RepFrame: An Efficient and Transparent Framework for Dynamic Program Analysis

Heming Cui[+*], Rui Gu[*], Cheng Liu[*], and Junfeng Yang[*]

[+]The University of Hong Kong [*]Columbia University

Abstract

Dynamic program analysis frameworks greatly improve software quality as they enable a wide range of powerful analysis tools (e.g., reliability, profiling, and logging) at runtime. However, because existing frameworks run only one actual execution for each software application, the execution is fully or partially coupled with an analysis tool in order to transfer execution states (e.g., accessed memory and thread interleavings) to the analysis tool, easily causing a prohibitive slowdown for the execution. To reduce the portions of execution states that require transfer, many frameworks require significantly carving analysis tools as well as the frameworks themselves. Thus, these frameworks significantly trade off transparency with analysis tools and allow only one type of tools to run within one execution.

This paper presents RepFrame, an efficient and transparent framework that fully decouples execution and analysis by constructing multiple equivalent executions. To do so, RepFrame leverages a recent fault-tolerant technique: *transparent state machine replication*, which runs the same software application on a set of machines (or replicas), and ensures that all replicas see the same sequence of inputs and process these inputs with the same efficient thread interleavings automatically. In addition, this paper discusses potential directions in which RepFrame can further strengthen existing analyses. Evaluation shows that RepFrame is easy to run two asynchronous analysis tools together and has reasonable overhead.

1 Introduction

Dynamic program analysis frameworks greatly improve software quality as they enable a wide range of powerful analysis tools (e.g., data race detectors [34, 41, 43] for multithreaded applications) at runtime. Existing analysis frameworks can be classified into two approaches depending on how a framework transfers an application's execution states to an analysis tool. Traditional analysis frameworks [8, 31, 34, 38, 41] take a "fully-coupled" approach: a framework in-lines an analysis with the execution, then the analysis can inspect all or most execution states. However, this approach can easily let the analysis slow down the actual execution. To improve performance, leveraging a fact that many analyses can be done asynchronously with the actual execution, some recent frameworks [10, 20, 22, 35, 42, 43] take a second "partially-decoupled" approach: only analysis-critical execution states (e.g., effective memory addresses and thread interleavings) are transferred to the analysis running on other CPU cores, then the actual execution can be less perturbed by the analysis.

Unfortunately, despite these great effort, most existing analysis frameworks are still hard to deploy in production runs, mainly due to three problems. The first problem is still performance. Traditional frameworks fully couple the analysis with the actual execution using the shadow memory approach, so whenever an analysis does some heavyweight work, the execution is slowed down (e.g., a popular race detection tool ThreadSanitizer [41], which uses a traditional framework, incurs 20X$\sim$100X slowdown for many programs). Recent frameworks that take the "partially-decoupled" approach have shown to run 4X$\sim$8X times faster [22, 43] than traditional frameworks because they transfer fewer execution states. However, the overall slowdown of these recent frameworks are still prohibitive in their own evaluation because the amount of execution states (e.g., effective memory addresses [22] and thread interleavings [43]) transferred to analysis tools are still enormous.

The second problem is that recent frameworks with the "partially-decoupled" approach heavily trade off transparency with analysis tools. To be able to transfer fewer execution states to an analysis tool, these frameworks require heavily carving the analysis tool as well as the transferred execution states. For example, a recent framework [43] for race detection tools leverages the record-replay technique [24, 25, 29] to reduce analysis work in the actual execution, but this framework requires significantly carving race detection tools into several phases to adapt to record-replay, which makes the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

APSys '15, July 27-28, 2015, Tokyo, Japan.
2015 ACM. ISBN 978-1-4503-3554-6/15/07$15.00.
http://dx.doi.org/10.1145/2797022.2797033

tools largely different from typical ones (Note that an analysis tool itself is already quite difficult to implement and maintain.)

The third problem is that existing frameworks, including both traditional ones and recent ones, have not shown to run multiple types of analysis tools together. Multiple analysis tools bring multiple powerful guarantees, which is attractive to today's applications. However, the traditional "fully-coupled" frameworks typically take the shadow memory approach to hold analysis results for each memory byte in the actual execution, then different analyses may require different copies of shadow memory per byte. It is not trivial to extend these frameworks to support multiple copies of shadow memory. Considering "partially-decoupled" frameworks, it is not trivial to extend them to support multiple types of analysis tools within one execution either, because these frameworks heavily carve the transferred execution states, which are hard to reuse for different types of analysis tools.

In sum, a fundamental reason for the three problems in existing analysis frameworks is that they run only one actual execution, so an analysis tool has to be fully or partially coupled with the execution in order to inspect execution states. Well, what if one can construct multiple equivalent executions efficiently and transparently?

To address this question, this paper presents REPFRAME, an efficient, transparent dynamic program analysis framework by leveraging a technique called *transparent state machine replication*, presented in CRANE [16]. This technique runs the same multithreaded application on a set of machines (or replicas) transparently without modifying these applications, and enforces that all replicas perform the same execution state transitions. To do so, CRANE combines two techniques, *state machine replication (or SMR)* and *deterministic multithreading (or DMT)*. SMR ensures that all replicas always see the same sequence of inputs as long as majority of replicas agree on these inputs. DMT efficiently enforces the same thread interleavings across replicas on each input.

Leveraging CRANE, REPFRAME can just run analysis tools on some replicas while the other replicas still run actual executions and process client requests fast. However, to achieve this goal, REPFRAME must address three practical challenges. First, even asynchronous analysis tools (e.g., race detectors) sometimes may decide to roll back to previous execution states and discard malicious inputs that triggered harmful events. To address this challenge, REPFRAME provides a transparent application-level checkpoint mechanism with expressive API for replicas to roll back consistently.

Second, it is unknown whether REPFRAME's replication techniques and existing analysis tools can benefit each other. To address this challenge, this paper discusses several types of analyses in which REPFRAME has the potential to strengthen and speedup existing tools themselves via REPFRAME's replication architecture. In addition, this paper points out that existing race detection tools can actually speedup REPFRAME. In a word, REPFRAME and existing analysis tools form a mutually beneficial eco-system.

Third, despite much effort, state-of-the-art still lacks evaluation to show that different types of analysis tools can actually run together. To address this challenge, we evaluated REPFRAME on a popular parallel anti-virus scanning server `ClamAV` [11] with three replicas (each has 24 cores) in Linux. Our evaluation shows that REPFRAME is able to transparently run two analysis tools together: one is a heavyweight analysis tool, the Helgrind race detector [34]; the other is a lightweight analysis tool, DynamoRio's code coverage tool `drcov` [8]. Moreover, REPFRAME incurred merely 2.1% overhead over the actual execution.

The main contribution of REPFRAME is the idea of applying transparent state machine replication to fully decouple an application's actual execution and analysis tools, which benefits applications, analysis tools, and frameworks. Note that: (1) unlike CRANE, REPFRAME does *not* aim to provide fault-tolerance, but aims to construct multiple equivalent executions so that applications can enjoy efficient and transparent analyses; and (2) REPFRAME does not aim to replace traditional "fully-coupled" frameworks, but aims to compensate them because REPFRAME can easily run them on replicas.

In the remaining of this paper, §2 introduces the background of the CRANE system. §3 gives an overview of REPFRAME, including its deployment model, its checkpoint design for analysis tools, and its potential benefits. §4 presents evaluation results, §5 introduces related work, and §6 concludes.

2 CRANE Background

CRANE's deployment model is similar to a typical SMR's. In a CRANE-replicated system, a set of `2f+1` machines (nodes) are set up within a LAN, and each node runs an instance of CRANE containing the same server program. Once the CRANE system starts, one node becomes the *primary* node which proposes the order of requests to execute, and the others become backup nodes which follow the primary's proposals. An arbitrary number of clients in LAN or WAN send network requests to the primary and get responses. If failures occur, the nodes run a leader election to elect a new leader and continue. Despite `f` nodes fail, CRANE still guaratees availability of the replicated application.

This section presents the background of SMR (§2.1) and DMT (§2.2), the two techniques that CRANE combines.

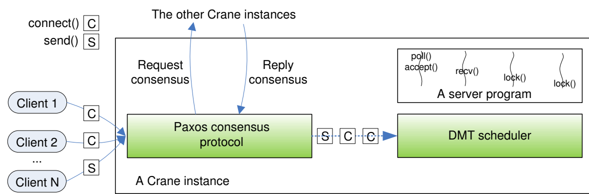

Figure 1: *The* CRANE *Architecture.* Key components are shaded (and in green).

2.1 State Machine Replication (SMR)

State machine replication (SMR) is a powerful fault-tolerance concept [33]. It models a program as a deterministic state machine, where states are important program data and the transitions are deterministic executions of program code under input requests. SMR runs replicas of this state machine on multiple nodes, tolerating many possible node and network failures. To keep the replicas consistent, it invokes a distributed consensus protocol (typically PAXOS [26, 28, 33]) to ensure that a quorum (typically majority) of the replicas agree on the input request sequence; under the deterministic execution assumption, this quorum of replicas must reach the same exact state. SMR is proven safe in theory, and provides high availability in practice.

To support general server programs transparently, CRANE leverages CRANE's PAXOS consensus protocol, which takes the POSIX socket API as consensus interface. This PAXOS protocol enforces two kinds of orders for socket operations. First, for requests coming from the clients, such as `connect()` and `send()` requests, this protocol enforces that all nodes see the same totally ordered sequence of these requests using the PAXOS and socket API interposition components. (this protocol does not need to order the blocking socket operations in the clients because we mainly focus on analyses for server applications.) Second, for server applications' blocking operations, this PAXOS protocol schedules them according to the matching operations from the clients (e.g., a `send()` from a client matches a `recv()` from the server within the same socket connection). This protocol does not schedule non-blocking operations in servers (e.g., `send()` to clients) because it focuses on replicating the server's execution states.

Figure 1 shows an instance of CRANE running on each node, and the PAXOS consensus component is the gateway of this instance. This component accepts socket requests from the clients and invoke a PAXOS consensus instance with the other replicas on this operation. Once a consensus is reached, this component forwards the operation to the DMT component. This component is also the only CRANE component that communicates among different CRANE instances.

In this paper, REPFRAME skips the fault-tolerance nature of CRANE, but leverages it to construct multiple equivalent executions for analysis tools.

2.2 Deterministic Multithreading (DMT)

DMT [3–6, 17, 21, 36] is an advanced threading technique that enforces the same schedule on the same inputs. This technique typically maintains a *logical time*[1] that advances deterministically based on the code run. It allows a thread to synchronize only at deterministic logical times. By induction, it makes an entire multithreaded execution deterministic. The overhead of DMT is typically moderate: one recent DMT system, PARROT [15], incurs an average of 12.7% overhead on a wide range of 108 popular multithreaded programs on 24-core machines.

The DMT component in Figure 1 runs within the same process as a server replica, and enforces the same logical clocks for inter-thread communication operations. CRANE leverages the PARROT [15] DMT runtime system because it is fast (i.e., 12.7% overhead for a wide range of 108 popular multithreaded programs) and transparent to the application.

Specifically, PARROT uses a runtime technique called LD_PRELOAD to dynamically intercept Pthreads synchronizations (e.g., `pthread_mutex_lock()`) issued by an executable and enforces a well-define, round-robin schedule on these synchronization operations for all threads, practically eliminating nondeterminism in thread synchronizations. Although PARROT is not designed to resolve data races deterministically, deploying a race detector in one replica can overcome this limitation (§3.2). CRANE augments the DMT component to schedule the return points of blocking socket operations in server replicas, too, to ensure that requests are admitted exactly at the same logical time across replicas.

3 REPFRAME Overview

REPFRAME's deployment model is similar to CRANE's except two things: (1) at least `f+1` nodes run an actual execution or a lightweight analysis tool on each so that they can reach consensus on inputs and process requests fast, and (2) at most `f` nodes run an heavyweight analysis tool on each. Empirically (see §4), this paper considers an analysis tool with no more than 30% overhead as lightweight tools (e.g., DynamoRio's drcov tool [8]), while the other tools heavyweight tools (e.g., the Helgrind race detector [34]).

A REPFRAME instance running on each node is the same as the one in Figure 1 except that a server program runs transparently in a REPFRAME instance with or without an analysis tool. Neither the server or the analysis is aware of REPFRAME's components. The re-

[1] Though related, the logical time in DMT is not to be confused with the logical time in distributed systems [27].

maing of this section first presents REPFRAME's checkpoint and rollback design for analysis tools (§3.1), and discusses REPFRAME's potential benefits (§3.2).

3.1 Checkpoint and Rollback Mechanism

To allow synchronous analysis tools (e.g., control flow integrity and buffer overrun protection tools) to recover from malicious events, we have designed a checkpoint mechanism for REPFRAME. Each checkpoint is associated with the index of the last executed socket operation, so that REPFRAME can consistently match up with the execution states of native executions and various analysis executions.

To perform checkpoints transparently without affecting application's executions and analyses, REPFRAME leverages CRIU [12], a popular, open source process checkpoint tool that supports CPU registers, memory, etc. Each checkpoint operation is only performed on the server program and the DMT scheduler; the PAXOS consensus component does not require checkpoints because we explicitly design it to be stateless (all socket operations have been persistently logged).

REPFRAME provides two functions, one for checkpoint and the other for rollback. When an analysis tool calls the `checkpoint()` function, REPFRAME uses its PAXOS consensus component to invoke an operation: "take a checkpoint associated with the global index of the latest socket operation". Once a consensus is reached, all replicas (including primary) do a checkpoint operation as the next consensus operation.

When the tool catches a malicious event and decides to roll back to a previous checkpoint associated with a socket operation index, the tool calls a `rollback(int index=-1, bool discard=true)` function. Ignoring the arguments in this function means rolling back to the last checkpoint within all replicas and discard all inputs since this checkpoint. This API works as three steps: (1) current replica uses the PAXOS consensus component to request an operation: "rollback to a previous checkpoint according to `index`"; (2) once consensus is reached, each replica invokes CRIU to do the actual rollback operation; (3) if `discard` is on, each replica discards all future inputs since the checkpoint. Overall, this API makes all replicas rollback and discard malicious inputs consistently.

REPFRAME's checkpoint mechanism has a few limitations: (1) although this mechanism's API is expressive, they moderately trade off transparency between an analysis tool and REPFRAME's framework; (2) it may defer the processing of network requests; and (3) it can not revoke information that have already leaked to clients. However, considering the benefits (§3.2.2) that REPFRAME may bring to tools, we argue that this checkpoint mechanism is still worthwhile.

3.2 Discussion

3.2.1 What Types of Analyses are Suitable to Run in REPFRAME?

We envision that an analysis can transparently run in REPFRAME if: (1) it can run asynchronously with the actual execution, and (2) it does not schedule conflicting order of Pthreads synchronizations that REPFRAME's DMT runtime enforces. Many reliability analyses (e.g., data race detection), profiling, and logging tools meet this requirement.

Some synchronous security analysis such as control flow integrity can not transparently run in REPFRAME because it monitors the execution synchronously for each control flow transfer. In addition, some other security analysis such as information leakage protection can not transparently run in REPFRAME because the actual execution may probably run faster than the analysis and leak information to clients before the analysis detects the leakage. Nevertheless, many security tools such as use-after-free vulnerabilities can be transparently deployed in REPFRAME. If a synchronous analysis tool would like to run in REPFRAME, it should use REPFRAME's checkpoint and rollback APIs (§3.1).

3.2.2 Can REPFRAME Strengthen Existing Analyses?

REPFRAME has the potential to strengthen existing analyses via its performance benefit. Previously, to mitigate the huge performance slowdown, some analysis tool developers sometimes have to weaken the guarantees of an analysis. For instance, ThreadSanitizer[41], one of the most practical race detectors, only logs the last four accesses for each memory byte instead of the complete access history, because logging and analyzing the complete history is quite heavyweight (e.g., 20X+ slowdown for some testing programs the authors evaluated). However, an incomplete access history may miss data races at runtime. With REPFRAME, developers can now strengthen the analysis by logging complete memory access history.

REPFRAME also has the potential to speedup existing analysis tools themselves via its replication architecture. For instance, the high logging overhead for failure reproductions on multi-processors is an open research problem [44]. Fortunately, with REPFRAME's architecture, these logging tools now can separate different logging aspects such as functions, libraries, and threads into different replicas (e.g., each replica logs for only one thread), greatly reducing the logging overhead. Sampling-based race detection tools (e.g., [18]) can also greatly reduce sampling overhead by dividing sampling work into replicas, or can significantly multiply sampling rates by running the same tool among replicas.

Moreover, REPFRAME provides an extra fault-

tolerance benefit for analysis tools. Existing reliability and security analyses tend to expose or detect harmful defeats in an application, which may crash the application. In addition, the actual executions across different nodes may fail as well due to hardware or OS failures. With REPFRAME's SMR architecture, the sequence of inputs are persistently and consistently enforced across replicas, and failing minor nodes (either an analysis or an actual execution node) do not affect the other nodes' executions and analyses.

3.2.3 Can Existing Analyses Benefit REPFRAME?

Interestingly, analyses running in REPFRAME can benefit REPFRAME's DMT runtime and improve REPFRAME's performance. Existing typical DMT systems use two approaches to enforce schedules. First, a DMT approach enforcing a total order order for shared memory accesses (for short, mem-schedules) is fully deterministic even with data races, but this approach incurs prohibitive slowdown. For instance, DTHREADS [30] incurs several times slowdown for many programs because typical applications have intensive amounts of shared memory accesses.

The other DMT approach is enforcing a total order for synchronizations only (for short, sync-schedules). This approach can be efficient because most code is not synchronization and can still run in parallel. For instance, Kendo, TERN, and PEREGRINE [13, 15, 36] incurred less than 16% overhead on most programs they evaluated. However, this approach is only deterministic when no races occur. Overall, despite much effort [2, 14, 15, 30], an open problem still exists: people lack a simple and deployable approach to enforce fully deterministic and efficient DMT schedules.

Fortunately, REPFRAME's replication architecture can address this problem by simply running a race detector in one replica, then REPFRAME can choose the faster DMT approach that enforces sync-schedules. If the race detector reports a race, application developers can easily diagnose and fix it because synchronization schedules enforced by DMT already make races easily reproducible [37]. By enforcing the same synchronization schedules across replicas, race detection results consistently hold for all replicas. In sum, REPFRAME and race detection tools form a mutually beneficial eco-system.

4 Evaluation

We evaluated REPFRAME on `ClamAV` [11], a popular anti-virus scanning server that scans files in parallel and deletes malicious ones. Our evaluation was done on a set of three Linux 3.2.14 machines within a 1Gbps bandwidth LAN, and each machine has 2.80 GHz dual-socket hex-core Intel Xeon with 24 hyper-threading cores and 64GB memory. To evaluate performance, we used `ClamAV`'s own client utility `clamdscan` to ask the `ClamAV` server to scan its own source code and installation directories, and we measured `clamdscan`'s execution time as the server's response time. The `ClamAV` server spawned eight threads to scan these directories in parallel. To avoid network latency, `clamdscan` was ran within the primary node. Adding network latency will mask REPFRAME's overhead. Each data point was ran for five times and we picked the median value.

To evaluate whether REPFRAME can transparently run analysis tools, we selected two popular tools: one heavyweight tool, the Helgrind race detector [34]; and one lightweight tool, DynamoRio's code coverage tool `drcov` [8].

Table 1 shows the performance results running `ClamAV` in REPFRAME. REPFRAME's bare replication framework incurred negligible overhead over the native execution. When running Helgrind only in one replica, REPFRAME incurred 5.8% overhead over the native execution, because both the other replica node and the primary ran the native executions and they reached consensus fast. When running the `drcov` tool only or running both tools, REPFRAME incurred only 2.1% to 3.6% overhead. This overhead is low because once the `drcov` tool reached consensus with the primary on the `clamdscan` utility's request, the primary just processed the request and responsed to `clamdscan` regardless of `drcov`'s execution. Considering performance jitters, we viewed REPFRAME's results running with one or both tools equal.

The `drcov` tool itself incurred a moderate 28.3% overhead compared to native execution, while Helgrind itself incurred a 40.4X slowdown. Table 1's results show that REPFRAME's replication architecture masked the huge performance slowdown of Helgrind, then `clamdscan` felt that `ClamAV` ran efficiently even both the two tools were turned on.

Note that Helgrind is carried in Valgrind, a traditional analysis framework that takes the "fully-coupled" approach. So does `drcov` for the DynamoRio analysis framework. Overall, this evaluation shows that REPFRAME is efficient, transparent to the Helgrind and `drcov` tools, and complementary to traditional analysis frameworks Valgrind and DynamoRio. In addition, to the best of our knowledge, REPFRAME's evaluation is the first one that has shown to run different types of

Approach	Response time (ms)
Native execution	991
REPFRAME bare framework only	992
REPFRAME with Helgrind	1,048
REPFRAME with drcov	1,036
REPFRAME with Helgrind and drcov	1,012
Helgrind only	41,018
drcov only	1,272

Table 1: *`ClamAV`'s performance running in* **REPFRAME.**

analysis tools together.

5 Related Work

Existing analysis frameworks can be classified into two approaches depending on how a framework exposes an application's actual execution states to an analysis tool. The first approach in-lines an analysis within an actual execution [8, 31, 34, 38, 41], which may cause prohibitive slowdown in the executions when an analysis does heavyweight work. Recently, researchers have proposed to decouple analysis from execution [10, 20, 22, 35, 42, 43] via executing analyses in parallel, record-replay, and so on. We classify these frameworks into "partially-decoupled" approach because they still need to frequently transfer execution states from the execution (e.g., effective memory addresses and thread interleavings) to analysis. These "partially-decoupled" frameworks have shown 4X$\sim$8X speedup over the traditional frameworks on the same analysis, which shows that decoupling analysis from execution is a promising direction. REPFRAME fully decouples analysis from execution via replicating equivalent executions.

SMR has been studied by the literature for decades, and it is recognized by both industry and academia as a powerful fault-tolerance technique in clouds and distributed systems [27, 40]. As a common standard, SMR uses PAXOS [28] as the consensus protocol to ensure that all nodes see the same input request sequence. In this standard, nodes first "agree" on a total order of input request as a input sequence, and then "execute" the requests that have reached this consensus. This typical SMR approach is called "agree-execute". SMR systems, including Chubby [9], ZooKeeper [1], and the Microsoft PAXOS [28] implementation, have been widely used to maintain critical distributed systems configurations (e.g., group leaders, distributed locks, and storage meta data). SMR has also been applied broadly to build various highly available services, including storage [39] and wide-area network [32]. Hypervisor-based Fault Tolerance [7] leverages a hypervisor to build a primary-back system for single-core machines. These systems focus on specific types of applications (e.g., distributed lock service [9], file system [1]) and thus they are not designed to be transparent to general multithreaded applications.

In order to support multithreading in SMR, Eve [23] introduces a new "execute-verify" approach: it first executes a batch of requests speculatively, and then verifies whether these requests have conflicts (e.g., different thread interleavings) that cause execution state divergence. If conflicts occur, Eve rolls back. Eve's execution divergence verification requires developers to manually annotate all shared states in application code, thus it is not transparent to applications.

Rex [19] addresses the thread interleaving divergence problem with a "execute-agree-follow" approach: it first records thread interleavings on the primary node by executing requests, and then replays these interleavings on the other backups. Rex requires frequently shipping thread interleavings across nodes, which may be slow. Furthermore, Rex requires application developers to build their own checkpoint-restore mechanism, thus this mechanism is not transparent to applications.

6 Conclusion

We have presented REPFRAME, an efficient, transparent dynamic program analysis framework. It leverages the transparent state machine replication technique to construct multiple equivalent executions in replicas, so that actual executions and analyses can be fully decoupled. Evaluation shows that REPFRAME can transparently run analyses on replicas with reasonable overhead. Furthermore, analyses and REPFRAME can be mutually beneficial. REPFRAME has the potential to promote the deployments of powerful analyses in applications' production runs.

Acknowledgments

We thank Jian Ouyang (our shepherd) and anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

References

[1] ZooKeeper. `http://hadoop.apache.org/zookeeper/`.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[5] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[7] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[8] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004. Supervisor-Amarasinghe, Saman.

[9] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, June 2008.

[11] Clam AntiVirus. `http://www.clamav.net/`.

[12] criu. Criu. `http://criu.org`, 2015.

[13] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[14] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[15] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[16] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[17] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[19] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multicore. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[20] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, 2009.

[21] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[22] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2013.

[23] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[24] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 155–166, 2010.

[25] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[26] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21 (7):558–565, 1978.

[28] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[29] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[30] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[31] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, 2005.

[32] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[33] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[34] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[35] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 308–318, 2008.

[36] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[37] 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 (SOSP '09)*, pages 177–192, Oct. 2009.

[38] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[39] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, Jan. 2011. ISSN 2150-8097. 10.1145/2797022.279703310.14778/1938545.1938549. URL `http://dx.doi.org/10.14778/1938545.1938549`.

[40] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[41] tsan. Threadsanitizer. `https://code.google.com/p/data-race-test/wiki/ThreadSanitizer`, 2015.

[42] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.

[43] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing data race detection. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 27–38, Mar. 2013.

[44] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 143–154, Mar. 2010.