

Native Evidence for Classifying Replayed Failures

Abstract

Backend failures can depend on effects that request logs omit. We present a replay gate that joins application dependency records to operating-system effects. The gate assigns a result only when the trigger, dependency sessions, native trace, and failure identity are complete. Otherwise it reports an unknown result. In a real Chrome run, a fixture application reproduced a numeric defect with value -1 against a nonnegative-value rule. The fixed build passed, the unchanged build was classified as a regression, and an unstable build was marked unknown. A Linux process test also reproduced an injected filesystem permission failure and distinguished its fix from an alternate failure and timeout. All outcomes matched their hand-authored test oracles. No negative control was reported as the target failure. These two authored scenarios do not estimate precision on production bugs.

1 Introduction

A backend request can fail because of a dependency response, a local file, process configuration, or runtime state. A request log preserves the trigger but may omit the effect that caused the failure. Replaying a partial trace can mislabel a different failure as the original one. A useful replay result must distinguish the same failure, a fixed failure, a different failure, and an execution that cannot be judged.

Application-level record and replay already captures selected inputs and dependency calls. R2 records interactions at application boundaries [1]. BugRedux reconstructs field failures from execution information [4]. Parikshan replays production service requests against cloned services [6]. Execution Reconstruction uses hardware-assisted tracing to reproduce failures [7]. These systems establish the value of replay at application, service, and process boundaries. They do not make an application trace complete when an effect bypasses its adapter.

The system checks dependency behavior inside a declared backend operation against a separate native trace. It keeps a failed operation only when every supported observation has one owner, every bounded session completes, all seven adapter classes are registered, and the native trace stays healthy. During replay, it matches requests to recorded digests, returns recorded outcomes, and rejects unconsumed records or unowned native effects. Its checker separates pass, regression, different failure, and unknown. The evaluation tests whether this distinction holds for a small set of controlled process failures.

2 System

2.1 Operation capture

An operation is one request-response call, ordered stream, or delivered task. The application defines its start, trigger input, and terminal result. At operation start, the SDK captures the process environment and a wall-clock baseline. Seven adapter classes observe clock, database, environment, filesystem, outbound HTTP, queue, and randomness effects.

An adapter opens an observation session around a supported dependency call. The session records a bounded request and its response or error. Each record carries its class and position, operation identity, causal parent when available, and request digest. Environment and filesystem state use content-addressed artifacts. Other dependency calls use request-response transcripts. The shared contract permits at most 1,024 observations and 65,536 bytes per semantic request or response. Environment artifacts have a separate 1 MiB limit.

Adapters differ by language and library. They preserve the live result during capture, then use the recorded result during replay. Unsupported transports, partial or oversized bodies, sensitive headers, ambiguous asynchronous ownership, and failed hooks keep the operation local. The system does not replace an unsupported live effect with an assumed response.

2.2 Native coverage gate

Semantic adapters show which dependency calls the application made. They cannot show that every relevant operating-system effect passed through an adapter. On Linux, a separate tracer observes selected syscall events across the process threads. It records only the thread, event class, and sequence number, not syscall arguments or application payloads. Its event classes include filesystem, network, clock, randomness, and process events.

When an adapter dispatches a dependency call, the SDK binds that observation to its operation, class, and thread. The tracer assigns a compatible event to that active observation. Database, HTTP, and queue calls may own the filesystem, network, clock, and randomness effects needed to perform the dependency call. A direct filesystem, clock, or random read can be owned only by its matching observation class. If no unique active observation owns an event, the sentinel marks active operations incomplete. A lost event, queue overflow, tracer exit, or invalid sequence also invalidates coverage.

The operation closes only if all of these conditions hold:

$$\begin{aligned} \text{eligible}(o) = & \text{failed}(o) \wedge \text{triggerComplete}(o) \\ & \wedge \text{sessionsComplete}(o) \wedge \text{allAdapters}(o) \\ & \wedge \text{nativeTraceClean}(o) \wedge \text{withinBounds}(o). \end{aligned} \tag{1}$$

The gate runs before candidate delivery. Successful operations are discarded. Incomplete, unsupported, ambiguous, or over-limit operations remain local. Package validation must also verify that registered adapter hooks are present. A registration alone does not prove that a hook is installed. Automatic native coverage targets Linux ARM64 and x86_64. macOS and Windows keep automatic captures local until matching providers are available.

2.3 Replay

A compatible Linux worker verifies the sealed capture and its content digests, restores the captured subject files and environment, and sends the original trigger to the subject. It serves recorded dependency exchanges locally. The replay adapter matches each request by class, position, and digest before it returns a recorded outcome. It never calls the live dependency. Replay fails if a request changes, a session is abandoned, a record is missing or unread, an extra call occurs, or the native trace reports an unowned effect. The worker compares the resulting failure identity with the captured identity. Replay is platform-compatible, not cross-platform: the subject runtime and operating system must match the capture contract.

3 Evaluation

3.1 Method

We ran two native Chrome integration tests on macOS ARM64 with Chrome 154.0.8037.57. They launch a local HTML, JavaScript, and CSS application in Chrome, performs six browser actions, and checks the observed DOM state. The baseline sets a numeric widget to -1 against a rule requiring a value of at least zero. The test then checks a fixed build, an unchanged build, and a build whose element geometry changes during observation. It also tampers with saved state evidence, verifies the result offline, blocks an external request, and checks browser and file cleanup. A second test cancels a live Node and Chrome process group and checks that both processes and their files are removed. Both tests passed.

We also ran the operating-system crate’s complete test suite on ARM64 Linux, Ubuntu 24.04, and Linux 6.17. All 62 tests passed. Five tests launch native subject processes through the Linux adapter. One integration test exercises a protected filesystem write, its corrected version, an unchanged failure, a different failure, a timeout, and memory exhaustion. Four more tests check UI trigger completeness, shared deadlines, deadline violations, and staging cleanup. A successful result is verified offline against its capsule, host evidence, process outputs, and cleanup proof.

Both suites use authored controls rather than bugs collected from applications. The expected result for each control is set by the test. An unknown result counts as an abstention, not as a correct failure match.

3.2 Observed outcomes

Table 1: Observed outcomes for the end-to-end browser and native process controls.

Run	Injected defect or control	Observed result
Chrome baseline	Numeric value -1 , minimum zero	Target reproduced
Chrome fixed	Numeric value zero	Pass
Chrome unchanged	Numeric value -1	Regression
Chrome unstable	Element geometry changes	Unknown
Chrome tampering	Saved state byte changed	Verification rejected
Linux baseline	Protected filesystem write	Target reproduced
Linux fixed / unchanged	Write fixed / remains denied	Pass / regression
Linux different / timeout	Alternate failure / no result	Unknown / unknown
Linux resource / trigger	Memory limit / incomplete trace	Limited / unknown

The three determinate Chrome decisions matched their labels: the baseline reproduced the numeric defect, the fixed build passed, and the unchanged build was a regression. The unstable build was unknown. The alternate-failure and timeout process controls were also unknown, so neither became a false target match. The memory-limited process stopped at its bound. Evidence tampering failed offline verification. Cancellation and deadline violations left no task processes or staged files.

The three determinate browser outcomes matched their expected labels. The fixed build was the only determinate negative browser control. The alternate failure and unstable run abstained, and neither was labeled as the target. This is not a field false-positive rate or a precision estimate: the study has one unique browser defect and one unique filesystem fault, with hand-authored expected results. It does not measure sensitivity across a representative set of bugs.

4 Related Work

R2 records application interactions, while PRES reconstructs missing nondeterminism from partial execution sketches [1, 2]. ReCrash and BugRedux use captured execution state to reproduce failures or generate tests [3, 4]. Timelapse replays web interactions [5]. Parikshan replays service requests against cloned services, and Execution Reconstruction uses hardware-assisted tracing and symbolic execution [6, 7]. Microservice traffic replay also identifies dependency boundaries for later replay [8].

The system combines application-level dependency records with a native ownership gate, then uses the resulting capsule to rerun a captured backend subject. Its claim is about this admission rule and its implementation across SDKs.

5 Limits and Implications

The Linux tracer classifies a bounded set of syscall events. The adapters cover only listed library boundaries. Untested effects, thread schedules, hardware state, and external services can still change an execution.

The evaluation uses one browser defect and one injected filesystem failure, not a historical bug corpus or production application. Its test labels are hand-authored, and the number of positive and negative cases is too small for a useful precision or recall estimate. The tests do not measure how many real failures the adapters capture, how often valid captures replay, production overhead, artifact size, or integration effort. The results support only the claim that the tested browser and Linux process paths distinguish these faults from their fixes and abstain on the tested incomplete or alternate outcomes.

6 Conclusion

The gate admits a replay only when application records and native effect evidence agree. In real Chrome, it reproduced one numeric UI defect, accepted its fix, flagged the unchanged build, and abstained on unstable state. Linux process tests produced the same distinction for one protected filesystem write. A broader bug corpus is needed to measure how well these labels generalize.

References

- [1] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008. <https://www.usenix.org/conference/osdi-08/r2-application-level-kernel-record-and-replay>.
- [2] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009. <https://doi.org/10.1145/1629575.1629593>.
- [3] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of ECOOP*, 2008. <https://homes.cs.washington.edu/~mernst/pubs/reproduce-failures-ecoop2008-abstract.html>.
- [4] W. Jin and A. Orso. BugRedux: Reproducing Field Failures for In-House Debugging. In *Proceedings of the 34th International Conference on Software Engineering*, 2012. <https://doi.org/10.1109/ICSE.2012.6227168>.
- [5] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, 2013. <https://homes.cs.washington.edu/~mernst/pubs/record-replay-uist2013-abstract.html>.
- [6] N. Arora, J. Bell, F. Ivančić, G. Kaiser, and B. Ray. Replay without Recording of Production Bugs for Service Oriented Applications. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018. <https://doi.org/10.1145/3238147.3238186>.
- [7] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, and B. Kasikci. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021. <https://doi.org/10.1145/3453483.3454101>.
- [8] J. Liu, J. Liu, P. Di, A. X. Liu, and Z. Zhong. Record and Replay of Online Traffic for Microservices with Automatic Mocking Point Identification. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, 2022. <https://doi.org/10.1145/3510457.3513029>.