International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 2, pages 347–372, July 2015

Using Checkpointing and Virtualization for Fault Injection

Cyrille Artho, Kuniyasu Suzaki
AIST/RISEC
Amagasaki/Tsukuba, Japan

Masami Hagiya, Watcharin Leungwattanakit, Richard Potter
The University of Tokyo
Tokyo, Japan

Eric Platon
Independent
Tokyo, Japan

Yoshinori Tanabe
Tsurumi University
Yokohama, Japan

Franz Weitl, Mitsuharu Yamamoto
Chiba University
Chiba, Japan

Received: January 29, 2015
Revised: April 23, 2015
Accepted: June 1, 2015
Communicated by Hiroyuki Sato

Abstract

The program monitoring and control mechanisms of virtualization tools are becoming increasingly standardized and advanced. Together with checkpointing, these can be used for general program analysis tools. We explore this idea with an architecture we call Checkpoint-based Fault Injection (CFI), and two concrete implementations using different existing virtualization tools: DMTCP and SBUML. The implementations show interesting trade-offs in versatility and performance as well as the generality of the architecture.

Keywords: Fault Injection, Checkpointing, Virtualization, Software Validation, Quality Assurance

1 Introduction

Software testing is a widely used and effective quality assurance technique [1]. However, testing has its limitations; in particular, non-determinism in a system means that even with a specific input,

the behavior of a system is not always fully reproducible.

Non-determinism may come from several sources. A particular source of non-determinism is input/output, where the system under test interacts with other components. These interactions may not be fully reliable or repeatable for several reasons. Sensitivity to communication latency can cause timeout conditions to happen only sometimes. Faults from hardware and software, such as full disks and dropped network connections, can also affect the execution of a system.

Fault injection [2] analyzes the outcome of such non-deterministic input/output operations. In particular, we focus on how the application software handles such faults, which are signaled by the system library as error codes or exceptions.

In principle, any input/output operation may either succeed or fail. Network operations are particularly likely to fail, especially on mobile networks. Fault injection investigates these two possible outcomes, either using random simulation or a more comprehensive analysis where both outcomes are investigated. This makes fault injection useful to analyze the robustness of communicating systems in the presence of network problems.

Fault injection typically focuses on simulating (non-deterministic) input/output errors, whose occurrence depends solely on the environment such as the underlying network. In principle, fault injection can be applied to other operations, including faults that are caused by malformed input. However, such faults are deterministic and can be tested through traditional means, if the right input can be found. Furthermore, simulating faults with correct data but a forced control flow through a branch that assumes faulty data, may produce results that are not consistent with actually possible program executions.

We show a generic tool architecture, checkpoint-based fault injection (CFI), that adapts virtualization and checkpointing for fault injection. We see that the scope of checkpointing operations influences their size and the performance of fault injection greatly. Our contributions are:

1. We introduce CFI, a generic architecture that reuses existing checkpointing tools (commonly based on virtualization) for fault injection. CFI controls input/output operations through virtualization and uses checkpointing to manage different outcomes.

2. We introduce two concrete implementations of fault injection tools based on CFI, using the DMTCP [3] and SBUML [4] platforms. The former uses process-level virtualization, while the latter uses OS-level virtualization. This fundamental difference has ramifications for the implementation and use of each tool.

3. Our experiments demonstrate how different virtualization platforms offer trade-offs in applicability against performance and scalability. We also investigate how well checkpointing scales with different types of virtualization.

This paper is an extended version from an earlier conference paper [5]. The conference paper was very limited in space, so we can greatly expand on background information, the description of each tool, the experiments, and related work in this paper. Furthermore, we provide a qualitative evaluation of the scalability of each checkpointing tool, which was not included in the previous paper.

This paper is organized as follows: Section 2 gives more background on fault injection. Section 3 introduces the architecture of CFI, from which we describe concrete implementations in Section 4. Experiments using our tools are described in Section 5, while Section 6 covers related work. Section 7 concludes this paper.

2 Background

Software testing can detect faults. However, even if a system under test (SUT) passes a certain test, this may not mean that the SUT always will pass that test. Sources of non-determinism in the system may change its behavior so that a test sometimes passes and sometimes fails. For this paper, we use the term *choice point* to refer to locations of such non-deterministic choices. If the outcome at a given choice point cannot be controlled by the user, a developer may not discover certain software defects even after running a test many times. This problem gives rise to several program analysis tools and techniques.

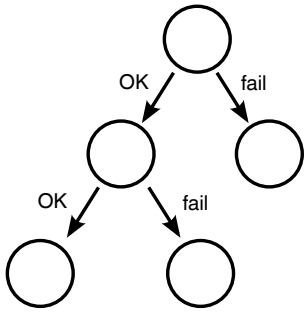


Figure 1: The state space of a system using functions that may fail.

Table 1: Common C network library functions and possible faults.

Function	Purpose	Possible faults
`socket`	Initialization of socket	Permission denied
		Protocol not supported
		Internal tables or memory full
`inet_pton`	Host name lookup	Wrong address type
		Buffer too small
`connect`	Connect to server	Connection refused
		Host unreachable
		Local network interface down
`accept`	Accept connection from client	Connection aborted
		Internal table full
		Client does not accept connection
`read`/`write`	Read from/write to connection	Connection reset by peer
		Timeout
		Socket is not connected

2.1 Fault injection

In *fault injection* [2], faults may come from a wide range of sources, from hardware production defects and physical effects affecting the hardware itself, to implementation defects that result in incorrect code. In this paper, we focus on the effects of network input/output (I/O) at run-time, because this link is likely to fail in practice. If a failure occurs, the system library that communicates with the hardware typically returns an error code or exception to the application. We focus on testing code that handles connection problems, and elide effects of network latency and multi-threading in this work, because these issues require more heavy-weight techniques [6, 7, 8]. Our goal is to increase branch coverage in C programs, so branches that handle network connection problems are covered. In programming languages that use exceptions (such as Java), we want to increase the coverage of exception handling code, according to the *e-deacts* criterion [9].

Fault injection treats I/O as non-deterministic operations: each call either succeeds or fails, for reasons outside the control of the application. To explore the impact of failures in I/O operations, faults are simulated (injected) by modifying the code of the system under test (SUT) or the library it uses. The combination of all possible non-deterministic executions creates a tree-like state space (see Fig. 1).[1]

To show how faults can be injected in code, let us consider a simple client example, which is a slightly abbreviated version from an online tutorial [10] (see Figure 2). The program first creates a new socket (a connection handle), converts a given host name to an IP address, connects to the server, and reads from it until the server closes the connection.

[1]Note that while a program often terminates when an external component fails, this is not necessarily always the case. For example, retrying an operation would result in a loop in the state space.

```
 1 int sockfd = 0, n = 0;
   char recvBuff[1024];
   struct sockaddr_in serv_addr;

 5 memset(recvBuff, '0',sizeof(recvBuff));
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
       printf("\n Error: Could not create socket \n");
       return 1;
   }
10
   memset(&serv_addr, '0', sizeof(serv_addr));

   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(5000);
15
   if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0) {
       printf("\n inet_pton error occured\n");
       return 1;
   }
20
   if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
      printf("\n Error: Connect Failed \n");
      return 1;
   }
25
   while ((n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0) {
       recvBuff[n] = 0;
       fputs(recvBuff, stdout);
   }
30
   if (n < 0) {
       printf("\n Read error \n");
   }

35 return 0;
```

Figure 2: An example program that connects to a server and reads from it [10].

Table 1 shows the most important C library functions for network I/O, including all functions used by the example in Figure 2, and a non-exhaustive list of possible faults that may occur. As can be seen, fault for the first two operations can be reproduced by restricting permissions in the execution environment (so a call to `socket` automatically fails) or by supplying an invalid host address (which causes the host name lookup to fail). For fault injection on I/O operations, the other types of faults are more interesting, as they are harder to test. For example, an unreachable host or connection timeout is commonly a sign of a network connectivity or routing problem; it would be cumbersome to simulate this on an actual network.

We therefore simulate such network problems on the software layer, by replacing the standard C library functions with functions that may return an error code instead of executing the actual library function. When returning an error code, the real library function is not executed, which results in a system state that mimics a failed execution of the library function. In the example (Figure 2), it is difficult to produce the third and forth error messages that related to a failed connection and a read error on the network. The code that handles the connection problems is therefore likely to be untested. By injecting a fault in the C library call, we can achieve 100 % branch coverage for the example.

2.2 Virtualization and checkpointing

Virtualization entails the transformation of a concrete resource into a set of abstract (virtual) resources of the same kind. A virtual resource can be used in a transparent manner, as if it was the concrete resource. Resources can be hardware components, such as memory, CPU, storage, or a network; as well as software components, such as operating systems or applications. Common virtualization and related technologies are KVM [11], Xen [12], VMware [13], the Java Virtual Machine [14], and Java WebStart [15].

The virtual resource is often called the "host", while the user of the virtual resource is the "guest". The concept of "cloud computing" relies on virtualization to provide both hardware and software resources as commodities: Typically a set of physical servers host virtualized operating systems that share the physical resources transparently.

Various degrees and scopes of virtualization are possible. Degrees range from full virtualization (the virtual resource provides the same service as the concrete one) to partial virtualization (using the virtual resource may require adaptations or additional libraries), to para-virtualization (the virtual resource exposes also an API to guests to negotiate and optimize performance). The scope of virtualization ranges from simple resource (e. g., CPU or a simple application process) to full systems (e. g., an entire hardware system or an operating system). Our work targets the verification of distributed applications, so we focus on virtualization at the process level, and the virtualization of network interfaces.

Virtualization enables a number of operations that are not possible with concrete resources. For example, live migration allows a guest system to be migrated to a different host while executing. In addition to fault injection on the level of the system library that interacts with a network device, our work is centered around checkpointing, which is usually used to provide migration capabilities in a virtualized environment.

A *snapshot* represents system information in a summarized way. A *checkpoint* is a snapshot from which a system can be restored to resume execution—on the same host or after migration. Most virtualization solutions like Xen or VMware provide checkpoints of entire operating systems. More granular solutions exist at the process level. A simple example is the GNU debugger: *gdb* can create memory dumps that can serve to restore a process state [16] (note that this is not a property of any debugger). More capable checkpointing tools can create snapshots of multiple processes [17]. Our work relies on two of these tools: DMTCP [3] and SBUML [4].

DMTCP (Distributed Multi-Threaded Checkpointing) virtualizes processes [3]. It can checkpoint several multi-threaded processes at once and controls I/O resources by overloading system libraries. DMTCP is lightweight: processes that are not of interest do not have to be virtualized. However, certain aspects of the OS, such as process IDs, are not retained when restoring a checkpoint.

SBUML (Scrapbook for User-Mode Linux) [4] is an extension of UML (User-Mode Linux), which

is a Linux kernel that runs as a user mode process on the host. Communication channels between guest processes, such as network connections, are entirely represented in user-mode data structures. SBUML snapshots capture the full OS and represent all its aspects faithfully.

2.3 Replay vs checkpointing

We call each location where a fault may be injected a *choice point*. An exhaustive analysis needs to cover all outcomes at each choice point. This requires a way to restore a program to a previous state, to investigate alternative outcomes of a choice, which can be done in two ways:

1. *Replay:* For each state, the history of inputs to reach that state is recorded. Inputs include user inputs and the outcome of each non-deterministic operation at a choice point. To restore a given state, the history of inputs (up to the desired target state) is replayed during a new test execution of the SUT.

2. *Checkpointing:* If a checkpoint was taken previously at a given state, that checkpoint can be restored directly.

A replay-based approach requires that all non-determinism from concurrency be controlled (through code instrumentation, wrapping, or by other means). When used on concurrent software, the thread schedule also needs to be reproduced when replaying an execution [18].

In contrast, checkpointing stores the full system state when reaching a choice point. To restore a given state, the checkpointed state can be directly restored. Checkpointing may introduce overhead, but it works reliably even when certain aspects of program execution, such as concurrency or the usage of external resources, may produce results that are not entirely repeatable. Checkpointing also avoids potential problems from replaying I/O operations on persistent data. For these reasons, we adopted checkpointing for our project.

Hybrid approaches are possible in that checkpoints may be created for some system states while replay is used to reach given states from the closest ancestor checkpoint [19]. Both checkpointing and replay have in common that those non-deterministic functions that are subject to fault injection need to be controlled. To achieve this, we modify the virtualized execution environment, as discussed in Section 3.

In any case, both approaches incur some overhead: replaying requires restarting a program, and checkpointing incurs overhead in duplicating a process image. The performance trade-off between checkpointing and replay [20] is outside the scope of this paper, but it should be noted that both approaches are only useful if the number of choice points is not very high. For I/O operations, this is the case; however, more fine-grained operations cannot be treated in this way. For example, Exhaustive fault injection for memory faults due to radiation (on space missions) is currently not feasible on actual system executions due to the very large number of memory accesses during program execution.

3 Architecture of Our Fault Injector

In our setting, the system under test (SUT) communicates with peers during execution. In the SUT, a *choice point* is any location in the code where an input/output (I/O) operation takes place. In each choice point, I/O may potentially fail. Faults are injected in the SUT using virtualized I/O operations. Peers are not subject to program analysis. Any choice point in the SUT, defined as a non-deterministic operation where faults may occur, requires that all possible outcomes be tested by fault injection.

3.1 Overview

Our approach, which we call checkpoint-based fault injection (CFI) systematically explores all given choice points, using checkpointing to restore previously visited states (see Fig. 3).[2]

[2]Both the SUT and the peer may consist of multiple processes, but only one process is shown for simplicity.

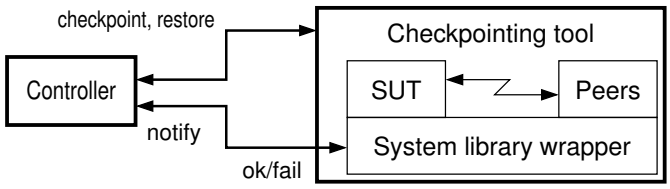


Figure 3: Architecture of checkpoint-based fault injection.

Our architecture, CFI, is independent of the actual tool used. In our design, a *controller* assumes the two key tasks:

1. Control of the decision at each choice point, by interacting with a system library wrapper that virtualizes I/O. This wrapper [21] translates generic decisions by our controller into specific actions on the virtualized resource, and vice versa for the returned result.

2. Creation and restoration of each SUT checkpoint, by controlling the checkpointing/virtualization tool within which the SUT and peer processes execute.

The controller executes as its own process, outside the checkpointing tool. It maintains two communication channels, one with the checkpointing tool, and one with the wrapper for the system libraries. Wrappers inject faults and manage the outcome of the I/O operations; see below.[3]

3.2 Controller

The controller explores all states of the SUT and all possible outcomes of I/O operation (in our case, *ok* and *fail*). System states are managed alongside with the choice taken at a given checkpoint; each choice point results in the creation of two choices ($\{c, ok\}$ and $\{c, fail\}$ where c denotes the checkpoint, followed by the choice taken.

3.2.1 Outline of algorithm

We first describe a simplified version of the algorithm to make it easier to understand. The procedure is outlined in Algorithm 1, depicted graphically in Fig. 4.

The outline uses an operation step(SUT) to advance the SUT by one instruction. We furthermore use depth-first exploration to simplify the discussion of the algorithm. After initializing both the SUT and its peers, the controller steps into the SUT execution until termination ("loop forever" in Algorithm 1). At each step, if the SUT state is a choice point (an I/O operation), the controller memorizes the state and resumes execution under successful choice condition. While the algorithm is running, The controller builds a set of states where execution will have to be resumed to explore new choices. All these memorized states represent unexplored failures at choice points.

Upon termination of the SUT, the controller restores the SUT to one of the memorized states, and executes again until termination. This procedure is repeated until there is no memorized state. Along the run, the controller reports any error encountered (reports are not represented in this overview algorithm).

3.2.2 Implemented algorithm

The actual algorithm also memorizes the set of visited states, to avoid redundancy in the state space search, and is shown in a version where the search strategy is not fixed (so depth-first search but also other search algorithms are possible).

The *state space* of the SUT comprises the set of all visited program states at choice points. The controller virtualizes I/O operations and decides their outcome during test execution (see Algorithm 2). The algorithm takes the SUT and its peers, along with a set of choice points. It explores

[3]Communication between the controller and the checkpointing tool is exempt from fault injection.

Algorithm 1 Outline of algorithm for exploring the SUT.

```
1  S ← ∅ // States to visit
2  init SUT and peers
3  loop forever {
4     s ← step(SUT) // Get next SUT state
5     if (s is a choice point) {
6        S ← S ∪ {s, fail}
7        continue with {s, ok}
8     } else if (s is end and S ≠ ∅) {
9        pick choice c from S
10       S ← S \ {c}
11       restore checkpoint from c and continue with given choice
12    } else {
13       break loop // end of exploration
14    }
15 }
```

the SUT until either the SUT terminates or a choice point is hit. At each choice point, a checkpoint of the current system state (which includes the SUT and its peers) is created, and is stored with the set of all possible choices (*ok* or *fail*). From this set, a new state and choice is taken each time a choice presents itself or the current SUT run terminates. The algorithm runs until the SUT has terminated and no checkpoints are left to be explored. An error found along the way is reported. Visited choices are remembered throughout the analysis so the same state is only analyzed once. Non-termination is handled by imposing a CPU time limit on the SUT.

Different search strategies employ a different choice of $\langle c', \gamma \rangle$ when a new checkpoint is chosen (line 16). We choose depth-first search, as the SUT and peer states do not have to be changed when the current execution has not terminated yet. In other words, the operation "restore SUT state" is redundant in that particular case, where $c' = c$ and the SUT it still running. Visited states are tracked by set V.

As Algorithm 2 does not control concurrency inside the SUT, or between the SUT and its peers, some outcomes from different interleavings between messages or memory accesses may be missed. An exhaustive concurrency analysis would increase the analysis time exponentially in the number of thread interleavings. Furthermore, Algorithm 2 does not limit the number of faults injected for a given execution. We can easily bound this by limiting the search depth at line 6.

In many cases, the user is interested in finding the first error that the tool detects; this is why the search terminates at line 13. If all errors in the SUT should be found, then the search should continue. Error states are recognized by abnormal termination of the SUT; see Section 4 for details.

3.3 Wrappers to control choice points

Fault injection at choice points is implemented by virtualizing I/O operations so the outcome of such operations can be controlled by our tool. We use *wrapper functions* [21] for each library function that is fault injected. The wrappers are called instead of the original function. They either inject a fault (to simulate a failure) or call the original function (to simulate success).[4] The controller (see Fig. 3) determines the outcome of the wrapped function (see Algorithm 3).

[4]The original function may of course also fail, but in practice non-deterministic failures in a test bed that uses a wired network are extremely rare and often point to a hardware defect. In cases where `actual_function` fails, successful execution *cannot* be simulated as the correct system state entails updates in internal buffers after a successful I/O operation.

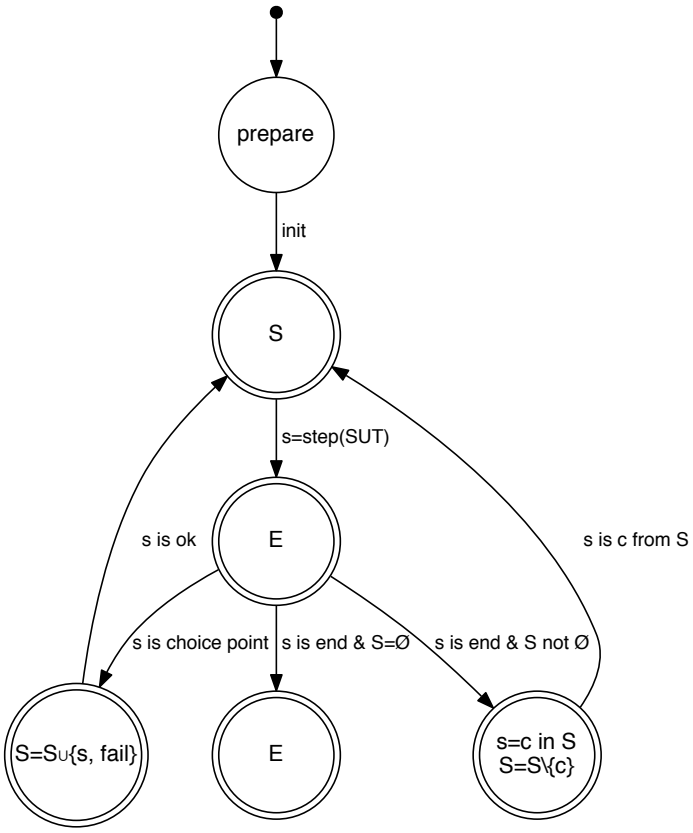


Figure 4: State-transition diagram of the controller.

The call to the wrapped function occurs inside the SUT; at this point, execution is transferred to the controller by helper function `controller_choice` (see Alg. 3, line 1). This function suspends the execution of the SUT, transfers information about the current choice point location to the controller, and waits for the result that the wrapped function should take. When the controller returns the result, either a fault is simulated, or the actual library function is called.[5]

3.4 Summary

The controller uses a custom runtime environment to manage the execution of the SUT and its peers. The custom environment uses I/O library wrappers to allow the controller to simulate failed executions of I/O operations. The locations where these operations occur are defined as choice points in the execution of the SUT; choice points are instrumented with code that uses the virtualization environment to transfer control of the execution from the SUT back to the controller.

4 Implementation Using DMTCP or SBUML

We present two fault injection tools based on CFI, using on DMTCP [3] and SBUML [4]. The first tool uses process-level checkpointing while the second one uses OS-level virtualization. Therefore, the tools differ in certain technical aspects even though their architecture is similar. Both tools use checkpointing to analyze all possible application states where faults can be injected, but do not cover other sources of non-determinism such as concurrency. The different level of virtualization

[5]Note that the controller may, based on its search strategy, choose to suspend the current system (saving it to a checkpoint) and resume the new system from a different checkpoint, possibly affecting the result of a different choice point than the one where execution was suspended at.

Algorithm 2 Algorithm for exploring the SUT.

```
1  C := ∅ // set of choices to explore
2  V := ∅ // set of visited choice points/choices
3  start SUT and peers from initial state
4  loop forever {
5    execute SUT till choice is hit or SUT terminates
6    if (hit choice point) {
7      create checkpt c from current state (SUT + peers)
8      C := C ∪ {⟨c, ok⟩} ∪ {⟨c, fail⟩}
9      // add choices for new checkpoint to C
10   } else { // SUT terminated
11     if (error state) {
12       report error
13       exit
14     } // else normal termination
15   } // end if (hit choice point)
16   repeat
17     if (C = ∅) exit // last state explored
18     choose ⟨c′, γ⟩ ∈ C
19     C := C\⟨c′, γ⟩ // remove current choice from C
20   until ⟨c′, γ⟩ ∉ V
21   V := V ∪ ⟨c′, γ⟩ // mark choice as visited
22   restore SUT state from ⟨c′, γ⟩
23 } // end loop
```

Algorithm 3 Pseudo-code outlining fault injection.

```
1  result := controller_choice(cp)
2  if (result = fail)
3    return ERROR; // return −1, or throw exception
4  return actual_function(...) // else (result = ok)
```

has consequences for the implementation of fault injection, control of the virtualized resources, and, more subtly, communication between the virtualization tool and the controller.

4.1 DMTCP-based fault injector

DMTCP virtualizes a group of processes [3] and controls both process execution and communication between processes it manages. A DMTCP-controlled program execution uses a *coordinator*, henceforth called DMTCP coordinator, to control all processes attached to it. The coordinator virtualizes I/O operations internally (for checkpointing) and is part of DMTCP.

A checkpoint in DMTCP includes communication links between all captured processes that are managed by the DMTCP coordinator. However, communication to processes outside DMTCP is not managed. For this reason, our setup executes both the SUT and its peers inside DMTCP, to ensure that their state is consistent before and after checkpointing (see Fig. 5).

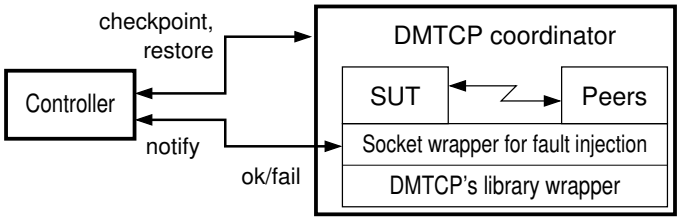


Figure 5: DMTCP-based fault injection tool.

```
#!/bin/bash

# Execute the specified command.
IS_SUT=1 "$@"
RV=$?

# Notify the controller.
echo "${DONE} ${RV}" > /dev/tcp/localhost/${PORT}
exit ${RV}
```

Figure 6: The SUT wrapper script for DMTCP.

4.1.1 Controller

Our controller is separate from the DMTCP coordinator and determines program correctness by the *exit status* of the SUT: In Unix-like operating systems, some exit statuses indicate severe failures such as an assertion error (134), a segmentation violation (139), etc.; these are displayed as an error report by our controller.

Communication between the controller, the SUT, and the DMTCP coordinator is implemented using TCP/IP sockets, on a fixed port for each channel. The wrapper functions of the SUT send a notification message to the controller, to check if a fault should be injected or not. The controller responds accordingly. After the SUT has terminated, a "done" message is sent to the controller to signal termination, followed by the exit status of the SUT.

A shell script wraps the execution of the SUT (setting `IS_SUT` to denote if fault injection is enabled) and handles the message signaling program termination (see Fig. 6). Note that we use a bash-specific feature which treats `/dev/tcp` like a device connecting to a given host and port. We use this because DMTCP showed problems when using the more portable netcat (`nc`) command, causing some of our tests to fail.

The state space of the search (checkpoints to be explored, and visited choices) is managed by keeping copies of checkpoints as files.[6] The performance of fault injection therefore depends on the file size of all snapshots. If the disk cache can contain them all, then the entire analysis can execute in RAM, allowing it to finish quickly (see Section 5).

4.1.2 Library function wrappers

DMTCP does not directly call C library functions related to I/O, but instead virtualizes all these functions to keep track of file descriptors and other data. This is necessary for its internal checkpointing mechanism. Our fault injection therefore operates at the level of the wrapper function used by DMTCP (see Fig. 5). With DMTCP, the system library wrapper is separated into two components: Our fault injection wrapper, and DMTCP's library wrapper that is called when no fault is injected. The fault injection function shown in Algorithm 3 therefore calls DMTCP's wrapper function as

[6]Our current implementation uses depth-first search; therefore, one choice is always immediately taken when a choice point is hit, eliminating the need for remembering a set of remaining choices per checkpoint.

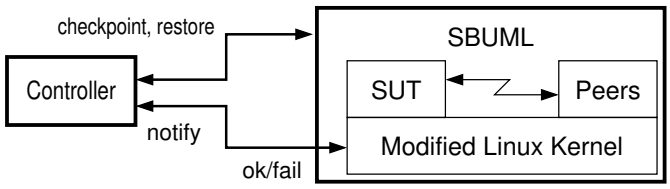


Figure 7: SBUML-based fault injection tool.

`actual_function` (line 4), which in turn calls the original C library function after keeping track of file descriptors and other internal data.

With DMTCP, the SUT and peer processes share the same environment. Hence an environment variable, `IS_SUT`, tracks whether fault injection is enabled. We support fault injection on `connect`, `accept`, `send`, and `recv`. A fault is injected by returning -1 in these functions, and setting an appropriate error code, such as `ECONNREFUSED` in `connect`. At this point our tool injects a typical (representative) error code for each type of operation, but it is easy to generalize this to a user-defined error code if needed. For I/O on file descriptors, we check the result of `fstat` to limit fault injection to sockets. If all file operations were fault injected, then loading the dynamic libraries at program start-up would also be fault injected and possibly fail. These special cases therefore require that fault injection be conditionally suppressed, for example, for system-specific code executing before main is executed. Internal control messages used by DMTCP are also exempt from fault injection.

4.1.3 Communication

The controller acts as a server and listens on a given port for notifications from the socket wrapper functions executing in the SUT. Communication is implemented to be short-lived so no active channel between the controller and DMTCP exists when checkpointing operations are used. This is necessary because DMTCP assumes full control of all communication channels, which is only possible for processes managed by it. Communication between the controller and the DMTCP coordinator is therefore terminated before each checkpointing operation, and re-established afterwards.

4.2 SBUML-based fault injector

Our second implementation uses SBUML, which virtualizes an entire operating system [4]. This has the advantage that all resources used by the SUT and peers are included in a checkpoint, down to details such as process IDs that are not virtualized by DMTCP. Disadvantages are the fact that system processes are also included, and that communication between the controller and SBUML cannot be easily done via sockets, as the guest kernel is frozen between snapshots.

4.2.1 Controller

The controller needs to communicate with SBUML to control system execution (see Fig. 7). Our fault injection builds on already existing debugging interfaces in SBUML, so that no additional kernel modifications are needed. The main such debugging feature is the ability to inspect and set global variables in the kernel, by directly modifying the SBUML kernel memory that is memory mapped to files on the host. Communication by writing to files is flexible because it can be done while the guest kernel is frozen. A higher-level inspection feature exists to walk through kernel data structures, list all guest processes and determine which process is currently executing. Because the inspection features already existed, only 69 lines of the kernel source were modified to introduce the wait-loop choice point and few global variables for controlling it.

The controller needs a way to determine the exit code of the SUT, which is running as a guest process that is controlled by the SBUML kernel. SBUML's debugging features cannot inspect guest processes running inside SBUML, so another technique must be used. We use *hostfs*, a feature inherited from UML that allows guest processes to write to files on the host, at for example

```
./vfi-sbuml.sh -explore snapshotname \
    "processname" /tmp/SUT-exit-code
```

Figure 8: The SUT wrapper script for SBUML.

Table 2: Comparison between both tools.

	Code size (lines of code)		
Tool	**Existing platform**	**CFI wrapper**	**CFI controller**
DMTCP-based fault injector	32,000 (C/C++)	600 (C/C++)	700 (Java)
SBUML-based fault injector	7,000 (C) + 8,000 (bash)	70 (C)	390 (bash)

`/tmp/SUT-exit-code`. Higher-level scripts running on the host can then easily poll and read such files.

A 350-line higher-level script performs fault injection based on these SBUML interfaces. It uses a manually prepared snapshot as a starting point. Preparation of that snapshot requires that the SUT and peer processes be installed into an SBUML guest file system, networking I/O choice points manually turned on, and then the SUT manually started. The SBUML kernel soon freezes on the first socket call, which could have been called by the SUT or any other guest process. The next step is to save this frozen kernel into a snapshot. Once this initial-state snapshot has been created, the script can be started (see Fig. 8).

The script then inspects every choice point and identifies those that happen inside the SUT, as identified by `"processname"`. The OS states that contain the choice points of interest are saved into snapshots. The SBUML kernel is allowed to continue normally, stopping at later choice points of interest. The exit code file(s) are checked after each iteration. If the SUT (or a peer, if desired) exits, a new search path is started by restoring one of the previously saved snapshots. Therefore, like with the DMTCP-based tool, we use depth-first search in this implementation.

4.2.2 Kernel I/O function wrappers

SBUML is based on Linux 2.4.24 and compiled as a modified Linux kernel. We introduce network I/O choice points into SBUML by modifying the Linux kernel source for the system call interface named `socketcall`, which handles all socket-related calls (`accept`, `connect`, etc.) for Linux 2.4 series kernels. The key part of the modification sleeps at the start of `socketcall`, effectively freezing the guest kernel and the processes inside it. Read and write calls in the kernel are changed in a similar way, to handle I/O using file descriptors.

Inserting the choice point into the kernel has the advantage that it is automatically applied to all network I/O for any SUT. The disadvantage is that it is too general and will cause networking from any process to block and even prevent the guest kernel and OS from booting. Therefore by default the network I/O choice point is turned off, and extra functionality is added to the kernel to control the choice points. This design shifts the key functionality to higher-level scripts so that harder-to-debug kernel modifications are minimized. These control scripts turn a choice point on and off, continue from a frozen choice point, set error codes to be applied upon continuation, and query about system call parameters and which process is currently executing. Fig. 7 shows how the communication connects the components.

4.2.3 Communication

In this tool, the controller manages SBUML via files that are memory-mapped to the kernel running inside SBUML. This avoids problems (time-outs or full buffers) when the guest kernel is frozen. The file system of the guest system is made visible to the host using *hostfs*.

Table 3: List of benchmark applications used.

Name	LOC	Description
darkhttpd 1.8	2,488	Web server
frox 0.7.18	7,847	Transparent FTP proxy server
netcat 1.1.0	2,102	Arbitrary connections and listens
prtunnel 0.2.7	2,685	HTTP tunnel server
alphabet server	153	Example server returning nth character in the alphabet
alphabet client	158	Client for alphabet server

4.3 Comparison

DMTCP is written in C and C++, with about 32,000 lines of code in total. SBUML adds about 7,000 source lines of C code to UML to provide checkpointing. The higher-level interfaces for the user and for debugging are written mostly in bash, and consist of about 8,000 source lines of shell script [22].

Reusing checkpointing tools greatly reduces the development effort for a fault injection tool. Our implementation uses only about 1,300 additional lines of C, C++, and Java code for DMTCP, and 460 extra lines of C and bash code for SBUML. This is about 3–4 % of the size of the checkpointing software used (see Table 2). The CFI controller for DMTCP is implemented in Java, and we could have shared much of the Java code for the SBUML-based fault injector. However, because process control works differently in both systems, it was more expedient to write the second tool mostly in bash. Because bash has many built-in operators for process control, its code is shorter than the Java-based controller.

5 Experiments and Discussion

Our implementations are reliant on DMTCP and SBUML, which have some restrictions: DMTCP does not virtualize certain OS-level features such as process IDs, and SBUML uses an older kernel version. However, common types of server software (HTTP, FTP) are supported, and both tools inject the same types of faults on network I/O. We have chosen three representative server programs for our experiments, along with netcat and an artificial benchmark that implements a minimalist server (see Table 3).

We furthermore present a benchmarking of DMTCP and SBUML itself, outside their use in CFI, to explain their performance differences.

5.1 Selection of examples

In order to be useful for our experiments, applications have to be supported by the platforms and compilers on our system, without dependencies on obscure third-party libraries, and with sufficient documentation so we can actually execute the system in our experiments. Furthermore, programs have to run without a graphical user interface or other user intervention. They have to be configurable or modifiable such that they would terminate after one connection, deviating from the typical behavior of running indefinitely, to avoid producing an infinite state space.[7]

We used a brute-force method to find enough interesting programs: First, we used `http://www.freecode.com/` to find a number of C programs that implement servers but do not require a graphical user interface or other features that make them unsuitable for automatic testing. A total of 79 positive and negative filter criteria were used, yielding 207 matching projects.[8]

Out of these projects, 175 contained a link that could be directly followed by a shell script (i. e., a link leading directly to an archive or a download page on `http://sourceforge.net/`). 134 of

[7] We compare checkpoints to avoid infinite loops when possible, but even slight changes in the program state such as incrementing a counter can easily produce an "infinite" state space, regardless of whether the change is important for program behavior.

[8] Unfortunately, the site has meanwhile been archived so a detailed search is no longer possible.

Table 4: Experimental results. Shown are the total analysis time taken, the total amount of disk space used by all checkpoints, and the number of checkpoints generated by fault injection. The search was configured to abort after 50 checkpoints.

SUT	DMTCP			SBUML		
	Time	Memory	# chkpts	Time	Memory	# chkpts
darkhttpd 1.8						
no error	16.02 s	47.40 MB	5	60.42 s	616.11 MB	5
seeded fault	8.46 s	47.40 MB	5	60.34 s	616.92 MB	5
frox 0.7.18	not compatible			481.95 s	5362.08 MB	50
netcat 1.1.0	not compatible			285.09 s	3156.18 MB	50
prtunnel 0.2.7	4.30 s	10.36 MB	1	36.57 s	298.08 MB	2
alphabet server	13.51 s	19.11 MB	4	54.28 s	490.44 MB	4
alphabet client						
no error	17.06 s	15.29 MB	3	70.96 s	784.79 MB	7
assertion error	3.39 s	10.18 MB	3	65.07 s	784.86 MB	7
segfault	6.30 s	15.29 MB	3	69.58 s	784.76 MB	7

these links were valid and usable, allowing access by a download script, requiring no browser cookies or user registration. 111 downloads yielded a working archive of a known archive type, containing C source code.

To test fault injection on a socket level, we looked for programs that use `recv`; we found 34 such programs, out of which seven compiled without any changes. The reason for this low yield is mainly that many less commonly used open-source programs are maintained infrequently and require small changes to work on newer systems, or at least special configuration options to compile. Indeed, out of the seven programs, only five contained sufficient documentation and complete configuration files to work on our system. We chose three representative programs (darkhttpd, frox, and prtunnel) for our experiments, along with netcat and an artificial benchmark that implements a minimalist server (see Table 3).

5.2 Tool comparison

We inject faults in five server programs (see Table 3) as the SUT, against a peer that is not fault injected, and also analyze the two components of the alphabet client/server pair in reversed roles, with the client being fault injected. Three variations of the client were tested in this experiment. One client produces no error. The other two produce an assertion error and cause a segmentation fault after a fault-injected function returns an error. We also seeded a similar fault in darkhttpd to confirm that our tools work as expected.

Table 4 shows the experimental results of the fault injection tool. The memory usage column shows the total amount of disk space taken by all checkpoints; since that amount is smaller than the disk cache, the checkpoints are effectively stored in memory. If the disk cache could not contain all checkpoints, though, then our analysis would require search pruning to remain useful in practice.

In those cases where it supports the application, DMTCP is a lot faster and uses much less memory than SBUML. This is mainly because DMTCP virtualizes only the processes of interest (the client/server processes of the SUT and peers), while SBUML virtualizes the entire operating system, including all daemon processes that are typically present on a system. In our case, 16 extra processes are included in each checkpoint, increasing both the size of a checkpoint and the time required to store and load it.

Unfortunately, DMTCP cannot handle every case, partially because some features of the C library are not yet fully supported by DMTCP, which is still under development. Furthermore, due to its approach, certain features are not faithfully restored from a checkpoint. Because DMTCP works on a process group rather than OS level, it is fundamentally very difficult to handle certain advanced features. In particular, we identify the following (non-exhaustive) list of features that caused problems or may cause problems when using DMTCP:

Table 5: Comparison of virtualization tools, when applied to one process.

Properties	**Virtualization tool**	
	SBUML	**DMTCP**
Virtualization level	OS	group of processes
Networking	emulated on OS	auto connection recovery
Disk image	reversible	irreversible
Support for process IDs, special files	yes	no
Checkpoint save time (ms)	1990	308
Checkpoint load time (ms)	1830	82
Checkpoint size (MB)	93	1.6

- Process IDs. DMTCP recreates a process when restoring a checkpoint. This causes restored processes to have a different process ID each time they are restored from a checkpoint. In particular, this causes programs that rely on on their process ID for inter-process communication, to fail. Many programs store their process ID in `/var/*.pid`, such as schedulers `atd`, `crond`, DHCP clients and servers, the network time daemon `ntpd`, the system log daemon `syslogd`, the secure shell server `sshd`, and many others. DMTCP is therefore not suitable to verify system daemons that are closely integrated into the server configuration.

- Special files. Such files include `/var/shm` (shared memory between processes with an interface to the file system) and the `/proc` file system. The latter gives access to information on other processes (via their process ID, which leads to the above-mentioned problems), and other statistics. Information can be read as if accessing a file, but the contents of these pseudo-files can change at any time, which may lead to inconsistencies after restoring a program from a checkpoint.

- We had issues with the `select` system call, where a connection was not correctly restored by DMTCP [23]. This is an implementation problem and not fundamentally related to process-level checkpointing. However, it highlights the fact that OS-level checkpointing works at a "natural" boundary by replacing a hardware execution environment with software, while process-level checkpointing interfaces with the SUT at the boundary between the priviledged kernel and the unpriviledged user-space SUT. Functionality that is embedded in the kernel or closely related to it, is more difficult or sometimes even impossible to emulate faithfully at that level.

OS-level checkpointing may not only produce overhead due to larger checkpoints, but sometimes produce more checkpoints as such, too. In our experiments, the SBUML-based fault injector sometimes produces more checkpoints, because the system calls are intercepted globally for any network operation, whereas on DMTCP, the network is virtualized at the C library level.

We found that for the given server programs, the lack of concurrency control did not affect the reliability of our analysis. The number of checkpoints depends on the number of connections and messages used, and the seeded defects are thread-local. Because of this, the outcome of the analysis is consistent across multiple executions.

5.3 Comparison of checkpointing platforms

Our experiments have shown that checkpointing performance has a significant impact on fault injection using checkpointing. The question therefore arises how much different scenarios impact the performance of checkpointing. To the best of our knowledge, there are many benchmarks measuring the performance of the *execution performance* of virtual environments [24, 25] but none measuring the *checkpointing performance.* Our comparison of checkpointing tools aims to find out how the design of a checkpointing tool affects its checkpoint time and size.

Table 5 compares the tools by key properties and gives an overview of the performance of each tool. We use the alphabet server application to measure the performance of checkpointing with

256 MB of RAM being reserved for the guest kernel in SBUML.

SBUML virtualizes a system at the OS level and saves file system changes in a disk image, which results in a heavy load during saving. Checkpoints contain the blocks read or written since the last cloned snapshot (SBUML). Table 5 only shows the size of VM state information, excluding the disk image. DMTCP does not keep track of file contents in a checkpoint, so the contents do not revert by checkpoint restart. It is not difficult to add a function to take a snapshot of a certain disk area into the coordinator, though. Indeed, the fact that checkpointing file contents is optional is an advantage, because it is not needed for many types of server programs that only read from disk. As can be seen from an initial test in Table 5, the performance of different tools varies greatly; in the case of OS-level checkpointing, it is also heavily affected by tuning the configuration, something which constitutes future work.[9]

Our benchmark of checkpointing tools examines their performance when checkpointing an SUT containing a certain number of processes. Figure 9 shows plots by three quantities: checkpoint size, save time, and restart time (using an average over nine runs). We also run the experiments for the case where each process allocates 2 MB of memory to see the change in performance (see Fig. 10). The extra memory is filled with random values so that checkpoint compression has minimal effect.

In case of the minimal server, process-level checkpointing performs best. However, OS-level checkpointing shows better scalability in all measures as the number of processes increases. The size of the checkpoints created by DMTCP grows linearly, since DMTCP creates one checkpoint file for each process. DMTCP recovers multiple processes by forking a number of child processes. The memory image of each newly created process is restored one by one. This explains why the overhead of this recovery process is proportional to the number of processes.

Overall, the performance of the two tools converges at about 50 processes. This is a rather large number and unlikely to be necessary for the purpose of fault injection, where the behavior of unreliable network connections is closely monitored on a small number of processes (usually a single target program and its child processes). If a program uses a main process to accept connections and a child process for each active connection, then $n + 1$ processes suffice to handle n connections. Even in a densely connected test setup, the number of 50 processes is usually not reached, which means that the more light-weight process-level checkpointing offers superior performance under normal circumstances. In fact, a test setup with two connections is often sufficient to find defects in connection handling [6]. It is therefore more important to know exactly what features a program uses, and choose DMTCP only if these features are all supported (see Section 5.2 above).

5.4 Discussion

We have evaluated two fault injection tools based on I/O virtualization and checkpointing. One tool is based on DMTCP and uses process-level checkpointing, the other one is based on SBUML and uses OS-level checkpointing. Process-level checkpointing is more lightweight and easier to use for all but very large programs (up to about 20–50 processes). It provides a better performance in our benchmarks but does not virtualize certain low-level resources such as process IDs faithfully; these may be needed for certain applications.

OS-level checkpointing handles complex applications but requires that a system image be prepared; in our experience, it is more difficult to configure a system image that is suitable for testing and also has the right parameters for good performance. Because OS-level virtualization includes many system processes in each checkpoint, checkpointing operations incur a significant overhead for fault injection.

This overhead drew our attention to the performance of checkpointing tools. Experiments confirm that OS-level virtualization suffers from the relatively large size of a default operating system system that provides the necessary resources for tests to execute. In normal usage of virtualization,

[9] We were able to get a better performance using OS-level checkpointing on smaller systems, when using less memory for the guest kernel. However, for a larger number of processes, 256 MB are needed. We therefore chose to run all experiments with the same setting. Process-level checkpointing does not require pre-setting the amount of memory needed, and is not affected by this issue.

checkpointing is rarely used; however, for fault injection, it makes sense to consider configurations of virtualization tools that can provide speedy checkpoint operations.

6 Related Work

Our work on checkpoint-based fault injection (CFI) is inspired by Java PathFinder (JPF) [26]. JPF targets non-determinism from concurrency and optionally from undetermined input and possible exceptions [26]. It defines choice points internally to represent when during program execution non-deterministic decisions are taken. By default, choice points are implemented for thread scheduling. Extensions of JPF define additional choice points for non-deterministic input, represented by symbolic values [27] or for faults on a network [6, 28].

6.1 Overview

CFI uses virtualization tools to apply the checkpoint-based fault injection to binaries. This is similar to other fault injection tools for networked systems [29, 30, 31]. CFI differs in that we use an existing tool to control all processes in a distributed system, instead of creating the whole platform from scratch.

Other such work combines inputs (workload) generation with fault injection [32]. It covers a wider range of problems than CFI, at a higher computational cost and generates an entire test suite under given constraints, in addition to injecting faults. Our work assumes a fixed input (given by a user-defined test suite) and injects only I/O related faults.

CFI is also related to other tools that inject faults on one given process. In many existing tools that target network I/O, the application code or execution is modified to return an error code or exception at random or for user-defined events; each execution covers one possible outcome [29, 30, 31, 34, 36, 37]. This means that only a subset of all possibilities is explored. Heuristics can improve the odds of finding a defect but there is no guarantee that all relevant system behaviors are covered.

A complete coverage of the state space can be achieved by systematically enumerating all possible faults (or combinations thereof). This has been implemented by either replaying, executing a system repeatedly [18, 33], sometimes combined with event workload generation [32]; or by using a platform that provides checkpointing internally [6, 7, 28, 35]. In the latter case, fault injection has been implemented for TCP/IP, where a fault from a dropped (unusable) connection results in an exception [6, 7, 28], and on UDP, where a packet may be dropped silently [35]. CFI is currently adapted to the case where a fault results an error code or exception but can in principle be modified to also support different non-deterministic outcomes, such as pocket loss and duplication.

Our key contribution is a general architecture that can be adapted to various checkpointing tools, making the approach less dependent on a given execution platform.

6.2 Classification of known approaches and tools

The combination of either randomized or exhaustive fault injection, with search approaches, gives rise to a classification of existing tools that inject faults on interactions of the SUT with external components (see Table 6).

When looking at the state space analysis capabilities of each tool, and how they are implemented, it is possible to classify the tools into six categories: by how the state space is explored, and whether replay or checkpointing is used. The state space can be explored exhaustively, by a user-defined search (in this case an exhaustive exploration is possible but may require state space management code by the user) or at random. In this classification, even though the sample size of 13 tools is not large, it is noteworthy that the checkpointing approach is not used for a randomized exploration of the search space (see Table 7). Most of exhaustive tools allow for a user-defined customization (pruning) of the search, so the cluster "checkpointing/user-defined" is only empty because we count tools that have a built-in capability of an exhaustive search in that category. However, while it is theoretically possible to build a checkpointing-based tools for randomized search, no such tool exists

to our knowledge, probably because of the overhead of checkpointing that makes it less attractive in the context of a randomized search, which is normally light-weight and scalable.

Another way to look at the tools is by looking at which platforms lend themselves to replay-based or checkpoint-based tools (see Table 8). It can be seen that all checkpoint-based tools use an existing facility to manage copies of a program state: Java-based tools use Java PathFinder [26], while tools that analyze binary code other use checkpointing (as presented in this paper) or fork (which internally creates a copy of the process image). Indeed, Java PathFinder is designed to be easily extensible and lends itself to creating specialized tools based on it; without it, probably fewer fault injection tools for Java would exist. Tools that use replay are more commonly developed from scratch. Our collection lists only one Java-based tool using replay, which is specialized for fault injection in unit tests [33]. In principle, there is no binary-specific technical hurdle for developing such a tool for binary code, which explains why many such tools exist.

Fault injection tools have in common that they generate faults, by simulating possible behaviors of the environment; however, the correctness of the test output is under modified conditions not verified. This has to be done by code inside the system under test (for example, using assertions to check safety properties). Outside such existing properties, fault injection tools are limited to crash testing. If it is desirable to model both possible faults and the required behavior under exceptional circumstances, model-based techniques can be used [38]. However, in this case full automation is no longer possible, as the output model has to be derived from the specification of the system.

7 Conclusions and Future Work

Input/output operations may fail unexpectedly due to hardware problems or network failures. Fault injection can simulate such failures at the software level. An exhaustive search of the impact of all possible faults requires that many possible paths be tested. Checkpointing provides a robust mechanism to achieve this goal.

We have shown a generic architecture for a fault injection tool based on checkpointing, and implemented such a tool using two checkpointing/virtualization tools: DMTCP and SBUML. DMTCP is a light-weight multi-process checkpointing tool, which offers excellent performance. SBUML virtualizes an entire OS, which incurs a higher overhead but models low-level features more faithfully. Our experiments have shown that it makes sense to choose the checkpointing tool based on the characteristics of the software to be tested.

Future work includes more options such search heuristics to find defects faster in larger test scenarios. A feature that replays a detected defect without running the full analysis again, will make CFI easier to use for debugging. We will also consider other checkpointing platforms [39].

In the long-term, we hope that checkpointing tools will also be developed with program analysis in mind, allowing them to be generalized further, under a common application programming interface (API). In the context of virtualization tools, our work has shown that the performance of checkpointing operations (and the size of checkpoints) matters when such tools are used for program analysis.

Acknowledgments

This work was supported by kaken-hi grants 23240003, 23300004, and 26280019 from the Japanese Society for the Promotion of Science (JSPS).

References

[1] G. Myers, *The art of software testing.* New York : Wiley, 1979.

[2] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[3] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. 2009 IEEE Int. Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE, 2009, pp. 1–12.

[4] "Scrap-Book User-Mode Linux," 2013, http://sbuml.sourceforge.net/.

[5] C. Artho, M. Hagiya, W. Leungattanakit, E. Platon, R. Potter, K. Suzaki, Y. Tanabe, F. Weitl, and M. Yamamoto, "Using checkpointing and virtualization for fault injection," in *Proc. 2nd Int. Symp. on Computing and Networking (CANDAR 2014)*. Shizuoka, Japan: IEEE, 2014.

[6] C. Artho, C. Sommer, and S. Honiden, "Model checking networked programs in the presence of transmission failures," in *Proc. IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*. Shanghai, China: IEEE, 2007, pp. 219–228.

[7] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi, "Modular software model checking for distributed systems," *IEEE Trans. on Softw. Eng.*, vol. 40, no. 5, pp. 483–501, 2014.

[8] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*. Palo Alto, USA: IEEE, 2013, pp. 169–179.

[9] S. Sinha and M. Harrold, "Criteria for testing exception-handling constructs in Java programs," in *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM 1999)*. Washington, USA: IEEE Computer Society, 1999, p. 265.

[10] H. Arora, "C socket programming for Linux with a server and client example code," http://www.thegeekstuff.com/2011/12/c-socket-programming/, 2011.

[11] Red Hat, Inc., "KVM," http://www.linux-kvm.org, 2012.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, New York, USA, 2003, pp. 164–177.

[13] VMWare, Inc., "http://www.vmware.com/," 2012.

[14] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999.

[15] M. Marinilli, *Java Deployment: with JNLP and WebStart*. Indianapolis, IN, USA: Sams, 2001.

[16] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB : the GNU source-level debugger*, 9th ed. Boston, MA : Free Software Foundation, 2002.

[17] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," in *Proc. Scientific Discovery through Advanced Computing (SciDAC 2006)*, Denver, USA, 2006.

[18] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006, pp. 131–146.

[19] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking distributed systems by combining caching and process checkpointing," in *Proc. 2011 IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, 2011.

[20] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking of concurrent algorithms: From Java to C," in *Proc. Conf. on Distributed and Parallel Embedded Systems (DIPES 2010)*, ser. IFIP AICT, vol. 329. Brisbane, Australia: Springer, 2010, pp. 90–101.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. New York, USA: Addison-Wesley Publishing Company, 1995.

[22] R. Potter and K. Kato, "SBUML: Multiple snapshots of Linux runtime state," *Computer Software*, vol. 26, no. 4, pp. 120–137, 2009.

[23] R. Potter, "Distributed MultiThreaded Checkpointing / mailing lists," http://sourceforge.net/p/dmtcp/mailman/message/31121809/, 2013, bug report on issue with `select` in DMTCP.

[24] K. Moeller, "Virtual machine benchmarking," 2007, diploma Thesis, Systems Architecture Group, University of Karlsruhe, Germany.

[25] VMWare, Inc., "VMmark virtualization benchmarks," http://www.vmware.com/products/vmmark/overview.html, 2012.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Softw. Eng. Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[27] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of java bytecode," in *Proc. of the IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2010)*. ACM, 2010, pp. 179–180.

[28] X. Li, H. Hoover, and P. Rudnicki, "Towards automatic exception safety verification," in *Proc. 14th Int. Symposium on Formal Methods (FM 2006)*, ser. LNCS, vol. 4085. Hamilton, Canada: Springer, 2006, pp. 396–411.

[29] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-Cloud: design of a software testing environment for reliable distributed systems using cloud computing technology," in *Proc. Int. Symp. on Cluster, Cloud and Grid Computing (CCGRID 2010)*. IEEE, 2010, pp. 631–636.

[30] P. Broadwell, N. Sastry, and J. Traupman, "FIG: a prototype tool for online verification of recovery," in *Proc. Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[31] J. Carreira, H. Madeira, and J. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. on Softw. Eng.*, vol. 24, pp. 125–136, 1998.

[32] D. Cotroneo, R. Natella, S. Russo, and F. Scippacercola, "State-driven testing of distributed systems." in *Proc. Int. Conf. on Principles of Distributed Systems (OPODIS 2013)*, ser. LNCS, R. Baldoni, N. Nisse, and M. van Steen, Eds., vol. 8304. Springer, 2013, pp. 114–128.

[33] C. Artho, A. Biere, and S. Honiden, "Exhaustive testing of exception handlers with Enforcer," *Post-proc. of 5th Int. Symposium on Formal Methods for Components and Objects (FMCO)*, vol. 4709, pp. 26–46, 2006.

[34] T. Tsai and R. Iyer, "Measuring fault tolerance with the FTAPE fault injection tool," in *Proc. 8th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems (MMB 1995)*. London, UK: Springer, 1995, pp. 26–40.

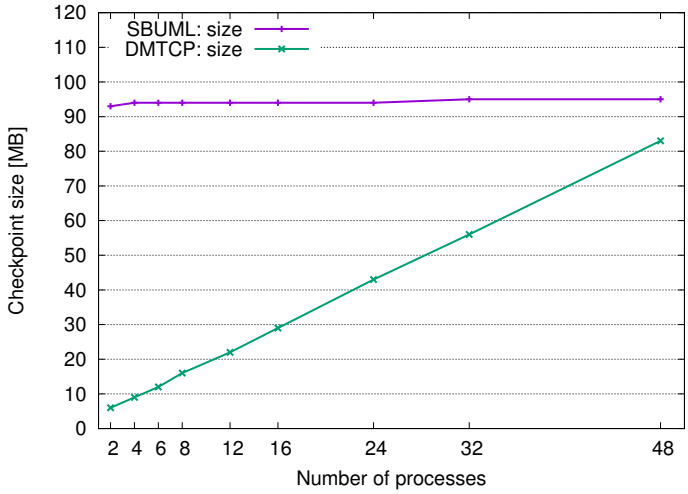
[35] N. Sebih, F. Weitl, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Software model checking of UDP-based distributed applications," in *Proc. 2nd Int. Symposium on Computing and Networking*. Shizuoka, Japan: IEEE, 2014, to appear.

[36] Z. Miller, T. Tannenbaum, and B. Liblit, "Enforcing Murphy's Law for advance identification of run-time failures," in *USENIX Annual Technical Conference*. Boston, Massachusetts: USENIX Association, 2012.

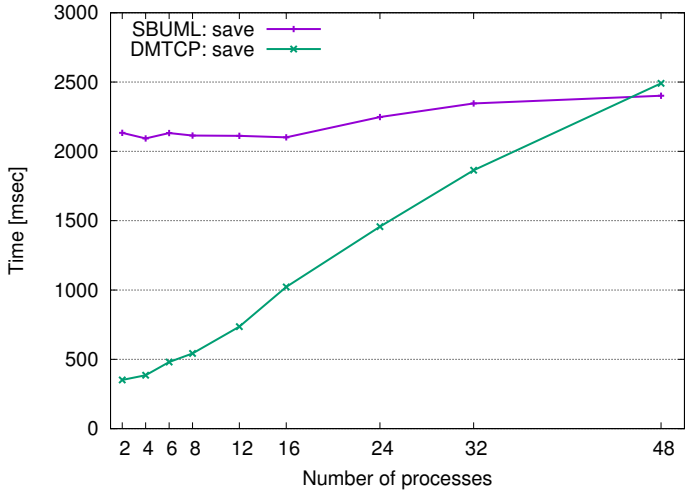
[37] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector." in *Proc. 2009 IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2009)*. Estoril, Portugal: IEEE, 2009, pp. 379–388.

[38] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, USA: Morgan Kaufmann Publishers, Inc., 2006.

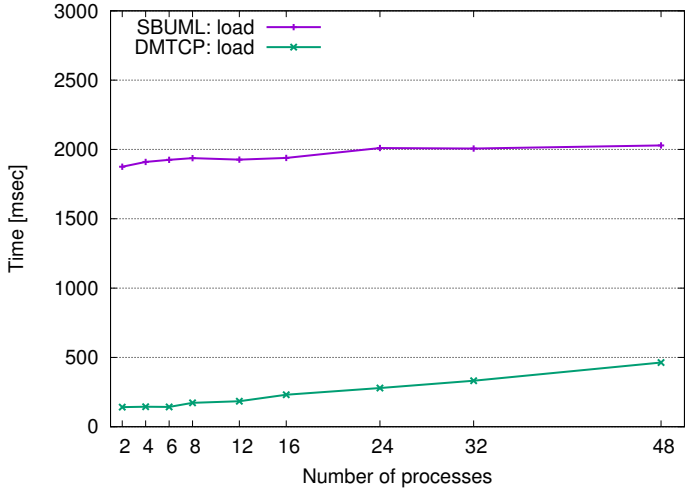
[39] "Checkpoint/Restore in User-space," 2014, http://criu.org/.



(a) Checkpoint size.

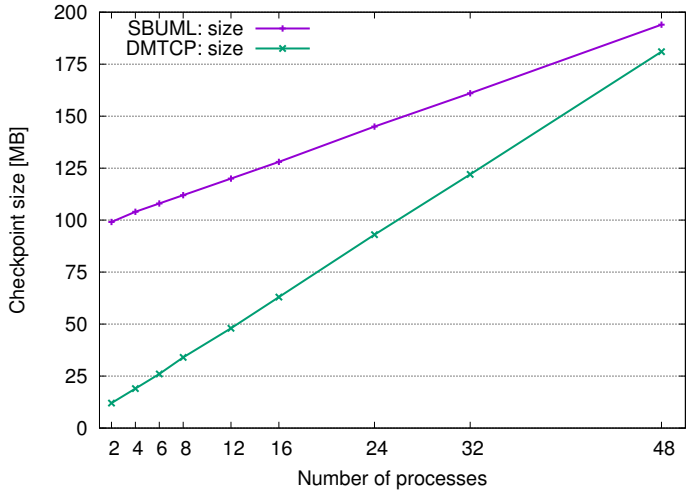


(b) Time to save checkpoint.

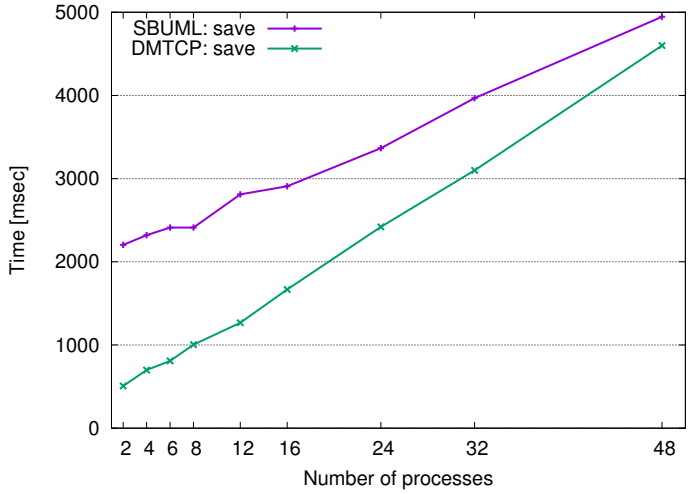


(c) Time to restore from checkpoint.

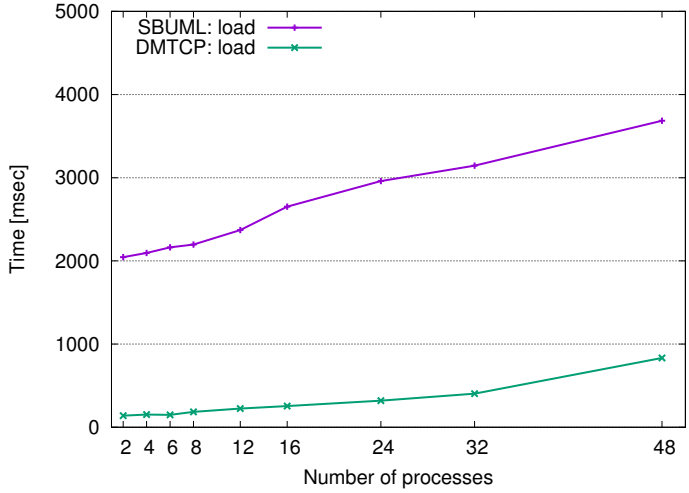
Figure 9: Performance for a given number of processes per checkpoint.



(a) Checkpoint size.



(b) Time to save checkpoint.



(c) Time to restore from checkpoint.

Figure 10: Performance for a given number of processes per checkpoint (each process uses 2 MB of random data).

Table 6: Comparison of existing fault injection tools.

Tool	**Target**	**Approach**	**Technology**	**SUT platform**
CFI [5]	network I/O	exhaustive	checkpointing (DMTCP or SBUML)	binaries
D-Cloud [29]	virtualized hardw.	user-defined	replay (multiple executions)	binaries
Enforcer [33]	network I/O	exhaustive	replay (multiple executions)	Java bytecode
Explode [18]	file I/O	exhaustive	depth-first search (using fork)	binaries
FIG [30]	library calls	user-defined	replay (multiple executions)	binaries
FTAPE [34]	I/O, others	randomized/heuristic	replay (multiple executions)	binaries
JPF + centralizer [6]	network I/O	exhaustive	checkpointing (using Java PathFinder)	Java bytecode
JPF net-iocache [7, 35]	network I/O	exhaustive	checkpointing (using Java PathFinder)	Java bytecode
JPF-based [28]	exceptions	exhaustive	checkpointing (using JPF with abstract interpr.)	Java bytecode
LFI [36]	kernel actions	randomized	replay (multiple executions)	binaries
Murphy [37]	library calls	randomized	replay (multiple executions)	binaries
Workload generator [32]	network I/O	evolutionary search	replay + workload generation	C++
Xception [31]	library calls	user-defined	replay (multiple executions)	binaries

Table 7: Classification of tools based on their state space generation (exhaustive, user-defined, or randomized) and implementation approach (replay or checkpointing).

	exhaustive	user-defined	randomized
replay	Enforcer [33] Workload generator [32]	D-Cloud [29] FIG [30] Xception [31]	FTAPE [34] LFI [36] Murphy [37]
checkpointing	**CFI** (this work) [5] Explode [18] (using fork) JPF + centralizer [6] JPF net-iocache [7, 35] JPF-based [28]		

Table 8: Classification of tools based on their implementation approach (replay or checkpointing) and platforms supported (Java or C++/binaries).

	Java	C++ or binaries
replay	Enforcer [33]	D-Cloud [29] FIG [30] FTAPE [34] LFI [36] Murphy [37] Workload generator [32] Xception [31]
checkpointing	JPF + centralizer [6] JPF net-iocache [7, 35] JPF-based [28]	**CFI** (this work) [5] Explode [18] (using fork)