimates are for a Linux system with a 3.7GHz Intel(R) Xeon(R) E5-1630 v3 (4-core) CPU, 32GB RAM and 1TB SSD.

After setting up the environment to execute Karousos (takes ≈ 20 minutes), reproducing the results for the paper requires: (1) compiling each of the target applications to produce the code that the server and verifier execute (takes ≈ 45 minutes). (2) running both the unmodified and the Karousos server on our target workloads to produce performance results for Karousos server overheads. Producing results only for Figure 6 takes ≈ 9 hours. Producing results for all figures (Figures 6, 9a, 10a, 11a, and 12a takes ≈ 20 hours. (3) running the unmodified application, a modified version of the server that collects advice both for Karousos and Orochi-JS, the Karousos verifier, and the Orochi-JS verifier on our target workloads to produce performance results for the verifier turnaround time and the advice size. Producing results only for Figures 7 and 8 takex ≈ 7.5 hours. Producing results for all figures (Figures 7, 8, 9b, 9c, 10b, 10c, 11b, 11c, 12b, and 12c) takes ≈ 13.5 hours.

A.2 Description & Requirements

A.2.1 How to access

Code is publicly available at https://github.com/nyu-systems/karousos/

A.2.2 Hardware dependencies

8G RAM

A.2.3 Software dependencies

Docker if using the artifact. If running locally, Node v12.16.1, NPM 6.13.4, and wrk [3]. However, we advise against this (details below).

A.2.4 Benchmarks

None

A.3 Set-up

We recommend using docker to run the experiments. Details on how to install docker are in the README at the repo.

You can also run the experiments locally. This requires installing a modified version of MySQL that collects advice for Karousos. Instructions are in the repo but process is complicated.

A.4 Evaluation workflow

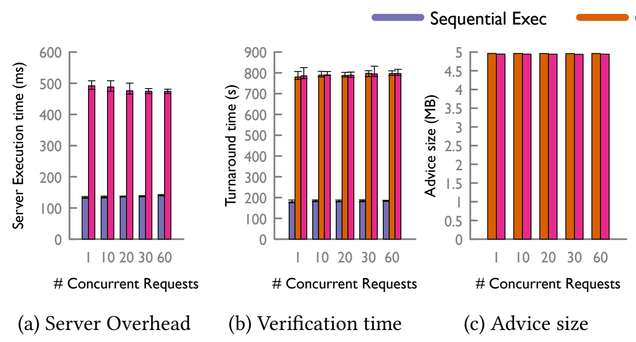
Major claims are Figures 6, 7, and 8. Secondary claims in Figures 9, 10, 11, and 12. Claims can be validated either with docker or by running in local machine. Docker is recommended. Instructions to execute in local machine are in the README.

Easiest way to reproduce data for Figures 6, 7, and 8 is executing `make produce-results`. This will set up the environment in the docker container, compile the applications for Karousos, and produce data for figures 6, 7, and 8.

Alternatively, you can use individual commands to create the container, set it up, and run the experiments. Instructions in the README. This way, you can also reproduce the data for figures 9, 10, 11, and 12.

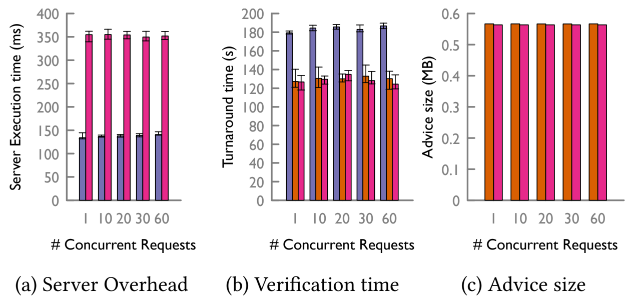
B Additional experimental plots

Below are the graphs for the workload types that are omitted from Section 6. To evaluate server overhead we use the same setup as in Figure 6. For verification performance we use the same setup as in Figure 7 and for advice size we use the same setup as in Figure 8.



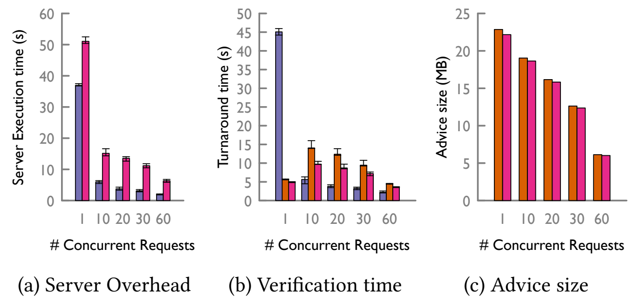
(a) Server Overhead (b) Verification time (c) Advice size

Figure 9. Karousos performance for MOTD (mixed)



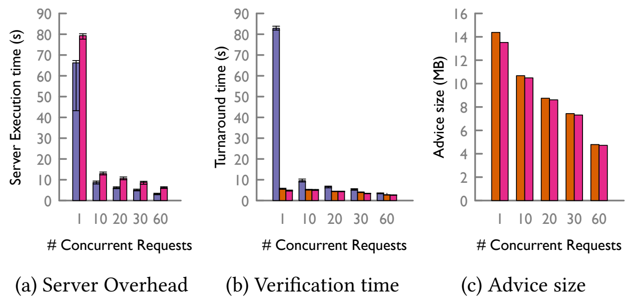
(a) Server Overhead (b) Verification time (c) Advice size

Figure 10. Karousos performance for MOTD (90% reads)



(a) Server Overhead (b) Verification time (c) Advice size

Figure 11. Karousos performance for stacks (mixed)



(a) Server Overhead (b) Verification time (c) Advice size

Figure 12. Karousos performance for stacks (90% writes)

C Karousos Algorithms And Correctness Proofs

C.1 Algorithms

In the following, a *global handler* is a handler that is registered by the initialization function (§3).

C.1.1 Annotating loggable accesses

The principal annotates a program P (§4.2) by identifying all loggable variables S and placing a special annotation called OnInitialize right after the initialization of each variable v in S.

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

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

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

C.1.2 Request ids, Handler ids, Variable IDs, and Transaction ids

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

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

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

C.1.3 Advice collection

The honest server collects the following advice:

- The control flow groupings (C) (§4.1).
- The handler logs *HL*s (§4.1): for each request, the ordered log of handler operations that the request issued. Each entry in the log is one of the kinds below. For all of these, *hid* is the id of the handler that issues the operation and opnum is the order of this operation among all operations that the handler issues:
  - register operations are tuples (*hid*, opnum, *functionID*, *eventNames*), where *functionID* is the id of the function, and *eventNames* is the set that contains the names of the events that the handler is registered for.
  - emit operations are tuples (*hid*, opnum, *eventName*), where *eventName* is a string that corresponds to the name of the event. An emit operation activates all functions that are registered for the event with name *eventName* (For more details on events and handler operations check Section 3).
  - unregister operations are tuples (*hid*, opnum, *functionID*, *eventName*), where *functionID* is the id of the function that is unregistered from event name *eventName*.
  - Check operations is a class of operations that inspect the handlers and the events. The server logs such operations as tuples (*hid*, opnum, *opInfo*), where *opInfo* is the name of the operation and any arguments that the operation is called with.
- The variable logs *VL*s (§4.2): We denote the variable log of a variable id v as VLv. VLv is a map from triplets (request id, handler id, opnum) to tuples of type (t: AccessType, v: Value, prec_rid: request id, prec_hid: handler id, prec_anum: Int). These are created during execution; on each variable access, the server follows the algorithms of Figure 13. AccessType is READ or WRITE. READ entries contain references to the write that they observe. WRITE entries contain the value written.
- The transaction logs *TXL*s (§4.4): for each transaction id, an ordered log of all operations that the transaction executes. Each entry is of the form:

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

  where
  - *hid* is the id of the handler that executes this operation

```
logs are of type $(requestid, handlerid, \mathbb{N}) \rightarrow (\{read, write\}, value, requestid, handlerid, \mathbb{N})$
READ entries contain references to the write that they observe.
WRITE entries contain the value written.
**procedure** OnInitialize($rid$, $hid$, opnum, $v$)
    Let $v$. log ← empty $VL$
    Let $v.value$ ← $nil$    //the most recent written value
    //Store the most write operation ($rid$, $hid$, opnum)
    Let $v.rid$ ← $rid$
    Let $v.hid$ ← $hid$
    Let $v$.opnum ← opnum

**procedure** OnRead($rid$, $hid$, opnum, $v$)
    **if** $Rconcurrent((rid, hid, \text{opnum}), (v.rid, v.hid, v.\text{opnum}))$ **then**
        //Check that the write that we read from has already been logged. If it has not, log it.
        **if** $v.\log\{v.rid, v.hid, v.\text{opnum}\} = nil$ **then**
            Let $v.\log\{v.rid, v.hid, v.\text{opnum}\} \leftarrow (write, v.value, nil, nil, nil)$
        //Log the read
        Let $v.\log\{rid, hid, \text{opnum}\} \leftarrow (read, nil, v.rid, v.hid, v.opnum)$
    **return** $v$

**procedure** OnWrite($rid$, $hid$, opnum, opcontents, $v$)
    **if** $Rconcurrent((rid, hid, \text{opnum}), (v.rid, v.hid, v.\text{opnum}))$ **then**
        //Check that the write observed by this one has already been logged. If it has not, log it.
        **if** $v.\log\{v.rid, v.hid, v.\text{opnum}\} = nil$ **then**
            Let $v.\log\{v.rid, v.hid, v.\text{opnum}\} \leftarrow (write, v.value, nil, nil, nil)$
        //Log the write
        Let $v.\log\{rid, hid, \text{opnum}\} \leftarrow (write, \text{opcontents}, v.rid, v.hid, v.\text{opnum})$
    //This write is the most recent write. So set the $v$ fields $value$, $rid$, $hid$, opnum
    //to those of this write operation.
    Let $v.value$ ← opcontents
    Let $v.rid$ ← $rid$
    Let $v.hid$ ← $hid$
    Let $v$.opnum ← opnum
```

Figure 13. Pseudocode for server's logic on reaching an annotation.

  - opnum is the order of this operation among all other operations that the handler executes.
  - optype is the type of operation, namely `tx_start`, `tx_commit`, `tx_abort`, `PUT` or `GET`,
  - *key* is the key for `PUT` and `GET` operations and null otherwise.
  - opcontents are null except for `PUT` and `GET` operations: For `PUT` operations they are the contents that are written and for `GET` operations they are the position in the logs of the write that they read from.
- *writeOrder* (§4.4): a single log that allegedly reflects the order in which the server applied the writes to shared external state.
- *responseEmittedBy*: a map from request ids to tuples (hid, opnum) s.t. the handler with id *hid* is the one that sends back the response and opnum is the number of operations that *hid* had issued prior to sending the response.
- *opcounts*: a map from the id (rid, hid) of every handler that is executed to the total number of operations that the handler issues (may be zero).

C.1.4 Verifier

The verifier's algorithms are in Figures 14, 16, 17, 18, 19, 20, 21:

```
**Input** Trace Tr, **Input** Advice $A$, **Input** Isolation level $I$
**Global** Graph $G$

**Global** Map OpMap : $(requestid, handlerid, \mathbb{N}) \longrightarrow (\text{"handler\_log"}, requestid, \mathbb{N}) \cup (\text{"tx\_log"}, txid, \mathbb{N})$:
   maps the $i$-th operation of a handler to the location of this operation in the logs.

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

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

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

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

**Global** Map $lastModification$: Map from $(requestid, handlerid, key)$ to an integer representing the order of the
   last operation of the transaction that modifies this key among all other operations that the transaction issues

procedure Audit
    Preprocess()
    ReExec() // Figure 18
    Postprocess()

procedure Preprocess
    Check Tr is balanced.
    Run the initialization phase and log all global handlers.
    $G_{\text{Tr}} \leftarrow$ CreateTimePrecedenceGraph() // [87, Figure 6]
    SplitNodes() // [87, Figure 6]
    AddProgramEdges()
    AddBoundaryEdges() // Figure 15
    AddHandlerRelatedEdges() // Figure 16
    AddExternalStateEdges() // Figure 16
    IsolationLevelVerification() // Figure 17

procedure Postprocess
    AddInternalStateEdges() // Figure 21
    if CycleDetect(G) then REJECT

procedure AddProgramEdges
    //This procedures adds all the nodes of each handler and program edges
    //between consecutive operations within a handler.
    for all $(rid, hid)$ in $A.opcounts$ do
        if $rid$ does not appear in Tr then REJECT
        //Add the handler end, start nodes
        $G.add\_node((rid, hid, 0))$
        $G.add\_node((rid, hid, \infty))$
        for $i \leftarrow 1, \ldots, A.opcounts[(rid, hid)]$ do
            $G.add\_node((rid, hid, i))$
            $G.add\_edge((rid, hid, i-1), (rid, hid, i))$
        $G.add\_edge((rid, hid, A.opcounts[(rid, hid)]), (rid, hid, \infty))$
```

Figure 14. Pseudocode for verifier's audit procedure in Karousos.

```
// Global Variables are the ones in Figure 14

procedure AddBoundaryEdges
    // For all ($rid$, $hid$) that are request handlers, add edge from ($rid$, 0) to ($rid$, $hid$, 0)
    for all ($rid$, $hid$) in $A.opcounts$ do
        if $hid.parent\_hid = null$ then
            $G.add\_edge((rid, 0), (rid, hid, 0))$
    // For each $rid$, ($rid$, $\infty$) represents delivering the response. For the handler ($rid$, $hid_r$)
    // that delivers the response for $rid$ (according to $A$), add an edge to ($rid$, $\infty$)
    // from the operation of $hid_r$ just prior to delivering the response, and an edge from ($rid$, $\infty$) to the
    // operation of $hid_r$ just after delivering the response.
    for all $rid$ in Tr do
        if $A.responseEmittedBy[rid] = null$ or $A.responseEmittedBy$ is not of type (handler id, i) where $i \in \mathbb{N}$ then
            REJECT
        Parse $A.responseEmittedBy[rid]$ as $(hid_r, \text{opnum}_r)$
        if $(rid, hid_r, \text{opnum}_r) \notin G.\text{Nodes}$ then REJECT
        $G.add\_edge((rid, hid_r, \text{opnum}_r), (rid, \infty))$
        if $\text{opnum}_r = A.opcounts[(rid, hid_r)]$ then
            //In this case the handler's next operation is handler exit
            $G.add\_edge((rid, \infty), (rid, hid_r, \infty))$
        else
            $G.add\_edge((rid, \infty), (rid, hid_r, \text{opnum}_r + 1))$
```

Figure 15. Pseudocode for verifier's AddBoundaryEdges procedure in Karousos.

```
// Global Variables are the ones in Figure 14

procedure AddHandlerRelatedEdges
    //add edges between consecuting operations in handler logs and activation edges
    for all $rid$ in $A.HL$ do
        if $rid$ does not appear in Tr then REJECT
        $Registered \leftarrow$ new Set()
        for $i \leftarrow 1, \dots, A.HL_{rid}.length$ do
            $op \leftarrow A.HL_{rid}[i]$
            CheckOpIsValid($rid$, $op$)
            OpMap$[(rid, op.hid, op.\text{opnum})] \leftarrow$ (“handler_log”, $rid$, $i$)
            //Add the handler op precedence edge
            if $i \neq 1$ then
                Let $prev\_op \leftarrow (rid, HL_{rid}[i-1].hid, HL_{rid}[i-1].\text{opnum})$
                $G.add\_edge(prev\_op, op)$
            if $op$ is a register operation then
                for all $eventName$ in $op.eventNames$ do
                    $Registered.add(eventName, op.functionID)$
            else if $op$ is an unregister operation then
                $Registered.remove(op.eventName, op.functionID)$
            else if $op$ is an emit operation then
                for all $(op.eventName, functionID)$ in $Registered \cup GlobalHandlers$ do
                    $hid' \leftarrow (functionID, op.hid, op.\text{opnum})$
                    //Check that the server has reported the activated handler
                    if $A.opcounts[(rid, hid')] = \emptyset$ then REJECT
                    $activatedHandlers[rid, op.hid, op.\text{opnum}].add(hid')$
                    //add the activation edge
                    $G.add\_edge((rid, op.hid, op.\text{opnum}), (rid, hid', 0))$

procedure AddExternalStateEdges
    //Bookkeeping for external state and edges described in Section 4.4
    for all $(rid, tid)$ in $A.TXL$ do
        //Check if the transaction is allegedly committed or not
        if last operation in the log $TXL_{(rid,tid)}$ is of type commit then
            $Committed.add(rid, tid)$
        Initialize map $MyWrites$
        for all $i \leftarrow 1, \dots, TXL_{(rid,tid)}$ do
            Let $op \leftarrow TXL_{(rid,tid)}[i]$
            CheckOpIsValid($rid$, $op$)
            OpMap$[(rid, op.hid, op.\text{opnum}] \leftarrow$ (“tx_log”, $tid$, $i$)
            if $i \neq 1$ then
            if $op.\text{optype} = \texttt{GET}$ then
                Let $(rid_w, tid_w, i_w) \leftarrow op.\text{opcontents}$
                Let $op_w \leftarrow A.TXL_{(rid_w,tid_w)}[i_w]$
                CheckOpIsValid($rid$, $op_w$)
                $G.add\_edge((rid_w, op_w.hid, op_w.\text{opnum}), (rid, op.hid, op.\text{opnum})$ //Add a read-from edge
                // Add this op to the dictating write's list of readers
                if $op_w.\text{optype} \neq \texttt{PUT} \lor op_w.key \neq op.key$ then REJECT
                $ReadMap[(rid, tid_w, i_w)].add(rid, tid, i)$
                //Make sure that if it reads a key that it has modified, it reads the last modification
                if $op.key \in MyWrites \land MyWrites[key] \neq (rid_w, tid_w, i_w)$ then REJECT
            else if $op.\text{optype} = \texttt{PUT}$ then
                //update MyWrites
                $MyWrites[op.key] \leftarrow (rid, tid, i)$
                if $(rid, tid) \in Committed$ then
                    $lastModification[rid, tid, key] \leftarrow i$

procedure CheckOpIsValid($rid$: request id, $op$: operation)
    if $A.opcounts[(rid, op.hid)] = \emptyset$ then REJECT
    if $op.\text{opnum} < 1 \lor op.\text{opnum} > A.opcounts[(rid, op.hid)] \lor$ OpMap$[(rid, op.hid, op.\text{opnum})]$ exists then
        REJECT
```

Figure 16. Pseudocode for verifier's AddHandlerRelatedEdges and AddExternalStateEdges in Karousos

```
// Global Variables are the ones in Figure 14

procedure IsolationLvlVer
    Initialize $DG$ to an empty graph
    //Add a node for each committed transaction
    for all $(rid, tid) \in Committed$ do
        $DG.add\_node((rid, tid))$
    $writeOrderPerKey \leftarrow$ ExtractWriteOrderPerKey()
    if $I$ = READ UNCOMMITTED then
        AddWriteDependencyEdges($writeOrderPerKey$)
        if CycleDetect($DG$) then REJECT
    else if $I$ = READ COMMITTED then
        AddWriteDependencyEdges($writeOrderPerKey$)
        AddReadDependencyEdges()
        if CycleDetect($DG$) then REJECT
    else if $I$ = SERIALIZABILITY then
        AddWriteDependencyEdges($writeOrderPerKey$)
        AddReadDependencyEdges()
        AddAntiDependencyEdges($writeOrderPerKey$)
        if CycleDetect($DG$) then REJECT

procedure ExtractWriteOrderPerKey
    if $writeOrder.length \neq |lastModification|$ then REJECT
    Initialize $writeOrderPerKey \leftarrow$ Map from keys to lists
    for all $(rid, tid, i)$ in $A.writeOrder$ in order do
        Let $op \leftarrow TXL_{(rid,tid)}[i]$
        if $lastModification[(rid, tid, op.key)] \neq i$ then REJECT
        $writeOrderPerKey[op.key].append(rid, tid, i)$
    return $writeOrderPerKey$

procedure AddReadDependencyEdges // w-r edges
    for all $(rid_w, tid_w, i_w)$ in $ReadMap$ do
        //check that if the write is not the last modification, no committed transaction reads from it
        if $(rid_w, tid_w, i_w) \notin writeOrder$ then
            for all $(rid_r, tid_r, i_r)$ in $ReadMap[(rid_w, tid_w, i_w)]$ do
                if $(rid_r, tid_r) \in Committed$ then REJECT
        else
            for all $(rid_r, tid_r, i_r)$ in $ReadMap[(rid_w, tid_w, i_w)]$ do
                if $(rid_w, tid_w) \in Committed \land (rid_w \neq rid_r \lor tid_w \neq tid_r)$ then
                    $DG.add\_edge(\langle (rid_w, tid_w), (rid_r, tid_r) \rangle)$

procedure AddWriteDependencyEdges($writeOrderPerKey$) // w-w edge
    for all $key \in writeOrderPerKey$ do
        Let $o \leftarrow writeOrderPerKey[key]$
        for $j = 1, \ldots, o.length - 1$ do
            //check that there's only one version per transaction
            $DG.add\_edge(\langle (o[j].rid, o[j].tid), (o[j+1].rid, o[j+1].tid) \rangle)$

procedure AddAntiDependencyEdges($writeOrderPerKey$) // r-w edges
    for all $k \in writeOrderPerKey$ do
        Let $o \leftarrow writeOrderPerKey[k]$
        for $j = 1, \ldots, o.length - 1$ do
            for all $(rid, tid, \_) \in ReadMap[o[j]]$ do
                Let $T_1 = (rid, tid)$ and $T_2 = (o[j+1].rid, o[j+1].tid)$
                if $T_1 \neq T_2 \land T_1 \in Committed$ then
                    $DG.add\_edge(\langle T_1, T_2 \rangle)$
```

Figure 17. Pseudocode for verifier's isolation level verification in Karousos (§4.4)

```
//Global Variables are the ones in Figure 14
procedure ReExec
    Re-execute Tr in groups according to $A.C$
        (1) Initialize a group as follows:
            Read in inputs for all requests in the group. Let $in$ be these inputs
            Allocate program structures for each request in the group
            Initialize $active$: a queue of tuples (handler id, inputs)
            Find the $functionID$s of the request handlers.
            if the $functionID$s of the request handlers don't line up across requests then REJECT
            for all $functionID$ in $functionID$s do
                Let $hid \leftarrow (functionID, null, 0)$
                $active.Enqueue(hid, in)$
                if $\exists rid$ in the group s.t. $A.opcounts[(rid, hid)] = \emptyset$ then REJECT
        (2) Execute the requests in the group with SIMD-on-demand:
        while $active \neq \emptyset$ do
            (a) The runtime picks the next handler $c$ to execute
            if $c \neq null$ then
                Compute $hid$ from the $functionID$ of the function, the parent handler and the event.
                if $hid \notin active$ then
                    continue;//Do not execute this handler.
                else
                    Name the handler $hid$ and set the inputs to the ones associated with $hid$ in $active$
                    Remove $hid$ from $active$
                    $idx[hid] \leftarrow 1$
                    Execute the activated handler for all requests in the group
            else
                //Pick the next handler to be executed from $active$
                $(hid, in) \leftarrow active.Dequeue$
                $idx[hid] \leftarrow 1$
                Execute the function $hid.functionID$ for all requests in the group with inputs $in$
            (b) ReExecute $hid$ for all requests:
                if execution within the group diverges then REJECT
                if the group makes an external state operation then
                    optype $\leftarrow$ the type of state operation
                    for all $rid$ in the group do
                        opcontents, $tid$, txnum $\leftarrow$ parameters from execution
                        $s \leftarrow$ CheckStateOp($rid$, $hid$, $idx[hid]$, optype, $tid$, txnum, $key$, opcontents)
                        if optype = GET then
                            state op result $\leftarrow$ s
                    $idx[hid] = idx[hid] + 1$
                if the group reaches an annotated operation then
                    For all $rid$ in the group:
                        if opnum $> A.opcounts[(rid, hid)]$ then REJECT
                        if it is a write or initialization then
                            Execute the operation
                        Execute the annotation according to Figure 20 where opnum is set to $idx[hid]$
                    $idx[hid] = idx[hid] + 1$
                if the group makes a handler operation then
                    optype $\leftarrow$ the type of handler operation
                    for all $rid$ in the group do
                        $info \leftarrow$ parameters from execution
                        CheckHandlerOp($rid$, $hid$, $idx[hid]$, optype, $info$)
                    if optype = emit then ActivateHandlers($hid$, $idx[hid]$, $active$)
                    Execute the handler operation
                    $idx[hid] = idx[hid] + 1$
                if the group sends back a response then
                    if $\exists rid$ in the group s.t. $A.responseEmittedBy[rid] \neq (hid, idx[hid])$ then REJECT
                    Write out the produced outputs
            (c) When the execution of the handler $hid$ exits
                if $\exists rid$ in the group s.t. $idx[hid] < A.opcounts[(rid, hid)]$ then REJECT
        (3) for all $rid$ in the group do
            if the produced outputs are not exactly the responses in Tr then REJECT
            //Check that there are no handlers in the advice that we did not execute
            if $\exists rid$ s.t. $\exists hid : A.opcounts[(rid, hid)]$ but $(rid, hid)$ was not executed by ReExec then REJECT
    return ACCEPT
```

Figure 18. Pseudocode for verifier's ReExec in Karousos

```
//Global Variables are the ones in Figure 14

procedure CheckStateOp($rid$, $hid$, opnum, optype, $tid$, txnum, $key$, opcontents)
    //Simulate and check logic for state operations (§2.3, §4.4)
    if opnum > $A.opcounts[(rid, hid)]$ then REJECT
    Let $(t, tid_c, \text{txnum}_c) \leftarrow \text{OpMap}[(rid, hid, \text{opnum})]$
    if $t \neq$ "tx_log" $\vee\ tid_c \neq tid \vee \text{txnum}_c \neq \text{txnum}$ then REJECT
    Let $op \leftarrow A.TXL_{rid}\,tid[\text{txnum}]$
    if $op$.optype ≠ optype ∧ $op$.optype ≠ tx_abort ∧ optype ≠ tx_commit then REJECT
    if $op.key \neq key$ then REJECT
    if optype ≠ GET then
        if $op$.opcontents ≠ opcontents then REJECT
    else
        Let $(rid_w, tid_w, i_w) \leftarrow op$.opcontents
        Let $op_w \leftarrow A.TXL_{rid_w}\,tid_w[i_w]$ return $op_w$.opcontents

procedure CheckHandlerOp($rid$, $hid$, opnum, optype, $info$)
    //Check that the handler operation matches the entry in the logs (§4.1)
    if opnum > $A.opcounts[(rid, hid)]$ then REJECT
    Let $(t, rid_c, i) \leftarrow \text{OpMap}[(rid, hid, \text{opnum})]$
    if $t \neq$ "handler_log" $\vee\ rid_c \neq rid$ then REJECT
    Let $op \leftarrow A.HL_{rid}[i]$
    if $info$ does not match the fields in $op$ then REJECT

// The following procedure is called by ReExec while it is executing a control flow group
// when it encounters an emit operation.
// It checks that all requests in the group induce the same handlers,
// and adds the handlers to $active$.
procedure ActivateHandlers($hid$, $i$, $active$)
    //Check that $(hid, i)$ activates the same handlers across all requests, according to the advice (§4.1)
    if exist $rid_1, rid_2$ in the group s.t. $activatedHandlers[rid_1, hid, i] \neq activatedHandlers[rid_2, hid, i]$ then REJECT
    Let $in$ the set of values of the emit operation across all requests.
    for all $hid' \in activatedHandlers[rid, hid, i]$ for some $rid$ in the group do
        $active.Enqueue(hid', in)$
```

Figure 19. Pseudocode for verifier's check op routines and activateHandlers routine in Karousos

```
all_variables ← {} // A set of all variables.
procedure OnInitialize(rid, hid, opnum, v)
    Let v.log ← VLv.variableID
    Let v.rid ← rid
    Let v.hid ← hid
    Let v.opnum ← opnum
    Let v.var_dict ← {} // Map from rid, hid, opnum to values.
    Let v.read_observers ← {} // maps from a write op to all readers who allegedly observed that op,
    // based on both server-supplied advice, and re-execution.
    Let v.write_observer ← {} // maps from a write op to 0 or 1 writers who allegedly observed that op,
    // based on either server-supplied advice or re-execution.
    Let v.initializer ← nil
    all_variables.insert(v)

procedure OnRead(rid, hid, opnum, opcontents, v)
    if v.log.contains(rid, hid, opnum) then
        // if a read is logged, then the server was supposed to
        // have logged the dictating write. So find the dictating
        // write in the log, and feed its value to the read.
        op, _, $rid_{op}$, $hid_{op}$, $opnum_{op}$ ← v.log{rid, hid, opnum}
        if op is not read or !v.log.contains($rid_{op}$, $hid_{op}$, $opnum_{op}$) then
            return nil
        op, value, _, _, _ ← v.log{$rid_{op}$, $hid_{op}$, $opnum_{op}$}
        if op is not write then
            return nil
        v.read_observers{($rid_{op}$, $hid_{op}$, $opnum_{op}$)}.insert((rid, hid, opnum))
        return value
    else
        //Below FindNearestRPrecedingWrite returns the last write by the nearest ancestor handler
        //by climbing up the handler tree and checking v.var_dict.
        Let $rid_p$, $hid_p$, $opnum_p$, value ← FindNearestRPrecedingWrite(v, rid, hid, opnum)
        if $rid_p = nil$ and $hid_p = nil$ then
            return nil
        v.read_observers{($rid_p$, $hid_p$, $opnum_p$)}.insert((rid, hid, opnum))
        return v
```

Figure 20. Code that verifier executes upon an annotated operation (§4.3), I

```
procedure OnWrite($rid$, $hid$, opnum, opcontents, $v$)
    Let $v.var\_dict\{(rid, hid, \text{opnum})\} \leftarrow$ opcontents
    if $v.\log.contains(rid, hid, \text{opnum})$ then
        Let $op, value, rid_o, hid_o, \text{opnum}_o \leftarrow v.\log\{rid, hid, \text{opnum}\}$
        if $op$ is not $write$ or $value \neq$ opcontents then
            return false // Operations or values don't agree.
        if $rid_o \neq nil$ and $hid_o \neq nil$ and $opnum_o \neq nil$ then
            if $v.write\_observer\{rid_o, hid_o, \text{opnum}_o\} \neq nil$ then
                return false // Two handlers cannot overwrite the same value.
            else
                Let $v.write\_observer\{rid_o, hid_o, \text{opnum}_o\} \leftarrow (rid, hid, \text{opnum})$
                return true
    else
        Let $(rid_p, hid_p, \text{opnum}_p, value) \leftarrow$ FindNearestRPrecedingWrite($v$, $rid$, $hid$, opnum)
        if $rid_p \neq nil$ and $hid_p \neq nil$ and $\text{opnum}_p \neq nil$ then
            Let $v.write\_observer\{rid_p, hid_p, \text{opnum}_p\} \leftarrow (rid, hid, \text{opnum})$
        else
            Let $v.initializer \leftarrow (rid, hid, \text{opnum})$
        return true

procedure AddInternalStateEdges
    for all $v \leftarrow all\_variables$ do
        Let $(rid, hid, \text{opnum}) \leftarrow v.initializer$
        while $rid \neq nil$ and $hid \neq nil$ and $\text{opnum} \neq nil$ do
            // Add WR (write-read) edges.
            for all $(rid_r, hid_r, \text{opnum}_r) \leftarrow v.read\_observers\{rid, hid, \text{opnum}\}$ do
                $G.add\_edge((rid, hid, \text{opnum}), (rid_r, hid_r, \text{opnum}_r))$
            if $v.write\_observer\{rid, hid, \text{opnum}\} \neq nil$ then
                // Add RW (anti-dependency) edges.
                for all $(rid_r, hid_r, \text{opnum}_r) \leftarrow v.read\_observers\{rid, hid, \text{opnum}\}$ do
                    $G.add\_edge((rid_r, hid_r, opnum_r), v.write\_observer\{rid, hid, \text{opnum}\})$
                // Add WW edge.
                $G.add\_edge((rid, hid, \text{opnum}), v.write\_observer\{rid, hid, \text{opnum}\})$
            Let $(rid, hid, \text{opnum}) \leftarrow v.write\_observer\{rid, hid, \text{opnum}\}$
```

Figure 21. Code that verifier executes upon an annotated operation (§4.3), II

C.2 Correctness Properties

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

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

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

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

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

  After the execution of a request's handler, the request is held, until the executor reschedules it. If a request is scheduled but the request has no active handlers, the executor immediately yields and chooses the next *rid* in S.
- At the end, output Tr′.

Our operation-wise execution differs from the one in given in Orochi [87] in that it explicitly constructs an alternate trace instead of consulting the observed one.

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

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

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

C.3 Proofs

We need the following definitions:

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

- $\mathit{rid} = \mathit{rid}'$ and $\mathit{hid} = \mathit{hid}'$ and $\text{opnum} < \text{opnum}'$, or
- $\mathit{rid} = \mathit{rid}'$ and *hid* is an ancestor of hid' in the handler tree.

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

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

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

For example:

$$(1, 0), (23, 0), (1, \mathit{hid}_1, 0), (23, \mathit{hid}_2, 0), (1, \infty), (1, \mathit{hid}_1, 1) \ldots$$

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

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

1. it is a permutation of the graph G that is constructed by Preprocess,

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

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

OOOAudit This procedure is shown in Figure 22.

Lemma 1 (Equivalence of well formed op schedules)**.** For all op schedules S_1, S_2 that are well-formed (with respect to Tr and advice A)

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

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

Now, assume that OOOExec(S_1) and OOOExec(S_2) are equivalent, meaning that (a) either both accept or both reject and (b) they access the same variables setting the *initializer*, *write_observer* and *read_observers* of each variable to the same values. Now examine the execution of Postprocess. (b) implies that the edges that are added to G by AddInternalStateEdges are the same in both executions and, thus, the constructed graph G is the same in both executions. CycleDetect thus runs the same in both executions. Therefore, either both executions accept or both reject.

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

1. Fix a request *rid*.
2. First notice that the only global state that is modified during OOOExec is the *active* map, per-variable dictionaries, lists of read_observers, write_observers, and initializer.
3. Base case: Because both schedules are well-formed, the first operation of a request is $(rid, 0)$: none of the data that this execution depends on get modified throughout OOOExec. So the execution of this operation is independent of its position in the log, and it is executed in the same way in both executions.
4. Induction: If both executions are about to execute operation k of request *rid*, and neither has rejected so far, the execution of operation k will proceed in the same way in both executions.
   - Assume that the next operation is (rid, ∞): The handler *hid* which both executions of OOOExec execute is the one in *A.responseEmittedBy*. Moreover, because the schedules are well formed, the latest operation of (rid, hid) that has been executed so far on both executions is
     $(rid, hid, A.responseEmittedBy[rid].\text{opnum})$. Thus, both executions will execute the handler that allegedly sends back the response, from the (allegedly) last operation prior to the response up until the next operation. Because of the induction hypothesis and the fact that the execution of a handler between operations is deterministic, the two executions will proceed in the same way up until right before the next event, producing the same state. Moreover, the next event will be the same in the two executions. If this event is not the emission of a response, both executions will reject. Otherwise, because both executions have the same state, the produced outputs will be the same.
   - Operations (rid, hid, i):
     – If it is a handler start operation ($i = 0$) then the executions do not depend on state that is modified except for the check in line 25. We will show that either both executions accept or both reject: Assume that this does not hold. Then, without loss of generality assume that the check passes in OOOExec(S_1) and fails in OOOExec(S_2). So in OOOExec(S_2), *hid* is not in *active*[*rid*], either because (i) *hid* was in *active*[*rid*] and removed from it, or (ii) *hid* was never added to *active*[*rid*]. We can rule out case (i) because the only place where *hid* could be removed is line 32 which, if it were executed, would mean that (hid, rid, ∞) appears before $(hid, rid, 0)$ in S_2, which is not possible, since S_2 is well-formed and in particular respects program order. So case (ii) holds.
       Now, in OOOExec(S_1), *hid* is in *active*[*rid*]. There are two places where *hid* could have been inserted: (a) line 16 during execution of $(rid, 0)$ or (b) line 49 during the execution of an emit operation $(rid, parent, j)$. Consider case (a). Because S_1 and S_2 are well-formed, $(rid, 0)$ appears before $(rid, hid, 0)$ in both S_1 and S_2. Also, as argued above, both OOOExec(S_1) and OOOExec(S_2) execute $(rid, 0)$ the same way, initializing *active*[*rid*] to the same value. Therefore, if case (a) holds for OOOExec(S_1) then correspondingly, *hid* would have been inserted in OOOExec(S_2) in the same line, in contradiction to case (ii) above. So case (b) holds.

```
1:  //Global Variables are the ones in Figure 14

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

Figure 22. Pseudocode for OOOAudit in Karousos.

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

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

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

□

C.3.1 Completeness

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

1. First, we establish that for any well-formed op schedule S, OOOAudit(Tr, A, S) accepts (Lemma 2).
2. Next, we show that Audit(Tr, A) is equivalent to OOOAudit(Tr, A, S') for a specific well-formed op schedule S' (Lemma 3). We take S' to be the op schedule that results from a "flattened" batch execution.

Lemma 2 (OOOAudit Completeness)**.** If the executor executes the given program (under the execution model given in Section 3 and the given advice collection procedure, producing trace Tr and advice A, then for any well-formed op schedule S (with respect to Tr and A), OOOAudit(S) accepts.

Proof. Because of Lemma 1, it is sufficient to prove that there exists some well-formed op schedule S' (with respect to Tr and A) for which OOOAudit(S') accepts.

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

- A *request event* happens when a request *rid* reaches the server, and is notated as $(rid, 0)$.
- A *response event* happens when the server issues a response for a request *rid*, and is notated as (rid, ∞).
- A *handler start event* happens when the server starts executing a handler (rid, hid), and is notated as $(rid, hid, 0)$.
- A *handler end event* happens when the server finishes executing a handler (rid, hid), and is notated as (rid, hid, ∞).
- A (rid, hid, i) event happens when the server either collects advice associated with a handler op or a state op, or when the handler executes an annotated operation.

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

Sub-lemma 2.1. S' is well-formed, with respect to the Trace Tr and advice A produced by the online execution.

Proof. First, we show that S' is a permutation of the nodes in graph G. Since the server is honest, S' contains exactly one request event and exactly one response event for each request in Tr; so does G (from the logic of CreateTimePrecedenceGraph and SplitNodes). Moreover, S' contains exactly one handler start event and exactly one handler end event for each handler (rid, hid) that is executed; so does G. This follows from the logic of AddProgramEdges, specifically, lines 39 and 40 of Figure 14, and the fact that the server faithfully executes the advice collection procedure, and sets the entries of $A.opcounts$ to exactly the handlers that are executed during online execution. Last, S' will contain exactly one entry for each handler operation/state operation/annotated operation that it executes. Because the honest server faithfully reports *A.opcounts*, S' will contain exactly one (rid, hid, i) for each $i < A.opcounts$. So will G (line 42 of Figure 14). S' contains no other entries other than the ones above and G contains no other nodes other than the ones above. Thus, S' is a permutation of the nodes of G, as required.

Moreover, S' respects program order (Definition 10): The server faithfully executes the given program and collects the advice. This means that the relevant order of events within a handler implied by the opnum field of the corresponding operations in the logs, and the order of the response event relative to the other events of the handler that issues the response implied by the contents *A.responseEmittedBy* reflect what happened online. As a result, from the logic of AddProgramEdges of Figure 14 and AddBoundaryEdges of Figure 15, the existence of a program edge or boundary edge $\langle n_1, n_2 \rangle$ in G implies that n_1 happens before n_2 during online execution. Thus, n_1 also appears before n_2 in S', by construction of S'.

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

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

in G, then the emit event $e = (rid, parent\_hid, i)$ activates handler (rid, hid) during online execution (because during a faithful execution a handler cannot start running until after the event that activates it is emitted). The activation edge is added to G at line 28 of Figure 16 only if *hid.functionID* is registered for the event e according to *GlobalHandlers* or according to *Registered*. We will show that in both cases, *hid.functionID* is registered for the event e during online execution and, thus, e activates handler (rid, hid). In the former case, *hid.functionID* is registered for e at the end of the initialization procedure at the verifier. Because the initialization procedure is deterministic, *hid.functionID* is registered for e at the end of the initialization procedure at the online server and, because requests don't modify global handlers, it is still registered when e is emitted, as required. In the latter case, *hid.functionID* is registered for e according to *Registered* only if there exists a register operation prior to e in HL_{rid}. Because the server executes the advice collection procedure faithfully, the order of operations in HL_{rid} reflects the order in which they are executed at the online server. This implies that *hid.functionID* is registered for e during online execution. □

Sub-lemma 2.2. Preprocess passes.

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

- Line 19 of Figure 14: Passes because the honest server always sends back a response for each request it receives.
- Line 37 of Figure 14: When the server is honest, it does not execute nor collect advice for any requests that are not in Tr.
- Lines 14 and 16 of Figure 15: The honest server executes exactly the requests in Tr and sends back responses for exactly those requests. Moreover, it faithfully executes the advice collection procedure setting the contents of *A.responseEmittedBy*$[rid]$ for each *rid* that appears in the trace to a tuple: (hid_r, opnum_r) s.t. $0 \leq \text{opnum}_r \leq A.opcounts[(rid, hid)]$. Consequently, the check of line 14 passes. Moreover, notice that because of the logic of AddProgramEdges and the fact that the honest server correctly sets the *A.opcounts*, $(rid, hid_r, \text{opnum}_r)$ is added to G before the check of line 16 which implies that the check passes.
- Line 6 of Figure 16: Because the server is well-behaved, it never includes a handler operations log in A for a request that is not in Tr.
- Line 25 of Figure 16: As argued in the proof of lemma 2.1, if a *functionID* is registered for an event e when e is emitted according to *Registered* or *GlobalHandlers* during AddHandlerRelatedEdges, this *functionID* is registered for e during online execution. This implies that all handler ids for which line 25 is executed are handler ids that are actually activated by this operation during online execution. Moreover, the server, being well-behaved, has these ids as keys in *A.opcounts*. Thus, the check passes.
- Invocation of CheckOpIsValid in Line 10 of Figure 16: When the server is honest, it correctly sets *opcounts* for each request in Tr and the contents of the logs so that each operation appears exactly once in the logs. Under these conditions the checks pass.
- Line 48 of Figure 16: When the server is honest, each `GET`(key) operation reads the contents of a `PUT`$(key, \cdot)$ operation. Moreover, the honest server correctly logs state operations in $A.TXL$. Under these conditions, the check passes.
- Line 51 of Figure 16: When the server is well-behaved, the execution at the database is internally consistent (Section D) meaning that if a transaction modifies a *key* and later reads it, it reads its latest modification. Moreover, the well-behaved server correctly sets the opcontents field of each `GET` operation to the position of its dictating write in $A.TXL$s. This implies that for each `GET` operation *op* that appears in some $A.TXL_t$ after a `PUT` operation op' with $op'.key = op.key$, *op*.opcontents corresponds to the last `PUT` operation to *op.key* that precedes *op* in $A.TXL_t$. Meanwhile, from the logic of AddExternalStateEdges, when a `GET` operation $op \in A.TXL_t$ is processed, $op.key \notin MyWrites$ iff there are no `PUT`

operations to $op.key$ prior to op in $A.TXL_t$. Otherwise, $MyWrites[op.key]$ is the last PUT operation to $op.key$ that precedes op in $A.TXL_t$. Thus, either $op.key \notin MyWrites$ or $MyWrites[op.key] = op$.opcontents. So, the check passes.

- Lines 23 and 27 of Figure 17: We show that both checks pass by showing that the entries (rid, tid, i) of $A.writeOrder$ are exactly the set of (rid, hid, i) s.t. $\exists key : lastModification[rid, tid, key] = i$. First, notice that because the server is well-behaved the entries (rid, tid, i) of $A.writeOrder$ correspond to the PUT operations that the server applied to the external state. These are exactly the last modifications of committed transactions: that is, the PUT operations op s.t. op belongs to a committed transaction and op is the last operation of the transaction that modifies a key. Moreover, the honest server correctly sets the entries of $A.TXL$s. Thus, the entries (rid, tid, i) of $A.writeOrder$ are exactly the operations $op = A.TXL_{(rid,tid)}[i]$ s.t. (1) the last operation of (rid, tid) is `tx_commit`, and (2) there exists no $j > i$ s.t. $A.TXL_{(rid,tid)}[j]$ is a PUT on $op.key$. From the logic of AddExternalStateEdges, these are exactly the (rid, hid, i) s.t. $\exists key : lastModification[rid, tid, key] = i$, as required.
- Line 36 of Figure 17: First, observe that from the logic of AddExternalStateEdges, $ReadMap$ maps each PUT operation that appears in the logs to the set of GET operations that read from it according to the advice. Thus, to show that this check passes for all GET operations in the range of $ReadMap$, we show that for each GET operation op that appears in some $A.TXL_t$ either $t \notin Committed$ or op.opcontents $\in A.writeOrder$. Observe that this line is executed only when the purported isolation is READ COMMITTED or SERIALIZABILITY. Consider the history of execution (Section D) at the honest server. The history is consistent with the isolation level and, thus, does not exhibit phenomena G1a and G1b. Consequently, during online execution, GET operations of committed transactions only read from operations that correspond to last modifications of committed transactions. These latter operations are exactly the ones that the honest server places in the $A.writeOrder$. Thus, GET operations of committed transactions only read from operations that are in the $A.writeOrder$. Because the server correctly logs state operations in $A.TXL$s, we deduce that for each GET operation op that appears in some $A.TXL_t$ either the last operation in $A.TXL_t$ is `tx_abort` and, thus, $t \notin Committed$ or op.opcontents $\in A.writeOrder$, as required.
- Line 11 of Figure 17: We need to show that if the server is honest, then the graph DG when this line is executed is acyclic. This line is executed only when the purported isolation level is READ UNCOMMITTED. Consider the history of execution H at the honest server (Section D). Because the server is well-behaved, H exhibits READ UNCOMMITTED which implies that H does not exhibit phenomenon G0: $DSG(H)$ contains no cycles consisting of write depend edges. We show that DG is acyclic by showing that DG is the subgraph of $DSG(H)$ that contains only write depend edges. First, DG and $DSG(H)$ have the same nodes: DG has a node for each transaction that commits during online execution whereas $DSG(H)$ has a node for each transaction in $Committed$. Because the honest server collects advice for each transaction that it executes and the last operation of each committed transaction t in $A.TXL_t$ is a `tx_commit` operation, AddExternalStateEdges adds exactly the transactions that commit during online execution to $Committed$ (line 35 of Figure 16). Thus, DG and $DSG(H)$ have the same nodes, as required. Now we argue that the edges of DG which are the write dependency edges (added at line 47 of Figure 17) are exactly the write depend edges of $DSG(H)$: Observe that the write depend edges of $DSG(H)$ are the edges$\langle t_1, t_2 \rangle$ s.t. t_1 writes a key and t_2 writes the next version of the key according to H (Section D). That is, $DSG(H)$ has a write depend edge $\langle t_1, t_2 \rangle$ iff there exist operations $op_1 = (t_1, i)$ and $op_2 = (t_2, j)$ s.t.
  1. op_1 appears before op_2 in the version order of H,
  2. $op_1.key == op_2.key$, and
  3. for each operation op' that appears between op_1 and op_2 in the version order of H, the key that op' writes is not key.

  Moreover, a well-behaved server sets $A.writeOrder$ to the version order of H, and correctly logs state operations in $A.TXL$s. Thus, the write depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist indexes i and j s.t.
  1. (t_1, i) appears before (t_2, j) in $A.writeOrder$,
  2. $A.TXL_{t_1}[i].key = A.TXL_{t_2}[j].key$, and
  3. for each operation (t, k) that appears between (t_1, i) and (t_2, j) in $A.writeOrder$, $A.TXL_t[k].key \neq A.TXL_{t_1}[i].key$.

  Meanwhile, from the logic of ExtractWriteOrderPerKey, t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$ iff they meet the above conditions. Thus, t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$ iff $\langle t_1, t_2 \rangle$ is a write depend edge of $DSG(H)$. Moreover, from the logic of AddWriteDependEdges the edges of DG are exactly the edges $\langle t_1, t_2 \rangle$ s.t. t_1 and t_2 are consecutive in some $writeOrderPerKey[key]$. Thus, the edges of DG are exactly the write depend edges of $DSG(H)$, as required.
- Line 15 of Figure 17: As in the previous case, we need to show that the graph DG when this line is executed is acyclic. This line is executed only when the purported isolation level is READ COMMITTED. Consider the history of execution H at the honest server (Section D). Because the server is well-behaved, H exhibits READ COMMITTED which implies that H does not exhibit phenomenon G1c: $DSG(H)$ contains no cycles consisting of write depend edges and read depend

edges. We show that DG is acyclic by showing that DG is the subgraph of $DSG(H)$ that contains the write depend and read depend edges of $DSG(H)$. Specifically, we show:

1. that DG and $DSG(H)$ have the same nodes,
2. that the write dependency edges of DG (added at line 47 of Figure 17) are exactly the write depend edges of $DSG(H)$, and
3. that the read dependency edges of DG (added at line 40 of Figure 17) are exactly the read depend edges of $DSG(H)$.

We show 1 and 2 as above (in the proof that the check at line 11 of Figure 17 passes). Now we show 3: First, observe that the read depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ s.t. some operation of t_2 reads a value written by t_1. Moreover, because the server is well-behaved, the history H does not exhibit phenomenon G1b: as explained above (in the proof that the checks at lines 23 and 27 of Figure 17 pass), this implies that all GET operations of committed transactions read from operations that are in H's version order. Thus, the read depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ for which there exist an operation op_1 that t_1 issues and an operation op_2 that t_2 issues s.t.

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

Because the server is well-behaved, it correctly logs all state operations in the $A.TXL$s and sets $A.writeOrder$ to H's version order. Moreover, as argued above (in the proof that the check at line 11 of Figure 17 passes), when the server is honest, *Committed* contains exactly the transactions that commit during online execution. Thus, the read depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist operations (t_1, i) and (t_2, j) s.t.:

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

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

- Line 20 of Figure 17: As in the previous case, we need to show that the graph DG when this line is executed is acyclic. This line is executed only when the purported isolation level is SERIALIZABILITY. Consider the history of execution H at the honest server (Section D). Because the server is well-behaved, H exhibits SERIALIZABILITY which implies that H does not exhibit phenomena G1c and G2 and, thus, $DSG(H)$ contains no cycles. We show that DG is acyclic by showing that DG is exactly $DSG(H)$. Specifically we show 1, 2, and 3 as in the previous case and, additionally, we show that the anti dependency edges of DG (added at line 56 of Figure 17) are exactly the anti depend edges of $DSG(H)$: The anti depend edges of $DSG(H)$ are the edges $\langle t_1, t_2 \rangle$ s.t. t_1 reads some version of a *key* and t_2 writes the next version of *key* according to the H's version order. Thus, the anti depend edges of $DSG(H)$ are exactly the edges $\langle t_1, t_2 \rangle$ for which there exist a transaction t_3, and operations op_1, op_2, and op_3 issued by t_1, t_2, and t_3 respectively:

– op_3 appears before op_2 in the version order of H,
– $op_3.key = op_2.key$,
– for each operation op' that appears between op_3 and op_2 in the version order of H, the key that op' writes is not $op_3.key$,
– op_1 reads the value written by op_3,
– $t_1 \neq t_2$, and
– t_1 commits

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

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

– $t_1 \in \mathit{Committed}$
From the logic of AddExternalStateEdges, ExtractWriteOrderPerKey and AddAntiDependencyEdges, these are exactly the anti dependency edges of DG.

□

Sub-lemma 2.3. The invocation of OOOExec(S'):

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

Proof. Proof outline: Induct on S':

Base case: The first operation in S' has no ancestors in G'. It can only be an operation $(rid, 0)$ for some $rid \in$ Tr. OOOExec handles this operation by allocating structures for running and reading in the inputs. This is the same behavior as online execution. Moreover, OOOExec finds all request handlers for *rid*, computes their handler ids and checks that for each handler id there is an entry in *opcounts* (Line 17). This check will pass because the honest server sees the same request handlers for the request during online execution, computes their handler ids in the same way as the verifier (the computation is deterministic), and has entries for each of them in *opcounts*.

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

- Case I: $op = (rid, 0)$: Same reasoning as in the base case.
- Case II: $op = (rid, hid, 0)$ where handler (rid, hid) is a request handler (that is, $hid.parent\_hid = null$): Because S' obeys program order, this operation appears after $(rid, 0)$ and before (rid, hid, ∞). This means that, because this handler is a request handler, when OOOExec executes this operation, *hid* has already been added to $active[rid]$ in line 16, has not been removed yet, and $active[rid][hid]$ has been set to the request inputs. Thus, the check of line 25 passes, OOOExec sets the handler's inputs to the request inputs and allocates structures for running the handler. This is the same behavior as online execution.
- Case III: $op = (rid, hid, i)$ where $i = 1$ and handler (rid, hid) is a request handler (that is, $hid.parent\_hid = null$) By the induction hypothesis and the fact that S' obeys program order, OOOExec and online execution had the same program state at $(rid, hid, 0)$. Because the server is well-behaved, both online execution and OOOExec will take the same next step in terms of handler op, handler exit event, external state op, or annotated operation. Since the server is well-behaved and $opcounts[rid][hid] > 1$, the next operation is a handler op, an external state op, or an annotated operation in both executions.
  - Handler Op: Similar arguments to the ones in case III of Sub-lemma 7b of Orochi [87]. The determinism of passing from $(rid, hid, 0)$ to $(rid, hid, 1)$ and the induction hypothesis imply that the program state of online execution and the program state of OOOExec right before executing the handler operation are the same. Being well-behaved, the server recorded this operation correctly in HL_{rid} and this is the operation that the verifier checks in CheckHandlerOp. Moreover, the contents of the log entry (optype and *info*) are the ones produced during online execution and consequently the ones produced during by OOOExec. Under these conditions CheckHandlerOp passes. Moreover, if the operation is an emit operation, the handler ids in *activatedHandlers* are exactly the ones that this operation activated and their inputs in *active* will be set to the inputs during online execution.
  - External State Op: First, observe that the operation has the same *tid* and txnum under both executions: If optype is `tx_start` then both online execution and OOOExec compute the same *tid* as (hid, opnum) and set txnum $= 0$, as required. Otherwise, because of the induction hypothesis, both executions have assigned the same *tid* to this transaction. Meanwhile, the operations of a transaction are not concurrent meaning that the order of operations within a transaction is consistent with program order and activation order. Because both online execution and OOOExec(S') follow program order and activation order, the transaction *tid* issues the same number of operations prior to *op* under both executions. Thus, txnum is the same under both executions as required. This implies that during OOOExec(S'), the operation is checked against the entry in the logs that the honest server records for this operation during online execution. Moreover, the determinism of passing from $(rid, hid, 0)$ to $(rid, hid, 1)$ and the induction hypothesis imply that the program state of online execution and the program state of OOOExec right before executing the state operation are the same. This implies that the parameters of the operation $(\text{optype}, \text{opcontents}, key)$ are the same under both executions except in the case where optype $=$ `tx_commit`: in this case the recorded operation in the logs may be `tx_abort` because during online execution, the transaction could not successfully commit. Meanwhile, because the server is well behaved, it correctly logs the operation in $A.TXL_{(rid,tid)}$. Thus, all checks of CheckStateOp pass. Moreover, the well behaved server correctly sets the opcontents field of a `GET` operation to (rid_w, tid_w, i_w) s.t. $A.TXL_{(rid_w,tid_w)}[i_w]$

is the dictating PUT operation. This implies that the value that OOOExec reads is the one written by the dictating PUT, which is the value read at online execution. Thus, the two executions have the same program state after executing the state operation.

- Annotated Operation: Both online execution and OOOExec execute the operation and call the annotation with arguments (rid, hid, i) for the same variable v. We will argue that the claim holds after executing the annotation for any handler hid and any i.
  - Initialization: In this case both executions execute the operation and then execute the annotation, which performs no checks. Thus, the two executions result in the same program state.
  - Write operation: Both executions execute the operation, assigning the same value to the v, and then execute the annotation. If during the execution of the annotation by OOOExec the operation is found in the logs, then the server, being well-behaved, has correctly recorded the operation in $v.\text{log}$. As a result, all checks of OOOExec pass. Otherwise, the verifier does no checks.
  - Read operation: We need to show that the value that OnRead returns in OOOExec is the value of v when the annotation is executed by online execution, which is the value written by its dictating write.
    If during the execution of the annotation the read operation is in $v.\text{log}$ then the online server, being well-behaved, has correctly logged both the read operation and its dictating write. As a result, all checks pass and OnRead sets the value of v to its value during online execution.
    We will now argue that OnRead returns the value of v at online execution when the operation is not in $v.\text{log}$. If the operation is not in $v.\text{log}$, then this is because when the operation is executed by the honest server, op reads the value written by some operation op' that is not R-concurrent with op (Definition 8). Thus, $op' <_R op$. Meanwhile, because OOOExec(S') follows program order and activation order, it executes op' prior to executing op. Furthermore, from the induction hypothesis, OOOExec(S') and online execution have the same program state when they execute op', meaning that the parameters of op' that OOOExec(S') records in $v.var\_dict$ (at line 2 of Figure 21) are exactly the parameters of op' during online execution. Thus, when OOOExec(S') executes op, there exists an entry in $v.var\_dict$ that maps op' to the value written during online execution. If op' is not the nearest R-preceding write of op in $v.var\_dict$, then there is a later ancestor, call it op'', such that op' was re-executed before op'', which was re-executed before op. Since op' and op'' R-precede op and since each handler has only one parent, we must have $op' <_R op'' <_R op$. But $<_R$ never inverts online execution, so op'' was also executed in between op' and op online, in which case op could not have observed op' without violating causality. Thus, there is no such op''. FindNearestRPrecedingWrite therefore returns op' and reads the value written by op' during online execution, as required.

- Case IV: $op = (rid, hid, i)$ where $i \in [2, A.opcounts[(rid, hid)]]$ and handler hid is a request handler (that is, $hid.parent\_hid = null$) Same arguments as in case III.
- Case V: $op = (rid, hid, \infty)$ where hid is a request handler (that is, $hid.parent\_hid = null$) An argument similar to one made elsewhere (Orochi [87], Sub-lemma 7b, Case II) establishes that the next operation is handler exit both in online execution and in OOOExec. OOOExec handles handler exit events in the same way as online execution.
- Case VI: $op = (rid, hid, 0)$ where hid is not a request handler. We need to show that the check of line 25 of Figure 22 accepts and that the inputs on which the handler is executed by OOOExec are the ones of online execution. Because (rid, hid) is not a request handler it is activated by some emit operation op' during online execution and, since the server is well-behaved, op' is executed before op during OOOExec. From the induction hypothesis, the program state of OOOAudit when it executes op' is the one of online execution. This implies that if line 49 of Figure 22 is executed for hid, $active[rid][hid]$ is set to the handler's inputs according to online execution. In order for this line to be executed for hid, it must be that $hid \in activatedHandlers[op']$. This can only happen if when op' is parsed during AddHandlerRelatedEdges $hid.functionID$ is registered for $op'.eventName$ according to $GlobalHandlers$ or $Registered$. We will now argue that this is indeed the case. Because op' activates (rid, hid) during online execution, $hid.functionID$ is registered for $op'.eventName$ when op' is executed at the online server. $hid.functionID$ is either a global handler, or there exists some operation op'' executed by the request rid that registers $op'.eventName$ for $hid.functionID$ during online execution. In the former case per the determinism of the initialization procedure $(op'.eventName, hid.functionID) \in GlobalHandlers$. In the latter case, because the server is well-behaved, op'' appears before op' in HL_{rid} and $(op'.eventName, hid.functionID) \in Registered$ when op is examined during AddHandlerRelatedEdges.
- Case VII: $op = (rid, hid, i)$ where hid is not a request handler and $i \in [1, A.opcounts[(rid, hid)]]$. We can show this using the same arguments as in cases III and IV above.
- Case VIII: $op = (rid, hid, \infty)$ where hid is not a request handler. We can show this using the same arguments as in case V above.

- Case IX: $op = (rid, \infty)$. Because S' is well-formed,

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

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

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

Last, as argued in case IX, all responses will match the ones of online execution and OOOExec accepts at line 52 of Figure 22. □

Sub-lemma 2.4. Postprocess passes.

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

Consider the edges that are added to G during Preprocess:

- Procedure SplitNodes: An edge $\langle (rid_1, \infty), (rid_2, 0) \rangle$ is added to the graph only if the response for rid_1 appears in the trace before the request rid_2 is issued. This implies that the response for rid_1 was issued by the server before the request rid_2 reached the server. Thus, (rid_1, ∞) happened before $(rid_2, 0)$, as required.
- Line 7 of Figure 15: An edge $\langle (rid, 0), (rid, hid, 0) \rangle$ is added because hid is a request handler for rid. All handlers for a request start executing after the request reaches the server, so the event $(rid, 0)$ happened before the event $(rid, hid, 0)$ during online execution
- Lines 43 and 44 of Figure 14: An edge $\langle n_1, n_2 \rangle$ is added to the graph because according to the advice, n_1 preceded n_2 during the execution of a handler. Because the server is honest, n_1 indeed preceded n_2 during online execution.
- Lines 17, 20 and 22 of Figure 15: For some rid, let hid be the handler that issued the response according to $responseEmittedBy$ and n be the last operation of hid prior to issuing the response according to $responseEmittedBy$. Because the honest server correctly sets the contents of $responseEmittedBy$ and send_response is a synchronous operation at the server, these lines add edges to indicate that a response is issued after n and before the next event of hid during online execution
- Line 15 of Figure 16: such edges are added because according to the advice a handler operation preceded another handler operation. Since the server is honest, this precedence held during online execution as well.
- Line 28 of Figure 16: such an edge $\langle (rid, parent\_hid, \text{opnum}), (rid, hid, 0) \rangle$ is added only if according to the advice the emit operation $(rid, parent\_hid, \text{opnum})$ activates handler (rid, hid). Because the server is well-behaved the emit operation $(rid, parent\_hid, \text{opnum})$ activates handler (rid, hid) during online execution, and, because a handler does not start running until the event that activates it is emitted, the handler start operation $(rid, hid, 0)$ happens after $(rid, parent\_hid, \text{opnum})$ during online execution as required.
- Line 46 of Figure 16: Such an edge $\langle n_1, n_2 \rangle$ is added only if the operation n_2 reads a value written by operation n_1 according to the advice. Because the server is well-behaved, n_2 truly reads from n_1 during online execution and, because an operation cannot read from the future, the operation n_1 executes before operation n_2 during online execution. Meanwhile, during online execution, the server collects advice for `PUT` operations before issuing them to the database and it collects advice for `GET` operations after their execution at the database completes. Thus, event n_1 precedes event n_2 during online execution, as required.

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

First, we argue that if at the beginning of Postprocess for two operations n_1, n_2, $v.write\_observer\{n_1\} = n_2$ or $n_2 \in v.read\_observers\{n_1\}$ for some variable v, then n_1 happens before n_2 during online execution. We will only argue this in the case where $v.write\_observer\{n_1\} = n_2$ because the $read\_observers$ case is similar. $v.write\_observer\{n_1\}$ can be set to n_2 at only two locations during OOOExec(S'). First, in line 11 of Figure 21 which is executed if n_2 is in the logs and the server has recorded n_1 as the previous write. Because the server is well-behaved, n_1 happens before n_2 during online execution. Second, in line 16 of Figure 21, which is executed if n_1 is identified as the nearest write by some ancestor of n_2. In this case n_1 appears before n_2 in S'. Because n_1 and n_2 operate on the same variable and we assume that variables are serializable, n_1 and

n_2 are ordered during online execution and, because S' follows the order of online execution on non-concurrent operations, n_1 happens before n_2 during online execution.

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

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

In case (i), we claim that $n_1 \in v.read\_observers\{n_2\}$. There are two sub-cases: either n_2 is an ancestor of n_1 during online execution, or n_2 and n_1 are concurrent during online execution. For the first sub-case: because OOOExec(S') follows activation order, n_2 is an ancestor of n_1 during OOOExec and n_1 is added to $v.read\_observers\{n_2\}$ at line 34 of Figure 20. For the second sub-case: because a faithful server logs concurrent accesses for which at least one is a write, OOOExec adds n_1 to $v.read\_observers\{n_2\}$ at line 26 of Figure 20. Combining $n_1 \in v.read\_observers\{n_2\}$ with $n_1 \in v.read\_observers\{n\}$ (from the fact of an rw edge $\langle n_1, n_2\rangle$), we have a contradiction, as an operation is only added to one *v.read_observers*.

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

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

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

□

Lemma 3 (Equivalence of OOOAudit and Audit). If the server executes the given program and advice collection procedure, producing trace Tr and advice A, then there exists a well-formed op schedule S' (with respect to Tr and A) such that Audit(Tr, A) and OOOAudit(Tr, A, S') are equivalent.

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

```
1. For each request id $r$ in $C$, append $(r, 0)$ to $S'$
2. Pick some request id $rid^*$ in the group $C$
3. Initialize a set $R$ that contains tuples (event name, function ID).
4. Initialize $active$ to the ids of the request handlers of $rid^*$. .
5. $I \leftarrow active$.
6. While $active \neq null$:
  a. If $I \neq null$:
    i. Pick some $hid$ from $I$ and remove this $hid$ from I.
    ii. If $hid \notin active$, go to step 6.
    Otherwise, pick some $hid$ from $active$.
  b. For opnum = $0 \ldots A.opcounts[(rid^*, hid)]$:
    i. For all requests $r$ in the group, append $(r, hid, \text{opnum})$ to $S'$.
    ii. If $A.responseEmittedBy[rid^*] = (hid, \text{opnum})$, then for all requests $r$ in $C$, append $(r, \infty)$ to $S'$.
    iii. $(t, rid^*_c, i) \leftarrow \text{OpMap}[(rid^*, hid, \text{opnum})]$.
    iv. if $t$ = "handler_log" and $A.HL_{rid^*}[i]$.optype = register, for all $eventName \in A.HL_{rid^*}[i].eventNames$
          $R.add(eventName, A.HL_{rid^*}[i].functionID)$
    v. if $t$ = "handler_log" and $A.HL_{rid^*}[i]$.optype = unregister,
          $R.remove(A.HL_{rid^*}[i].eventName, A.HL_{rid^*}[i].functionID)$
    vi. If $(rid^*, hid, \text{opnum})$ is in $activatedHandlers$,
```

A. add all hid' in $activatedHandlers[(rid^*, hid, \text{opnum})]$ to $active$.
B. $(t, rid_c^*, i) \leftarrow \text{OpMap}[(rid^*, hid, \text{opnum})]$.
C. $eventName \leftarrow A.HL_{rid^*}[i].eventName$.
D. For all f s.t. $(eventName, f) \in R \cup GlobalHandlers$, add (f, hid, opnum) to I

c. For all requests r in the group, append (r, hid, ∞) to S'.
d. Delete hid from $active$

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

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

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

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

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

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

and

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

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

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

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

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

and that the forward direction implies that

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

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

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

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

Now we show that relation (3) holds. The forward direction holds because an hid is added to $active$ if it is a request handler or it is in $activatedHandlers[rid^*, hid', i]$ for some hid'. In the former case there is an entry in $A.opcounts[rid^*]$ because the server is well-behaved and in the latter there is an entry in $A.opcounts[rid^*]$ because all entries in $activatedHandlers$ are in $A.opcounts$ as argued in the proof of lemma 2.2. For the backwards direction of (3), notice that if hid is in $A.opcounts[rid^*]$, then because the server faithfully sets the contents of $A.opcounts[rid^*]$, hid is activated for rid^* during online execution. If *hid* is a request handler then it is added to *active* at the beginning of the process. Otherwise let $op_1, \ldots, op_n$ be the sequence of emit operations that led to the activation of *hid* during online execution. Because the server correctly logs the handler operations to reflect what happened during online execution, $activatedHandlers[op_i]$ for each i will contain the handler that emits op_{i+1}. Moreover, op_1 is issued by a request handler. Under these conditions, the above process will add all handlers that execute $op_1, \ldots, op_n$ to *active*, will examine each operation, and when it encounters op_n it will add *hid* to *active*, as required.

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

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

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

1. ReExec checks that the number of operations that each handler issued matches the purported number of operations in the advice. OOOExec has no such explicit check but it does have an (rid, hid, ∞) case. An argument similar to the one in case (i) of Theorem 10 of Orochi [87] implies that the difference is superficial.
2. OOOExec executes the requests in a Round-Robin fashion whereas ReExec does SIMD-style execution. An argument similar to the one in case (ii) of Theorem 10 of Orochi [87] implies that the difference is superficial.
3. ReExec checks that the execution of requests does not diverge inside each handler. An argument similar to the one in case (iii) of Theorem 10 of Orochi [87] implies that the difference is superficial.
4. When OOOExec starts executing a handler, it checks that it is in *active*. ReExec does not do this check. The difference is superficial because due to the fact that the server is well-behaved, the check always passes during OOOExec.
5. ReExec checks that when a group makes an emit operation, all requests in the group activate the same handlers. This difference does not affect the execution because when the server is honest all requests in the same group activate the same handlers from corresponding emit operations which means that requests in the same group have the same entries in *activatedHandlers* and, thus, ReExec's check passes.
6. ReExec keeps track of the number of ops that a handler has executed so far in *idx*. OOOExec uses the *i* field in the op schedule entry as the number of ops that the handler has issued so far. The difference is superficial: $i = idx$ at all times because *idx* and *i* are both the running counter of operations that the handler has executed so far.
7. When a group sends back a response, ReExec checks that the contents of *A.responseEmittedBy* match re-execution. In OOOExec there is no such check, but there is a (rid, ∞) case. This difference is superficial: both executions reject if the contents of *A.responseEmittedBy* do not match the ones produced during re-execution and reject otherwise.
8. ReExec lets the runtime pick the next handler to execute at line 16 of Figure 18, whereas OOOExec picks itself the next handler to execute from S'. Observe that from the logic of ReExec and the way S is constructed, in both cases, the handler that is executed is a handler whose id is in *active*. Moreover, the two executions pick the same handler to execute under the condition that the activated handlers under ReExec are exactly the handlers that are in I when the operation is added to S' (during the construction of S'). We now show that this condition holds: Initially I contains exactly the request handlers of the request which are exactly the handlers that the request activates under ReExec. Now, we show that the handlers that each emit operation activates during ReExec are exactly the handlers that are added in I: The handlers that are activated by each emit operation under ReExec are exactly the handlers registered for the event by the request and the global handlers registered for the event. Meanwhile, from the logic of the algorithm that we use to construct S', and the semantics of handler operations, when each operation is executed by ReExec, the set of handlers that are registered by the request contains exactly the entries of R when the operation is added to S'. Thus, the handlers activated by each emit operation are the ones registered for the emitted event in *GlobalHandlers* and the ones in R when the operation is added to S'. These are exactly the handlers that are added in I when the emit operation is added to S', as required.

□

Composing Lemmas 2 and 3, we have proved:

```
//Global Variables are the ones in Figure 14

procedure ActualHandlerOps(op schedule $S$)
    Preprocess()
    return ActualHandlerOpsExec($S$)

procedure ActualHandlerOpsExec(op schedule $S$)
    $\mathrm{Tr}' \leftarrow []$
    for each $op$ in $S$ do
        if $op = (rid, 0)$ then
            Read inputs $in$ of the request from Tr
            Allocate program structures
            $\mathrm{Tr}'.append((\mathrm{REQ}, rid, \mathrm{inputs}))$
            Find the $functionID$s of the request handlers
            for all $functionID$ in $functionID$s do
                Let $hid \leftarrow (functionID, null, 0)$
                Name the instance of the handler $hid$
        else if $op = (rid, \infty)$ then
            Let $hid \leftarrow A.responseEmittedBy[rid].hid$
            Run the handler $(rid, hid)$ up to and including the next event
            if it is not a send response operation then REJECT
            $\mathrm{Tr}'.append((\mathrm{RESP}, rid, \mathrm{outputs}))$
        else if $op = (rid, hid, i)$ then
            if $i = 0$ then
                if ($hid$ is not an activated handler) then REJECT
                Read in the handler's inputs and allocate structures for running the handler
            else if $i = \infty$ then
                Run the handler $(rid, hid)$ until the next event
                if it is not a handler exit operation then REJECT
            else
                Run the handler $(rid, hid)$ until the next event
                if the next event is an external state operation then
                    optype $\leftarrow$ the type of state operation
                    opcontents, $tid$, txnum $\leftarrow$ parameters from execution
                    $s \leftarrow$ CheckStateOp($rid, hid, i$, optype, $tid$, txnum, $key$, opcontents)
                    if optype = GET then
                        state op result $\leftarrow$ s
                else if the next event is an annotated operation then
                    if it is a write or initialization then
                        Execute the operation and skip the annotation
                    else
                        Return the current value of the variable
                else if the next event is a handler operation then
                    Execute the handler operation
                    if the operation is an emit operation then
                        for all functions that the operation activates do
                            $hid' \leftarrow (functionID, hid, i)$
                            Name the instance of the handler that is activated $hid'$

    return $\mathrm{Tr}'$
```

Figure 23. Pseudocode for ActualHandlerOps

Theorem 1 (Audit Completeness). If the executor executes the given program (under the concurrency model given in Section 3 of the paper) and the given advice collection procedure, producing trace Tr and advice A, then Audit(Tr, A) accepts. □

C.3.2 Soundness

In the following we assume no external state operations. To show that Definition 6 is satisfied, we will show that whenever the verifier accepts an input trace Tr and advice A, there exists a well formed op schedule (with respect to Tr and A) that causes OOOAudit to accept (Lemma 5) which in turn implies the existence of a request schedule RS s.t. $\mathrm{Tr} \in O_{RS}$ (Lemma 4).

Lemma 4 (OOOAudit Soundness)**.** Given trace Tr and advice A, if there exists a well-formed op schedule S for which OOOAudit(Tr, A, S) accepts then there exists a request schedule RS s.t. $\text{Tr} \in O_{RS}$.

Proof. If OOOAudit(Tr, A, S) accepts, then there are no cycles in graph G. We consider an op schedule S' that is a topological sort of G in which the order of $(rid, 0)$ and (rid, ∞) events matches Tr. We show that such an op schedule exists (Lemma 4.1). S' is well-formed (which follows from the remark after Definition 10). Thus, by Lemma 1, OOOAudit(Tr, A, S') accepts. Then, we define an execution ActualHandlerOps as in Figure 23. ActualHandlerOps is the same as OOOExec of Figure 22 but:

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

Then, we prove that if OOOAudit(Tr, A, S') accepts, then ActualHandlerOps(Tr, A, S') outputs Tr (Lemma 4.2).

Subsequently, we define an execution Actual as in Figure 24 that is the same as ActualHandlerOps of Figure 23 except that it executes external state operations against a database instead of simulating them by reading from the $A.TXL$s. Because the execution at the database is non deterministic, each `GET` operation that Actual issues may return more than one outputs meaning that Actual has many possible output traces.

Then, we show that if ActualHandlerOps(Tr, A, S') outputs Tr then Tr is a possible output of Actual(Tr, A, S') (Lemma 4.3).

Last, we show that if one of the possible outputs of Actual(Tr, A, S') is Tr, then $\text{Tr} \in O_{RS}$, where RS is the request schedule derived from S' by discarding the handler id and opnum components (Lemma 4.4).

Sub-lemma 4.1. If G is acyclic, then there exists a topological sort S' of G in which the order of $(rid, 0)$ and (rid, ∞) events matches Tr.

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

We create an ordered list S' as in Figure 25.

If the procedure ConstructS of Figure 25 does not reject, the constructed S' is a topological sort of G with the required property: It is a topological sort because a node v is not added to S' until after all nodes that have a path to v have been removed from G and added to S'. Moreover, from construction, nodes that correspond to request/response events are always added in the order that they appear in Tr.

We now prove that ConstructS of Figure 25 does not reject. Assume that it does reject. This can happen only if all nodes in *frontier* correspond to request/response events (that is, items in Tr) and none of them is the node u that corresponds to $\text{Tr}[i]$. *Claim:* There exists a request or response node v such that v appears in Tr after u yet v has a directed path in to u in G. We now justify this Claim. Denote as G_i the graph G at line 3 of Figure 25 is executed for the i-th iteration. Because u is in G_i but not in *frontier*, u has in-degree larger than 0 in G_i. Because G_i is acyclic (being a subgraph of G), there exists a path in G_i, and hence also in G, to node u from some node v that has in-degree 0 in G_i. By inspection of the algorithm, v is in *frontier*. Because all nodes in *frontier* are request or response nodes, there exists a j s.t. $\text{Tr}[j]$ corresponds to v. Meanwhile, for $j < i$, all nodes that correspond to $\text{Tr}[j]$ are not in G_i (again by inspection of the algorithm). Thus, $j > i$, which implies that the node v appears in Tr after u.

In the following, we use $v_1 \overset{G}{\rightsquigarrow} v_2$ to denote that there is a directed path from v_1 to v_2 in G and $rid_1 <_{\text{Tr}} rid_2$ to denote that (RESP, rid_1, $\cdot$) appears before (REQ, rid_2, $\cdot$) in the trace Tr. Similar arguments as in the proof of Lemma 2 of Orochi [87] imply that

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

Now we use this observation to analyze cases:

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

```
//Global Variables are the ones in Figure 14

procedure ACTUAL(op schedule $S$)
    Preprocess()
    return ActualExec$S$)

procedure ACTUALEXEC(op schedule $S$)
    $\mathrm{Tr}' \leftarrow []$
    for each $op$ in $S$ do
        if $op = (rid, 0)$ then
            Read inputs $in$ of the request from Tr
            Allocate program structures
            $\mathrm{Tr}'.append((\mathrm{REQ}, rid, \mathrm{inputs}))$
            Find the $functionID$s of the request handlers
            for all $functionID$ in $functionID$s do
                Let $hid \leftarrow (functionID, null, 0)$
                Name the instance of the handler $hid$
        else if $op = (rid, \infty)$ then
            Let $hid \leftarrow A.responseEmittedBy[rid].hid$
            Run the handler $(rid, hid)$ up to and including the next event
            if it is not a send response operation then REJECT
            $\mathrm{Tr}'.append((\mathrm{RESP}, rid, \mathrm{outputs}))$
        else if $op \leftarrow (rid, hid, i)$ then
            if $i = 0$ then
                if ($hid$ is not an activated handler) then REJECT
                Read in the handler's inputs and allocate structures for running the handler
            else if $i = \infty$ then
                Run the handler $(rid, hid)$ until the next event
                if it is not a handler exit operation then REJECT
            else
                Run the handler $(rid, hid)$ until the next event
                if the next event is an external state operation then
                    Execute the state operation against the database
                else if the next event is an annotated operation then
                    if it is a write or initialization then
                        Execute the operation and skip the annotation
                    else
                        Return the current value of the variable
                else if the next event is a handler operation then
                    Execute the handler operation
                    if the operation is an emit operation then
                        for all functions that the operation activates do
                            $hid' \leftarrow (functionID, hid, i)$
                            Name the instance of the handler that is activated $hid'$

    return $\mathrm{Tr}'$
```

Figure 24. Pseudocode for Actual

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

3. $u = (rid_1, 0)$, $v = (rid_2, \infty)$. Since there is a path from v to u in G, the left-to-right direction of relation (4) implies that $rid_2 <_{\mathrm{Tr}} rid_1$. This implies that v appears before u in Tr, which is a contradiction.
4. $u = (rid_1, 0)$, $v = (rid_2, 0)$. From the construction of G, the only incoming edges to u are from nodes (rid_3, ∞) that appear before u in Tr. Thus, $v \overset{G}{\rightsquigarrow} v'$ for some $v' = (rid_3, \infty)$. Meanwhile, v' appears before v in Tr (because v' appears before u and u appears before v), so $v' \overset{G}{\rightsquigarrow} v$, hence a cycle exists between v and v', impossible.

□

.

```
procedure ConstructS(graph $G$)
    Initialize $S'$ to empty, a set *frontier* to the set of all in-degree 0 nodes of $G$ , and set $i = 0$;
    while $G$ is not empty do
        while there exists a node $v$ in *frontier* which is not request/response do
            ProcessFrontier($v$, $G$, $S'$, *frontier*);
        Let $u$ be the node that corresponds to $\mathrm{Tr}[i]$ in *frontier*. If $u$ is not in *frontier* then REJECT
        ProcessFrontier($u$, $G$, $S'$, *frontier*)
        $i \leftarrow i + 1$

procedure ProcessFrontier(Graph $G$, Node $v$, op Schedule $S'$, *frontier*)
    Remove $v$ from *frontier* and from $G$. Also remove the outgoing edges of $v$ from $G$
    Append $v$ to $S'$
    Add all nodes of $G$ that have in-degree 0 to *frontier*
```

Figure 25. Algorithm for creating S'

Sub-lemma 4.2. If $\mathrm{OOOAudit}(\mathrm{Tr}, A, S')$ accepts, $\mathrm{ActualHandlerOps}(\mathrm{Tr}, A, S')$ outputs the trace Tr.

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

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

Base case: Before processing any operation, the two runs have the same program state (because we assume that initialization is deterministic), *active* is empty in OOOAudit and there are no activated handlers in ActualHandlerOps. Thus, the invariants hold before processing the first operation of S'. The first operation in S' has the form $(rid, 0)$. Both executions read inputs from Tr, allocate program structures and, subsequently, perform operations that do not affect program state. Thus, since both executions start from the same program state, the two executions have the same program state after processing *op*. Moreover, because invariant 2 holds prior to processing *op*, it holds after processing *op*: both executions compute the same handler ids for *rid*'s request handlers, which OOOAudit adds to *active* at line 16 of Figure 22 and ActualHandlerOps uses to name the new activated handlers at line 16 of Figure 23.

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

- Case $op = (rid, 0)$: A similar argument as the one used in the base case implies that the invariants hold after processing *op*.
- Case $op = (rid, \infty)$: Let $hid = A.responseEmittedBy[rid].hid$. Since invariant 1 holds, prior to processing *op*, the two executions have executed handler (rid, hid) up until the same operation op_l and have the same program state. From the logic of OOOAudit and ActualHandlerOps, the two executions resume the execution of (rid, hid) from op_l until its next special operation. Since both executions proceed deterministically between operations, the program state of OOOAudit when it reaches line 21 of Figure 22 is the same as the program state of ActualHandlerOps when it reaches line 20 of Figure 23. This implies that the next operation will be the same in both executions and, thus, either both checks at the aforementioned lines pass or both fail. Because the work that the two executions perform past these checks does not affect program state, we conclude that the two executions have the same program state after *op*.
  Moreover, observe that invariant 2 holds prior to processing *op*, OOOAudit does not modify *active* as part of handling *op* and ActualHandlerOps does not modify the activated handlers as part of handling *op*. These imply that invariant 2 holds after processing *op*.
- Case $op = (rid, hid, 0)$: The two executions handle this operation in the same way except that OOOExec checks if *hid* is in *active* and ActualHandlerOps checks if *hid* is the name of some activated handler. Because from the induction hypothesis the ids of all activated handlers of ActualHandlerOps are exactly the handler ids in *active*, either both checks pass or both fail. Thus, the two executions reach the same program state after processing *op*. Moreover, invariant 2 holds prior to processing *op* and while processing *op* neither the activated handlers are modified by ActualHandlerOps nor *active* is modified by OOOAudit. Thus, invariant 2 holds after processing *op*.

- Case $op = (rid, hid, \infty)$: The two executions start from the same program state, pick the same handler, run it until the next event and check that it is a handler exit event. Thus, the two executions result in the same program state. Moreover, upon reaching the handler exit event, ActualHandlerOps truly executes the handler exit operation, and this operation removes the handler with id *hid* from the activated handlers. On the other side, OOOAudit removes *hid* from *active* at line 32. This implies that the invariant holds after processing *op*.
- Case $op = (rid, hid, i)$ where *op* is an external state operation: The result follows from the induction hypothesis, the fact that execution proceeds deterministically between operations and the fact that both executions handle external state operations in the same way.
- Case $op = (rid, hid, i)$ where *op* is a handler operation. From the induction hypothesis and the fact that execution proceeds deterministically between operations we conclude that the two executions have the same program state right before they process *op*. The processing of *op* in ActualHandlerOps and OOOAudit does not affect program state (which, recall, excludes the set of registered handlers and emitted events). Thus, invariant 1 holds after processing *op*.
  Now we argue that invariant 2 holds. From the induction hypothesis the invariant holds before processing *op*. If *op* is not an emit operation, ActualHandlerOps does not modify the activated handlers while processing and OOOAudit does not modify *active*. Thus, invariant 2 holds after processing *op*. On the other hand, assume *op* is an emit operation. We will argue that the handler ids that are in $activatedHandlers[(rid, hid, i)]$ (which are the ones added to *active* by OOOAudit at line 49) are exactly the ones activated in ActualHandlerOps at lines 45–47 of Figure 23. Let *eventName* be the event that *op* emits. Let C be the set of function ids c s.t. $(eventName, c) \in Registered \cup GlobalHandlers$ when *op* is processed by AddHandlerRelatedEdges and C' the set of function ids that *op* activates during ActualHandlerOps. In the following we will sometime abuse notation and refer to the handler's function as handler.
  Claim: $C = C'$. Denote C_g the set of function ids c s.ts. $(eventName, c) \in GlobalHandlers$ and C_r the set of function ids c s.t. $(eventName, c) \in Registered$. Obviously,
  $$C = C_g \cup C_r.$$
  Moreover, because each function that *op* activates during ActualHandlerOps is either a global handler or a function registered for *eventName* by *rid*,
  $$C' = C'_g \cup C'_r,$$
  where C'_g is the set that contains the ids of all global handlers that are registered for event *eventName*, and C'_r is the set that contains the ids of all functions that are registered for event *eventName* over the course of *rid*. To establish the Claim, we show that $C_g = C'_g$ and $C_r = C'_r$:
  1. $C_g = C'_g$: First, observe that C_g is exactly the ids of the functions registered for *eventName* over the course of the initialization procedure of OOOExec. Because the initialization procedure is deterministic, it registers the same functions for *eventName* under both OOOExec and ActualHandlerOps. Thus, C_g is exactly the ids of the functions registered for *eventName* over the course of the initialization procedure of ActualHandlerOps. Because requests don't modify global handlers, $C_g = C'_g$.
  2. $C_r = C'_r$: C'_r contains the ids of the functions that are registered by *rid* for *eventName* at the time when *op* is executed. Because ActualHandlerOps follows S', these are exactly the ids of the functions H s.t.
     a. There exists an operation op_r that registers H for *eventName* and appears before *op* in S', and
     b. For all operations op' between op_r and *op* in S', $op'.rid \neq rid$ or op' does not unregister H from *eventName*.

     Meanwhile, the induction hypothesis and the fact that execution proceeds deterministically between operations imply that OOOExec and ActualHandlerOps have the same program state right before executing every register and unregister operation that precedes *op*. This implies that the parameters of each register or unregister operation op' (these are the *functionID* and *eventNames* for register operations, and *functionID* and *eventName* for unregister operations) are the same under ActualHandlerOps and under OOOExec. Moreover, OOOExec checks these parameters against the corresponding entry in $A.HL_{rid}$ (line 23 of Figure 19). The above implies that C'_r contains exactly the function ids c s.t.:
     a. There exists a register operation op_r with parameters c and *eventNames* in $A.HL_{rid}$ that appears before *op* in S' and for which $eventName \in eventNames$, and
     b. For all operations op' between op_r and *op* in S', either $op'.rid \neq rid$ or the entry in $A.HL_{rid}$ that corresponds to op' is not an unregister operation with parameters c and *eventName*.

     Meanwhile, because S' is a topological sort of the graph G and G has edges between consecutive handler operations in $A.HL_{rid}$ (line 15 of Figure 16), the order of the handler ops of *rid* in S' matches their order in $A.HL_{rid}$. Thus, we conclude that C'_r contains exactly the function ids c s.t.
     a. There exists a register operation op_r with parameters c and *eventNames* in $A.HL_{rid}$ that appears before *op* in $A.HL_{rid}$ and for which $eventName \in eventNames$, and

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

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

Let

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

By definition of C, C_{id} is exactly the set of handler ids that AddHandlerRelatedEdges places in $activatedHandlers[(rid, hid, i)]$ at line 26 of Figure 16, and, because $C = C'$, also exactly the handler ids that ActualHandlerOps uses to name the handlers that op activates at line 47 of Figure 23. So, these two sets are equal, as required.

- Case $op = (rid, hid, i)$ where op is an annotated operation. Since in this case the activated handlers under ActualHandlerOps and *active* under OOOAudit are not modified, if invariant 2 holds prior to this step, it holds after this step.
  Now, we argue that invariant 1 holds. As argued in some of the previous cases, the induction hypothesis and the determinism of execution between operations implies that the program state right before op is processed is the same across executions. If the annotated operation is either a write or initialization, then both executions execute the operation, which results in the same program state. Then, OOOAudit executes the annotation, which ActualHandlerOps skips. However, the annotation does not affect program state on OOOAudit and, consequently, the two executions have the same program state after executing the annotation, as required. Now, we argue that they have the same program state when op is a read. In this case, ActualHandlerOps reads the current value of the variable whereas OOOAudit reads the value returned by the OnRead function of Figure 20. We argue that the value of the variable under ActualHandlerOps (which is the most recent value written) is the value returned from the OnRead annotation in OOOAudit. Let op' the write operation that op reads from in OOOAudit and v the variable that these operations access. We will show that op' is the most recent write operation to v prior to op in S'. From the logic of OnRead, $op \in v.read\_observers\{op'\}$. This implies that there exists a read edge $\langle op', op\rangle$ in G. Moreover, because G contains anti-depend edges and write-depend edges, for any other write op'' to v either there exists a path from op'' to op' consisting of write-depend edges or there exists a path from op to op'' in which the first edge is an anti-depend edge and the rest are write-depend edges. Thus, because S' is a topological sort of G, the last write op to v prior to op in S' is op'. Because ActualHandlerOps follows S', this implies that op' is the most recent write to v prior to op and, thus, the value of v under ActualHandlerOps is the value written by op' as requested.

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

Now, we show that Tr′ is a permutation of Tr. First, we argue that Tr and Tr′ contain entries for the same request ids: This follows from (1) the fact that G's $(rid, 0)$ and (rid, ∞) nodes are exactly those for which $rid \in$ Tr (this follows from the logic of CreateTimePrecedenceGraph and SplitNodes) (2) the fact that S' is a topological sort of G and (3) that Tr′ has exactly one request entry for each $(rid, 0)$ node in S' and one response entry for each (rid, ∞) node in S'. Moreover, the request contents of each request in Tr′ are those in Tr because of the logic of lines 10 and 12 of Figure 23. Last, because invariant 1 holds, for each *rid*, the program state of OOOExec at line 21 of Figure 22 is the same as the program state of ActualHandlerOps when it reaches line 20 of Figure 23. This implies that the response contents for *rid* that ActualHandlerOps writes in Tr′ are those that OOOExec checks against Tr at line 52 of Figure 22.

Last, from the construction of S' (lemma 4.1, Figure 25), the order of $(rid, 0)$ and (rid, ∞) operations in S' corresponds to their order in Tr. Moreover, the order of the operations in Tr′ matches their order in S'. Consequently the order of operations in Tr′ matches their order in Tr, and Tr′ = Tr as required. □

Sub-lemma 4.3. If ActualHandlerOps(Tr, A, S') outputs the trace Tr, then Tr is a possible output of Actual(Tr, A, S').

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

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

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

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

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

1. Constraint 1a: First, because CheckStateOp at ActualHandlerOps does not reject when called for external state operations and ActualHandlerOps follows S', the order of the operations of a transaction t in S' is consistent with their order in $A.TXL_t$. This implies that S' preserves the order of all operations within the transaction. Because E is consistent with S', so does E, as required.
2. Constraint 1b: Because S' is a topological sort of G and G contains read-from edges (line 46 of Figure 16), for each GET operation op in S', $op_w = op$.opcontents, precedes op in S'. Because the order of operations in E is consistent with their order in S', op_w precedes op in E, as required. Furthermore, because the check at line 48 of Figure 16 passes, $op_w.key = op.key$, and op_w.optype = PUT, as required.
3. Constraint 1c: From the logic of AddExternalStateEdges (Figure 16), when the i-th operation op of $A.TXL_t$ is examined, $MyWrites$ has an entry for each key for which there exists at least one PUT operation op' prior to op in $A.TXL_t$ with $op'.key = key$. Moreover, $MyWrites$ maps each such key to the latest PUT operation that modifies it according to $A.TXL_t$. Because ActualHandlerOps passes, the check at line 51 of Figure 16 passes which implies that for each $op \in A.TXL$s: if op.optype = GET and $op.key \in MyWrites$ then op.opcontents = $MyWrites[op.key]$. Thus, the dictating write of each operation op of transaction t that reads a key that has been previously modified by t according to $A.TXL_t$, is the operation op' that last writes this key according to $A.TXL_t$. Meanwhile, because CheckStateOp does not reject when called for external state operations and ActualHandlerOps follows S', the order of the operations of a transaction t in S' is consistent with their order in $A.TXL_t$. Thus, the dictating write of each operation op of transaction t that reads a key that has been previously modified by t according to S', is the last PUT operation op' issued by t that modifies key according to S'. Furthermore, from the definition of E, the order of operations is consistent with S' and the dictating write of each operation is consistent with the $A.TXL$s. Thus, E is internally consistent, as required.
4. The version order V is a list of unique tuples (rid, tid, m) s.t. $(rid, tid, m) \in V$ iff (a) $(rid, tid, \text{PUT}, key, m, v)$ in E, (b) there exists no $(rid, tid, \text{PUT}, key, m', \cdot)$ in E with $m' > m$, and (c $(rid, tid, \texttt{tx\_commit})$ in E: First, observe that V is consistent with $A.writeOrder$. Furthermore, observe that because the checks at lines 23 and 27 of Figure 17 pass, the entries in $A.writeOrder$ are exactly the entries (rid, tid, m) s.t. there exists a key s.t. $lastModification[rid, tid, key] = m$. Meanwhile, from the logic of AddExternalStateEdges, the entries in $lastModification$ are exactly the (rid, tid, key) s.t. transaction (rid, tid) modifies key according to $A.TXL_{(rid,tid)}$ and $(rid, tid) \in Committed$. $lastModification$ maps each such entry (rid, tid, key) to the index of the last operation that writes key in $A.TXL_{(rid,tid)}$. Furthermore, from the logic of AddExternalStateEdges, $(rid, tid) \in Committed$ iff it issues a `tx_commit` operation according to $A.TXL_{(rid,tid)}$. Thus, the entries of $lastModification$ (which correspond to the entries in the version order) correspond to exactly the PUT operations op s.t. (a) there exists a transaction (rid, tid) s.t. $op \in A.TXL_{(rid,tid)}$, (b) there exists no PUT operation op' to $op.key$ that appears after op in $A.TXL_{(rid,tid)}$, and (c) there exists a `tx_commit` operation in $A.TXL_t$. Meanwhile, as argued above, the operations in E correspond to exactly the operations in $A.TXL$s, and the order of operations in E is consistent with the order of the corresponding operations in $A.TXL$s. Thus we conclude that the entries in the version order V are exactly the operations (rid, tid, m) s.t. (a) (rid, tid, m) appears in E (b) there exists no $(rid, tid, \text{PUT}, key, m', \cdot)$ in E with $m' > m$, and (c) $(rid, tid, \texttt{tx\_commit})$ in E, as required.

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

First, observe that:

1. when the isolation level is READ COMMITTED or SERIALIZABILITY, H does not exhibit phenomena G1a and G1b: Phenomena G1a and G1b require that each read of a committed transaction in E should read from an operation in V. As argued above, the entries of E correspond to the entries of $A.TXL$s and the operations in V correspond exactly to the entries in *lastModification*. Thus, we need to show that each `GET` operation in $A.TXL_t$ s.t. t's last operation is `tx_commit`, reads from an entry in *lastModification*. This is exactly the check that IsolationLvlVer performs in the case of READ COMMITTED or SERIALIZABILITY in line 27 of Figure 17.
2. $DSG(H)$ and DG have the same nodes: $DSG(H)$ contains exactly the transactions that commit according to H. Meanwhile, from the construction of H, these transactions are exactly the transactions t s.t. there exists a `tx_commit` operation in $A.TXL_t$. From the logic of AddExternalStateEdges these are exactly the transactions in *Committed* which IsolationLvlVer adds to G. Thus, $DSG(H)$ and DG have the same nodes as required.
3. the edges that AddWriteDependencyEdges adds to DG are exactly the write depend edges of $DSG(H)$: the write depend edges of $DSG(H)$ are exactly the edges between T_1 and T_2 s.t. T_1 installs a version of some key and T_2 installs the next version according to V. On the other side, from the logic of ExtractWriteOrderPerKey and AddWriteDependencyEdges, the white dependency edges of DG are exactly the edges $\langle T_1, T_2\rangle$ s.t. T_1 installs a version of some key and T_2 installs the next version according to $A.writeOrder$. Because V exactly matches the $A.writeOrder$, we conclude that the write depend edges of $DSG(H)$ are exactly the write dependency edges of DG.
4. the edges that AddReadDependencyEdges adds to DG are exactly the read depend edges of $DSG(H)$ when the isolation level is READ COMMITTED or SERIALIZABILITY: Observe that for these isolation levels, because the history does not exhibit phenomena G1a and G1b, the dictating write of each `GET` of a committed transaction is an operation in V. This implies that the read depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2\rangle$ for which there exist operations $op_1 \in T_1$ and $op_2 \in T_2$ s.t. op_2 reads from op_1 according to E, $op_1 \in V$ and T_2 commits according to E. Meanwhile, V matches $A.writeOrder$ and the dictating writes of operations in E match $A.TXL$s from construction. Moreover, the committed transactions according to E are exactly those in *Committed*: from the logic of AddExternalStateEdges, *Committed* contains exactly the committed transactions according to $A.TXL$s and, from the definition of the execution above, these are exactly the transactions that commit according to E. Thus, the read depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2\rangle$ for which there exist $op_1 \in A.TXL_{T_1}$ and $op_2 \in A.TXL_{T_2}$ s.t. op_2.opcontents $= op_1$, $op_1 \in A.writeOrder$ and $T_2 \in Committed$. From the logic of AddExternalStateEdges and AddReadDependencyEdges, these are exactly the read dependency edges of DG as required.
5. the edges that AddAntiDependencyEdges adds to DG are exactly the anti depend edges of $DSG(H)$: The anti depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2\rangle$ for which there exists a transaction T_3 and operations $op_1 \in T_1$, $op_2 \in T_2$, $op_3 \in T_3$ s.t. the dictating write of op_1 is op_3 according to E, T_1 commits according to E, and op_3 installs a version of a key and op_2 installs the next version according to V. Meanwhile V exactly matches $A.writeOrder$, and the dictating writes of `GET` operations in E exactly match their dictating writes according to $A.TXL$s. Last, as argued above *Committed* contains exactly the committed transactions according to E. Thus we conclude that the anti depend edges of $DSG(H)$ are exactly the edges $\langle T_1, T_2\rangle$ for which there exists a transaction T_3 and operations $op_1 \in A.TXL_{T_1}$, $op_2 \in A.TXL_{T_2}$, $op_3 \in A.TXL_{T_3}$ s.t. the dictating op_1.opcontents $= op_3$, $T_1 \in Committed$, and op_3 installs a version of a key and op_2 installs the next version according to $A.writeOrder$. These are exactly the anti dependency edges of DG as required.

When the required isolation level is READ UNCOMMITTED, H exhibits the isolation level because it does not exhibit phenomenon G0: First, the results 2 and 3 imply that the subgraph of $DSG(H)$ that contains only write depend edges is exactly DG. Moreover, DG is acyclic because the check at line 11 of Figure 17 accepts. This implies that $DSG(H)$ does not exhibit phenomenon G0 as required.

When the required isolation level is READ COMMITTED, H exhibits the isolation level because it does not exhibit phenomenon G1: First, result 1 implies that H does not exhibit phenomena G1a and G1b. Moreover, the results 2, 3, and 4 imply that the subgraph of $DSG(H)$ that contains only write depend and read depend edges is exactly DG. Moreover, DG is acyclic because the check at line 15 of Figure 17 accepts. This implies that $DSG(H)$ does not exhibit phenomenon G1c, as required.

When the required isolation level is SERIALIZABILITY, H exhibits the isolation level because it does not exhibit phenomena G1 and G2: First, result 1 implies that H does not exhibit phenomena G1a and G1b. Moreover, the results 2, 3, 4, and 5 imply that $DSG(H)$ is DG. Moreover, DG is acyclic because the check at line 20 of Figure 17 accepts. This implies that $DSG(H)$ is acyclic and, thus, does not exhibit phenomena G1c and G2, as required. □

Sub-lemma 4.4. If Tr is a possible output of Actual(Tr, A, S'), then Tr is a possible output of Operation-wise execution on input Tr by following RS.

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

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

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

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

Denote $\text{Actual}_{\text{Tr}}$ the execution of Actual that outputs Tr on input Tr and S'. $\text{Actual}_{\text{Tr}}$ captures both the execution at the server and the execution at the database. To show that Tr is a possible output of Operation-wise, we will show that there exists an execution of Operation-wise on input Tr and *RS* that is identical to $\text{Actual}_{\text{Tr}}$. We do the proof by induction: we show that if $\text{Actual}_{\text{Tr}}$ and Operation-wise have proceeded in the same way up until the $(i-1)$ step, the next step of $\text{Actual}_{\text{Tr}}$ is a step that Operation-wise can take that will result in the two executions having the same program state and database state after step i.

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

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

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

Now assume that this is not the first occurrence of *rid* in *RS*. Because *RS* is constructed from the well formed S' by dropping all fields other than *rid*, the corresponding operation in S' is either of the form (rid, hid, i) or (rid, ∞). In either case, $\text{Actual}_{\text{Tr}}$ "resumes" the execution of a handler that is activated. Because the two executions are identical up to step $i-1$, the activated handlers are the same. This implies that the activated handler that $\text{Actual}_{\text{Tr}}$ picks to execute is an activated handler in Operation-wise. Thus, Operation-wise can take the same next step as $\text{Actual}_{\text{Tr}}$ executing the same handler up until its next special operation. Because execution between special operations is deterministic and does not modify the database state, both executions have the same program state and database state up until they execute the next special operation. Moreover, the handling of all special operations other than external state operations is the same in $\text{Actual}_{\text{Tr}}$ and Operation-wise and such operations don't modify database state. Thus, both executions reach the same program state and database state after step i when the special operation at step i is not a state operation. It remains to show that $\text{Actual}_{\text{Tr}}$ and Operation-wise reach the same program state and database state after step i, when the special operation at step i is an external state operation: Because the two executions have the same program state up until executing the i-th special operation, the parameters of the state operation are the same across executions and, thus, both executions issue the same operation to the database. Because two databases that start from the same state and receive the same operation can execute this operation identically, we conclude that the i-th state operation can be executed by Operation-wise in the same way that it was executed by $\text{Actual}_{\text{Tr}}$. In this case, the database returns the same result under Operation-wise that it returns under $\text{Actual}_{\text{Tr}}$, and the two executions have the same program state and database state after executing the i-th operation as required. □

□

Lemma 5. Given trace Tr and advice A, if Audit(Tr, A) accepts, then there exists a well-formed op schedule S that causes OOOAudit(Tr, A, S) to accept.

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

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

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

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

- It contains all nodes $(rid, 0)$ and (rid, ∞) s.t. $rid \in \text{Tr}$ otherwise the produced outputs are not exactly the outputs in the trace and ReExec rejects in line 62 of Figure 18.
- S contains nodes $(rid, hid, 0)$, (rid, hid, ∞) for each $(rid, hid) \in A.opcounts$: Notice that S contains nodes $(rid, hid, 0)$ and (rid, hid, ∞) for each (rid, hid) that is executed by ReExec. To show that these nodes are exactly the handler start and handler end nodes of G (lines 39 and 40 of Figure 14) we need to prove that for each rid, the set H of all hid s.t. (rid, hid) in $A.opcounts$ and the set H' of all hid that are executed by ReExec are equal. We show that $H' \subseteq H$ and that $H \subseteq H'$.
  $H' \subseteq H$: Notice that from the logic of ReExec, H' is exactly the hids that are in *active* during the execution of the group in which rid belongs to. We will show that for each hid that is added in *active* during the execution of rid, $opcounts[(rid, hid)] \neq \emptyset$. *active* is initially empty and entries are added to it in line 12 of Figure 18 and in ActivateHandlers. In the former case, ReExec rejects if (rid, hid) is not in *opcounts* (line 13 of Figure 18). In the latter case, hid is added to *active* from $activatedHandlers(rid, \cdot, \cdot)$. Notice that hid can only be added to $activatedHandlers(rid, \cdot, \cdot)$ at line 26 of Figure 16 after the check at line 25 of Figure 16 passes. Thus, for any $hid \in activatedHandlers(rid, \cdot, \cdot)$, $A.opcounts[rid][hid] \neq \emptyset$ as required.
  $H \subseteq H'$: This follows from the fact that ReExec does not reject at line 64 of Figure 18.
- For each (rid, hid) in $A.opcounts$ it contains all nodes (rid, hid, j) s.t. $j \in [0, A.opcounts[(rid, hid)]]$. This follows from the previous bullet and the fact that for each (rid, hid) the value of j in S prior to the insertion of (rid, hid, ∞) is $A.opcounts[(rid, hid)]$: ReExec does not reject at line 60 of Figure 18 and, thus $j \geq A.opcounts[(rid, hid)]$. Moreover, $j \leq A.opcounts[(rid, hid)]$ because otherwise ReExec rejects in line 43 of Figure 18, in CheckStateOp or CheckHandlerOp.

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

Now, we prove that OOOAudit(Tr, A, S) accepts. OOOAudit(Tr, A, S) (Figure 22) is the same as Audit(Tr, A) (Figure 18) except the differences that we describe below. For each of them, we show that they do not result in different program state or OOOAudit rejecting.

1. ReExec executes the requests in SIMD style whereas OOOExec round-robins the execution from operation to operation for a group of requests. This does not affect program state; the flow and ordering is the same across both executions. Thus, the produced output is the same.
2. When ReExec executes a group, it picks the next handler to run from *active* whereas OOOExec picks the next handler to run from S. This difference is superficial because S is derived from ReExec.
3. There is a difference in how handler end events are processed. In OOOExec there is an (rid, hid, ∞) case that checks that the next event is a handler exit operation. In ReExec handler exit events are processed in case 2c where the number of operations issued by the handler is checked against $A.opcounts$ (line 60 of Figure 18). Similar arguments to those made elsewhere (Orochi [87], lemma 8) establish that this difference is superficial.
4. When OOOExec starts executing a handler, it checks that it is in *active*. ReExec does not do this check but picks which handler to run from *active*. The difference does not result in different program state because both executions just require that when a handler starts executing, it must be in *active*.
5. ReExec keeps track of the number of ops that a handler hid has executed so far in $idx[hid]$. OOOExec uses the i field in the op schedule entry as the number of ops that the handler has issued so far. The difference does not result in different program state because both i and $idx[hid]$ correspond to the running counter of operations that the handler issues and thus $i = idx[hid]$ at all times.
6. When a group sends back a response, ReExec checks that the contents of $A.responseEmittedBy$ match re-execution. In OOOExec there is no such check, but there is a (rid, ∞) case. The difference is superficial; both executions reject if $A.responseEmittedBy$ does not match re-execution.

□

D Definitions of isolation levels according to Adya

In this section we briefly present Adya's definitions for consistency models [7]. We should note that we modify these definitions to make them consistent with our terminology. Additionally, we modify them to make them suitable for transactional key-value stores; for instance, we erase the parts of the definitions that refer to predicates.

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

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

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

   The TxOp order must obey the following constraints:

   a. It preserves the order of all operations within a transaction including its commit and abort operations
   b. A transaction $T_{rid,tid}$ cannot read version $key_{rid',tid'}$ before it has been produced by $T_{rid',tid'}$. Formally, if an operation $(rid, tid, \texttt{GET}, key, rid', tid', m)$ exists in H, it is preceded by $(rid', tid', \texttt{PUT}, key, m, \cdots)$ in H.
   c. If a transaction modifies a key and later reads it, it will observe its last update to the key. Formally, if an operation $(rid, tid, \texttt{PUT}, key, m, \cdots)$ is followed by an operation $(rid, tid, \texttt{GET}, key, rid', tid', m')$ in H without the interleaving of an operation $(rid, tid, \texttt{PUT}, key, m'', \cdots)$, it should be $rid = rid'$ and $tid = tid'$ and $m = m'$. We call this property *internal consistency*
2. An order all key versions (version order) V created by committed transactions in E i.e. a list of unique tuples (rid, tid, m) s.t. $(rid, tid, m) \in V$ iff (a) $(rid, tid, \texttt{PUT}, key, m, v)$ in E, (b) there exists no $(rid, tid, \texttt{PUT}, key, m', \cdot)$ in E with $m' > m$, and (c $(rid, tid, \texttt{tx\_commit})$ in E.

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

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

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

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

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

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

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

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