Using Checkpointing and Virtualization for Fault Injection

Cyrille Artho*, Masami Hagiya†, Watcharin Leungwattanakit†, Eric Platon‡,
Richard Potter†, Kuniyasu Suzaki*, Yoshinori Tanabe§, Franz Weitl¶, and Mitsuharu Yamamoto¶

*AIST/RISEC, Amagasaki/Tsukuba, Japan
c.artho@aist.go.jp, k.suzaki@aist.go.jp
†The University of Tokyo, Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp, le.watcharin@gmail.com, potter.richard@gmail.com
‡Independent, Tokyo, Japan
eric.platon@gmail.com
§National Institute of Informatics, Tokyo, Japan
y-tanabe@nii.ac.jp
¶Chiba University, Chiba, Japan
franz@chiba-u.jp, mituharu@math.s.chiba-u.ac.jp

***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**

I. Introduction

Fault injection [1] simulates input/output failures to analyze the robustness of systems against network problems. 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 greatly influences the size of checkpoints and the performance of fault injection. 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 [2] and SBUML [3] platforms.
3) Our experiments demonstrate how different types of virtualization platforms offer trade-offs in applicability against performance and scalability.

This paper is organized as follows: Section II gives more background on fault injection. Section III introduces the architecture of CFI, from which we describe concrete implementations in Section IV. Experiments using our tools are described in Section V, while Section VI covers related work. Section VII concludes this paper.

II. Background

Fault injection simulates defects; modern virtualization tools have capabilities that lend themselves to the implementation of fault injection tools.

A. Fault injection

In *fault injection* [1], faults may come from a wide range of sources, from hardware defects to implementation defects. 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 [4], [5].

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 injected by modifying the code of the system under test (SUT) or the library it uses.

B. 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 then be used as if it was the concrete one. 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 [6], Xen [7], VMware [8], the Java Virtual Machine [9], and Java WebStart [10]. The virtual resource is often called the "host", while the user of the virtual resource is the "guest".

Various degrees of virtualization exist; full virtualization provides the same service as the concrete resource. Our work targets the verification of distributed applications, using full virtualization at the OS and process level including I/O.

A *snapshot* represents system information in a summarized way. A *checkpoint* is a snapshot from which a system can be restored to resume execution. 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 [11]. More capable checkpointing tools can create snapshots of multiple processes [12]. Our work relies on two of these tools: DMTCP [2] and SBUML [3].

DMTCP (Distributed Multi-Threaded Checkpointing) virtualizes processes [2]. 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) [3] 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.

C. 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 restore a program to a previous state, to investigate alternative outcomes of a choice. This can be done in two ways:

1) *Replay:* The history of inputs to reach each state is recorded. 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 [13].

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.

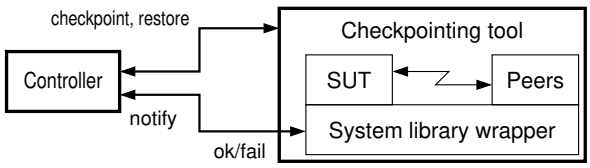

Figure 1: Architecture of checkpoint-based fault injection.

III. 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.

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

1) Decisions taken at each choice point, by interacting with a system library wrapper that virtualizes I/O.
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.[1]

A. Controller

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 1). The algorithm takes the SUT and its peers, along with a set of choice points. It explores 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.[2]

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

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

[2]Non-termination is handled by imposing a CPU time limit on the SUT.

Algorithm 1 Generic 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 2 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)
```

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 1 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 1 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 IV for details.

B. Wrappers to control choice points

Fault injection at choice points is implemented by virtualizing I/O operations. We use *wrapper functions* for each library function that is fault injected. The wrappers are called instead of the original function and either inject a fault (to simulate a failure) or call the original function (to simulate success). The controller (see Fig. 1) determines the outcome of the wrapped function (see Algorithm 2).

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. 2, 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.[3]

C. Summary

The controller manages the state space for fault injection. Choice points are managed by wrapping library functions that are fault injected. When a choice point is hit while the SUT is executing, execution is transferred to the controller. The wrapper obtains the desired simulation result from the controller, which resumes the execution of the SUT.

IV. Implementation Using DMTCP or SBUML

We present two fault injection tools based on CFI, using on DMTCP [2] and SBUML [3]. The first tool uses process-level checkpointing while the second one uses OS-level virtualization. The different level of virtualization has consequences for the implementation of fault injection, control of the virtualized resources, and, more subtly, communication between the virtualization tool and the controller.

A. DMTCP-based fault injector

DMTCP virtualizes a group of processes [2] and controls both process execution and communication between processes it manages. A checkpoint in DMTCP includes communication links between all captured processes. 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. 2).

1) Controller: Our controller determines program correctness by the *exit status* of the SUT: A status from an assertion error (134), segmentation violation (139), etc., is 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. The state space of the search (checkpoints to be explored, and visited choices) is managed by keeping copies of checkpoints as files.

[3]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.

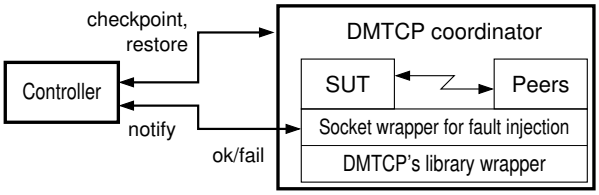

Figure 2: DMTCP-based fault injection tool.

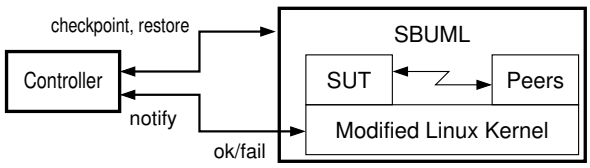

Figure 3: SBUML-based fault injection tool.

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. 2). 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 2 therefore calls DMTCP's wrapper function as `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`. For I/O on file descriptors, we check the result of `fstat` to limit fault injection to sockets. Internal control messages used by DMTCP are also exempt from fault injection.

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.

B. SBUML-based fault injector

Our second implementation uses SBUML, which virtualizes an entire operating system [3]. 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.

1) Controller: The controller needs to communicate with SBUML to control system execution (see Fig. 3). 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 `/tmp/SUT-exit-code`. Higher-level scripts running on the host can then easily poll and read such files.

The controller 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, which is used by the controller as a starting point. Once this initial-state snapshot has been created, our tool executes fault injection on a given target process, identified by its name.

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

Table I: Comparison between both tools.

Tool	Wrapper	Controller
DMTCP-based fault injector	C/C++ 600 LOC	Java 700 LOC
SBUML-based fault injector	C 70 LOC	bash 390 LOC

Table II: 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

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. 3 shows how the communication connects the components.

3) Communication: In this tool, the controller manages SBUML via files that are memory-mapped to the kernel running inside SBUML. The file system of the guest system is made visible to the host using *hostfs.*

C. Comparison

DMTCP is written in C and C++, with about 32,000 lines of code. 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 [14].

Reusing checkpointing tools greatly reduces the development effort for a fault injection tool. Our implementation uses only about 1,300 (DMTCP) and 460 (SBUML) lines of code, which is about 3–4 % of the size of the checkpointing software used (see Table I). 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.

V. 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 minimalistic server (see Table II).

A. Tool comparison

We inject faults in five server programs (see Table II) 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 III 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 such as the process ID 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. 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. In these cases, the outcome of the analysis is consistent across multiple executions.

B. Discussion

We have evaluated our two CFI tools. We observe that process-level checkpointing is more lightweight and provides a better performance on our benchmarks, but does not virtualize certain low-level resources such as process IDs faithfully; these are needed for certain applications.

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

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.

VI. Related Work

Checkpoint-based fault injection (CFI) is closely related to the usage of fault injection inside Java PathFinder (JPF), an analysis platform for Java bytecode [4], [15]. JPF targets non-determinism from concurrency and optionally from undetermined input and possible exceptions [16].

CFI uses virtualization tools to apply the same concept to binaries. This is similar to other fault injection tools for networked systems [17], [18], [19]. CFI differs in that we one tool to control all processes in a distributed system.

Other such work combines inputs (workload) generation with fault injection [20]. This covers a wider range of problems than CFI, at a higher computational cost.

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 [17], [18], [19], [22], [23], [24]. This means that only a subset of all possibilities is explored.

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 [13], [21], sometimes combined with event workload generation [20]; or by using a platform that provides checkpointing internally [4], [5], [15]. Our contribution is a general architecture that can be adapted to various checkpointing tools, making the approach less dependent on a given execution platform.

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 IV).

VII. 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.

Future work includes more options such as 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 [25].

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.

References

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

[2] 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.

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

Table IV: Comparison of existing fault injection tools.

Tool	Target	Approach	Technology	SUT platform
Checkpoint-based fault injection (CFI)	network I/O	exhaustive	checkpointing (DMTCP or SBUML)	binaries
D-Cloud [17]	virtualized hardw.	user-defined	replay (multiple executions)	binaries
Enforcer [21]	network I/O	exhaustive	replay (multiple executions)	Java bytecode
Explode [13]	file I/O	exhaustive	depth-first search (using fork)	binaries
FIG [18]	library calls	user-defined	replay (multiple executions)	binaries
FTAPE [22]	I/O, others	randomized/heuristic	replay (multiple executions)	binaries
JPF + centralizer [4]	network I/O	exhaustive	checkpointing (using Java PathFinder)	Java bytecode
JPF net-iocache [5]	network I/O	exhaustive	checkpointing (using Java PathFinder)	Java bytecode
JPF-based [15]	exceptions	exhaustive	checkpointing (using JPF w. abstract interpr.)	Java bytecode
LFI [23]	kernel actions	randomized	replay (multiple executions)	binaries
Murphy [24]	library calls	randomized	replay (multiple executions)	binaries
Workload generator [20]	network I/O	evolutionary search	replay + workload generation	C++
Xception [19]	library calls	user-defined	replay (multiple executions)	binaries

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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