Improving the Efficiency of Fuzz Testing Using Checkpointing

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Improving the Efficiency of Fuzz Testing Using Checkpointing

Master Thesis

E. Zachow

April 1, 2014

Advisors: Prof. Dr. D. Basin, Dr. M. Torabi Dashti, P.Tsankov, M. Guarnieri
Department of Computer Science, ETH Zürich

Abstract

Fuzz testing, applied to protocols, attempts to provoke misbehaviour of the programs running those protocols by tampering with the messages that are being exchanged. In general, there are too many ways how we can manipulate protocol runs, so we cannot try them all. Instead, we just try to run as many test cases as we possibly can. For each test case we waste time by rerunning the protocol from the start up to the state where we can make our modification.

We want to reduce this overhead by using a checkpointing tool to take snapshots of the relevant states so we can simply restore the state of a protocol rather than performing a whole new protocol run. We present SnapIntercept, a tool that uses a novel approach involving snapshots to increase the efficiency of fuzz testing on protocols. We also provide some preliminary empirical results showing the benefits of our approach.

Contents

Introduction

In this thesis we focus on fuzz-testing network protocols. This can be done by randomly changing the protocol's messages and examining the resulting system behaviour. Since protocols are usually stateful, we have to exercise the system ideally in all possible states. For instance, according to the specification of the Internet Key Exchange protocol (IKE) [3], the nodes must encrypt their keep-alive messages with the session key. Therefore, to test whether IKE handles malformed keep-alive messages, the protocol nodes must have completed the key-exchange phase of the protocol.

The effectiveness of fuzz-testing depends on the number of test cases we can run. Due to the random nature of fuzz-testing, the number of tests required to thoroughly test a system is generally large. Empirical research [5] suggests that we need to run half a million test cases *after we have found the last bug* before we can stop fuzz testing.

Even for simple stateful protocols like IKE, executing a single test case can take one second on average [6]. Most of this time is spent on taking the protocol nodes to the desired state. For instance, to test the keep-alive message of IKE, we have to perform the expensive key-exchange phase each time.

Our goal is to improve the effectiveness of fuzz-testing by removing the overhead caused by taking the nodes to a desired state. Our hypothesis is that we can do this by using *checkpointing*, which is a technique for saving and restoring program states. Our approach is as follows. First we run the protocol and take a snapshot of all protocol nodes before a message is delivered to a node. Then, to run a test we restore the desired snapshot, fuzz the pending message, and deliver it. Note that we may run the protocol multiple times to acquire more snapshots.

To illustrate the advantages of our approach, consider testing IKE's keep-alive messages. We take a snapshot of IKE's nodes just before the delivery of a keep-alive message. For each test, we just need to restore the snapshot, modify the keep-alive message, deliver it, and observe whether a failure

has occurred. In this way, we replace the cost of taking the protocol to the desired state with the cost of restoring the snapshot. If the latter is faster than the former, then our approach improves the efficiency of fuzz testing. We remark that for complex security protocols, this is usually the case.

Contributions: We propose a novel approach to fuzz-testing that leverages the capabilities of checkpointing tools. The idea is to intercept all protocol messages on the transport layer and store snapshots of the whole system just before we fuzz a message. In this way we can restore the system to that state again and skip rerunning the whole protocol.

As a proof of concept, we present SnapIntercept, a fuzz-testing tool for network protocols that uses DMTCP [2], a state of the art checkpointing tool. Our tool provides the mechanisms for intercepting and injecting messages as well as taking snapshots. To use our tool, the testers must provide fuzz-operators and the test oracle.

Outline: We provide a short background for fuzz-testing and checkpointing in Chapter 2. The requirements for our new tool SnapIntercept will be shown in Chapter 3. We explain the architecture of of SnapIntercept in Chapter 4 and measure it's performance in comparison with the conventional method of protocol reruns in Chapter 5. We will briefly report about the development process of SnapIntercept in Chapter 6 before we summarize our findings in Chapter 7 and talk about possible future directions and development options for our new tool.

Background

2.1 Fuzz Testing

Fuzz testing is about testing software by introducing changes to the inputs at different states of the problem [4]. These changes are called *fuzz operators* and a proper selection of fuzz operators can increase the likelihood of a successful fuzz testing outcome. The selection of fuzz operators can also be done in various ways ranging from a selection that is uniform at random to more strategic approaches based on previous results or classification of fuzz operators.

We focus on fuzz-testing network protocols. This requires that we can intercept the transmitted messages of a protocol, tamper with the content that is relevant to the protocol we are trying to test and inject the messages back into the system instead of the original message. Depending on the protocol there may be different more or less elaborate ways to tamper with the message. Maybe for some test case we just try to flip individual bits and see if the protocol nodes misbehave or we spent some time to understand the different fields and the range of values they can possibly take and change the message based on those fields, for example we could swap two numbers that are part of some message or randomize them.

2.2 Checkpointing

Application checkpointing describes the idea to freeze a program in some state and store all relevant information about that frozen state on the hard drive, so that the execution can be later resumed at that particular state. Motivation to use checkpointing can vary, for example the user may wish to stop execution of a program and simply resume it later or migrate to a different host system in between. Checkpointing is a very active area of development and in the recent past there have been multiple changes introduced to the Linux Kernel so that snapshot tools are better supported.

Current problems that are also relevant for our tool relate the various types of connections between systems and how to checkpoint these connections in a reasonable way. For example think about a stateful connection between two systems where we try to checkpoint one end of the connection, but not the other. It is also an issue in general to run a monitor like valgrind attached to some program we try to checkpoint. Nevertheless, we want to discuss an application for fuzz testing in this document and implement a new tool on top of an existing snapshot tool.

There are different checkpointing tools that try to implement a way to take snapshots of a program's state with all relevant information such as file descriptors and open connections. We do not want to restrict our approach to one specific tool, but we have to use one for the implementation of our SnapIntercept tool. Because of their different approach and usability, we will note two different checkpointing tools here that are worth looking into. For the SnapIntercept tool, we decided to use DMTCP:

CRIU Checkpoint/ Restore In Userspace [1] is a tool that allows the user to take a snapshot of a running application and later restore it. The developers have recently committed a number of changes to the main Linux Kernel.

DMTCP Distributed MultiThreaded Checkpointing [2] provides a framework with a central coordinator. Programs can be launched within that framework and become attached to the coordinator by communicating over a dedicated DMTCP port. DMTCP can keep track of resources that each program tries to access because it gets attached to a program upon launch and the coordinator provides a way to coordinate the snapshot and restore process. For a snapshot, every program gets stored in a separate file. These checkpoint files can be either manually restored, which is what we do for our SnapIntercept tool, or we can run a script that the coordinator automatically generates to restore the whole system.

Requirements

This chapter defines some general requirements towards a new tool like SnapIntercept.

We can immediately formulate the first two requirements based on the core functionality we desire. We want to be able to intercept messages in such a way that we can apply fuzz operators to them and take snapshots of every involved program at exactly the state after a message has been sent by a protocol node, but before it has been delivered to another protocol node.

Requirement 1 The system has to provide a method for intercepting and injecting messages.

Snapshots are defined as the content of the intercepted message and the state of the involved programs at the exact point in time where the read/write access to that message is gained.

Requirement 2 We must be able to take snapshots of protocol nodes.

Furthermore, we want our system to be applicable to a variety of protocols and snapshot tools rather than just one. The best we can do is to require the existence of an interface that provides read/write access to the intercepted message without forcing restrictions on the developer of the code that fuzzes the message (from here on called the *fuzzer*). For example, he should not be required to use a specific programming language or be restricted to use a stateless fuzzer. It follows that the fuzzer cannot be part of the snapshot, because the developer may decide that he wants to keep track of the test cases he has already tried which is impossible if we roll the fuzzer back to a previous state.

Requirement 3 The system has to be applicable to a wide range of protocols rather than being tailored towards one specific protocol.

Requirement 4 There must be an interface which provides access to the intercepted message for a fuzzer program whose state is not captured by the snapshot tool.

Finally, the amount of work required to implement and use our system should be justifiable in comparison to other methods. We do not want to reinvent solutions to complex issues that have already been successfully solved and do not even occur for other approaches. For example, we do not want to invent a new operating system or transport layer just for this purpose.

Requirement 5 The implementation and usage of the new tool should require an amount of time and work that is justifiable in comparison to other methods such as restarting the protocol.

Tool Architecture

Our system consists of six main components besides the protocol nodes that we run. The setup looks as shown in Figure Figure 4.1. We need at least one component for a minimal fuzz-testing architecture in order to intercept a connection and modify the protocol messages. Since we want to be able to intercept multiple connections at the same time, but do not want to distribute our fuzzer across machines, we need another component that receives all intercepted messages and provides read/write access to the messages on the same machine, which we will call the *tool-machine*.

The user of our tool has provide the fuzz-operators and the test oracle. Since we want to decouple SnapIntercept from protocol specific functions, we declare both the fuzzer and the oracle to be separate components.

The final two components coordinate the snapshot taking process. The tool-machine needs a controller that can notify DMTCP at the right moment to take a snapshot and also contact every other machine in the network to perform local operations like storing snapshot files or restoring a certain snapshot. The controller must be another component because he must control DMTCP and not be affected by it. The last component is the program running on the regular machines that receives the commands from the controller.

There are two modes of operation. During *snapshot*-mode, we perform at least one, but depending on the protocol possibly multiple, *clean* protocol run without message modifications and take snapshots of the different states after messages have been sent, but directly before fuzz operators are applied to them. In *testing*-mode we have already acquired the snapshots we are interested in and possibly performed preprocessing operations on the pending messages that will be fuzzed upon snapshot restoration. We keep running test cases in this mode where one test case is the restoration of some snapshot and following execution of the whole component-system until we decide to end the test case.

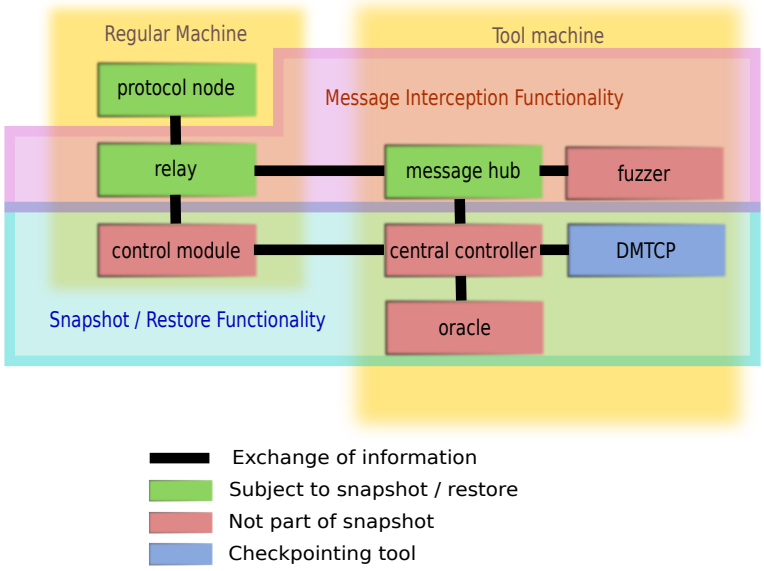

Figure 4.1: Basic setup and communication layout (DMTCP architecture simplified)

4.1 Components

Here is a brief description of all components and their tasks, we will discuss them in more detail below. Aside from the protocol nodes only the relays and control modules do not run on the tool-machine. Again, the basic setup is shown in Figure 4.1.

Relay: intercept messages, send them to the tool-machine, inject response, see Section 4.2

Message Hub: receive messages from relays on tool-machine, provide message access to fuzzer, send messages back to correct relays, see Section 4.3

Fuzzer: apply fuzz-operators to message provided by message hub, see Section 4.3

Oracle: decide whether a test passes or fails, see Section 4.4

Central Controller: coordinate snapshot/restore process by utilizing DMTCP and commanding control modules on other machines, see Section 4.4

Control Module: execute commands received from central controller for local machine, see Section 4.4

As we will see in the next sections, we take snapshots of the application programs as well as the relays and the message hub. The fuzzer, oracle and control programs keep running and are not affected by snapshot or restore processes. The fuzzer and oracle need to be tailored towards the specific

protocol we want to test since there is no general set of fuzz operators that apply to every protocol in existence and there is also no standard way to decide when to end a test case and whether the test has passed or failed that is a reasonable choice for all protocols. This means that the fuzzer and oracle have to be provided by the tester and require some interface in order to interact with the rest of the tool.

4.2 Relay

According to Requirement 3, we should be able to intercept messages for several different protocols. Right away we can see a conflict with Requirement 5. If we want our system to be able to handle TCP and UDP, we need to be able to intercept messages on the network layer. This means for example that we must reimplement large portions of the TCP protocol in order to avoid fragmentation of messages. Because of this reason it makes sense to relax Requirement 3 so that it only applies to protocols that operate above the transport layer and focus on one protocol of the transport layer and the layers below. For our tool SnapIntercept we have decided to focus on TCP. Another good choice based on popularity of the protocol would be UDP, but we have chosen to use DMTCP as a snapshot tool which currently does not support UDP.

TCP addresses programs by their port number and the IP of the machine they run on. We will try to exploit this. For every TCP connection one end has to be the server and listen on an open port for incoming connections and the other end will be the client and connect to that port. We will try to implement a relay that runs in between both ends.

If we run the relay on the same machine as the application listening on the server port, there are two ways to achieve this. One way is to configure the client to connect to the relay port and have the relay connect to it's own machine, localhost, on the real server port. The other way is to configure the server to run on a different port and have the relay listen on the default server port. This way an unmodified client will connect to the relay and the relay will forward the connection in the same way to the server. Every message from one end will be relayed towards the other end and vice versa. Figure 4.2 shows the two possible setups for our relay in comparison to the usual setup without a relay. Since every message passes through the relay, which we have implemented, we can intercept the message and tamper with it in any way we like before we forward it to the intended recipient.

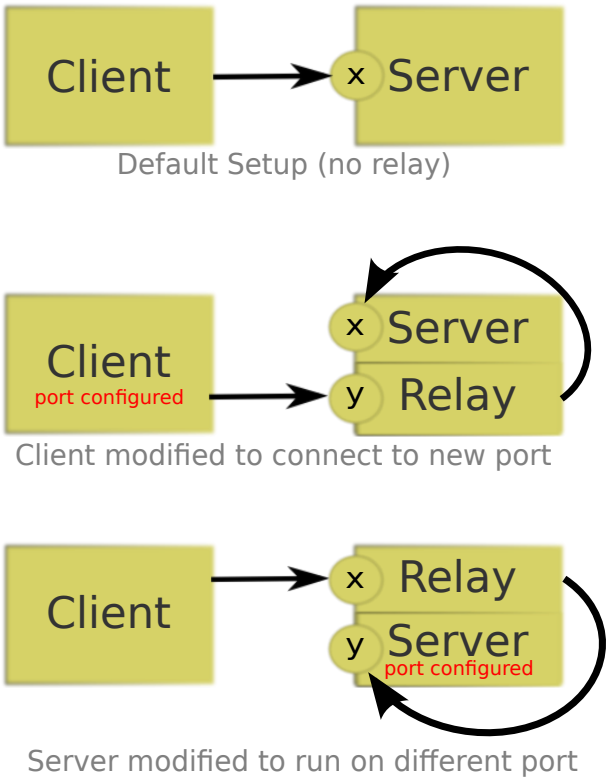

Figure 4.2: Relay setup, port x is the default server port, y is a new unused port

4.3 Message Hub and Fuzzer

Relays intercept protocol messages so that we could technically implement our fuzz operators at the relay. However, this would also require that we must distribute the fuzzing operation to every node where a relay is running. This may not work for stateful fuzzing strategies which may require tedious synchronization among the nodes.

We have required that we do not want to burden the fuzzer implementation with unnecessary constrains, so it makes sense to find a way to perform fuzzing in a centralized way. It follows that every relay has to send the intercepted messages to some dedicated machine where a program, called the message hub, receives all messages, provides access to the fuzzer and sends the modified messages back to the correct relays. You can refer to Figure 4.3 to see the setup of the message hub and understand how every intercepted message gets passed along to the fuzzer and back.

Note that we have separated the message hub from the fuzzer. This is not just a design choice because we want to clearly separate fuzzer and oracle, which are supplied by the tester, from the rest of the architecture. It is nec-

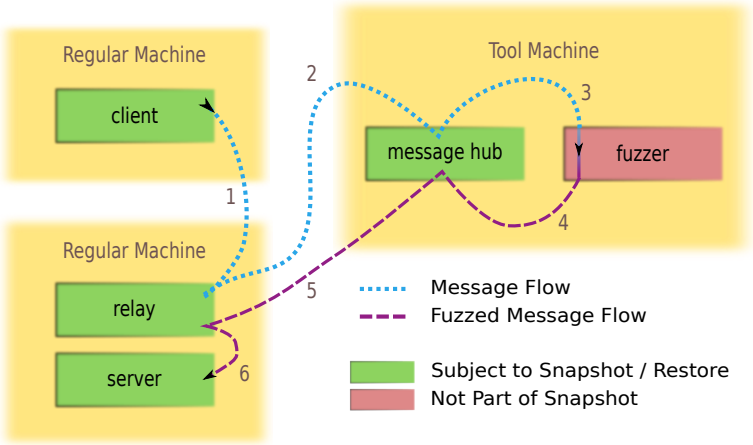

Figure 4.3: Message Hub Setup and Intercepted Message Flow

essary to do so, because we have required the fuzzer to be unconstrained by the snapshot procedure, while the message hub's state must be stored in every snapshot because the intercepted messages travel along open connections and taking snapshots of only one end of an open connection is a serious cause for errors for the current state of snapshot tools. It is even questionable whether a generally applicable solution exists for taking snapshots of only one connection end, so we try to avoid this scenario altogether.

However, we need to extract the message from the message hub to make it accessible for the fuzzer. Instead of using a connection, we will just write the message to a file and later read the fuzzed message from another file. The message file, written by the message hub, is part of the snapshot for that state, so we can perform preprocessing like decryption operations after we have acquired the snapshots that we want. Another benefit of this approach is that we can implement the fuzzer in any programming language that we want.

The fuzzer will only be required to read the message from a file and write the modified message to another file. Whatever happens in-between is subject to the specific implementation. The message hub only provides the intercepted message in a file, but does not communicate with the fuzzer in any other way in order to minimize all unnecessary communication between programs that are part of the snapshot and those who are running without interruption. The fuzzer will be told when to start fuzzing a message by the central controller via regular TCP connection, respond once the fuzzer is done for that message, and wait for the next trigger message.

4.4 Oracle, Central Controller and Control Modules

As we have pointed out in Section 2.2 on the example of CRIU and DMTCP, the checkpointing tools can vary significantly in their approach. In practice we found that the checkpointing tool can impose the most significant restrictions on the usage of our tool, because the whole system fails if the checkpointing tool does not support every critical aspect of the program we want to run with it. At this moment we found for our approach that DMTCP handles multiple restorations with open TCP connections better than CRIU, so we chose this program for our SnapIntercept tool. However, we will try not to exploit program behaviour that is specific to just DMTCP, so the same method can be applied to different checkpointing tools as well.

We have already said that SnapIntercept can be run in two different modes. One mode is used to get the relevant snapshots, the other is used for the fuzz-testing process. For the snapshot-mode we want to take the snapshots just after the message hub has written a message to the message file. Since the central controller coordinates the snapshot taking process, the message hub has to signal the central controller somehow. Again, we want to avoid external connections, so we create a trigger file whose existence signals the central controller that a new message is accessible. Later the central controller will signal the message hub in a similar way to continue execution. For more details about this approach, see the paragraph about development issues in Chapter 6.

The central controller coordinates the process of checkpointing. He will contact the DMTCP-coordinator in order to invoke the DMTCP checkpoint mechanism that creates snapshot files for each program attached to DMTCP on the machine it is running on. Some additional file operations are necessary to store those files in a new folder so they do not get overwritten. The central controller can store the snapshot for the message hub this way because the message hub is running on the tool-machine, but the other programs run on different machines, so he needs to contact all control modules and command them to perform the necessary file operations for their local machine. This means that we have a distributed snapshot and every machine has the relevant information for it's local programs. Figure 4.4 shows the sequence of events if SnapIntercept takes a snapshot.

During snapshot mode we can ignore the oracle. We could already use the fuzzer to apply fuzz operators if we want to reach states that require fuzzing, but most likely we are only interested in clean protocol runs at this point. It is up to the tester whether or not he wants to use the fuzzer in this mode. During testing mode, the oracle can decide to trigger the central controller at any point in time. This signals that the current test case is over and we restore some snapshot to start the next test case. The central controller will restore the message hub in the relevant state, overwrite the

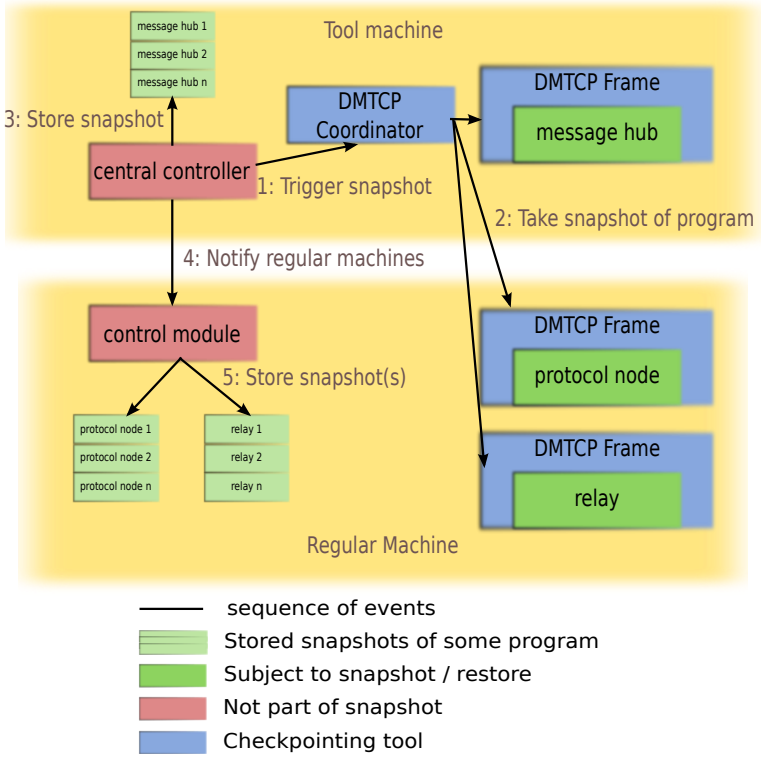

Figure 4.4: Snapshot mode

message file, so the correct message is accessible for the fuzzer and contact all control modules, so they can restore the local application programs and relays and reattach them to DMTCP. The sequence of events for the snapshot restoration is shown in Figure 4.5

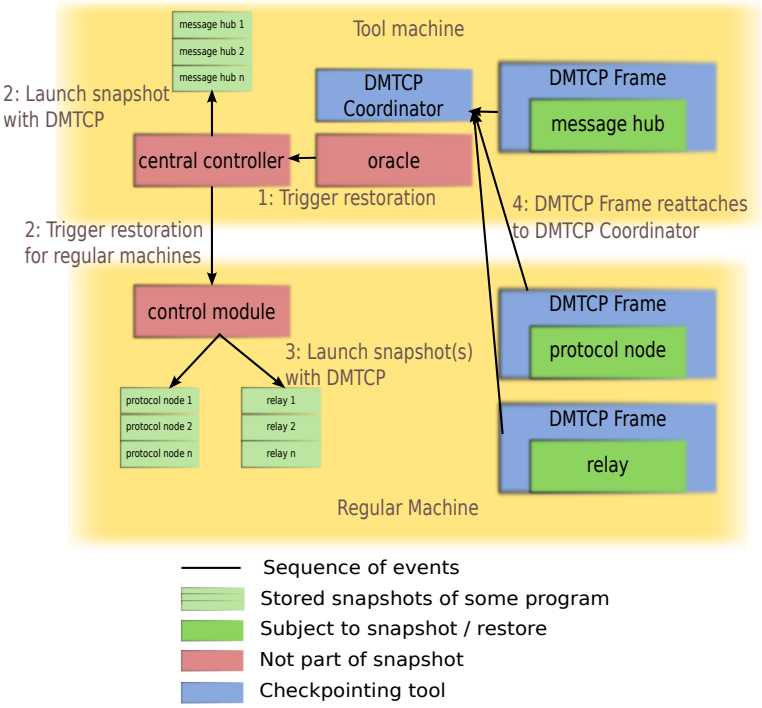

Figure 4.5: Restoration mode, event 3 and 4 are done in parallel

Experiments

We want to compare the performance of our approach that uses checkpointing techniques to the conventional method that uses protocol reruns. For that purpose we have written a simple program capable of sending mail according to the SMTP protocol and use a python class that implements an SMTP server capable of running the SMTP protocol without actually sending the mail away. This provides a minimal environment where we can reasonably perform comparisons between both methods. Our primary interest is the average execution time and standard deviation of each test case, but we will also provide information about the size of the snapshots.

We have mentioned that it is necessary to implement the fuzzer and oracle for the specific protocol at hand. For our tests we are simply interested in measuring the time overhead we introduce for initializing test cases, so we do not fuzz messages at all and simply write the input of the fuzzer directly to the output. The oracle decides when a test case ends. Again, we are only interested in the overhead of initializing a test case, so we immediately end it once we are sure that every program has been restored. For this purpose we distribute the oracle to all nodes, so we can query them whether or not all programs are running.

For SMTP the inputs are e-mails. For the purpose of this experiments, a test case consists of a sequence of e-mails where the last one will be fuzzed. By test case size we mean the number of e-mails in the test case. For our experiments, we used a machine with an Intel i4770 CPU and 32 GB of RAM running Linux. We used virtualbox to setup our tool architecture with two virtual machines running the protocol nodes and one dedicated tool-machine for the central components of the tool.

We measure the time required to run test cases of different sizes using protocol reruns and SnapIntercept. The results are given in Figure 5.1. Our results show that on average it takes 4.33 ms to run a test case of size 1 using protocol reruns. This time increases linearly in the size of test cases. For example it takes 110.34 ms to execute a test case of size 30. Using

Table 5.1: Disk Usage of Snapshot Files

Program	Snapshot Size (bytes)
Message Hub	25,927,680
Relay	30,253,056 - 32,342,016
SMTP Client	27,459,584
SMTP Server	26,554,368 - 26,562,560

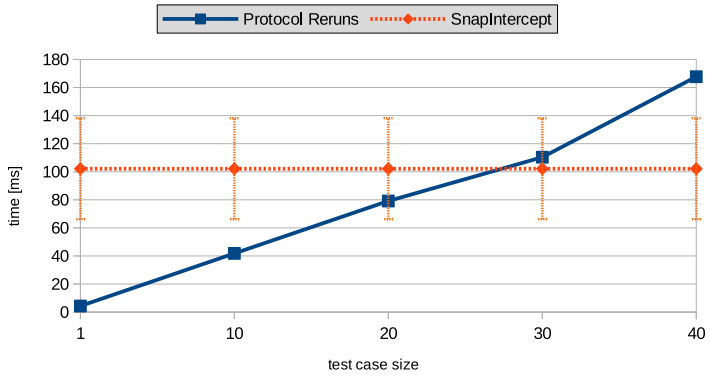

Figure 5.1: Time comparison of SMTP mail transfers (blue) and SnapIntercept performance (orange)

SnapIntercept it takes 102.14 ms to run a test case. This time does not depend on the size of the test case, because we send only the last (fuzzed) e-mail in the test case. In the case of SMTP, SnapIntercept improves the efficiency of fuzz-testing for test cases larger than size 30. We expect that the benefits of SnapIntercept are more significant in more complex protocols with expensive initialization phases. For instance, in order to test IKE keep-alive messages, it takes one second to reach the required state.

For the snapshot size, it is worth pointing out that the measured disk usage across all snapshots seems constant to the byte for the involved programs. The relay has a large drop in size by two megabytes, which occurs only for the last snapshot. This is likely because of the fact that SMTP exchanges 'Bye' messages for the final two protocol messages and one end has already torn the connection down before we intercept the final 'Bye' message. Another observable change in size occurs for the SMTP Server after he has received the actual mail text content. The size was constant before and after that.

Challenges

This part of the thesis is not crucial to the understanding of SnapIntercept or it's architecture and performance, but reports some of the issues and findings that occurred during development.

Virtual Machine Approach: First of all, if someone tries to implement a tool running on a network of virtual machines that somehow utilizes snapshots and is satisfied with the performance of snapshot mechanisms implemented in the virtualization software, then it is possible to use a simpler approach and just take snapshots of the whole virtual machines. However, if we care about speed, then it is likely that we can do better by using a tool that can handle checkpointing of programs, since we do not need to restore a whole operating system. If we do not want to implement our own checkpointing tool, then we are currently left with no other options.

We want to mention that we have also implemented another tool with related functionality to SnapIntercept that takes snapshots of whole virtual machines instead of using some checkpointing tool for programs. The main challenge here is to send a signal from within the virtual environment to the host system, since the host system controls the virtualization tool, virtualbox in this case, and can take or restore snapshots.

To solve this, we can integrate one of the host's network interfaces in a network with all virtual machines and use UDP, which is stateless, to notify the host system that it can take or restore a snapshot from within the virtual environment. The notification has to be stateless, so we do not break any connection during snapshot restoration. The rest of the architecture can be implemented similar to SnapIntercept.

The performance of this approach depends on the configuration of the virtual environment such as the memory size, but even for reasonably small memory sizes of 512 MB we found that the restoration of a virtual machine snapshot takes virtualbox multiple seconds, which is too long in order to be practical for our purpose.

Development issues: During development of SnapIntercept, the first serious issue we had was the communication between the programs that are are subject to checkpointing and those who are not. As mentioned before, we could not use stateless communication like UDP in order to bridge this gap because DMTCP does not support it currently. Simply using the existence of a file to notify the central controller that a new message is ready for fuzzing seemed to be the most robust choice in this case. However, this is only suitable as a communication option for two programs at a time because we have to delete the trigger file immediately in order for this approach to be effective multiple times. We will introduce race conditions if multiple programs change behaviour based on the existence of the same file and delete it concurrently.

Another problem worth mentioning is that killing and restarting processes with open connections on some port should be done with care. If the parent of a child process has not collected the child's return value, the child will linger in the system even if it has been killed and we cannot reuse the same port, so restarting the same program from it's snapshot multiple times will fail and it is not immediately obvious why.

Checkpointing constraints: Both snapshot tools mentioned in this paper (DMTCP [2] and CRIU [1]) are currently under active development. For SnapIntercept, some of the minor tasks might be implemented in a simpler way now because of very recent updates, for example the first version of DMTCP that we used would not allow us to name snapshot files, but assign long, unintuitive names that change for every program. Restarting a program therefore required a workaround to get the file name that is no longer necessary with the current version.

A more serious problem occurred when DMTCP would not let us restart a snapshot with superuser privileges, which was necessary for our experiments. This would have had a serious impact on our ability to run experiments, but fortunately this issue was solved with an update.

In general, our whole approach is mostly constrained by the support of the checkpointing tool. DMTCP already supports a lot of programming and scripting languages, but in practice we could not successfully checkpoint several protocol nodes. We tried approaches for different protocols like IKE and telnet, but ran into issues especially for checkpointing the daemon. We also tried SSH, which is supported by DMTCP, but we felt that this does not demonstrate the approach we want to show appropriately because it is treated as a special case in DMTCP and DMTCP handles checkpointing in that case slightly different. This means that the first thing someone should try after verifying that a protocol runs over TCP is to try and checkpoint just the protocol nodes before doing the whole setup with SnapIntercept. If the

protocol nodes are supported by DMTCP, then we only need to configure the port where the relay listens and the port where the relay forwards the traffic to. The tool's internal ports and file names can remain the same.

Conclusion

We proposed a novel approach to fuzz-testing that uses checkpointing techniques to decrease the time overhead caused by restarting protocols to reach a specific state of interest. Furthermore, we presented SnapIntercept, a new tool that implements this novel approach.

Our tool is capable of intercepting messages sent over TCP connections while avoiding low level networking problems like fragmentation. It can take a snapshot of programs involved in the execution of a protocol in a controlled way, so that an intercepted message has not reached it's intended recipient yet and can immediately be subject to fuzz testing techniques if we choose to restore that particular snapshot later. This eliminates the need for protocol reruns from the start. We also performed a preliminary evaluation of the benefits of our tool using the SMTP protocol.

Future Work: First and foremost, we need to apply SnapIntercept to more protocols with deeper state spaces than SMTP and measure the benefits of our tool over the conventional method of protocol reruns.

We have utilized DMTCP for the snapshot and restore process, so the system is constrained to implementations of the protocol that are supported by DMTCP. We can extend SnapIntercept to support other checkpointing tools such as CRIU.

For the DMTCP implementation, it may be worth investigating the impact of configuration options like state compression or which monitors are supported if any and how they affect performance.

Another obvious way to improve SnapIntercept is to adapt the system to support message interception for UDP as well. In principle we only need to care about both of the relay's connection points to the application ends and leave the more internal parts of the system as they are.

Before we run SnapIntercept on large networks of machines, we can add

more parallelism especially for the central controller connection to the individual nodes which is currently done in sequence.

There may be other applications besides fuzz testing for our tool whenever we want to examine a sequence of messages with the ability to continue execution of a system at the moment that message was sent. After we have obtained our snapshots, we can simply investigate and even change the message that was sent last. This may be used for a network oriented debugging process of programs or reverse engineering of protocols.

Finally, the idea of improving the effectiveness of testing using checkpointing goes beyond fuzz testing and can be applied to any dynamic analysis. We plan to generalize our approach and investigate whether we can reduce the time required to run general test cases.

Bibliography

[1] Checkpoint/Restore In Userspace. http://www.criu.org.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[3] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[4] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[5] Lars Opstad, Jason Shirk, and Dave Weinstein. Fuzzed Enough? When It's OK to Put the Shears Down, 2008. Presented at BlueHat v8, Redmond, WA, USA.

[6] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST'12, pages 1–7. ACM, June 2012.