Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures

Hiep Nguyen, Daniel J. Dean, Kamal Kc, Xiaohui Gu
Department of Computer Science
North Carolina State University
{hcnguye3,djdean2,kkc}@ncsu.edu, gu@csc.ncsu.edu

Abstract

Online service failures in production computing environments are notoriously difficult to debug. When those failures occur, the software developer often has little information for debugging. In this paper, we present Insight, a system that reproduces the execution path of a failed service request onsite immediately after a failure is detected. Upon a request failure is detected, Insight dynamically creates a shadow copy of the production server and performs *guided binary execution exploration* in the shadow node to gain useful knowledge on how the failure occurs. Insight leverages both environment data (e.g., input logs, configuration files, states of interacting components) and runtime outputs (e.g., console logs, system calls) to guide the failure path finding. Insight does not require source code access or any special system recording during normal production run. We have implemented Insight and evaluated it using 13 failures from a production cloud management system and 8 open source software systems. The experimental results show that Insight can successfully find high fidelity failure paths within a few minutes. Insight is light-weight and unobtrusive, making it practical for online service failure inference in the production computing environment.

1 Introduction

Although online services[1] are expected to be operational 24x7, recent production service outages [2, 1] show great challenge to meet such an expectation. Unfortunately, when those online services experience failures in a production computing environment, the software developer is often given little information for debugging.

Particularly, we focus on *non-crashing failures* where the server does not crash but fails to process some requests. Different from crash failures that often receive immediate attention, those non-crashing failures often go unnoticed. We observe that those failures are common in online services based on our experience with the virtual computing lab (VCL) [3] which is a production cloud computing infrastructure. Users who experience frequent service failures will be seriously discouraged to use the service again. Most production servers are well engineered to avoid fatal crash failures and strive to capture all the request failures with error messages. However, those error messages do not tell us *why* a service request has failed and can be misleading sometimes [7, 36].

To debug a production-run failure, software developers generally need to reproduce the failure at the developer-site to understand what happened during the production run in order to infer the root cause. Much effort has been devoted to explore the right balance between recording overhead and debugging effectiveness, ranging from deterministic record-replay techniques [19, 17, 18, 13] to partial record-replay [6, 14]. However, production infrastructures are often reluctant to adopt any intrusive system recording approaches due to deployment and privacy concerns.

In this paper, we present *Insight*, a system that can infer the execution path of a failed service request *inside* the production environment *without* any intrusive system recording. We view Insight as a first-step failure inference tool for the developer to gain useful knowledge about *how* a service request fails in the production computing environment. Insight can significantly expedite the debugging process by narrowing down the scope of diagnosis from thousands of functions to a few of them. Moreover, the failure paths reproduced by Insight can be fed into a debugger (e.g., GDB) or a symbolic execution engine [10, 8] for further analysis.

The key idea of Insight is to perform *in-situ* failure path inference *inside* the production environment. The rationale behind our approach is that the production com-

[1] The online services considered in this paper refer to those request and response services such as a web server or a virtual machine (VM) reservation service in an infrastructure-as-a-service cloud.

puting environment provides many useful clues for us to perform failure inference more efficiently than offline approaches. Those clues include both *environment data* (e.g., input logs[2], configuration files, state of the faulty component, interaction with other production servers such as database query results) and *runtime outputs* (e.g., console logs, system call traces). Our experiments show that using environment data and runtime outputs can greatly *reduce the failure path search scope* and provide *important guidance* for us to find the correct failure path.

When a request failure is detected, Insight dynamically creates a shadow component of the faulty production server which produced the error message or was identified by an online server component pinpointing tool [11, 20, 27]. We detect a request failure by intercepting error messages or employing system anomaly detection tools [32, 15]. Since the production server is still alive during non-crashing failures, the shadow component can inherit the failure states of the faulty production server. Moreover, the shadow component allows us to decouple failure inference from the production operation. The production server can continue to process new requests without worrying about losing important diagnostic information. Our current prototype implements dynamic shadow component creation by augmenting the live virtual machine (VM) cloning technique [9, 22, 26]. Our scheme allows the shadow component to acquire environment data and runtime outputs from the production environment while imposing minimum disturbance to the production operation.

Insight proposes a novel *guided* binary execution exploration scheme that can efficiently leverage the production environment data and runtime outputs as guidance to search the failure paths. We make careful design choices in our failure inference algorithm in order to meet the following requirements: 1) *binary-only* since we cannot assume source code is available in the production environment; 2) *fast* path search in order to leverage the "fresh" environment data at the failure moment (i.e., the environment does not change much and the failure-triggering inputs or similar inputs are still in the buffer of the recent input log.); 3) no *intrusive* recording; and 4) support *both interpreted and compiled* programs.

Our guided binary execution exploration starts by replaying the last input in the input log when a failure is detected. However, Insight does not require the exact failure-triggering input to find the failure path since our binary execution exploration scheme can inherently handle incorrect environment data (e.g., different inputs, outdated or missing query results). During replay, we use the runtime outputs as guidance to stop searching along wrong paths, that is, if the replay produces a mismatched output, we roll back the execution to the previous branch point and flip the branch condition value (e.g., from `true` to `false`) to search a different path. If no matched path is found using the current input, we replay the next input in the input log and repeat the above process. We also support concurrent multi-path search to further shorten the failure path search time. Multi-path search also allows Insight to find multiple candidate failure paths that match the output of the production run.

We consider both *console log messages* and *system call traces* in the output matching. Most production servers already record console logs. If the production program produces many console log messages, our experiments show that Insight can rely on console log messages to produce high fidelity failure paths. However, if the production program includes very few console log messages, we propose to use system calls as hints to search paths between console logs. We chose to match system calls because they often represent key operations and can be collected using kernel-level tracing tools [5, 16] with low overhead ($< 1\%$).

We intentionally skip the constraint checking during the binary path search in order to achieve *fast* failure path inference in the production computing environment. With the help of environment data, we observe that Insight only needs to flip a small number of branches and the chance of finding an infeasible path is small. To filter out infeasible paths in our final result, we can apply constraint solver [12, 29, 24] to the candidate failure paths found by Insight, which is much faster than applying constraint solver during the path search.

We make the following contributions in this paper:

- We propose to perform *in-situ* failure path inference using a dynamically created shadow server *inside* the production computing environment.
- We present a *guided* binary execution exploration algorithm that can use available *environment data* (e.g., inputs, configuration files, states of interacting components) and *runtime outputs* (e.g., console logs, system calls) as guidance to quickly find the failure path over binary code directly.
- We evaluate Insight using real system failures. Experiments show that in-situ failure path inference is feasible. Insight can efficiently use the environment data and runtime outputs when they are present to find high fidelity failure paths within minutes.

The rest of the paper is organized as follows. Section 2 compares our work with related work. Section 3 presents the design and implementation of Insight. Section 4

[2] We observe that most production servers buffer a set of recent inputs. Although it might be impractical to assume the input log access for *offline* diagnosis (e.g., the privacy concern), it is easy to acquire the input log within the production computing environment.

presents the experimental results. Finally, the paper concludes in Section 5.

2 Related Work

Production-run failure debugging is a well known challenging task. In this section, we focus on reviewing the work that is most related to Insight and describing the difference between Insight and previous approaches.

Triage [33] first proposed an onsite production run failure diagnosis framework. It uses checkpoint-replay with input/environment modification to perform just-in-time problem diagnosis by comparing good runs and bad runs. Although Insight shares the same idea of onsite failure analysis with Triage, Insight differs from Triage in the following major ways. First, Triage performs onsite debugging on the production server directly, which can cause significant downtime to the online service. In contrast, Insight creates a shadow server to decouple the failure inference from the production operation. Second, Insight does not rely on repeated replays with input/environment modifications, which can incur a long failure analysis time and sometimes difficult to achieve in production systems. In comparison, Insight provides a fast binary execution exploration approach that uses the environment data and runtime outputs as guidance to search the failure paths on a dynamically created shadow component.

Alternatively, previous work (e.g., [19, 17, 18, 13, 30, 6, 28, 25, 39]) has proposed to introduce application-level or system-level instrumentation and infer the failure path based on instrumentation data. However, large-scale production computing environments are reluctant to adopt continuous intrusive system recording approaches due to overhead and deployment concerns. For example, *Aftersight* [13] proposed to decouple complex program analysis from normal executions using VM record and replay techniques. However, VM recording can impose high overhead to the normal production execution (e.g., worst case overhead reached 31% and 2.6x for some workloads [13]). Crameri et al. [14] proposed to use static and dynamic analysis to identify those branches that depend on input and only record those branches for failure reproduction. In comparison, Insight does not record any branch during the production run but instead exploits production environment data and runtime outputs to find the correct failure path onsite immediately after the failure occurs.

Another alternative is to perform *offline* failure inference using static source code analysis [37, 38]. For example, *Sherlog* [37] uses static source code analysis to infer the possible failure paths from console logs. *ESD* [38] uses program source code and bug reports (i.e., core dump information) to reproduce a failure execution. ESD first statically analyzes the source code to infer the control path capable of reaching the bug location, and then symbolically executes the program along the inferred control path to reproduce the failure execution. Because reproducing a production run failure outside the production environment is challenging [33], offline analysis cannot leverage the production environment data (e.g., inputs, configuration files, interaction results) or some runtime outputs that are difficult to obtain offline (e.g., system calls). Moreover, it is difficult for the offline approach to localize environment issues (e.g., network failure, wrong database query results). S^2E [12] provides an in-vivo multi-path analysis framework using selective symbolic execution over binaries for finding all potential bugs. In contrast, Insight aims at quickly finding the execution path for a specific occurring production-run failure. S^2E also does not consider runtime outputs when finding the failure path.

We view Insight as a first-step light-weight failure inference tool that can be used inside the production environment. We can apply the static/dynamic program analysis or symbolic execution to the candidate failure paths found by Insight to further validate the feasibility of the failure paths and localize root cause related branches.

3 System Design and Implementation

In this section, we describe the design and implementation details of the Insight system. We first present the dynamic shadow server creation scheme. We then describe our guided binary execution exploration algorithms.

3.1 Dynamic Shadow Server Creation

When a service request failure is detected, Insight dynamically creates a shadow component of the production server on a separate physical host using live VM cloning [9, 22]. Since Insight targets non-crashing failures and performs immediate cloning, we assume that the state of the shadow component is similar to the state of the production server when the failure occurs. We found this assumption holds for all the server failures we tested in our experiments.

The current prototype of Insight uses a pre-copy live KVM VM cloning system [26]. However, we can integrate Insight with other VM cloning techniques easily. Insight only requires a brief stop-and-copy phase (e.g., $<$ 100 milliseconds [26]) where the production component is paused temporarily for transferring any remaining dirty pages. During the stop-and-copy phase, the production server just pauses its processing but can continue to receive the user requests in its input buffer. For all the server systems we tested, Insight can complete the

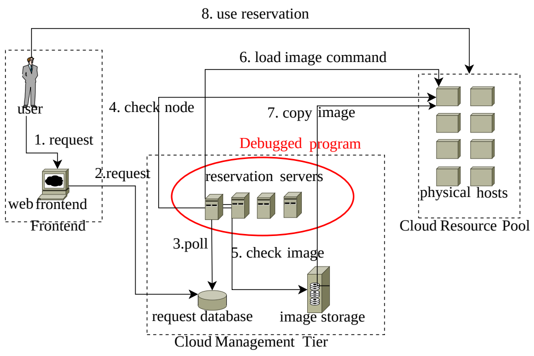

Figure 1: Our field study production server: VM reservation servers in the VCL cloud computing infrastructure [3]. The user makes a VM reservation request via a web interface. The request is stored in a database which is continuously polled by the reservation server. The reservation server forks a new process for handling each VM reservation request. First, the reservation server allocates a set of physical hosts for the user. If these hosts do not have the VM images required by the user, the reservation server then loads requested images from an image database. The reservation server then starts the `ssh` service and creates a user account for the user.

whole shadow component creation process within tens of seconds. Additionally, Insight performs transparent fast disk cloning to make the shadow component completely independent of the production server [26].

After the cloning is done, we need to reconfigure the shadow server to prepare it for the failure reproduction. Note that all the reconfigurations do not require any modification to the server software. Because live VM cloning makes the shadow server inherit all the state from the production server, which includes the IP address, the shadow server may immediately send out network packets using the same IP address as the production server, causing duplicate network packets and application errors. To avoid this, we first disconnect the network interface of the shadow server, clear the network buffer, and then reconnect the network interface of the shadow server with a new IP address.

To leverage the production environment for failure reproduction, we need to allow the shadow server to interact with other servers in the production environment for retrieving needed information. Figure 1 shows our field study production server which is a VM reservation server in an infrastructure-as-a-service cloud. The reservation server needs to interact with a MySQL database server to search for available physical hosts, look up the VM image name, and update the reservation state. Insight registers the shadow server with the database server using event-driven application auto-configuration [26]. Other interactions can be enabled in a similar way.

If the interaction requires the shadow server to read information from the environment (e.g., query from a database), the interaction is allowed. However, if the interaction requires the shadow server to update some information in the environment (e.g., write to a database), the interaction will be filtered to avoid undesired disturbance to the production server. We use an interaction filtering proxy to intercept outputs from the shadow server and drop selected outputs based on the query type. The proxy runs outside the shadow server software but on the same physical host with the shadow server. For example, our field study production server is written in Perl. We implemented the interaction filtering proxy within the Perl interpreter. We can also perform interaction recording on the *shadow server* to log important environment data which will be helpful for developers to diagnose a failure caused by an environment issue.

Insight is resilient to false alarms by providing light-weight runtime failure path inference and flexible cloning. If a false alarm is confirmed by the online anomaly detection tool before the shadow server is started, we simply cancel the live VM cloning operation. If a false alarm is confirmed after the shadow server is already started, we issue a delete command to the shadow server and release all resources allocated to the shadow server. In our field study server system, we use the critical error messages for detecting failures, which has few false positives [21]. We can also combine the error message detection with other failure prediction tools [31] to further reduce the false alarms.

3.2 Guided Binary Execution Exploration

Insight performs guided binary execution exploration in the shadow component to find the failure path. The execution exploration engine intercepts conditional jump statements (e.g., JZ, JNE, JE) in the binary code and explores different execution paths by manipulating the jump conditions (`true` or `false`). We assume all the conditional statements including the `switch` statements are translated into one or multiple conditional jump statements in the binary code. For example, in C/C++ program, we can compile the code using the `fno-jump-tables` option in `gcc`.

To start the execution exploration, we first replay the last input in the input log when the failure is detected. We employ an input replay proxy to retrieve the input log from the production server when the failure is detected. As mentioned in the Introduction, most production servers buffer recent inputs in an input log file. For example, a web server stores its input (i.e., HTTP requests) in the `access_log` file. For VCL reservation server, the inputs (i.e., VM reservation requests) are stored on a database server. Although our experiments show that inputs play a crucial role in the failure path inference, Insight does not require the exact failure-triggering input to find the failure path.

During the replay, we check whether the shadow component produces the same outputs (i.e., console log messages, system call sequences) as the failed service request. We will describe the output matching scheme details in the next section. A replayed path can produce mismatched outputs either because we did not replay the exact failure-triggering input or because some environment data (e.g., database content) was changed during the shadow component creation. We use an *unmatched output as a hint* to stop searching along a wrong path. Under those circumstances, the execution is rolled back to the previous branch point and we flip the branch condition to search a different path. If rolling back to the previous branch point still cannot produce any matched failure path, we rollback to the branch point before the previous branch point and so on. To avoid redundant search, we stop the rollback process when we see the previous console log message again. If no matched path is found using the current input, we replay the next input in the input log and repeat the above process. To support the above mechanism, Insight performs process checkpointing at each branch point and each console log output. We implement the process/thread checkpointing using $fork$.

Insight supports concurrent multi-path search to achieve fast failure reproduction. We implement the concurrent multi-path search by using a set of probing processes/threads called probes to explore different execution paths simultaneously. When the probe encounters a conditional jump statement, it forks a new child probe for exploring both the `true` and the `false` branches concurrently. To avoid overloading the system with a large number of concurrent searches, we set a concurrency quota *CQ* to limit the number of probes that can simultaneously run. When the number of probes exceeds CQ, we make the parent probe wait and allow the child probe to explore either the `true` or `false` branch. If the child probe produces an unmatched output, we kill the child probe to discontinue the search along the wrong path and release one concurrency quota. If the parent probe of the terminated child probe is waiting for the quota, the parent probe will be signaled to continue its exploration. When a probe produces the next matched output (i.e., console log message or system call), we stop the exploration and switch back to concrete execution mode (i.e., continue the execution without forking).

If an explored path contains a loop, Insight forks a new child probe at the beginning of each iteration by default. The parent probe will then exit the loop (i.e., the `false` branch) and allow the child probe to continue to execute the next iteration of the loop (i.e., the `true` branch). However, if the program does not produce any console log messages or system calls within the loop, Insight will never get any hint on when to stop exploring the loop. To avoid unnecessary loop explorations, Insight performs loop detection by checking for repeated program counters within one function. If no console log message or system call is produced within the loop, we disable exploration for that loop branch statement (i.e., do not fork new child probe) and let the loop exit naturally as its normal execution.

When a probe produces the same complete console log and system call sequences as the failed request, Insight marks the execution path explored by the probe as one *matched failure path*. Our approach can also find multiple matched failure paths simultaneously. The failure path inference will be terminated after the target number of matched failure paths are found or the search process times out. We also annotate each reproduced path with useful diagnostic information such as which branch points were manipulated by our exploration process and what the environment values were when the branch points were flipped by our system. Developers can use this information to decide the fidelity of the reproduced paths and perform informed value inferences.

Since Insight works on binaries directly, most Insight components can be applied to compiled or interpreted programs written in different languages without any modification. The only program-specific parts are how to intercept branch statements and change branch conditions. Insight currently supports Perl and C/C++ programs. For Perl programs, we modified the Perl interpreter to intercept the conditional jump statement. The jump condition value is stored in the interpreter's execution stack. We modify the jump condition value by changing the execution stack value. For C/C++ programs, Insight uses the Pin tool [23] to intercept the conditional jump statements and modify the jump conditions by changing the appropriate flags (i.e., jump flag, carry flag, overflow flag, and parity flag) in the EFLAGS register. Note that the above system modification and instrumentation are only applied to the shadow server during the execution exploration time. Insight does not perform any modification or instrumentation to the production server.

3.3 Runtime Output Matching

Insight uses runtime outputs as hints to check whether it explores a correct or incorrect path. We chose to match two different types of runtime outputs: console log messages [35, 37] and system calls for the following reasons. Production systems often produce console log messages for debugging production-run failures [35, 37]. In today's practice, console logs often provide the sole information source for diagnosing production run failures. Since console log messages are inserted by software developers for recording operations considered to be important, they are often able to provide useful clues about key program execution states. However, we also observe some systems (e.g., open source software) contain a limited number of log messages. Under those circumstances, we propose to match system call traces because system calls can be easily collected using kernel-level tracing tools with negligible overhead ($<$ 1% CPU load) and system calls often denote the key operations in the program. Different from user-level tracing tools such as ptrace [4], kernel-level tracing tools impose little overhead by avoiding context switches. We use SystemTap [5] in our current prototype.

During binary execution exploration, we continuously match the console log messages and system call sequences produced by the explored execution path with those from the failed production-run request. For console log matching, we adopt the same strategy as previous work [35, 37] by only considering the static text parts called *message templates* since the variable parts (e.g., timestamp, variable values) typically differs over different runs. Those message templates can be easily extracted from the source code and provided to Insight by software developers. Alternatively, we can extract the message templates directly from log files [34], which is however orthogonal to our work. We can also leverage parameter run-time values in the console log messages to extract more hints about the failure. We might be able to increase the failure path accuracy using those parameter values by incorporating Insight with taint analysis techniques. However, doing so will probably increase the runtime overhead. Our current results show that Insight can still successfully infer the failure paths without using those parameter values.

If the console log is too sparse, Insight still faces the challenge of large exploration scope. Thus, we use system calls as hints to guide our path search between any two consecutive console log messages L_1 and L_2. We observe that each console log message is written into the console log file using a sequence of `sys_write` system calls. The system call sequence S in-between those two groups of `sys_write` calls are marked as the system call sequence between L_1 and L_2. We use `readlink` and file descriptor contained in each `sys_write` to identify whether it writes into the console log file. When we perform failure path search between L_1 and L_2, we match the system call sequence S. We currently only consider system call types when we perform matching. We could also consider system call arguments or return values, which, however, might increase the system call tracing and matching overhead significantly.

When a mismatched system call is encountered, we roll back the exploration to the previous branch point and flip the branch condition to execute a different path. During our experiments, we observe that requiring an exact match sometimes prevents us from finding any matched path. The reason is that the same function call such as `malloc` might invoke slightly different numbers of system calls (e.g., `mmap`) depending on the application's heap usage. During those circumstances, we allow k mismatches (measured by string edit distance) to occur during system call sequence matching. We start from $k = 0$ and gradually increase k until we either find a matched path or our search times out. During our experiments, we find k needs to be no more than 2.

4 System Evaluation

In this section, we present the experimental evaluation for the Insight system. We first present our evaluation methodology followed by our experimental results.

4.1 Evaluation Methodology

Case study systems. We first test Insight using the virtual computing lab (VCL) [3] which is a production cloud computing infrastructure. VCL has been in production use for 9 years and has over 8000 daily users. Figure 1 shows the architecture of VCL. The key control part in VCL is the cluster of reservation servers which are written in about 145K lines of Perl code. The database server is configured to allow access from hosts in the same subnet, thus allowing the access from the shadow component. In our experiments, we deploy the Insight system on all the reservation servers and perform the in-situ failure path inference over the reservation servers which produce the reservation failure messages.

We also test Insight with several real software bugs in a set of open source softwares (Apache, Squid, Lighttpd, PBZIP2, aget, and GNU Coreutils).

Case study failures. We evaluated Insight using a set of real failures listed in Table 1. We also report the number of function calls and branch points contained in each failure execution path along with the root cause function of each failure. Each failure contains one error message. In our experiments, we detect failures by intercepting error messages: console log messages containing *critical* or *fatal* keyword or are written into *stderr*.

Evaluation metrics. We first evaluate whether the reproduced failure execution path is useful for debugging by checking whether the reproduced execution shows the same failure symptom (i.e., throwing out the same error messages), covers the root cause functions and branch statements. We then evaluate the precision and efficiency of Insight using the following metrics:

System name	**System description**	**LOC**	**Failure path length**		**Num. of console log msgs**	**Failure description**	**Root cause function**
			Num. of functions	**Num. of branches**			
VCL (v2.2.1)	VM reservation server	145K	112	378	132	**Overlapping reservation failure**: User tries to request the same VM reservation twice.	*computer_not_being_used*
VCL (v2.2.1)	VM reservation server	145K	299	1331	290	**Network failure**: The management node fails to create the VM reservation on a physical host due to the network failure on the host.	*_ssh_status*
VCL (v2.2.1)	VM reservation server	145K	298	1328	409	**Authentication failure**. The management node is unable to login into the reservation host due to a missing public key.	*run_ssh_command*
VCL (v2.2.1)	VM reservation server	145K	147	601	178	**Image corruption failure**. The VM image file corresponding to the user request is corrupted and cannot be copied.	*load*
Apache (httpd-2.0.55)	Web server	176K	176	21212	1	**Authentication failure**. Apache rejects a valid request due to incorrect file name setting for AuthGroupFile option (#37566).	*groups_for_user*
Apache (httpd-2.2.0)	Web server	209K	164	4983	1	**CGI failure**. Apache does not handle a malformed header generated by CGI script correctly (#36090).	*ap_scan_script_header_err_core*
Squid (v2.6)	Web cache and proxy server	110K	588	19679	195	**Non-crashing stop failure**. Squid is not able to handle a long value of “ACL name” option (#1702).	*aclParseAcl List*
Lighttpd (v1.4.15)	Web server	38K	730	4308	3	**Proxy failure**. Lighttpd could not find the back-end server when configured as a reverse proxy for 1 backend server with round-robin policy (#516).	*mod_proxy_check_extension*
PBZIP2 (v1.4.15)	Multithreaded data compression	3.9K	41	58	14	**Decompression failure**. The program fails to decompress files with trailing garbage (#886625).	*decompress ErrCheck Single*
aget (v0.4.1)	Multithreaded download accelerator	1.5K	2	8	1	**Download failure**. The program fails to download a file when setting the number of threads bigger than the maximum concurrent connection allowed in the server holding the file.	*http_get*
rmdir (v4.5.1)	GNU coreutils	0.2K	2	24	2	**Option failure**. The program does not handle trailing slashes with the “-p”.	*remove _parents*
ln (v4.5.1)	GNU coreutils	0.6K	1	47	1	**Option failure**. The program does not handle “target-directory” correctly.	*do_link*
touch (v7.6)	GNU coreutils	0.5K	1	7	1	**Time failure**. The program rejects a valid input with the leap second.	*main*

Table 1: Real system failures used in our experiments. All the failures have one error log message produced during the failure run.

1) *Call path difference* denotes the deviation of the call path discovered by Insight from the original call path of the failed production run. The call path consists of a sequence of invoked functions during the execution. We used the *string edit distance* to measure the deviation between two compared call paths. We instrumented all the tested programs to record the original call path of the failed production run. Generally, the call path difference reflects how close the reproduced execution is to the original execution.

2) *Normalized branch difference*. We use the branch difference to denote the deviation at the branch level between the path reproduced by Insight and the original failure path. We also use the *string edit distance* to measure the branch difference between two execution paths. To perform comparison between different application failures, we normalize the branch difference of each failure using its maximum string edit distance between the reproduced path and the original path (i.e., no overlapping at all).

Generally, the call path difference and the branch difference reflect how close the reproduced execution is to the original execution. The branch difference is a more fine-grained comparison than the call path difference.

3) *Percentages of flipped branches* denotes the percentage of the branches whose conditions are manipulated by Insight due to incomplete environment information or different input.

4) *Exploration time* defines the time taken by Insight to discover the target number of the matched failure paths.

5) *Performance impact and overhead.* We evaluate the runtime performance impact of Insight to the production system by comparing the per-request processing time between with and without the Insight system. We also report the overhead of the Insight system in terms of additional resource consumptions.

Impact of environment data. To understand the impact of the environment information on the accuracy of our in-situ failure path inference, we compare the failure inference accuracy results under three different

Failure name	Environment setting	Call path difference	Branch difference	Cover root cause functions	Cover root cause branches
VCL overlapping reservation failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	0	Yes	Yes
	No environment data	0	0	Yes	Yes
VCL network failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	3.4%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
VCL Authentication failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	2.7%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
VCL Image corruption failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	3.1%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
Apache authentication failure	Original input	0	0	Yes	Yes
	Same input type + console log	17	66%	Yes	Yes
	Same input type + console log + system call	11	61.5%	Yes	Yes
Apache CGI failure	Original input	0	0	Yes	Yes
	Same input type + console log	140	41.8%	Yes	Yes
	Same input type + console log + system call	9	14.8%	Yes	Yes
Squid failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0.0001%	Yes	Yes
	Same input type + console log + system call	0	0.0001%	Yes	Yes
Lighttpd failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0.8%	Yes	Yes
	Same input type + console log + system call	0	0.8%	Yes	Yes
PBZIP2 failure	Original input	0	0	Yes	Yes
	Same input type + console log	1	4.4%	Yes	Yes
	Same input type + console log + system call	0	1.3%	Yes	Yes
aget failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0	Yes	Yes
	Same input type + console log + system call	0	0	Yes	Yes
rmdir failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	17.5%	Yes	Yes
	Same input type + console log + system call	0	5.3%	Yes	Yes
ln failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	25.3%	Yes	Yes
	Same input type + console log + system call	0	9.1%	Yes	Yes
touch failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	53.5%	Yes	Yes
	Same input type + console log + system call	0	0	Yes	Yes

Table 2: Summary of Insight failure path inference accuracy results.

environment contexts in the VCL system: 1) *complete environment data* where all the query results from the database are assumed to be the same during the whole failure path inference process; 2) *partial environment data* where all the database entries that are related to the failed reservation request are deleted to emulate the case when the failure inference is triggered after the reservation server clears up the failed requests. However, the shadow server can still access some general database information such as "computer load state" and "OS type" needed by the failure path finding; and 3) *no environment data* where all the query results from the database are unavailable. This emulates the case of offline failure reproduction.

Since all the C/C++ server failures are produced under stand-alone mode, we could not evaluate the impact of the environment data on the open source systems.

Impact of input. We evaluate the impact of the input by performing failure reproduction using the original failure triggering input or using a different input that does not trigger the failure but is of the same type as the original input.

We define the *same type of input* for different open source systems as follows: 1) *Apache authentication failure*: the same type of input is a http request to access a webpage with a correct AuthGroupFile setting; 2) *Apache CGI failure*: the same type of input is a http request to execute a normal CGI script; 3) *Squid failure*: we use a default configuration file with a normal "ACL name"; 4) *Lighttpd failure*: we use a http request using a reverse proxy with two back-end servers instead of one back-end server that makes the system fail; 5) *PBZIP2 failure*: we use a compressed file with no trailing garbage; 6) *aget failure*: we use a request that does not have restriction on the maximum concurrent connection; 7) *rmdir failure*: we use a command without the "-p" option; 8) *ln failure*: we use a command without the "target-directory" option; and 9) *touch failure*: we use an input that does not have a leap second.

Since the inputs (i.e., reservation requests) in the VCL system are stored in the database, they are considered as part of the environment data.

We conducted all the experiments on a computer cluster in our lab. Each cluster node is equipped with a quad-core Xeon 2.53GHz CPU, 8GB memory, and is connected to Gigabit network. Each host runs CentOS 6.2 64-bit with KVM 0.1.2. The guest VMs run CentOS 6.2 32-bit and are configured with one virtual CPU and 2GB memory. We repeated each experiment five times and report the mean and standard deviation values. In all experiments, we set the concurrency quota *CQ*=20.

4.2 Results and Analysis

Failure path accuracy result summary. Table 2 summarizes the accuracy of the failure paths reproduced by Insight for different failures. We observe that the environment data in VCL allows Insight to find the exact failure paths for all the VCL failures. With partial environment data, Insight can still achieve high accuracy with 0 call path difference and small ($<$ 5%) branch difference. However, when we remove all the environment data, emulating the offline failure reproduction situation, we cannot find any matched path for three out of four VCL failures after searching for several hours. This indicates that environment data plays a crucial role in timely failure path finding because they can greatly reduce the search scope for the binary execution exploration.

For the open source software systems, we observe that with the original failure-triggering inputs, Insight can always reproduce the exact failure path for each failure. When given the same type of input (see Section 4.1 for the definition of the same input type), Insight can still reproduce high fidelity failure paths with 0 call path difference and small ($<$ 10%) branch difference for most failure cases. The only exceptions are those failures that include only 1 error message without any other console log messages. This is expected as Insight has too few runtime outputs to guide the exploration. However, we observe that system call sequences can greatly help improve the failure path inference accuracy for the failure cases where sparse console logs are present. The branch difference reduction can be up to 100% (i.e., the branch difference of the touch time failure is reduced from 53.5% to 0).

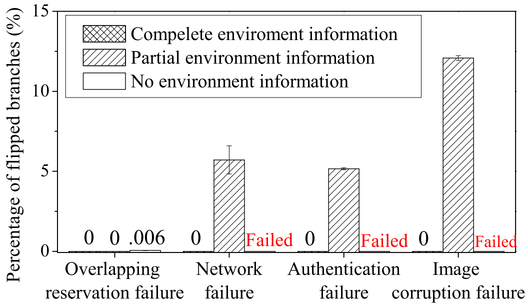

Figure 2: Percentage of flipped branches for VCL failures.

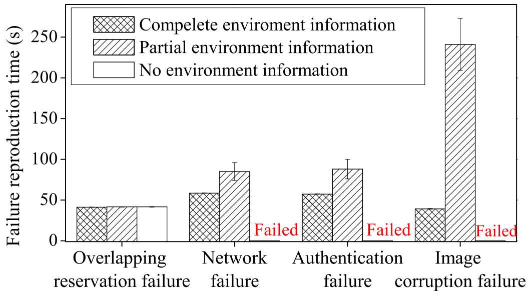

Figure 3: Total failure reproduction time (i.e., the shadow component creation time + failure path search time) for reproducing VCL failures.

We then validate whether the failure paths reproduced by Insight cover the root cause functions and branches by manually analyzing the source code. We observe that the failure paths found by Insight always cover the root cause functions and branches. Another interesting observation we observe is that the root cause branch points often do not appear right before the error message is produced, but reside in the middle of the execution path. For example, in the VCL overlapping reservation failure case, the error message "Reservation failed on vmsk1: process failed because computer is not available" does not provide the correct clue that the reservation failure is caused by an overlapping reservation not by the machine is not available. However, the failure path found by Insight covers the root cause branch where *pgrep* returns a process matching the request ID, indicating the same reservation has been made on the machine. For the Lighttpd failure, the reproduced path shows that the failure is caused by the back-end server lookup operation returning *empty* when the round-robin policy is employed and there is only one back-end server. The buggy code segment does not appear right before the error message.

We also examined how VM cloning helps Insight to find the failure paths. For example, the shadow component of the VCL reservation server inherits the configuration files that specify the supporting VM types (e.g., xCat, KVM), VM image locations, and public keys. Without those configuration parameters, it is extremely difficult to perform any replay. Similarly, the configuration file of Squid defines the permissions

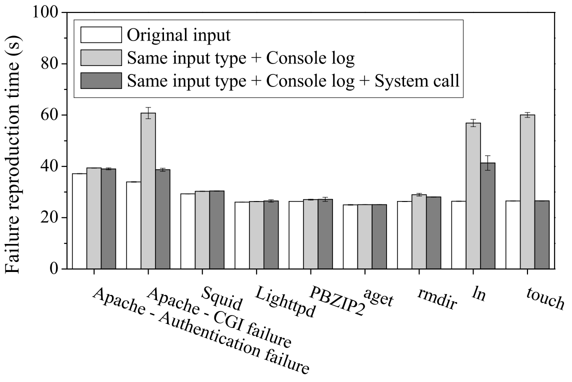

Figure 4: Failure reproduction time results for open source software bugs.

associated with the "ACL" name which are needed by the failure reproduction. In Lighttpd, code modules such as `mod_proxy` and back-end servers are specified in the configuration file. Additionally, VM cloning also ensures the same third-party libraries are installed on the shadow component.

Detailed VCL failure reproduction results. We now present the detailed failure reproduction results for all the VCL failures. Table 2 shows the branch difference for different VCL failures. As mentioned in the accuracy result summary, Insight can find a failure path with little branch difference compared to the original failure path when in-situ failure reproduction is performed.

Figure 2 shows the percentage of branch points that are flipped by our binary execution exploration engine during path finding. We observe that Insight only flips a small number of branches when part of the environment data is not available.

Figure 3 shows the failure reproduction time for the VCL failures. The reproduction time includes the time for Insight to create the shadow component and the time taken to search the failure path. We observe that Insight can reproduce the failure path within a few minutes. It took Insight about 30 seconds to create the shadow component using live VM cloning. We also observe that the environment data has an impact on the failure reproduction time. When complete environment data is available, Insight can quickly reproduce the failure path within tens of seconds. When part of the environment data is missing, the reproduction time is longer, taking up to 250 seconds to complete. As mentioned before, when no environment data is available, Insight cannot find any matched failure paths after searching for several hours.

Detailed open source software failure reproduction results. We now present the results for the open source software bugs. Table 2 shows the normalized branch difference for the open source system failures. Figure 4 shows the failure reproduction time. We observe that with the original inputs, Insight can always reproduce the exact failure paths within tens of seconds including the shadow component creation time.

Given the same types of inputs, Insight can still reproduce the failure paths for Squid, PBZIP2, aget, and all the Coreutils failures within several minutes. However, Insight cannot reproduce the Apache authentication failure, the Apache CGI failure, and Lighttpd failure within a short period of time ($<$ 1 hour) which is a requirement for our in-situ failure reproduction. The reason is that those open source systems produce zero or very few (i.e., 3) console log messages except the error message during their failure executions. With such little guidance, Insight is faced with a large path search scope. Under those circumstances, Insight uses a code selector in a similar way as S^2E [12] to limit the path exploration within a specified target code module. For the Apache authentication failure, the target code module is the authentication module. For the Apache CGI failure, the target code module is the CGI component that handles CGI scripts. For Lighttpd, `mod_proxy` is the target code module. After limiting the path exploration scope, Insight is able to find the failure paths within tens of seconds.

We observe that system call sequences can greatly reduce the branch difference for those failures with few console logs. We also notice that the branch difference in the Apache authentication failure is significantly larger than the other failure cases when the original input is absent. The reason is that the program includes a large loop that includes many branch points but does not generate any console log message or system call. Because of the input difference, Insight executes the loop with a different number of iterations from the original failure execution, which causes the high branch difference.

Generally, we observe a higher branch differences in the open source failures than those in the VCL failures. This is expected as the open source software systems have less environment data to leverage under the stand alone mode and contain fewer console logs than the VCL system. Insight can definitely benefit from a rich set of environment data and a system with a good number of console logs. Based on our observation and feedback from our industry partners, we believe most production systems do contain abundant console logs as they are the sole information source for the software developer to diagnose production failures.

We also wish to compare Insight with existing static analysis and symbolic execution approaches. Unfortunately, none of them can support the Perl program that forms the main part of the VCL production cloud management service. For open source software

System	Production runtime overhead		Logging overhead (1 day)				Shadow creation time	Stop-and-copy time
	With system call tracing	With shadow component	Console log	Input log	Interaction log	System call log		
VCL	N/A	< 0.3%	0.49 ± 0.01 GB	0.13 ± 0.01 GB	0.86 ± 0.01 GB	N/A	26.7 ± 2.3 s	49.6 ± 15.9 ms
Apache	< 1%	< 0.2%	0.3 ± 0.01 MB	19.6 ± 0.1 MB	N/A	11.9 ± 0.01 MB	23± 1.3 s	38.6 ± 6.5 ms

Table 3: Performance and resource overhead of the Insight system. Request rate in VCL: 120 VM reservation requests per minute. Request rate in Apache: 50 HTTP requests per second.

systems, we found that Insight can achieve much faster failure reproduction. For example, static analysis techniques need up to 28 minutes to analyze an Apache failure [37]. Symbolic execution requires up to 6 hours to explore a program with 1.3 KLOC [12]. This is expected as Insight can leverage many environment data and runtime outputs to greatly reduce the path search scope.

Insight system overhead. Table 3 shows the performance and resource overhead of the Insight system for the VCL reservation server and the Apache server. The results for other open source servers are omitted as they are similar to the Apache results. Insight does not require any system instrumentation during the production run except the system call tracing. We observe that the system call tracing imposes < 1% performance impact and <1.5% CPU load to the production server. The performance impact is measured by comparing the per-request processing time when running systems without system call tracing and with system call tracing. We also measure the performance impact for the production operation when the production server runs concurrently with the shadow server. Again, we observe very little performance impact. We also study the logging overhead incurred by Insight. We can see the logging overhead is small compared to the capacity of modern storage systems. Finally, we measured the shadow component creation time and stop-and-copy time for different servers. The results show that we can finish the live VM cloning and shadow server configuration within 30 seconds. During the shadow server creation, we only need to pause the production server for less than 100 milliseconds.

5 Conclusion

We have presented Insight, an in-situ failure path inference system for online services running inside the production computing environment. Insight uses a shadow component to achieve efficient onsite failure inference while imposing minimum interference to the production service. Insight employs a guided binary execution exploration process to achieve accurate failure path inference by exploiting the production-site environment data and two different types of runtime outputs (i.e., console logs, system calls).

Our initial prototype implementation shows that Insight is both feasible and efficient. We tested Insight using real request failures collected on a production cloud computing infrastructure and a set of real software bugs in open source software systems. Our experiments show that Insight can efficiently use the environment data and runtime outputs to find the failure paths with high fidelity (i.e., little difference from the original failure path) within a few minutes. Insight is lightweight and unobtrusive, imposing negligible overhead to the production service.

Acknowledgment

We thank the anonymous reviewers and our shepherd Erik Riedel for their valuable comments. We also thank VCL system administrators Aaron Peeler and Andy Kurth for providing us with the log data and their generous help on validation. We thank Anwesha Das for helping with the experiments. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] The 10 biggest cloud outages of 2012. http://www.crn.com/slide-shows/cloud/240144284/the-10- biggest-cloud-outages-of-2012.htm/.

[2] Amazon EC2 Service Disruption Summary. http://aws.amazon.com/message/65648/.

[3] Apache VCL. https://vcl.ncsu.edu.

[4] Process trace. http://linux.die.net/man/2/ptrace/.

[5] Systemtap. https://sourceware.org/systemtap/.

[6] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[7] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[8] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder-second generation of a java model checker. In *Workshop on Advances in Verification*, 2000.

[9] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *EuroSys*, 2011.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[11] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[13] J. Chow, T. Garnkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[14] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Eurosys*, 2011.

[15] D. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, 2012.

[16] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Linux Symposium*, 2006.

[17] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *NSDI*, 2007.

[18] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX ATC*, 2006.

[19] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI*, 2008.

[20] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM*, 2009.

[21] K. Kc and X. Gu. ELT: efficient log-based troubleshooting system for cloud computing infrastructures. In *SRDS*, 2011.

[22] H. A. Lagar Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[23] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[26] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

[27] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[28] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *FSE*, 2005.

[30] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS*, 2011.

[31] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *PODC*, 2010.

[32] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS*, 2012.

[33] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.

[34] R. Vaarandi. Mining event logs with slct and loghound. In *NOMS*, 2008.

[35] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[36] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[37] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

[38] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Eurosys*, 2010.

[39] J. Zhou, X. Xiao, and C. Zhang. Stride: search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, 2012.