Test Execution Checkpointing for Web Applications

Marco Guarnieri
Department of Computer Science
ETH Zurich, Switzerland
marco.guarnieri@inf.ethz.ch

Petar Tsankov
Department of Computer Science
ETH Zurich, Switzerland
ptsankov@inf.ethz.ch

Tristan Buchs
Swiss Finance Institute
EPLF, Switzerland
tristan.buchs@epfl.ch

Mohammad Torabi Dashti
Department of Computer Science
ETH Zurich, Switzerland
torabidm@inf.ethz.ch

David Basin
Department of Computer Science
ETH Zurich, Switzerland
basin@inf.ethz.ch

ABSTRACT

Test isolation is a prerequisite for the correct execution of test suites on web applications. We present Test Execution Checkpointing, a method for efficient test isolation. Our method instruments web applications to support checkpointing and exploits this support to isolate and optimize tests. We have implemented and evaluated this method on five popular PHP web applications. The results show that our method not only provides test isolation essentially for free, it also reduces testing time by 44% on average.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Test Execution, Checkpointing, Web Applications

ACM Reference format:
Marco Guarnieri, Petar Tsankov, Tristan Buchs, Mohammad Torabi Dashti, and David Basin. 2017. Test Execution Checkpointing for Web Applications. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092710

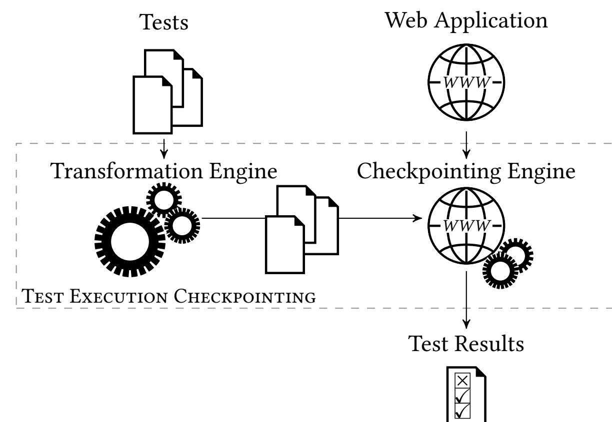

Figure 1: An overview of Test Execution Checkpointing.

1 INTRODUCTION

Testing is a widespread quality assurance method for web applications [2, 6, 7, 33, 42]. A web application test consists of a sequence of HTTP requests. These sequences often have side effects, where executing one test influences the results of subsequent tests by changing the web application's session variables, its databases, and so forth. These side effects can alter the test verdicts, introducing false positives and false negatives [34, 38, 48, 53]. Test isolation prevents these side effects and is thus imperative in practice.

Isolating web application tests is challenging. Straightforward *state-restore* approaches, such as resetting the application's state between successive test executions, are prohibitively expensive. This is not surprising because resetting a real-world web application requires restoring its database state, its session variables and session files, and restarting the web server. Alternatively, test engineers can manually modify tests to enforce isolation. But *test modification* is not only error prone [38, 53], it must be repeated each time the application is modified. In short, state-restore approaches are universally applicable and can be automated, but are inefficient. In contrast, test modification approaches can be efficient but are application-dependent and largely manual.

In this paper, we propose *Test Execution Checkpointing* (TEC), a test execution method for web applications that achieves the best of the two aforementioned approaches. Namely, it provides automatic test isolation in an efficient and application-independent manner. It allows test engineers to focus on writing tests and delegate isolation entirely to the execution engine. Our method's key ingredient is to extend test execution engines with checkpointing capabilities to save and restore application states with minimal overhead. Our empirical results show that TEC not only provides test isolation essentially for free, but it significantly reduces testing time.

Test Execution Checkpointing. TEC is an automated method, which relies both on state-restoring and test modification, for isolating and optimizing tests. The inputs to TEC are a web application and a test suite, i.e., a set of tests. Our solution, illustrated in Figure 1, consists of two main components: a *Checkpointing Engine* and a *Transformation Engine*. The checkpointing engine instruments 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.
ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092710

web application to support efficient checkpointing, which is a technique for saving a snapshot of an application's state and restoring the state from a previously saved snapshot.

The transformation engine modifies the tests in an automated and provably correct manner. The modifications, which amount to adding *save state* and *restore state* commands to the tests, are application-independent. The tests are then executed sequentially on the instrumented application. The modifications guarantee test isolation: in its simplest form, the application's initial state is saved and each test is modified by adding a command to *restore the initial state*. This reinitializes the application before executing each test. In contrast to the aforementioned straightforward state-restore approach, our tailored checkpointing engine renders the reinitialization overhead negligible. In addition to isolation, we also reduce testing time significantly by carefully choosing where *save* and *restore* commands are added to the tests. Web application tests, i.e., sequences of HTTP requests, typically have shared prefixes. By simply saving the application's state after a test prefix is executed and restoring the application to that state, we avoid re-executing the prefix. Our transformation engine automatically detects the longest shared test prefixes and applies the corresponding test modifications. We prove that the modifications are optimal.

Contributions. We propose Test Execution Checkpointing, a method for automatically isolating, optimizing, and executing web application tests. We also develop WebCheck, a tool that instantiates TEC for PHP web applications, and use it to evaluate TEC's benefits on five popular PHP applications.

We develop an efficient and scalable checkpointing engine tailored towards web applications. Our engine checkpoints session and static variables, databases, accessed files, random seeds, and timestamps. This is sufficient to correctly checkpoint a large class of PHP web applications, as we demonstrate in our experiments. Our engine efficiently checkpoints web applications, taking less than 2 milliseconds on average. This enables WebCheck to isolate tests almost for free, with less than 2% average overhead.

We also propose an algorithm that isolates and optimizes tests by exploiting the features provided by our checkpointing engine. The algorithm is the heart of the transformation engine in Figure 1. We prove that (1) our algorithm isolates the tests, and (2) it is optimal, meaning that there is no application-independent test transformation that achieves isolation and results in tests with a smaller number of HTTP requests. Using our algorithm, WebCheck significantly reduces the tests size and reduces testing time by 44%.

Scope and Limitations. TEC is applicable to server-side tests, i.e., tests that consist of sequences of HTTP requests. However, it applies neither to client-side code, such as Javascript, nor to web applications not supported by our checkpointing engine. In more detail, our checkpointing engine supports web applications that (1) store state in session variables, static variables, and persistent storage, such as files and databases, and (2) access random values and timestamps using standard APIs. We discuss the limitations of our checkpointing engine in §3.2.

Organization. In §2 we explain the importance of isolation in web application testing. In §3 and §4 we describe our checkpointing technique for web applications and our test transformation

t_1 `login.php?username=alice&password=1234,`
`add.php?id=1&shipping=standard`

t_2 `login.php?username=alice&password=1234,`
`edit.php?id=1&shipping=overnight`

(a) Tests

```
1 <?
2   include(check_login.php);
3   $order = $db.getOrder($_GET["id"]);
4   $order->shipping = $_GET["shipping"]; //fault
5   $db.insertOrder($order);
6 ?>
```

(b) edit.php

Figure 2: The two test cases for our web application for managing client orders, and the fauly code snippet of `edit.php`.

algorithm. We present and empirically evaluate WebCheck in §5. Finally, we discuss related work in §6 and we conclude in §7.

2 OVERVIEW

In this section, we first present an example and illustrate the problem of test isolation in web application testing. Afterwards, we overview TEC and illustrate how it isolates and optimizes tests.

2.1 Motivating Example

Consider a simple web application for managing client orders, which stores all orders in a database. Figure 2a shows two tests that exercise the application's functionality for adding and editing orders. Each test is a sequence of HTTP requests. Both tests first log in the user by passing the username `alice` and the password `1234` as parameters to the web page `login.php`. Afterwards, the first test adds an order with identifier 1 and shipping type `standard`, whereas the second test changes the shipping type of the order with identifier 1 to `overnight`.

The PHP implementation of `edit.php` is given in Figure 2b. Clients pass the order identifier and the shipping type as parameters. The web page retrieves the order object using the `getOrder` method, updates the shipping type, and commits the updated order to the database. The `getOrder` method returns `null` if an order with the given identifier does not exist in the database. Therefore, this web page contains a null-pointer dereference error at line 4. The web application throws an error when it dereferences the field `shipping` whenever the value of the variable `order` is `null`.

2.2 Test Isolation

Test isolation guarantees that the test results are identical to those obtained when executing each test with the web application starting in its initial state. Therefore, the results of the isolated tests are independent of the other tests, the order in which the tests are executed, and the number of times the tests are executed.

Web application testing, however, does not guarantee test isolation. Standard test execution practice is to (1) start the web application, (2) for each test, sequentially send the HTTP requests to the web server, and (3) observe the results of the respective computations. We call this approach *standard test execution*. In contrast

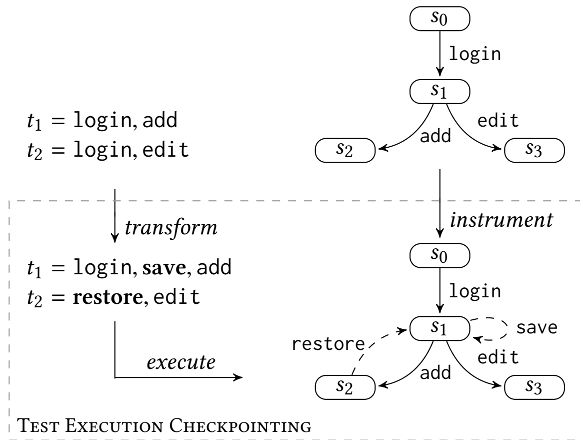

Figure 3: An overview of TEC. The test execution engine takes as input a test suite (top left) and a web application (top right), which we abstractly depict as a transition system. The transformation engine isolates the tests (bottom left) by adding `save` and `restore` requests and optimizes them by removing shared prefixes. The checkpointing engine instruments the web application (bottom right) to support checkpointing (depicted using dashed arrows).

to traditional software testing, the web application is not restarted between tests as the time required for this is usually prohibitive. Tests, however, often have persistent side effects, such as changes to session variables and database content. These side effects may influence the execution of the remaining tests, resulting in both false positives and false negatives [4, 38, 53].

To illustrate, consider the two tests t_1 and t_2 in Figure 2a. Both tests have side effects: t_1 adds an order entry to the database, and t_2 modifies an order entry in the database. Executing the test t_1 followed by t_2 results in a false negative, because t_1 adds an order with the identifier 1 and masks the null-pointer dereference error in `edit.php`. However, we detect the aforementioned pointer dereference error if the tests are executed in isolation.

Resetting a web application to its initial state is, unfortunately, non-trivial. In our example, we would need to restore the database and session variables to their initial states. This can be prohibitively time consuming: our experiments in §5 show that reinitializing a web application may take several seconds, even for very small databases. This is a significant delay since, in comparison, an HTTP request can be executed in a few milliseconds. We remark that the time required to reinitialize the web application amounts, on average, to 88% of the test execution time; see §5.

2.3 Test Execution Checkpointing

As illustrated above, web application testing may result both in false positives and false negatives due to the lack of test isolation. In the following, we describe how we address this problem using TEC. Our method relies on two key components: (1) a checkpointing engine, and (2) a transformation engine. The two components and the main steps performed by TEC are illustrated in Figure 3.

Checkpointing Engine. The checkpointing engine instruments the application, augmenting its functionality with methods for saving and restoring checkpoints, which are invoked using two designated HTTP requests `save` and `restore`. We illustrate the instrumentation in Figure 3. The web application is abstractly depicted as a transition system with an initial state s_0. The transitions are the HTTP requests accepted by the application. For example, the application changes its state from s_0 to s_1 upon receiving the HTTP request `login`. The instrumentation adds new events with associated transitions to existing states. In Figure 3, we depict two relevant transitions labeled with the designated HTTP requests `save` and `restore`. Executing the HTTP request `save` saves the state s_1, and executing the HTTP request `restore` restores the application's state back to s_1.

Transformation Engine. The transformation engine isolates and optimizes test suites. The modified tests rely on the checkpointing features of the instrumented application, as we illustrate below.

A straightforward, but inefficient, way to isolate the tests is to simply restore the web application to its initial state before executing each test. But one can do better. In particular, web application tests often share test prefixes, and the execution of such prefixes usually dominates the overall test execution time; see our experimental results in §5. For example, the two tests t_1 and t_2 both start with the HTTP request `login`. The execution of login steps is often time consuming as it requires database queries to check the user's credentials as well as cryptographic operations. By avoiding the re-execution of such shared prefixes, we can significantly reduce the overall test execution time.

Our test execution engine avoids re-executing a shared prefix by checkpointing the web application's state immediately after executing the shared prefix. To illustrate, consider the two tests t_1 and t_2. The transformation first inserts the HTTP request `save` in the test t_1 after the HTTP request `login` to save the application's state. Afterwards, the transformation replaces the shared prefix in the test t_2 with the HTTP request `restore`, which restores the application to the state s_1 and thereby avoids executing the HTTP request `login`. Executing the tests in this way is equivalent to executing both t_1 and t_2 starting in the application's initial state, and the HTTP request `login` is executed just once.

3 CHECKPOINTING WEB APPLICATIONS

We present here our technique for checkpointing web applications.

3.1 Web Application States

We consider the server-side of web applications. A web application stores state in its *data layer*, which consists of session variables and persistent storage such as databases and files. Most web applications are stateful as they modify their data layer when processing HTTP requests. For example, the web page `edit.php` in our motivating example updates the database. Furthermore, web applications often use random values and timestamps, for instance to generate cookies and session identifiers. These values influence the web application's executions [36] and are also part of the web application's state.

3.2 Checkpointing Engine

Our checkpointing engine, depicted in gray in Figure 4, intercepts all HTTP requests sent to the web application as well as all communication between the application and its data layer. The engine

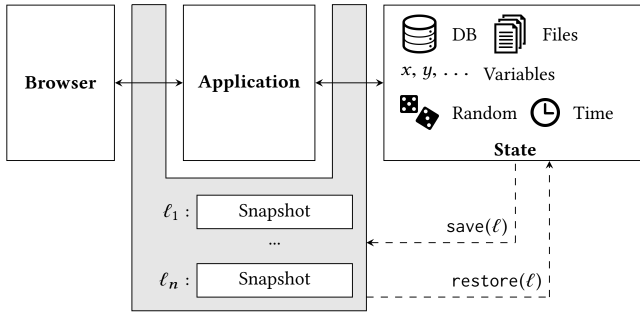

Figure 4: Checkpointing engine for web applications.

augments the web application with support for two additional HTTP requests, namely save(ℓ) and restore(ℓ), which are used to save and restore snapshots of the web application's state. Upon receiving a save(ℓ) request, the checkpointing engine saves the web application's state and labels it with ℓ, e.g., a natural number. This label can later be used to refer to the saved data. Upon receiving a restore(ℓ) request, the checkpointing engine restores the web application's state labeled with ℓ. All HTTP requests except the designated save(ℓ) and restore(ℓ) requests are forwarded to the web application intact. Below we describe how the checkpointing engine saves a snapshot of the state's main components.

Database. We leverage the transaction mechanisms provided by SQL databases to efficiently save the database state, since storing a complete snapshot of the database is impractical. Specifically, the checkpointing engine creates a single SQL transaction with the database and forwards all SQL commands issued by the web application to this transaction. Effectively, the checkpointing engine unifies all SQL connections used by the web application into a single database transaction. Then, to save the database state, the checkpointing engine creates a transaction savepoint using the SQL command SAVEPOINT ℓ. The checkpoint engine uses the SQL command ROLLBACK TO SAVEPOINT ℓ to rollback the transaction to the savepoint labeled with ℓ, without aborting the transaction.

Session and Static Variables. The values stored in session variables persist across HTTP requests and the checkpointing engine saves and restores them. The concrete mechanism for capturing the values of session variables depends on the language used to implement the web application. In PHP, for example, session variables are stored in local files and their values are easily accessed by saving and restoring copies of the relevant files.

Static variables may also persist across HTTP requests and therefore must be saved and restored. In JSP web applications, for example, they are persistent and can be handled using lightweight virtualization containers, such as [4]. In other languages, such as PHP, static variables do not preserve values across requests and need not be checkpointed.

Files. Web applications may access local files. As files are accessed through standard APIs, it is sufficient to monitor the calls to the relevant APIs to detect the concrete paths of the files accessed at run time. The size of such files is usually small, and for most practical examples it is sufficient to store a complete copy of the accessed files. To efficiently checkpoint large files, one can just store file differences rather than complete files.

Random values and Time. Both random values and time are accessed through standard APIs. The checkpointing engine instruments all calls to these APIs to control their return values, similarly to [36]. The checkpointing engine defines an initial seed *seed* and timestamp *time*, and keeps two counters c_{seed} and c_{time}. Each call to the time API returns $time + c_{time}$ and increments c_{time}. Similarly, each call to the random API returns $rng^{c_{seed}}(seed)$, where *rng* is a random number generator initialized with *seed*, and increments c_{seed}. Checkpointing random values and time then amounts to saving and restoring the values of *seed*, *time*, c_{seed}, and c_{time}. We refer the reader to §5.1 for our concrete implementation for PHP.

Limitations. In general, a web application may store state-relevant values in other components, such as remote web services. To handle such applications, one must either checkpoint these additional components or reinitialize them programmatically inside the tests. For instance, for multi-threaded web applications, the scheduler's state, which is part of the web application's state, can be checkpointed using techniques similar to [13]. In practice, however, most web applications store their state in the components described above. We support this claim empirically with our experiments; see §5.

4 TRANSFORMATION ENGINE

Our transformation engine modifies test suites by adding the save and restore requests in the tests and removing redundant test prefixes. The modified tests are guaranteed to be executed in isolation by sequentially sending all HTTP requests contained in the tests to the instrumented web application. In §4.1 we describe the algorithm implemented in our transformation engine, and in §4.2 we prove that it achieves test isolation and is optimal.

4.1 Transforming Test Suites

The transformation engine takes a set of tests T as input and outputs a sequence of tests T' that are isolated and optimized. We remark that the order of the tests in T' is essential both for achieving isolation and for the well-formedness of the tests, e.g., a save request should be executed before the corresponding restore request.

A simple way to isolate tests using checkpointing is to save the web application's state before executing the first test and then to restore the web application to its initial state before executing each test. This is achieved by prepending the save request to the first test and then prepending the restore request to all remaining tests. This suffices to isolate the tests, but it does not remove the redundant HTTP requests.

In the following, we define an algorithm that isolates the tests *and* optimizes them by removing shared test prefixes.

Background. We use the following notation to define the algorithm. A test suite is a set $\{t_1, \ldots, t_n\}$ of tests. Each test t_i consists of a sequence $[q_1^i, \ldots, q_{m_i}^i]$ of HTTP requests. Given two tests $t = [q_1, \ldots, q_n]$ and $t' = [q_1', \ldots, q_m']$, we write $t \cdot t'$ for the test $[q_1, \ldots, q_n, q_1', \ldots, q_m']$ obtained by appending t' to t.

Without loss of generality, we assume that each web application has a designated initial state. Thus, executing a test suite on a web application amounts to executing the test suite from the initial state.

Our algorithm uses *prefix trees* [28]. A prefix tree (also called a trie) is a tree that compactly stores strings over an alphabet Σ. We represent a prefix tree as a pair (N, E) where N is a set of nodes and

E is a set of labeled edges. Each edge is of the form (n_1, q, n_2), where n_1 and n_2 are nodes in N and $q \in \Sigma$ is a character in the alphabet. Each node represents the string obtained by concatenating the characters along the path from the root to the node. In our setting, the alphabet consists of HTTP requests.

Our algorithm constructs a prefix tree over the HTTP requests in the test suite provided as input. To illustrate, consider the test suite $T = \{t_1, t_2, t_3\}$, where $t_1 = [a, b, c, d]$, $t_2 = [a, b, c, e]$, and $t_3 = [a, f]$. The test suite T and its corresponding prefix tree are shown in Figure 5. Each path in the prefix tree represents a test. We denote the prefix tree derived from a test suite T by $\mathit{prefixTree}(T)$.

Algorithm. Algorithm 1 takes a test suite T as input and outputs a sequence of tests T'. The algorithm first constructs a prefix tree from the input test suite T. It then iteratively constructs the optimized test suite by traversing the prefix tree in a depth-first manner. The stack S, constructed during the depth-first traversal, contains pairs of the form (n, t), where n is a node in the prefix tree and t is a test, whereas T' contains the optimized tests. Initially, the stack S contains a pair $(n_0, [])$, where n_0 is the prefix tree's root and $[]$ stands for the empty sequence. Each iteration of the while loop processes one node in the prefix tree as well as the associated partial test. If the node is a leaf node, the test is fully modified and appended to T'. Otherwise, the node is an inner node. The algorithm inserts `save` and `restore` HTTP requests whenever the inner nodes are branching, that is, where tests have a shared prefix. Note that the running time of the algorithm is linear in the number of HTTP request in the test suite.

Without loss of generality, we assume that the test suite does not contain tests that are prefixes of other tests. For example, we do not consider test suites such as $\{[a], [a, b]\}$. These test suites can be supported by appending a designated request end to all tests, e.g., $\{[a, \mathsf{end}], [a, b, \mathsf{end}]\}$.

Example. Consider the test suite given in Figure 5. The figure also depicts how the stack and the generated tests evolve during the execution of Algorithm 1. Initially the stack contains the pair $(n_0, [])$. The first iteration of the while loop follows the second branch (lines 16–18) because n_0 has just one child, namely n_1, which is reachable from n_0 by issuing the HTTP request a. After the first iteration, the stack S contains the pair $(n_1, [a])$, which associates to the node n_1 the partial test a, and the test suite T' is still empty. The second iteration of the while loop executes the if branch (lines 10–15) because n_1 has two children, one reachable using b and the other reachable using f. After the second iteration, the stack S contains the pairs $(n_2, [a, \mathtt{save}(\ell_1), b])$ and $(n_6, [\mathtt{restore}(\ell_1), f])$. In the third iteration, we extend the first partial test by appending the HTTP request c. At the end of the iteration, the stack contains $(n_3, [a, \mathtt{save}(\ell_1), b, c])$ and $(n_6, [\mathtt{restore}(\ell_1), f])$. In the fourth iteration, we execute again the if branch (lines 10–15) because n_3 has two children. We therefore pop $(n_3, [a, \mathtt{save}(\ell_1), b, c])$ from the stack and replace it with $(n_4, [a, \mathtt{save}(\ell_1), b, c, \mathtt{save}(\ell_2), d])$ and $(n_5, [\mathtt{restore}(\ell_2), e])$. We have now visited the entire prefix tree. Therefore, in the last three iterations we collect the modified tests from the stack and append them to T'. Finally, the algorithm returns the tests $[t'_1, t'_2, t'_3]$, where t'_1 is $[a, \mathtt{save}(\ell_1), b, c, \mathtt{save}(\ell_2), d]$, t'_2 is $[\mathtt{restore}(\ell_2), e]$, and t'_3 is $[\mathtt{restore}(\ell_1), f]$.

Algorithm 1: Isolating and optimizing test suites.

```
   Input: A test suite T.
   Output: A sequence of tests T'.
 1 begin
 2   (N, E) := prefixTree(T)
 3   n0 := root(N, E) // n0 is the root of prefixTree(T)
 4   S := [(n0, [])] // S is a stack
 5   c := 0 // c is a counter
 6   T' := []
 7   while S ≠ [] do
 8     (n, t) := pop(S)
 9     O := {(n, q, n2) ∈ E}
10     if |O| > 1 then
11       Pick (n, q, n2) ∈ O
12       c := c + 1
13       for (n, q', n2') ∈ (O \ {(n, q, n2)}) do
14         push(S, (n2', [restore(ℓc), q']))
15       push(S, (n2, t · [save(ℓc), q]))
16     else if |O| = 1 then
17       Pick (n, q, n2) ∈ O
18       push(S, (n2, t · [q]))
19     else if |O| = 0 then
20       T' := T' · [t]
21   return T'
```

Test Oracles. Test oracles are used to determine whether tests pass or fail. Our transformation engine guarantees that the results of state assertions (i.e., oracles that check a property of a web application's state) are the same for both the transformed tests and the original tests, provided the latter are properly isolated. Furthermore, navigation assertions (e.g., checking whether a web page is accessible after the user has logged in) can be also easily checked as one can construct the original traces from those observed when executing the transformed tests. Note that oracles that do not depend on the web application's state may not be preserved by our transformation. For example, oracles checking non-functional requirements, such as performances, are, in general, not preserved by our transformation.

Integration with Other Techniques. Our transformation engine can be directly integrated with test selection techniques by first selecting a subset of the tests and then applying our transformation. Our algorithm currently does not support test prioritization, since the latter orders the tests while our algorithm takes as input a set. We leave this extension as future work.

4.2 Transformation Engine Correctness

We first formalize test isolation and optimality. We then prove that the output of Algorithm 1 satisfies these two properties. We start with several definitions.

Without loss of generality, we model a *web application* as a deterministic Labelled Transition Systems (LTS) $\langle S, \Sigma, \delta, s_{\mathsf{init}} \rangle$ where

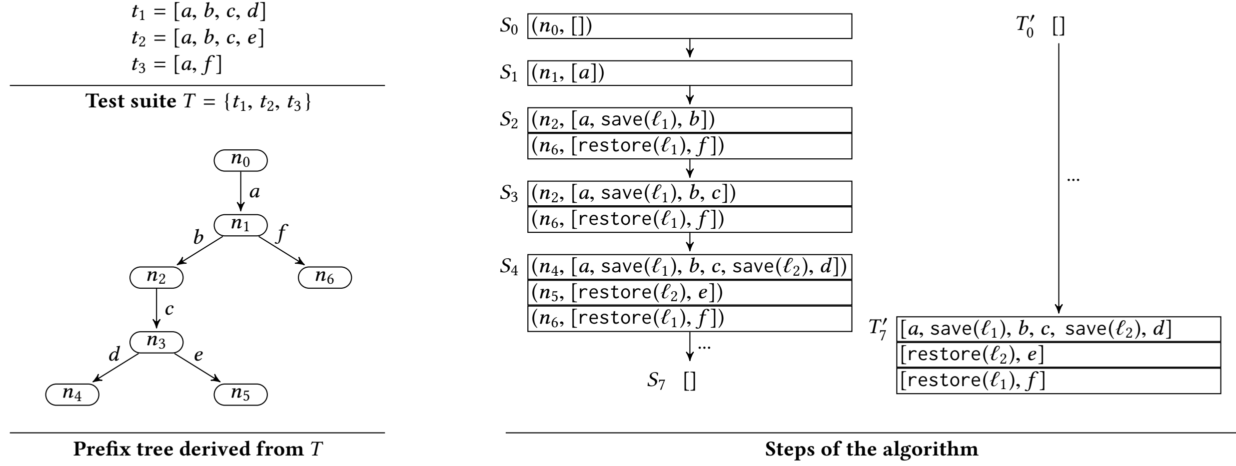

Figure 5: A test suite and its corresponding prefix tree (left) and the evolution of the stack and the generated tests (right). Each S_i and T'_i correspond to the state of the stack and, respectively, generated tests, at the i-th loop iteration of Algorithm 1.

Domains

n, m	$\in$	N	Set of nodes
ℓ	$\in$	L	Set of labels
μ	$\in$	M	Set of maps from N to L
q	$\in$	Σ	Set of requests
e	$\in$	$E \subseteq (\Sigma \cup \{\mathtt{save}(\ell), \mathtt{restore}(\ell) \mid \ell \in L\})^*$	Set of extended tests
$tree$	:	$E \to Trees$	
$build$	:	$N \times E \times M \to Trees$	

Tree building

$tree(e) = build(e, n, \emptyset)$

$build(\mathtt{save}(\ell) \cdot e, n, \mu) = build(e, n, \mu[\ell \mapsto n])$

$build(\mathtt{restore}(\ell) \cdot e, n, \mu) = build(e, \mu(\ell), \mu)$

$build([], n, \mu) = $ (n)

$build(q \cdot e, n, \mu) = $ (n) $\xrightarrow{q}$ $build(e, m, \mu)$

Example

$e = [a, \mathtt{save}(\ell), b, d, \mathtt{restore}(\ell), c]$

$tree(e) =$

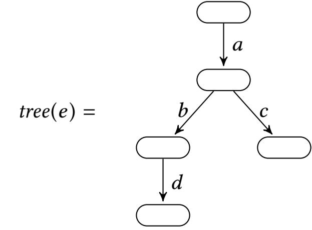

Figure 6: The function *tree* for building trees from extended tests. The nodes n and m are fresh to ensure that the result is a tree.

S is a set of states, Σ is a set of HTTP requests, $\delta \subseteq S \times \Sigma \to S$ is a transition function, and $s_{\mathsf{init}} \in S$ is the initial state.

Let $W = \langle S, \Sigma, \delta, s_{\mathsf{init}} \rangle$ be a web application. A *test* for W is a finite sequence over Σ, and a *test suite* for W is a set of tests for W. A *trace* of W is a sequence of states in S that starts from the initial state s_{init} and respects the transition function δ. Given a web application W and a test t, we write $apply(W, t)$ for the trace obtained by executing t on W. Here $apply(W, t)$ captures the web application's behavior where t is executed from the initial state. Formally, $apply(W, t) = s_0, \ldots, s_{|t|}$ such that $s_0 = s_{\mathsf{init}}$ and $(s_i, q_i, s_{i+1}) \in \delta$ for all $i \in \{0, \ldots, |t| - 1\}$, where $|t|$ is the number of requests in t and q_i is the i-th request in t. We lift the function *apply* to test suites in the standard way: $apply(W, T) = \{apply(W, t) \mid t \in T\}$. Note that $apply(W, T)$ represents the behavior of W where the tests in T are correctly isolated, i.e., each test is executed from the initial state.

To represent instrumented applications and modified tests, we introduce the notion of extended tests, which are tests extended with `save` and `restore` commands. Formally, an *extended test* for W is a finite sequence over $\Sigma \cup \{\mathtt{save}(\ell) \mid \ell \in L\} \cup \{\mathtt{restore}(\ell) \mid \ell \in L\}$, where L is a fixed set of labels, such as the set of natural numbers $\mathbb{N}$. To guarantee well-formedness, we further require that in any extended test e, each $\mathtt{restore}(\ell)$ is preceded by exactly one $\mathtt{save}(\ell)$. Note that any $\mathtt{save}(\ell)$ can be restored multiple times. We remark that the concatenation of the tests output by Algorithm 1 is an extended test.

Next, we define the *apply* function for extended tests. Let e be an extended test with $n \in \mathbb{N}$ restore commands. We first construct a tree of requests from e, where each path in the tree corresponds to a (non-extended) test for W. We then define the *apply* function as the set of $n + 1$ traces obtained from these tests. We formalize this in the following.

We define the function *tree* in Figure 6, which maps an extended test to a directed labelled tree where each edge is labelled with an element of Σ. We illustrate the definition with an example in the figure. The tree induces a test suite, denoted by $T_{tree(e)}$, in the standard manner: each path from the tree's root to a leaf corresponds to a test. Then, we define $apply(W, e)$ as the set $apply(W, T_{tree(e)})$. Note that we have overloaded the symbol *apply* in the above definitions.

Also note that $apply(W, e)$ captures the semantics of the save and restore requests.

We are now ready to define test isolation and optimality. Let W be a web application, T be a set of tests for W, and e be an extended test for W. We say e and T are *semantically equivalent* if $apply(W, T) = apply(W, e)$. A (test) *transformation* τ is a function that maps a set of tests to an extended test. We say τ achieves *test isolation* if, for any web application W and any test suite T for W, $\tau(T)$ is semantically equivalent to T. We denote by $\mathcal{TR}$ the set of transformations that achieve test isolation.

We now turn to optimality. A natural measure of the *cost* of executing a test is the number of HTTP requests it contains. We ignore the save and restore HTTP requests because the time for saving and restoring the state is negligible compared to the time needed to execute the HTTP requests, as our experiments in §5 demonstrate. A transformation $\tau \in \mathcal{TR}$ is therefore *optimal* if for any web application W, any test suite T for W, and any transformation $\tau' \in \mathcal{TR}$, we have $|\tau(T)| \leq |\tau'(T)|$, where $|e|$ denotes the number of HTTP requests in the extended test e.

Theorem 4.1. *Algorithm 1 achieves test isolation and is optimal.*

Proof. Let T be an input to the algorithm and e be the output extended test. Then, $prefixTree(T)$ is identical (up to isomorphism) to $tree(e)$; this can be established by a straightforward induction on the size of the input's prefix tree. Test isolation immediately follows. The optimality of the algorithm is then simply a consequence of the minimality of prefix trees. □

5 EVALUATION

Here, we describe WebCheck and report on the efficiency and scalability of its checkpointing engine, and on the impact of its optimization algorithm on the overall test execution time. We use WebCheck to isolate, optimize, and execute test suites for five popular PHP web applications. Our experiments show that even without optimization WebCheck isolates web application tests with negligible overhead — 2% on average. Using our optimization technique, WebCheck significantly reduces the test execution time — 44% on average — over the standard test execution time. In the following, we first describe WebCheck and our experimental setup, and then we report on our experiments.

5.1 Implementation

To conduct our experiments, we have instantiated TEC for PHP web applications. Our tool, called WebCheck, along with the test suites and all scripts used in our experiments are publicly available.[1] WebCheck implements the checkpointing engine described in §3 and Algorithm 1 given in §4. A preliminary version of WebCheck's checkpointing engine is described in [10].

WebCheck intercepts all HTTP requests sent to the web application and handles the save(ℓ) and restore(ℓ) requests. Furthermore, it instruments all calls to the SQL API. The SQL queries are forwarded to an SQL proxy, which maintains a persistent SQL connection to the database. All SQL queries are thus executed within a persistent SQL transaction, which enables WebCheck to save and restore the database using the standard SQL queries SAVEPOINT ℓ and ROLLBACK TO SAVEPOINT ℓ. All session variables in PHP are stored in files. To save and restore session variables, WebCheck copies and replaces these files. To handle local files, WebCheck instruments file system API calls and makes a copy of all accessed files. Finally, as described in §3, WebCheck instruments the rand, mt_rand, random_int, and uniqid APIs for producing random values, the microtime and time APIs for getting timestamps, and the session_id, session_start, and session_regenerate_id APIs for generating session identifiers. To instrument the APIs, WebCheck uses the PHP extension runkit.

To test a web application using WebCheck, the test engineer provides the URL of the web application, the SQL connection details, and the path to the web application's test suite. WebCheck automatically retrieves the path to the web application's session files using the PHP API. WebCheck then transforms the test suite as described in Algorithm 1, and sequentially executes the transformed tests on the web application.

[1] See http://www.infsec.ethz.ch/research/software/webcheck.html .

5.2 Experimental Setup

As test subjects, we have selected five popular open-source PHP web applications, using MySQL as database backend:

- **phpBB** v.3.1.6 [40]: a bulletin board application.
- **OSCommerce** v2.3.3 [39]: an e-commerce application.
- **WordPress** v.4.3.1 [51]: a content management system.
- **BambooInvoice** v0.8.9 [1]: an invoicing software.
- **Gallery3** v3.0.9 [20]: a photo album organizer.

Our test subjects do not ship with system-wide tests. Therefore, for each test subject, we automatically generated a test suite consisting of 1000 tests. Each test consists of up to 20 HTTP requests, exercising different portions of the application. We generate each test using w3af [49], a state-of-the-art web application crawler that automatically exercises the test subject. For each test subject, we run w3af 1000 times with a one-minute timeout and intercept all the HTTP requests produced by the tool. We then filter the resulting requests to remove duplicates and requests that do not exercise the web application, such as requests that retrieve Javascript files and images. The test is then obtained by selecting the first 20 requests from the filtered sequence of HTTP requests. We have performed all the experiments on a Linux machine, with an i7-4770 CPU, 32GB of RAM, running PHP v5.5.9 and Apache v2.4.7.

5.3 Experiments

In the following, *test execution time* is the total time spent to execute *all* requests, including the save and restore requests, whereas *checkpointing time* refers to the time spent to handle only the save and restore requests.

Checkpointing Engine Correctness. Here we evaluate whether WebCheck correctly checkpoints web applications. To this end, we compare WebCheck's save and restore capabilities with those provided by Virtual Machines (VMs). First, we execute each test suite using WebCheck. Afterwards, we run the web server within a VM, and we execute the test suites again by reinitializing the VM's state before each test. In both cases, we record the HTTP responses and the web application's state after each HTTP request; we refer to these as *outputs*.

We compare the collected outputs. Note that when the web application's state is reinitialized using VMs, the resulting outputs may differ slightly due to variations, such as timestamps. To account for this, we execute each test twice using the VM and we identify those portions of the output that do not change in both executions. We then compare these portions of the VM's output with WebCheck's output. If the outputs are identical, then WebCheck checkpoints our test subjects as correctly as virtual machines do. Using this method, we confirmed that WebCheck correctly checkpoints our test subjects. Along with Theorem 4.1, this guarantees that WebCheck correctly isolates test suites. Therefore, false positives and false negatives due to the lack of test isolation are avoided.

Efficiency. To evaluate the efficiency of our checkpointing engine and the benefits of our optimization algorithm, we measure the test execution time using five different test configurations, summarized in Figure 7. Each configuration defines the test isolation mode and the optimization mode. There are three test isolation modes:

- **Interfering (I)**. There is no test isolation, i.e., the tests are executed without reinitializing the web application's state.
- **Script-based (S)**. We save the web application's database state by dumping it in an SQL script and we restore it by executing the same script. The other components of the web application's state are treated as described in §5.1.
- **Checkpointed (C)**. WebCheck's checkpointing engine is used to isolate the tests.

We did not use VMs for isolating tests as saving and restoring the web server's virtual machine is very expensive (> 10*s*).

As for the test optimization, there are two modes:

- **Unoptimized (U)**. The tests are not optimized, i.e., the tests are just sequentially executed on the web application.
- **Optimized (O)**. The tests are optimized prior to execution using WebCheck's transformation engine, as described in §4. The optimized tests are sequentially executed against the instrumented web application.

A configuration is a pair of letters XY, where $X \in \{I, S, C\}$ and $Y \in \{U, O\}$, indicating the isolation and optimization modes.

Existing checkpointing tools do not fully support web applications; see §6. Therefore, we do not have a canonical baseline to compare our results with. This motivated us to thoroughly compare WebCheck, which corresponds to the configuration *CO*, with four other configurations, namely *IU*, *SU*, *SO*, and *CU*. The configuration *IU* represents the standard test execution, i.e., the currently adopted testing technique, which does not provide test isolation and sequentially executes all the tests. To evaluate the impact of our checkpointing engine, we compare the test execution time of the configurations *SU* and *SO*, which employ a straightforward checkpointing technique based on SQL scripts, with that of the configurations *CU* and *CO*, which use WebCheck's checkpointing engine. Similarly, to evaluate the impact of WebCheck's transformation engine, we compare the test execution time of the configurations *SU* and *CU*, which do not optimize the tests, with that of the configurations *SO* and *CO*, which optimize tests using WebCheck's transformation engine.

For each test subject and test configuration, we measure the time to execute all tests in the subjects' respective test suites, and we depict these results using a bar chart in Figure 8. For each test subject, this figure shows five bars, one for each configuration. The y-axis shows the test execution time in milliseconds using a logarithmic scale. We also measure the checkpointing time in the script-based and the checkpointed modes. These results are given in Figure 9. For each test subject, this figure shows four bars, one for each configuration that achieves test isolation. The y-axis shows the checkpointing time using a logarithmic scale in milliseconds. Note that the configuration *IU* does not isolate test suites and is thus not shown in Figure 9. Also observe that Figure 9 reports only the time to execute the `save` and `restore` commands, while Figure 8 also reports the time to execute the requests.

Configuration	Test Isolation	Optimization
IU	Interfering	Unoptimized
SU	Script-based	Unoptimized
SO	Script-based	Optimized
CU	Checkpointed	Unoptimized
CO	Checkpointed	Optimized

Figure 7: Test configurations.

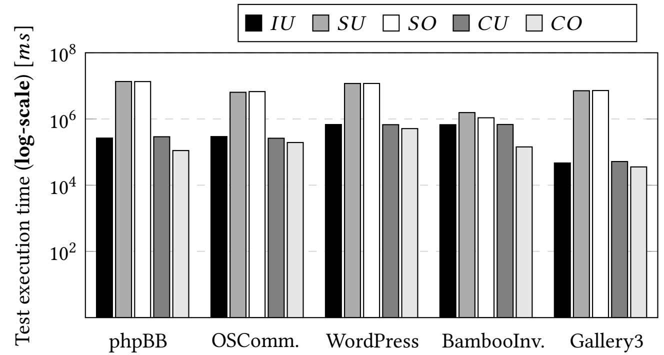

Figure 8: Test suite execution times for each test subject using the five configurations.

Our baseline is the configuration *IU*, which does not achieve test isolation. As expected, achieving test isolation using the straightforward solution based on SQL scripts, namely the configuration *SU*, introduces a significant overhead — from 1.3x for BambooInvoice to 150x for Gallery3, even though the test subjects' databases are relatively small (about 1 MB). For this configuration, most of the checkpointing time is spent for restoring the database state — 7.5 seconds on average per `restore` request — whereas saving the database state takes, on average, less than a millisecond per `save` request. In contrast, as shown by the measurements for the *CU* configuration, our checkpointing engine achieves test isolation for our test subjects almost for free. The overhead is less than 2% on average whereas the checkpointing time is, on average, less than 2 milliseconds per `restore` request. Note that, for the phpBB application, the overall checkpointing time for the *SU* configuration is 3.7 hours, as opposed to 2.7 seconds for the *CU* configuration.

The overall testing time can be further reduced by optimizing the tests using WebCheck's transformation engine, as shown by WebCheck's configuration *CO*. Our transformation engine reduces the test execution time with respect to the *CU* configuration — the

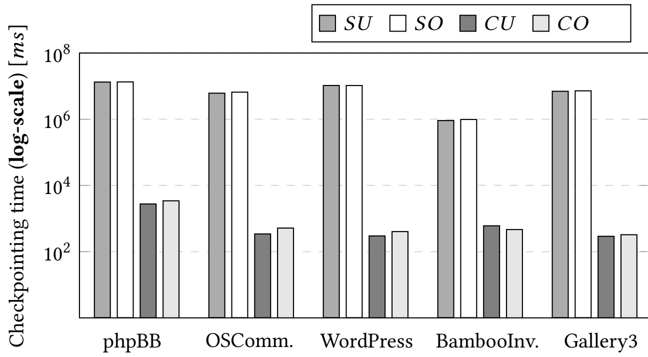

Figure 9: Checkpointing times for each test subject using the configurations *SU*, *SO*, *CU*, and *CO* that use checkpointing.

Test Subject	Checkpoints	% Optimized
phpBB	624	60%
OSCommerce	550	31%
WordPress	419	26%
BambooInvoice	483	35%
Gallery3	79	71%

Figure 10: Total number of checkpoints and average optimization for the test suites.

savings range from 25% for OSCommerce to 78% for BambooInvoice. We remark that WebCheck outperforms also our baseline configuration *IU* — with an average reduction of 44% in the test execution time. Therefore, test engineers can use WebCheck to achieve test isolation almost for free and to reduce the overall test execution time with respect to currently adopted testing techniques. The test transformation time for our subjects is less than 10ms.

Figure 10 describes the number of checkpoints and the reduction in the test size for the optimized test suites, i.e., those generated using WebCheck's transformation engine. Gallery3 is the test subject with the highest optimization rate and with the lowest number of checkpoints. We manually inspected the test suite and discovered that they share long prefixes. In contrast, the number of checkpoints for the other applications is comparable.

Quite surprisingly, combining straightforward script-based checkpointing with our optimization algorithm, namely the *SO* configuration, increases the overall checkpointing time. The reason is that in the *SU* configuration, each test suite contains just one `save` request and 999 `restore` requests, whereas, in the *SO* configuration there is a larger number of `save` requests, as shown in Figure 10, and the same amount of `restore` requests.

Scalability. The size of a web application's state naturally influences the time required to save and restore it. Here, we report how the two modes of test isolation, namely the script-based and the checkpointed modes, scale in the size of the database. We ignore the other components of a web application's state as they have a negligible impact on the checkpointing engine's performance.

We select phpBB for our scalability experiments because it is the test subject with the longest checkpointing time; see Figure 9. We generate 10 database instances of different sizes, ranging between

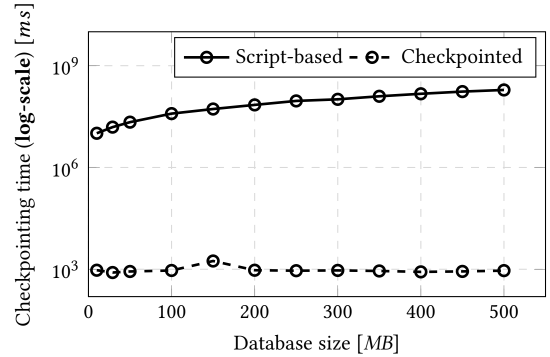

Figure 11: Impact of the database size on the checkpointing time when using the script-based and checkpointed modes of test isolation.

0.5MB and 500MB. These database instances are populated with random data. For each database instance, we run phpBB's test suite and measure the time required to restore a database's snapshot using the script-based and the checkpointed test isolation modes. Note that WebCheck uses the checkpointed test isolation mode.

We depict the results in Figure 11. The script-based test isolation mode is depicted using a solid line and the checkpointed test isolation mode using a dashed line. The y-axis shows the checkpointing time on a logarithmic scale. The results show that while the checkpointing time for the script-based test isolation mode increases linearly in the size of the database, the time for the checkpointed isolation mode remains constant in the database's size. This is because the checkpointed test isolation mode relies on transaction database features. The script-based test isolation mode is impractical even for small database, while the checkpointed test isolation mode scales to arbitrarily large databases.

5.4 Threats to Validity

The main threat to the validity of our study is the choice of the test subjects, which may not be representative of all PHP applications. We believe, however, that our test subjects cover a wide spectrum of common PHP applications. They include widely used bulletin boards and content management systems, as well as data-intensive applications such as e-commerce and invoicing applications.

Another threat to the validity of our results is the nature of the test cases. Although the choice of tests does not impact the correctness results, the savings in the test execution time directly depend on the tests. We used w3af to automatically explore the web application under test. Test suites generated using other techniques, e.g., manually created test suites, may produce different results.

6 RELATED WORK

Test Isolation. Test isolation has attracted considerable attention in recent years [3–5, 21, 30, 34, 48, 53]. Most existing work studies *test dependence*, a property related to test isolation. A test is *dependent* if there is a subset of the original tests that can be reordered to modify the test result [53]. Note that test isolation guarantees test independence. Several empirical studies [30, 34, 48, 53] highlight that dependencies between tests, and missing test isolation

in general, can cause false positive and false negative test results. The lack of isolation can also affect the results of testing techniques that implicitly rely upon it, such as test prioritization, test selection, and test parallelization [4, 5, 30, 38, 53].

As noted by Bell et al. [4], it is common practice to isolate each JUnit test in its own process. This provides test isolation when the tests depend only on static variables; but it can be extremely inefficient: the overhead introduced by isolating tests in this way is about 600%. A similar technique is proposed by Muşlu et al. [38], who execute each test in an isolated environment by restarting the Java Virtual Machine before each test. Their results are consistent with our findings that straightforward state-restore isolation techniques cause considerable overhead. A more efficient checkpointing engine for the in-memory state, i.e., the static variables, of Java applications is implemented in VmVm [4]. This checkpointing engine avoids the re-execution of the `setUp` methods in JUnit tests and achieves up to 60% speedups when compared to traditional test execution. This agrees with our findings that checkpointing engines are essential to efficiently isolate and optimize tests. In contrast to our checkpointing engine, VmVm does not deal with other components of the web application's state, such as databases and session files, and it cannot be used to checkpoint web applications. Furthermore, WebCheck removes all shared test prefixes, as opposed to VmVm which removes only the test setup.

An alternative approach to deal with the test isolation problem is to detect the lack of test isolation and just notify the tester about the possibly unsound results. Zhang et al. [53] prove that detecting test dependencies is NP-complete, even without considering the complexity of test execution, and propose four dependency detection algorithms. Dynamic techniques for detecting dependencies have also been proposed [5, 21]. Gyori et al. [21] present a technique for finding potential dependencies between tests by identifying tests that *pollute* the state shared between different tests, whereas Bell et al. [5] identify dependencies by leveraging data dependencies and anti-dependencies. Instead of detecting the lack of test isolation, TEC transparently provides it.

Test Optimization. Test optimization techniques based on removing redundant test steps without modifying the test suite semantics have been proposed in [15, 19, 27, 55]. Fraser and Wotawa [19] present an optimization algorithm for tests generated using a model-checker. Similarly to WebCheck, their algorithm removes shared test prefixes. They use a model checker to generate *gluing sequences*, which reinitialize the system to a desired state, similarly to our `restore` commands. Gluing sequences, however, are not guaranteed to exist and can be expensive to compute and execute. In contrast, restoring a checkpoint is always possible and can be done in negligible time. The algorithm in [19] requires a formal model of the system, whereas TEC works directly with the system.

Similarly to WebCheck, Khalek et. al. [27] optimize tests by avoiding the re-execution of common prefixes. Their approach relies on manually-defined undo operations. In contrast, WebCheck focuses on web applications, is fully automatic, and is guaranteed to provide test isolation. Devaki et al. [15] present *test merging*, a technique that optimizes tests by merging independent tests that share common steps. Unlike TEC, test merging neither isolates tests nor optimizes dependent tests. Zhongsheng [55] presents an algorithm for optimizing web application tests. The algorithm removes redundant tests, i.e., tests that are prefixes of other tests. In contrast to TEC, it does not remove redundant test prefixes, such as the prefix `login` of the tests [`login`, `add`] and [`login`, `edit`], and it does not provide test isolation.

Test minimization techniques [11, 12, 22, 23, 25, 26, 46, 50, 54] aim to reduce testing time without impacting thoroughness. To do so, they rely on heuristics such as code or requirement coverage. While also reducing testing time, these techniques modify the test suite semantics and may thus reduce its fault detection ability [19, 24, 26, 43]. Moreover, test minimization techniques implicitly assume that tests are correctly isolated. Indeed, as shown in [4, 5, 30, 53], missing isolation can affect the soundness of test minimization techniques. In contrast, TEC guarantees that the optimized tests are semantically equivalent to the original ones and it provides, rather than assumes, test isolation.

Application Checkpointing. Checkpointing [8, 9, 16, 17] is a technique for saving and restoring application states. It was originally developed to extend applications with fault-tolerant features and also for load-balancing by migrating applications between hosts [17]. State-of-the-art general-purpose checkpointing tools, such as [14, 29, 32], are limited to storing local variables, while web applications also store state in session files and databases. Our checkpointing engine supports these features.

Language-specific checkpointing tools have been used to provide test optimization [4], incremental checkpointing [31], recovery techniques [41], or capture and replay capabilities [52]. These tools rely on a mix of static and dynamic analysis for tracking state changes to efficiently save and restore a program states. However, these tools do not support persistent storage, which web applications often use. They can be integrated with our checkpointing engine to extend TEC with support for other languages, e.g., JSP.

Database transactions have been used to isolate unit tests for data-intensive applications and database procedures [18, 35, 37, 47]. WebCheck's checkpointing engine provides a similar functionality for PHP applications. Additionally, it checkpoints the other components of a web application's state, i.e., session variables, local files, randomness, and timestamps.

Capture and replay tools for web applications save and restore the web application's state to faithfully replicate executions. Existing tools [44, 45], however, only account for database and cookies. Furthermore, to checkpoint the database, they rely on inefficient SQL scripts. In contrast, WebCheck captures additional components of a web application's state, such as session files and variables, and exploits transactions to efficiently checkpoint the database.

7 CONCLUSION

Test isolation is essential to ensure correct test results [30, 34, 48, 53]. Test engineers usually adopt straightforward techniques to achieve it [4], thereby incurring significant overheads. We have proposed Test Execution Checkpointing, a novel method that exploits checkpointing to isolate and optimize tests. We have developed WebCheck, a tool that instantiates TEC for PHP web applications. We have shown that WebCheck achieves test isolation with a negligible overhead of 2% on average, and that our optimization technique even reduces test execution time by 44% on average.

REFERENCES

[1] Derek Allard. *Bamboo Invoice*. http://github.com/derekallard/BambooInvoice (Retrieved: January 2016).

[2] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. 2005. Testing web applications by modeling with FSMs. *Software and Systems Modeling* 4, 3 (2005), 326–345.

[3] Jonathan Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 799–802.

[4] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 550–561.

[5] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 770–781.

[6] Francesco Bolis, Angelo Gargantini, Marco Guarnieri, and Eros Magri. 2012. Evolutionary testing of PHP web applications with WETT. In *International Symposium on Search Based Software Engineering*. Springer, 285–291.

[7] Francesco Bolis, Angelo Gargantini, Marco Guarnieri, Eros Magri, and Lorenzo Musto. 2012. Model-driven testing for web applications using abstract state machines. In *Current Trends in Web Engineering: ICWE 2012 International Workshops*. Springer, 71–78.

[8] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 84–94. https://doi.org/10.1145/781498.781513

[9] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. 2004. Application-level Checkpointing for Shared Memory Programs. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 235–247. https://doi.org/10.1145/1024393.1024421

[10] Tristan Buchs. 2015. *Checkpointing-based testing*. Master's thesis. ETH Zurich, Switzerland.

[11] Tsong Yueh Chen and Man Fai Lau. 1998. A new heuristic for test suite reduction. *Information and Software Technology* 40, 5-6 (1998), 347–354.

[12] Tsong Yueh Chen and Man Fai Lau. 1998. A simulation study on some heuristics for test suite reduction. *Information and Software Technology* 40, 13 (1998), 777–787.

[13] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/281035.281041

[14] CRIU. *Checkpoint/Restore In Userspace*. http://criu.org (Retrieved: January 2016).

[15] Pranavadatta Devaki, Suresh Thummalapenta, Nimit Singhania, and Saurabh Sinha. 2013. Efficient and Flexible GUI Test Execution via Test Merging. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 34–44. https://doi.org/10.1145/2483760.2483781

[16] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. 1992. The performance of consistent checkpointing. In *Proceedings 11th Symposium on Reliable Distributed Systems*. 39–47. https://doi.org/10.1109/RELDIS.1992.235144

[17] Elmootazbellah N. Elnozahy and James S. Plank. 2004. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE Transactions on Dependable and Secure Computing* 1, 2 (2004), 97–108.

[18] Sequel: The Database Toolkit for Ruby. *Testing with Sequel*. http://sequel.jeremyevans.net/rdoc/files/doc/testing_rdoc.html (Retrieved: January 2016).

[19] Gordon Fraser and Franz Wotawa. 2007. Redundancy based test-suite reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*. 291–305.

[20] Gallery. *Gallery - Your photos on your website*. http://galleryproject.org/ (Retrieved: January 2016).

[21] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 223–233. https://doi.org/10.1145/2771783.2771793

[22] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-demand Test Suite Reduction. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 738–748.

[23] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 3 (1993), 270–285.

[24] Mats PE Heimdahl and Devaraj George. 2004. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. IEEE, 176–185.

[25] Dennis Jeffrey and Neelam Gupta. 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering* 33, 2 (2007).

[26] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering* 29, 3 (2003), 195–209.

[27] S. A. Khalek and S. Khurshid. 2011. Efficiently Running Test Suites Using Abstract Undo Operations. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*. 110–119. https://doi.org/10.1109/ISSRE.2011.20

[28] Donald E. Knuth. 1998. *The Art of Computer Programming* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[29] Berkeley Lab. *Berkeley Lab Checkpoint/Restart (BLCR) for LINUX*. http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/ (Retrieved: January 2016).

[30] Wing Lam, Sai Zhang, and Michael D. Ernst. 2015. *When tests collide: Evaluating and coping with the impact of test dependence*. Technical Report UW-CSE-15-03-01. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.

[31] J. L. Lawall and G. Muller. 2000. Efficient incremental checkpointing of Java programs. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*. 61–70. https://doi.org/10.1109/ICDSN.2000.857515

[32] LibCkpt. *A Portable Checkpointer for Unix*. http://web.eecs.utk.edu/plank/plank/www/libckpt.html (Retrieved: January 2016).

[33] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, Francesco Faralli, and Ugo De Carlini. 2002. Testing web applications. In *ICSM*.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[35] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[36] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/nsdi10-0/mugshot-deterministic-capture-and-replay-javascript-applications

[37] Microsoft MSDN. *How to: Write a SQL Server Unit Test that Runs within the Scope of a Single Transaction*. http://msdn.microsoft.com/en-US/library/jj851217(v=vs.103).aspx (Retrieved: January 2016).

[38] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 496–499. https://doi.org/10.1145/2025113.2025202

[39] OSCommerce. *OSCommerce - Creating Online Stores Worldwide*. http://www.oscommerce.com/ (Retrieved: January 2016).

[40] phpBB. *phpBB - Free and Open Source Forum Software*. http://www.phpbb.com/ (Retrieved: January 2016).

[41] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/1095810.1095833

[42] Filippo Ricca and Paolo Tonella. 2001. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE Computer Society, Washington, DC, USA, 25–34.

[43] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the 14th International Conference on Software Maintenance*. IEEE, 34–43.

[44] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. 2004. Composing a Framework to Automate Testing of Operational Web-Based Software. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*. IEEE Computer Society, Washington, DC, USA, 104–113.

[45] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. 2005. Automated Replay and Failure Detection for Web Applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/1101908.1101947

[46] Sriraman Tallam and Neelam Gupta. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/1108792.1108802

[47] tSQLt. *tSQLt – Database Unit Testing for SQL Server*. http://tsqlt.org/ (Retrieved: January 2016).

[48] A. Vahabzadeh, A. M. Fard, and A. Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 101–110. https://doi.org/10.1109/ICSM.2015.7332456

[49] w3af. *Open Source Web Application Security Scanner*. http://w3af.org/ (Retrieved: January 2016).

- [50] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. 1997. A Study of Effective Regression Testing in Practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE '97)*. IEEE Computer Society, Washington, DC, USA, 264–274.
- [51] Wordpress. *Wordpress - Blog, Tool, Publishing Platform, and CMS.* http://wordpress.org/ (Retrieved: January 2016).
- [52] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. 2007. Efficient Checkpointing of Java Software Using Context-sensitive Capture and Replay. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/1287624.1287638
- [53] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 385–396. https://doi.org/10.1145/2610384.2610404
- [54] Hao Zhong, Lu Zhang, and Hong Mei. 2006. An Experimental Comparison of Four Test Suite Reduction Techniques. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 636–640. https://doi.org/10.1145/1134285.1134380
- [55] Qian Zhongsheng. 2010. Test case generation and optimization for user session-based web application testing. *Journal of Computers* 5, 11 (2010), 1655–1662.