Technical Report IR-CS-47

Multi-tier intrusion detection by means of replayable virtual machines

Auke Folkerts
auke.folkerts@gmail.com

Georgios Portokalidis
porto@few.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Vrije Universiteit Amsterdam
Department of Computer science
De Boelelaan 1081a
1081 HV Amsterdam

August 2008

Abstract

In order to protect our computer systems from attacks, it is useful to be able to replay an attack once it has been detected, so that we may see how the attack works in greater detail. To this end, we have implemented a deterministic replay feature for Argos, a virtual machine system which is capable of detecting network attacks. Replay is based on capturing all events of the execution of a so-called *capture run.* Those events can later be used to re-create exactly the same execution in a *replay run.* This document describes our experiences designing and implementing this system.

We will show that replaying a captured execution can be performed faster than the original capture run. This can be explained by the fact that our solution is based on capturing the results of systemcalls during the capture run. During playback, we do not have to perform the actual call, since we have stored the results.

Furthermore, we have applied the technique of replaying to create a two-tier architecture which consists of a lightweight capturing machine that captures the execution of a virtual machine to a log, and any number of replay machines read from this log. Those secondary machines can replay the primary machines' execution in real time.

Secondary machines may be equipped with additional instrumentation to examine the execution of the machine in greater detail. We show that the extra time required for this extra instrumentation can be offset by the speed gained from replaying.

With this architecture, it is possible to run multiple secondary machines each with their own additional instrumentation, effectively parallelizing the additional instrumentation performed on the primary virtual machine's execution.

1. Introduction

Modern computer operating systems and applications are complex pieces of software. In this era where the Internet interconnects most computer systems and we have come to rely on the availability of these systems, it is crucial to ensure that the systems function correctly. Unfortunately, malicious individuals and organizations attack both human and technological weaknesses in order to gain control over our computers.

In order to protect ourselves from those malicious individuals and organizations, it is vital to learn as much as possible about the exact functioning of the attacks they use. The ability to replay an attack can help us discover exactly which part of the system, and which vulnerability, was exploited. This information can then be used to either fix the vulnerability in the software, or it can help in generating a signature which allows malicious network traffic to be filtered at the network level.

1.1 Problem case description

In order to protect its' customers networks, SURFnet develops and maintains an intrusion detection system (IDS) named SURFids. This intrusion detection system is based on Argos [16], a full system emulator designed for use in honeypots.

The Argos project uses a virtual machine approach to detect security breaches. By performing dynamic taint analysis on all data that enters the virtual machine, Argos is capable of detecting and avoiding attacks.

Once an attack has been detected, Argos will run a maintainer–supplied program to extract information from the virtual machine, and then shut down the machine. This means that all information not captured by the program will be lost forever. The problem with this approach is that it is very difficult to predict what information is relevant. Furthermore, because only the final state of the machine is available, it is difficult to detect how the attack worked.

The problem can be solved by implementing a replay feature in Argos. If the execution of the virtual machine can be re–done in *exactly* the same way, it is possible to inspect the individual steps of the machine execution in greater detail.

Furthermore, being able to replay the execution is likely to improve Argos' performance. During capture, we can limit the instrumentation to its bare minimum, only detecting whether an attack occurred. Once an attack is detected, we can perform the replay in an enhanced version of the virtual machine monitor, which performs a computationally expensive data–flow analysis.

1.2 Objective

The first goal of the project is to research the requirements for successful re–execution ("replaying") of a virtual machine, as well as to provide an implementation to do so.

This implementation will consist of a modification of the Argos virtual machine. The system will be enhanced to support the option to capture the execution of a virtual machine, as well as the option to later playback an earlier captured execution. Logging the execution of the virtual machine will be done in such a way that replay can be performed multiple times, by multiple machines in parallel.

On top of this basic replay functionality, we explore the possibility of capturing virtual machine execution in a lightweight machine, and replaying the execution in a virtual machine which performs a more thorough analysis of the data flow within the virtual machine.

Finally, performance is an issue that should not be neglected. If the overhead of capture and replay is too high, we might be better of running the enhanced instrumentation directly on the primary virtual machine and do away with the capture/replay mechanism.

1.3 About SURFnet

The research is performed at the SURFnet headquarters in Utrecht, the Netherlands. SURFnet has the mission to facilitate groundbreaking education and research through innovative network services.

The SURFnet network is the national computer network for higher education and research in the Netherlands. This network interconnects the universities, colleges and academic hospitals and provides full internet connectivity to those associated organizations. This makes SURFnet the Internet Service Provider for its customers.

SURFids

One of the services SURFnet offers to its customers is SURFids. SURFids is an open source Distributed Intrusion Detection System based on passive sensors. The goal is to provide an early warning system which lets system administrators correlate known and unknown exploits to attacks directed towards their networks.

SURFids uses Argos as a tool to distinguish malicious network traffic from legitimate traffic. The proposed modification to Argos will enhance the usability of Argos within the SURFids service.

1.4 Structure of the report

Section 2 of this thesis describes **general background information** on research that has been conducted towards replaying network attacks in general, as well as information on replaying virtual machines and generic programs.

In **Sections 3 and 4** we describe the **design** and **implementation**, respectively, of Argos–replay, our enhancement to Argos which facilitates replaying the execution of the virtual machine. We also describe alternate designs we considered during the development phase of Argos–replay.

Section 5 shows how we have **applied** replaying to create a system that is capable of replaying a captured execution on a virtual machine with more extensive instrumentation.

Section 6 shows our **evaluation** of the replaying functionality. It includes measurements of performance, both in execution time and disk usage for logging. Also included is a section which lists limitations and future work. Finally, **Section 7 concludes** and **summarizes** our work.

2. Background and related work

The technique of virtual machine replaying as described in this thesis is used to replay network attacks. The research towards virtual machine replay started out initially as an effort to solve the more generic problem of replaying a network attack.

In this section we describe other techniques which have been explored to replay attacks. We also describe the limitations of those techniques, which have lead us to believe that virtual machine replaying is the most suitable technique to replay network attacks.

Furthermore we will present an overview of other projects which have studied replay of virtual machines, as well as projects which have focussed on replaying generic programs.

Finally, we include a section on Argos' inner workings. This section will provide enough information on how Argos works to understand the changes we made to facilitate playback.

2.1 Replaying network attacks

Let us start describing the simplest approach to replaying a network attack. Since a network attack at the lowest level consists of network packets sent to a computer, arguably the simplest way to replay the attack is to capture the packets using a tool such as tcpdump [1], and to put those packets back on the wire in order to replay the attack. Various tools [2][3][4] implement this technique.

This approach is fairly straightforward. Tcpdump is capable of creating a dump file, where it stores all network data that passes through a host, possibly refined by an expression which limits the packets that are logged to those that match a certain condition. The format of this dump file is standardized format (pcap), meaning that other tools can easily read and process the file.

After the capture is performed, a separate tool is consequently used to replay the contents of the dump file. Typically some processing will have to be done. For example, the dump file generally contains both inbound and outbound traffic. When replaying a network connection, we typically only are interested in replaying one of the sides of the connection, as the machine we are talking to will provide the other half. Also, in some cases some parts of the packets, such as IP or MAC addresses, will have to be rewritten.

Tools that implement this type of replaying are commonly used to test traditional intrusion detection systems, because they are capable of reproducing the captured network traffic at a higher rate than the original capture, thus testing the IDS under heavy load.

An important observation here is that to test traditional intrusion detection systems, it is not necessary to replay the full application dialog of an attack, instead, those systems trigger on specific patterns which may be present in incoming packets. Sending a stream of network packets at varying rates can test the system under various loads.

Although tools that implement network replaying by outputting the captured packets back on the wire may be suited for testing Intrusion Detection systems, they have a few properties that make them unsuited for replaying all but the simplest computer attacks. In exceptional cases, they may be used to replay a computer attack in full. An example of such a simple attack is the Slammer worm which consisted of a single UDP packet. Replaying such an attack is as simple as resending the packet.

In the following paragraphs we will describe the three most crucial limitations of replaying an attack by resending the packets in more detail.

Limitation 1: Combining packets into streams

The first limitation of this method is the inability to relate different network packets to a single *stream*. A stream, or *flow,* is the collection of all packets that together form the application dialog between two endpoints. An example of this is an HTTP transaction, or a file download via FTP.

This limitation is caused by the fact that a dump of packets lists the packets in the order they are encountered by the capturing machine. This means that IP packets might arrive fragmented, TCP segments could be missing, arrived out–of–order, or duplicate, etc.

Any replay system that tries to combine these captured packets to a coherent flow should implement enough of the TCP/IP stack to understand the notion of IP fragments, TCP frame ordering, etcetera. However, even if the system contains a sophisticated network stack, it might differ from the network stack of the operating system that processed the stream. This could lead to the system have a different idea of which packets formed the stream.

Although a case can be made for replay systems which blindly copies the network packets, thus alleviating the need for a sophisticated network stack in such systems, such a system would not be able to detect the flows in the network traffic, nor could it detect which protocols are present in the network packets. This in turn removes the possibility to dynamically adapt certain protocol fields (such as port numbers or checksums) in network packets that are resent. This makes the technique unsuitable for replaying attacks in general.

Limitation 2: Multi–stream attacks

In some cases, an attack consists of multiple seemingly unrelated network flows. The Blaster worm[6] for example consists of four separate network connections between an attacker and a victim. In order to successfully replay such an attack, all separate connections should be replayed. This requires the replay system to identify which streams are related.

A technique that attempts to detect related streams is demonstrated in the Taser project [18] It is based on the observation that in order for multi–session attacks to work, the internal state of the victim machine must be changed in the earlier stages, so that the later stage can make use of the change. If the later stage would be executed without the earlier stage performed, it would fail, or have a different result.

For example, instructions in a first stream could add an account to the password database, so that in a second flow, an attacker can connect to the machine using the newly added account and perform some malicious action. If the first step, adding the account, is not performed, the attacker will not be able to connect to the machine in the second step; the connect would simply fail.

These dependencies can initially be outlined by relating file access. If a process A writes to a file, and process B consecutively reads from that file, it can be said that B depends on A. Once one has established this rough indication of which processes depend on each other, one can verify the degree of dependency, by measuring how much the execution of B changes if A is not executed first.

The drawback of this technique is that it depends on replaying all individual applications independently of each other. This is a very complicated and time consuming process, which depends on the possibility to undo the execution of any application that has executed.

Also, another category of multi–session attacks consists of attacks where a first stream detects stack or heap addresses, which may be used in a buffer overflow attack from a second stream. This approach will not be able to detect such an attack since the first stream does not change the internal state of the machine under attack.

Finally, this technique essentially relates application executions, not networks streams. Network traffic is only considered as a byproduct from the applications that are executed.

Limitation 3: Variable fields

Some network dialogs cannot simply be replayed by retransmitting packets. In a number of cases, protocol fields will have to be dynamically updated. For example, some protocols contain session identifiers, which are selected by one endpoint and should be mimicked in the response by the other endpoint. As another example, during FTP transfers, the server will select the port number on which data transfers will occur. Even more challenging are challenge–response authentication mechanisms which are specifically designed to counter replaying.

Although this limitation is not likely to be a problem if the purpose is to test an intrusion detection system, it makes the method unfit for replaying any application dialog, such as a computer attack.

A solution to this problem is described in the RolePlayer project [8] and in the Scriptgen project [12]. Both projects depend on capturing at least two different examples of the application dialog. From these two copies an algorithm is capable of finding the differences in the network streams. Combined with some protocol knowledge it is possible to identify the different fields in the network stream, such as length fields, application data, and identifiers which should be included in a reply. This solution originates from bio–informatics, where it is used to match DNA–sequences [13].

For replaying network attacks this technique is not viable however, as it depends on having at least two separate copies of the network stream. Although this can easily be set up in a lab environment, it is often impossible to capture two samples of an attack against a machine connected to the internet.

2.2 Replay of virtual machines to replay attacks

As shown above, replaying an attack from the network level poses a number of problems. Most of these problems are caused by the fact that a system that only mimics captured packets (possibly with some rewriting) has no knowledge on the internal states of the communicating endpoints.

By moving the monitored system into a virtual machine, the internal state of all processes that are communicating is encapsulated in a single application (the VMM) running on the host system. By replaying the execution of the virtual machine, we can replay everything that occurred inside the virtual machine, including attacks.

Argos integration
An added benefit of performing the replay of attacks from a virtual machine, is the fact that the Argos virtual machine is capable of detecting intrusions. The combination of Argos' ability to detect network intrusions together with the ability to replay the execution yields a very powerful system that can be used to learn more about attacks.

2.3 the ReVirt project

The ReVirt project [10][11] employs virtual machine re-execution for intrusion analysis. ReVirt aims to provide a secure and complete system logger. In order to achieve this, they move the monitored system into a virtual machine, and log below this virtual machine. This way, they can guarantee integrity, even if an intruder replaces the target (virtual) operating system. The approach of logging from outside the monitored service means that replay of all steps before, during and after any intrusions is possible.

The ReVirt project is based on the UML virtual machine. We essentially employ some of the techniques described in the ReVirt project to achieve replaying in the Argos virtual machine.

2.4 the Aftersight project

Much to our surprise, during the development of our project implementing replay for Argos, researchers from the commercial VMware project published their findings on deterministic replay of the VMware virtual machine [9]. The similarities in our researches are large; both the Aftersight project and our Argos-replay project aim to replay the execution in an enhanced virtual machine monitor which may consist additional instrumentation.

The Aftersight project is based on decoupling program analysis from the primary virtual machine's execution. This is achieved by capturing the virtual machine's execution in a lightweight virtual machine, and replaying this execution on a secondary virtual machine with added instrumentation in real time.

2.5 The Jockey project

Jockey [17] is an execution record/replay tool for debugging Linux programs. It records invocations of system calls and CPU instructions with timing-dependent effects and later replays them deterministically. Jockey is implemented as a shared-object file that runs as a part of the target process. It overwrites the addresses of all overloaded functions, which in effect means that whenever the target application performs a systemcall, jockeys' code is executed. This code performs the actual call and stores the result, or simply returns the captured result, depending on whether the system is capturing or replaying execution.

Unfortunately, Jockey makes use of the fact that Linux-2.4 kernels export a modifiable list of systemcalls. This behavior has been changed in more recent Linux kernels, which makes the tool unusable on modern Linux kernels.

2.6 Qemu overview

Argos is based on the Qemu virtual machine. In order to understand how Argos works it is important to know some of the internals of Qemu [7]. Qemu is a virtual machine monitor that emulates a full PC in software. Instructions that are executed by the virtual machine are translated to native instructions for the host computer, and then executed. This process of translating instructions is known as *dynamic translation*.

Instructions are translated in batches. Those batches are referred to as *basic blocks,* and a block of translated instructions is called a *translated block*. A basic block of instructions consists of the sequence of instructions up to and including the first *branching* instruction, such as a jump or a call. The rationale behind this is that a block will always be fully executed, since the control flow may only be altered by the last instruction.

Furthermore, this means that there are at most two possible blocks to execute after each block: either the branch is taken, in which case the execution continues at the jump target onward, or the branch is not taken, in which case execution continues with the next instruction (which is also the start of a new block). In case of an unconditional jump or a call instruction, it is also known at which address execution will continue.

At the time a block is translated, it is possible that one or both of its jump targets are already known and translated, because it has been executed before. If this is this case, Qemu will *link* the blocks together. This means that Qemu does not have to search and possibly translate the next block. Instead, execution will continue directly at the next block. Qemu keeps a cache of the blocks specifically for this purpose. This technique is called *block chaining,* it is an optimization which is used to speed up Qemu's execution.

The basic Qemu operation consists of a loop, in which the blocks are translated and executed. This loop is interrupted by an alarm signal which triggers every 10ms. The signal handler performs actions such as updating the graphical output and reading from and writing to the virtual IO devices. In hardware terms, this signal handler can be regarded as the hardware clock interrupt.

2.7 Argos overview

Argos enhances the Qemu virtual machine by implementing *dynamic taint analysis* to detect intrusions [16]. The technique of dynamic taint analysis is based on the observation that under normal operation, the only instructions that a machine executes are those that originate locally, ie. from the binaries that are present on the system. If a machine executes instructions that originate from the network (or any other untrusted source) we can state that the machine is being exploited.

Intrusion detecting

Argos keeps track of memory locations that contain data that entered the machine through the network. These memory locations are marked as *tainted*. Whenever tainted memory is used in an operation, the resulting memory locations are marked as tainted as well. This is referred to as *taint propagation*.

Because Argos has full control over all instructions that are executed by the virtual CPU, it can easily detect if tainted memory is used in program execution. This is the case when the instruction pointer points to tainted memory, or when an instruction `jmp`'s to a tainted memory location. Furthermore, when Argos is provided a hint on the operating system that is running inside of it, it is also capable of detecting the use of tainted data in system calls executed by the virtual operating system.

Signature generation

Once Argos has detected an intrusion, it proceeds to collect data from the exploited process. In order to do this, Argos inserts a specific piece of *shellcode* in the running applications address space, which extracts the process ID of the currently running application. (Note that this again requires a hint on the operating system running inside Argos). This step is required because Argos has no notion of which processes are running in the virtual operating system.

Secondly, Argos dumps all tainted memory ranges out to a file. This information can be used for later offline analysis. Also, this memory dump is correlated to the captured network stream in order to find the overlapping bytes. Currently, Argos uses an algorithm which is based on the Longest Common Substring (LCS) algorithm to find which parts of the network trace are also found in memory. The general idea is that the bytes in memory that were also present in the network stream play a crucial role in the attack. The signature is based on those bytes.

Prospector

An alternative to this approach is taken in a variant of Argos known as Prospector [19]. By performing a more thorough analysis of *why* data is marked tainted, it is possible to trace the offending bytes that contribute to the attack back to individual bytes of the network trace. This technique can be used to generate much more accurate signatures. The drawback is that it slows down Argos because of the more involved calculations.

3. Design

3.1 Goals and Requirements

We have chosen to enhance the functionality of the Argos virtual machine. The working title of this project is Argos-replay. It is our intention that the replaying functionality eventually becomes part of the core version of Argos.

While designing Argos-replay, we have set a number of requirements for the design. First, we aim to minimize the number of changes made to Argos. The rationale behind is simple: Argos is based on the Qemu virtual machine. Whenever the Qemu project releases a new version of Qemu, Argos needs to be ported to this new version. The more of Argos' internals we change, the more chance we have of a newer version of Qemu breaking the functionality.

Furthermore, the research is performed as the final masters' project research. Because of this, we have chosen to keep the design as simple as possible, due to time constraints.

Finally, we aim to keep the replay functionality contained in the Argos-replay package. This means we should not depend on changes being made to the underlying operating system libraries or kernel.

3.2 Overview

In essence, the Argos virtual machine is a user-space program which emulates a full computer in software. It does not need any elevated permissions, and runs on top of an existing operating system like any other program. Because of this design, the problem of replaying the virtual machine execution can be reduced to replaying a userspace application.

In order to replay a user space application deterministically, we must ensure that all instructions executed during replay are identical to the instructions performed during capture. Since the instructions stem from the same binary file, this will generally be the case. In other words, a computer program that calculates `1 + 1` and displays the result, will always print out `2`.

Given this observation, replaying the virtual machine can initially be achieved by restarting the program (with the same arguments and the same virtual machine image).

Nondeterministic input

One thing we must take into account is the fact that nondeterministic data can enter the application, and that the application may use this data in its execution. This means that the application may behave differently upon re-execution, because the input differs across multiple executions. Nondeterministic input can originate from two sources.

The most common source is the application itself: the application may request input from an external source, by performing a systemcall. The application has no control over the return value of such a systemcall, but the actions the application performs are affected by it. This means that the application might behave differently if a different value is received upon re-execution. In order to replay faithfully, we must ensure that the application receives the same input during replay as during capture.

Asynchronous events, sent from the underlying operating system are another source of non-deterministic input. An example of this is a signal sent to the process. Since a program typically performs some action upon receipt of such an event we must ensure that the events occurs at exactly the same point in execution during playback, as was seen during capture.

One point of importance here is that this mechanism facilitates true replay *from the application point of view.* Any actions that are performed by the underlying operating system based on the call made from the application, such as sending data over the network, or the altering of a file, may or may not be executed again. The important issue is that, from the application's perspective, any action taken yields exactly the same result during playback as it did during capture.

3.3 Design

We have established that we can essentially replay a virtual machine by restarting the Argos binary, as long as we ensure that (1) all calls made during replay have exactly the same results as during capture, and (2) any asynchronous events within Argos occur at exactly the same point in execution during replay. The design of Argos-replay is centered around solving those two issues.

3.4 Ensuring same-result calls

Operating systems create an abstraction layer above the hardware, by providing an interface to applications above it. This interfaces enables applications for example to read or write data in a uniform fashion, regardless of whether the data is coming from a disk drive or through a network interface. The main benefit of this abstraction is that applications do not need to bother with the details of either the network device or the disk drive.

This abstraction is made up of two layers. At the lowest level, the operating system *kernel* provides access to *systemcalls*. These systemcalls provide low-level access to the systems' resources, generally with none to very limited error checking.

In a layer above that, a *library* that resides in userspace makes use of this low-level interface, and provides a more feature-rich interface consisting of *library* calls to applications running on the system. This library is commonly referred to as the c-library, or libc, on unix-like systems. Typically, applications use library calls only, which my lead to one or more systemcalls. It is important to distinguish *library calls* from *system calls*.

The impact of this design for our replaying mechanism, is that we are faced with the choice at which layer we wish to perform our interposing. We can either inspect all *systemcalls* made on behalf of the application, or we can track the *library functions* which are being called by the process.

Regardless of the choice where we choose to interpose the calls made by the application, the general model is that during a *capture* phase we let all calls happen undisturbed, and we record all results. The results of a function consist of any changes to the system which are visible to the calling application.

The most visible result is of course the functions' return value, but we must also record any changes to the functions' arguments (so–called value–return parameters) and other memory locations that are accessible by the application which have been changed by the function.

During *playback*, we use the stored results to set up a context which is identical to the one seen during capture. Furthermore, we *prevent* the call from being made. The first reason for this, is that we don't *have* to. We already have captured everything that is relevant; any results from performing the actual call would be overwritten by our captured results. This would slow down the system unnecessarily. Furthermore, it is possible that a blocking function simply does not return at all during playback, because the data that unblocks it never becomes available. Consider a trivial application that reads data from the keyboard, and displays the input that is given. During capture, this data is read and recorded for later replay. If the same instruction to read data from the keyboard is issued during replay, the replay system would halt until the user typed his input. Obviously this is unwanted behavior.

3.5 Interposing system calls

The first design we considered was to intercept calls made by the application on the systemcall level, between the c–library and the kernel. Linux offers a mechanism for this by means of `ptrace`. Ptrace allows one process to trace the execution of another process. In this model, we would have an external program that examines the system calls made by the Argos program. This external program would then store enough information for each systemcall to facilitate replaying during a later stage.

This approach functions on a low level. An application performs a system call by storing the system call number in one of the CPU registers, the function arguments in other registers, and then signaling the kernel that a system call is to be performed. The kernel then executes the system call and stores the return value in one of the CPU registers.

If the process is being ptraced, the tracing process is notified twice for each systemcall that is executed: the first notification is sent right before control switches to the kernel to perform the systemcall, and secondly a notification is sent right after control switches back from the kernel to the application, just after the systemcall is performed. Upon receipt of those events, the ptracing process is capable of reading or modifying the value of these registers. It should be noted that changing the value of the registers right before the systemcall is performed can affect the systemcall being executed.

The main benefit of this approach is the completeness of tracing. The tracing process is notified of all system calls that are performed by the traced process and can take appropriate action.

Another benefit is that the interface that is exported by the kernel is well defined and side–effect free, meaning that only the registers, and memory locations pointed at by those registers, are modified by a systemcall. This helps us identifying the pieces of the system that are changed and thus should be recorded for later playback.

However, the approach also has some severe limitations, as described below.

Execution speed

The most problematic issue is caused by the fact that we are interfering with an applications execution from another process' address space. This means that we can only use the ptrace instructions to get information on which system call is being executed, resulting in a slowdown of several orders of magnitude. Since Argos by itself is already CPU intensive the additional slowdown turned out to be unbearable.

The slowdown imposed by ptrace is caused mostly by memory copies between the tracer and the tracee. During capture, we need to copy all CPU registers from the traced process to the tracing process for each system call to retrieve the calls results. Furthermore, in the case that one of the arguments is a pointer to a memory location that is used as a result (for example, the buffer in which `read()` places its data), the contents of this buffer must be copied too. To make matters worse, ptrace only allows the tracing process to read 4 bytes (the size of an individual CPU register on a 32–bit machine) at a time, which results in a lot of overhead when copying all data that a read–systemcall returns.

Avoiding making calls during playback

A second issue arises because we must ensure that not all calls are executed during playback, as described before. When using ptrace, the tracing process is notified when the traced process is about to make a system call. Avoiding the actual systemcall from being made at that point requires that we make a change in the address space of the traced program. We can either adjust the *instruction pointer* to point at the instruction after the systemcall, so that the systemcall is not performed, or we can change the register which contains the number of the systemcall, replacing it with a trivial systemcall such as `getpid()`.

A problem with this technique is that we cannot simply prohibit *every* system call from being made. Some systemcalls, such as `brk()` or `mmap()` alter the address space of the traced process, and the correct functioning of the traced program depends on the execution of this systemcall. Suppose that during recording phase a `brk()` call is issued to extend the memory available to the process. If, during playback, we simply return the value captured during the recording phase without the underlying operating system actually reserving this memory, our program will crash once it tries to access this memory.

This limitation results in a scheme where we would have to let a number of systemcall pass–through as–is, others would have to be interposed. This means that the benefit of completeness of this model diminishes. Considering how interposing at the systemcall level also resulted in a serious performance hit, we have decided to move the interposing of calls up to the library level.

3.6 Interposing library calls

When an application requests any input or makes any request to 'the underlying system' it does not talk to the operating system directly. Instead, all features that are available to applications are provided by a layer of libraries that sit between the application and the operating system. These libraries extend the low–level bare bones functionality of the interface that is provided by the kernel in the form of systemcalls. Typically these libraries provide more extensive argument checking, and clearer error reporting.

When an application is compiled from source, the resulting binary is *linked* with the libraries that are present on the system. This means that for any given function that is used by the application, a reference is created to the library that provides this function. At *runtime* the code in this library is loaded so that it can be executed. This mechanism is called *dynamic linking.*

An alternative to this mechanism is called *static linking*. In this approach, during compilation of the program, the code for the function that is called is compiled into the resulting binary. This approach has the benefit that it does not depend on the library being available during runtime of the application. The drawback is that the resulting binary program will be much larger as it also contains all code of referenced functions. Furthermore, if the library is updated in a later stage, the program will not benefit from this until it is recompiled.

In order to interpose the functions made by Argos, we must ensure that Argos calls our versions of the functions that are being used. Instead of calling the operating system provided `read()` instruction for example, we want to execute our own code. Our own code would call the real `read()` during capture, and store all the results. During playback, the earlier stored results should simply be returned. This mechanism is often referred to as a *wrapper function*.

There are three options to achieve that Argos calls our own code instead of the 'real' functions. The first is to change the library that provides the functions (the c library in our case). This is not a viable solution, as other programs use this library too, and affecting their behavior is unwanted.

The second solution is to create a library containing our versions of the functions that Argos uses, and instructing Argos to use this library instead of the real library at runtime. This can be done through tweaking the dynamic loader with the LD_PRELOAD environment variable. This solution is commonly applied to programs where the source code is not available, since they allow for changed behavior of the program without modification or recompilation of the source.

The third option is to link our library with wrapper functions with Argos at compile time. This approach is similar to using the wrapper library as described above. The difference is that we actually compile the wrapper library into the resulting binary program. This requires access to the source code and build environment of Argos, but since it is an open source program this is not an issue. The benefit of this approach is that we create a standalone binary, with no requirements from the underlying system in terms of modified libraries. This is the approach we have chosen for Argos–replay.

3.7 Solving asynchronous events within Argos

Now that we have a mechanism in place that ensures that all calls made by Argos return the same value during playback as that was captured during the capture phase, we need to make sure that the calls made by Argos are performed in the same order. While in general this will be the case, because the instructions stem from the same application binary, there are some areas that require extra attention.

Argos basic mode of execution consists of translating virtual machine assembly instructions to host machine assembly instructions, and executing those. Since this is the most time–critical part of Argos' execution, this happens inside a highly optimized loop.

Argos uses a signal handler to receive a signal every 10ms. This signal is used to have Argos break out of this inner loop, in effect emulating the timer interrupt which is present in normal hardware. Once Argos has broken out of this loop, it performs IO such reading user input, reading data from disk as requested from the virtual machine, and updating the graphical output as visible to the user.

It is obvious that the execution of Argos depends on at which point during execution these interrupts are generated. In order to ensure deterministic playback we should ensure that this loop is broken at exactly the same point in execution.

Details on how we solved this issue, as well as a few others that affect Argos' determinism are described in Section 4.3.

4. IMPLEMENTATION

4.1 Implementation overview

Argos-replay is based on the Argos virtual machine monitor, as described in the previous section. The changes to Argos are twofold.

First, we provide a library that is linked to Argos which provides wrappers around all the library functions Argos uses. The goal of this wrapper library is to ensure that functions called by Argos return the same value upon re-execution.

Second, Argos-replay changes some Argos internals to ensure that the application behaves deterministically during playback, by making sure that internal time-based events occur at exactly the same point in execution.

This section will describe the implementation of these two aspects of Argos-replay.

4.2 Ensuring same-result calls

As described above, we have implemented a library which provides overloaded versions of the functions that Argos uses. Our wrapper functions' behavior depends on whether we capture or playback a virtual machine.

During capture, the real library call is being made, but we store all results of the call. During playback, instead of making the real call, we return the previously recorded results.

We implement this library by providing an implementation for each function that is called by Argos which is normally linked from the c library. These functions must adhere to the prototype defined by the c library, since we are not changing the way Argos calls the functions.

Implementing the wrapper function

When implementing the wrapper functions, we should keep in mind that from Argos' perspective, it calls the 'real' function. This means that it should behave exactly like the real function it wraps around.

The basic implementation of an overloaded function call looks as follows. Do note that for simplicities sake error checking has been omitted:

```
1    int read(int fd, char *buf, int len)
2    {
3         sc_info *sc = sc_info_next();
4
5    #ifdef CAPTURE
6         int (*real_read)(int, char*, int) = dlsym(RTLD_NEXT, “read”);
7
8         /* Store basic information about call */
9         sc->number = NR_READ;
10        sc->result = real_read(fd, buf, len);
11        sc->error = errno;
12
13        /* Store value-return parameters ('buf' in this case) */
14        sc->len = sc->result;
15        sc->ptr = malloc(sc->len);
16        memcpy(sc->ptr, buf, sc->len);
17   #endif
18
19
20   #ifdef PLAYBACK
21        /* Check if we are executing the correct call */
22        if (sc->number != NR_READ) {
23          error(“Captured %s, but program executed 'read()'”,
24                lookup_call(sc->number)
25        }
26
27        /* Restore function arguments */
28        memcpy(buf, sc->ptr, sc->len);
29
30        /* Restore errno variable */
31        errno = sc->errno;
32   #endif
34
35        /* Return like the normal function */
36        return (sc->result);
37   }
38
```

Listing 1: An example overloaded function

It should be noted that the same function is used for capture and playback. Depending on how the library is compiled, it generates code for either capture or replaying systemcalls. When compiling our version of Argos-replay, we compile the library twice, for capture and replay. This way, we end up with two separate binaries. The main benefit of this is speed – at runtime we will not have to check whether we are capturing or replaying, which saves a check for each function that is called.

The call to `sc_info_next()` provides us with an 'sc_info' structure in which information about the systemcall is stored. During *capture*, this function returns an empty structure in which we can store our data. During *playback*, this function returns the stored record which is next in line.

Capture

As stated before, during capture, we want to execute the actual call, while recording the results. This means we need a way to get to the actual `read` library call. This is done in line 6. The function `dlsym` returns the memory location of the requested symbol. `RTLD_NEXT` is specifically designed to find the next occurrence of the symbol, as the first occurrence will be the overloaded function itself. By explicitly casting the memory location to a function pointer of the correct type, we can call the function as normal.

Once we have a reference to the actual function, we record the name of the system call, the return value, which is typically an integer, and the value of the *errno* variable, which is set on the execution of most library calls. This happens in lines 9–11.

Furthermore, any data returned in memory locations pointed at by function arguments, such as the buffer in which `read()` places its data, are copied as well. This is shown in lines 13–16. In general, all data that must be saved in this manner is stored in `sc->ptr`. The size of this data is recorded in `sc->len`.

Playback

During playback, we first retrieve the next set of results we have stored during the capture phase. We then perform a check to ensure that the systemcall we are executing matches the previously recorded call (lines 21–25). If this is not the case, apparently we are not performing an identical replay of the binary. This check is only present to prove the correct functioning of Argos–replay during development; it can safely be removed for performance reasons. (In fact, in the actual code this checking is performed by a macro. Redefining this macro to do nothing removes the check altogether).

After we have ensured that the call we are executing is indeed the call we expect, we proceed to restore the value of the arguments to the state they had during the capture state (lines 27–31). To achieve this, we simply copy back the data from the opaque `sc->ptr` memory block. Since we know how we copied the data into this buffer, we can use the same procedure in reverse to restore the values. Also, we restore the value of *errno* to the value it had after the call was captured.

Finally, we can proceed to return the captured return value. (Lines 35–36).

Because we have restored the value of the memory locations pointed at by the function call arguments, the value of errno, and the return value, the function has exactly the same results during playback as it had during capture.

4.3 Changing Argos' internals to ensure determinism

As described before, apart from interposing calls made by the application, we need to ensure that any events that occur within the program that may affect its internal state occur at precisely the same point in execution during playback. As stated in the Design of Argos–replay, we have identified a few points in Argos' execution where we have to perform some work in order to guarantee this.

Signals

As described in Section 2.6, Argos' inner main loop essentially is an infinite loop of fetching, translating and executing the blocks of assembly instructions. As detailed before, a signal is sent every 10ms to break out of this loop and perform IO. In order to ensure identical playback, we need to ensure that this signal is delivered at exactly the same point in execution during playback.

A technique that can be used for this, is to record the point in execution where the signal was delivered during capture, and recreating the signal at exactly that point during playback. Obviously the normal generation of signals should be disabled during playback in this case.

We can pinpoint the exact point in execution of a machine by looking at the instruction pointer. Because instructions may loop and thus end up at the same address multiple times, we also need to include the number of branches executed. Finally, because the x86 architecture allows interrupts to be delivered in so–called *repeat operations* we also need to capture the value of ECX. Repeat operations are instruction of the form `rep <instruction>`, where `<instruction>` is executed as often as the value of ECX. The ECX register is decremented each time the instruction is executed once, thus by tracking the value of ECX, we know how many iterations were left in such a case. This technique is described by and used in the ReVirt project [10][11].

Examining the instruction pointer and the ECX register is straightforward, since we can simply read out those registers when a signal occurs. Tracking the number of executed branches is more complicated. The hardware provides us with a register that stores the number of executed branches since the last interrupt. We can use this register, but we would have to keep track of which branches are executed on behalf of Argos, and which are made on behalf of the virtual machine. An alternative solution would be to adjust Argos' internal translation engine, to count the number of branches executed.

Furthermore, in order to re–insert the signal at the exact same time during playback, we would have to add instrumentation which checks the branch–counter, instruction pointer and ECX register to interrupt program flow at the required point. This will slow down performance.

Disable signals

An alternative solution we explored is to disable the signals that are being sent altogether, and instead interrupting the inner loop manually after a specified number of blocks has been executed. We have found that this solution is much easier to implement, although we have to disable *block chaining* to make this work.

Recall that each translated block ends with a branching instruction. This means that after any block, only two different blocks can be executed. If those blocks have been executed before, they are already present in the cache Qemu keeps. In this case, the blocks are linked together, so that Qemu is capable of directly continuing executing the next block of instructions without having to invoke the handler that finds the next block. It turns out that the most common case is that Qemu executes large chains of linked blocks.

It is also possible that one or more blocks blocks are linked in a *cyclic* way. This can for example happen when the virtual machine enters an idle loop, expecting to be woken up by the timer interrupt. An other example is a machine that performs an idle loop until some data becomes available. If a virtual machine is executing cyclic blocks, it is essentially stuck in an infinite loop.

Under normal execution this is not a problem, since every 10ms the signal handler interrupts the execution in order to perform the necessary IO, and the data being available will make sure that a different branch is taken so that the loop is ended. However, with the signal handler disabled, the IO will never occur, and the virtual machine will freeze. We have indeed observed this behavior in typical virtual machine execution.

The solution we have found for this is to disable the block chaining. This way, control will be released to Qemu's algorithm to find the next block, where we can hook in our code to perform the necessary IO after a specified number of executed blocks.

It should be noted that by hooking into the way Qemu executes chained blocks, we may be able to detect cyclic blocks, or we could be able to break a chain after a specified number of blocks. This would remove the need to disable block chaining. However, this approach hooks deep into the dynamic translation of Qemu, and has not been explored due to time constraints.

Asynchronous IO

Argos uses asynchronous IO operations to speed up execution. Essentially, whenever data is required from the virtual machine hard drive, Argos issues an asynchronous read call, which returns control immediately to Argos, and creates a new thread which reads the requested data in the background. Obviously, having multiple threads issuing various library calls independently of each other wreaks havoc on the order in which calls are being registered by our capturing library.

We have found that replacing the asynchronous IO–functions by synchronous versions which simply block until the data is read poses only a minor slowdown, while removing the source of non–determinism that it caused.

Read Time Stamp Counter (RDTSC)

We have found one occasion where nondeterministic data enters the Argos virtual machine without passing through the layers provided by the user space libraries or the operating systems system calls, namely by invoking the `rdtsc` assembly instruction directly. This assembly instruction provides the caller with information on the amount of clock ticks that have occurred since the CPU has reset. The `rdtsc` instruction is commonly used as a high resolution and low cost way of getting CPU timing information

Argos uses the TSC value from the physical CPU whenever the virtual CPU issues an `rdtsc` instruction. Therefore, in order to ensure that the same values enter the Argos virtual machine during playback, we must capture the result of this instruction during the capture phase, and use these captured values during playback.

We have implemented this by creating a wrapper around the function Qemu calls to obtain this value. This wrapper behaves similar to the other wrapper functions – during capture the regular call is performed, and the result is stored. During playback, we simply return the stored value.

5. Application

Although replaying the exact execution of an Argos or Qemu virtual machines is useful on itself, we have applied the technique to create an architecture where the replaying machine is capable to perform additional checks while replaying a captured virtual machine execution.

5.1 Adding instrumentation during replay

The execution trace we create for a running virtual machine is based on the library calls made by the virtual machine. This leads to the observation that as long as the system call profile remains identical, the replaying virtual machine is able to perform additional checks that were not present in the capturing virtual machine.

We have tested this hypothesis by replaying a virtual machine's execution captured in Qemu, in the Argos virtual machine. As described before, Argos performs additional checks during execution to ensure that only instructions originating from trusted sources are executed. In program terms, this boils down to a lot checks which compare values of memory locations, but the system call profile is still identical. We have found that the execution trace created by Qemu can be used in the Argos virtual machine with only minor modifications.

Recall that the log created by the capturing virtual machines contains the results of all the system calls that are made by the virtual machine during its execution. During playback, the systemcalls made by the replaying virtual machine will be the same that were made during the capture run, so the results can be directly returned.

During playback, we check for each systemcall that is executed whether it is the systemcall we expect (ie. the one that was made during capture). This check exists as safety net, so that we are able to tell when the execution of the replaying machine diverts from the capture run.

Obviously, when the addition instrumentation added to the replaying machine performs systemcalls, those systemcalls will not be present in the execution trace. Hence, we need to ensure that we can call these functions directly, bypassing the wrappers we have implemented around the functions. We achieve this by using the same technique our wrapper functions use, by obtaining a reference to the actual library call from the underlying operating system and calling this function directly.

The only changes that are needed in Argos to be able to replay an execution trace captured by Qemu are a call to `srand()` that Argos makes during initialization, to set up a random number generator, as well as all systemcalls that Argos makes once it has detected an exploit. Those systemcalls include print–statements to inform the user of the exploit, and write–calls to dump the memory area's of interest to a file. The benefit we have is that all those actions are performed from one location, namely the function that Argos calls once it has detected an attack.

In general, any version of Qemu or Argos can be used to replay the captured execution of a virtual machine, as long as all systemcalls that were not made during the capture run are made directly, bypassing the wrapper function.

5.2 Setting up a two-tier architecture

Since we have the option to replay the execution in a modified version of Qemu or Argos, it is a small step to extend this functionality to create a tiered architecture where one machine captures the virtual machine's execution, and a secondary machine reads the log and replays the machine at 'real time'. We use the term 'real time' loosely here, as there will be a small delay because the secondary machine has to wait until the primary machine has written out the log, which occurs every few seconds.

The secondary machines can contain any additional checks imaginable. For example Argos could be used as a secondary machine to check for exploits against the primary machine. As another example, a secondary machine could implement a virus scanner for each file that is opened within the virtual machine.

At this point there is no communication from secondary virtual machine back to the primary, so any anomalies detected by the additional instrumentation of the secondary machine cannot be communicated back to the primary machine. Any information obtained by the additional instrumentation in the secondary virtual machine(s) is only usable to inform the user. We envision that it should be possible to communicate findings by secondary machines back to the primary

This architecture is inspired by the work of the Aftersight project [9]. It was not originally one of the goals of our replaying project. However, since the two projects share a fair amount of similarities, we were interested in seeing to what extend our project was capable of using the same architecture as the Aftersight project. We were pleased to find that our approach is able to separate the *analysis* from the virtual machine *execution*, much like the Aftersight project does.

5.3 Countering the slowdown imposed by more instrumentation

It is obvious that the added instrumentation during replay makes the virtual machine slower. Depending on the amount of added checks, this slowdown might range from negligible to significant. Although a slowdown of the replaying virtual machine is not an issue if we use replay at a later stage for offline analysis, it might lead to problems if we use the two-tiered architecture where the replaying machine replays the execution in real-time.

However, it turns out that replaying a virtual machine's captured execution can be done faster than the original capture run. Although both the capturing and the replaying virtual machines dynamically translate and execute instructions, as described in Section 2.6, only the capturing virtual machine has to perform systemcalls such as `select()` and `read()`. During playback, there is no need to wait until such blocking calls return, as we already know what the return value will be. Having these values available directly means that the replaying machine will be able to complete its execution faster.

It should be noted that even though the virtual machine's execution is performed faster, it is still an identical replay of the captured execution. In this regard one could compare the replay to the playback of a movie at slightly increased speed. Its still the exact same movie, although its finished earlier.

We have observed that a replaying machine with extra checks, such as Argos, is capable of replaying a capturing machine in 'real time'. The term 'real time' is used loosely here, as we have observed delays up to a minute in the replaying virtual machine, under CPU intensive operation. Section 6.2 shows more measurements on the replaying of virtual machines.

6. EVALUATION

We evaluate the effectiveness of our replaying solution across various metrics. An important aspect of our replaying solution is speed. Although it can be expected that logging all nondeterministic events in the virtual machine imposes some overhead, it is important that the slowdown is kept within an acceptable limit. We evaluate speed by measuring the overhead imposed by the capturing mechanism, as well as the pace at which the system is able to replay the virtual machine's execution.

Furthermore, another metric of performance of our system by the amount of disk space required to store all nondeterministic events

6.1 Capture overhead

In order to measure the slowdown imposed by our capture mechanism, we have set up a Linux virtual machine running a vanilla version of Ubuntu 8.04. We run this virtual machine image in various versions or Qemu and Argos, and perform various tests. The speed at which these tests complete is a metric for how fast the virtual machine software is.

The versions of Qemu and Argos we have compared are

1. Vanilla Qemu. We have tested Qemu without using the kernel accelerator module. This version is relevant because capturing the execution is not possible while the kernel module is used. This is because with the kernel module, Qemu does not have control over non-deterministic input that enters the virtual machine through for example direct memory access (DMA). Furthermore, Argos does also not use the kernel module. (label: qemu-vanilla)

2. Our version of Qemu, capturing its execution. (label: qemu-capture).

3. We have also created a version of Qemu which only disables the block chaining. This version is used primarily to get an idea of the performance hit imposed by disabling the block chaining in Qemu. (label: qemu-nochain)

4. One of the Qemu developers has implemented an instruction counter in one of the development versions of Qemu[1]. This instruction counter can be used to deliver interrupts at specified intervals. Although the feature is not currently available in released versions yet, is is available in development snapshots of Qemu. (label: qemu-icount).

1 The instruction counting version of Qemu has been developed in parallel with our replaying mechanism. If it had been available sooner, we would have considered it as the basis for deterministic replay. However, it was made available only during the last few weeks or our project. Because of this, we have not been able to incorporate its techniques in our project. We include measurements for this version for completeness only.

5. Using the instruction counting version of Qemu instead of our own modifications to ensure interrupts being delivered at known, specific intervals, we have implemented deterministic replay using the instruction counting version of Qemu.

   Essentially, we have taken the instruction counting version of Qemu and added our wrapper library that ensures that system calls return the same values upon playback. In effect, the instruction counting replace our changes to Qemu internals to ensure that the system calls are made in the same order (see Section 3.7)
   (label: qemu–icount–capture)

6. Argos, version 0.4.2. (label: argos–vanilla)

7. Our version of Argos, based on version 0.4.2, capturing its execution. (label: argos–capture)

CPU load

We have performed two separate tests. First we test the virtual machine under CPU–intensive load. To this end, we use the open source NBench benchmarking program. This program performs several algorithmic calculations over a set time. It generates an index score based on how many iterations of the algorithm were able to complete during the time the test ran. The results are shown below

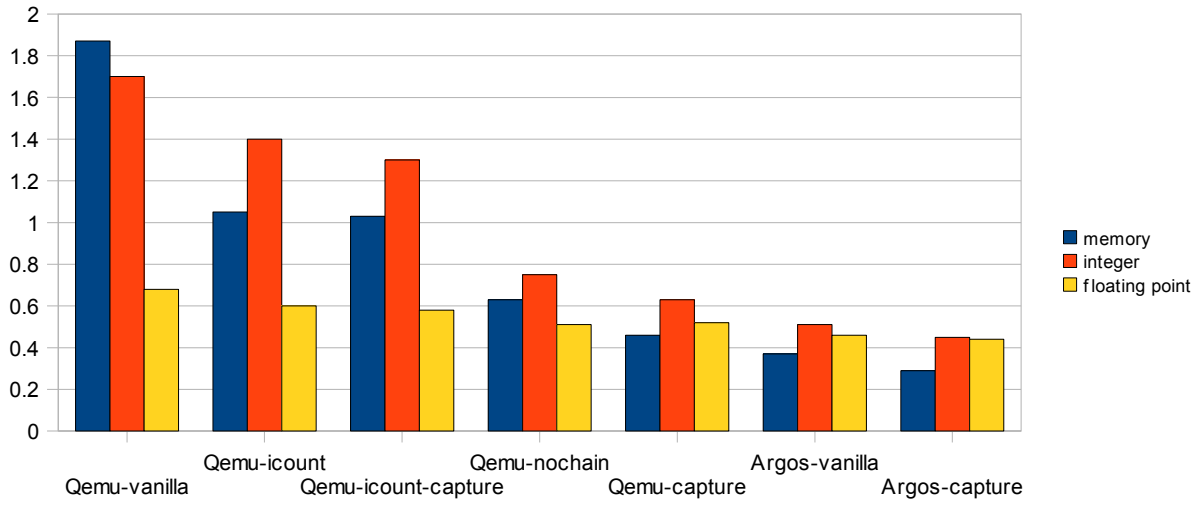

Illustration 1: NBench results for various versions of Qemu and Argos

Note that we have not included Qemu using the accelerator module in this graph. This is done for scaling reasons. Qemu using the kqemu module achieved an index of 14.1, roughly eight times the score of Qemu–vanilla. This shows the power of the kqemu module. Unfortunately, this module makes our implementation of deterministic replay impossible, as it allows Qemu to directly access the hardware without intervention. Because of this, we are unable to intercept the non–deterministic data that enters Qemu.

We can draw a number of conclusions from this graph. First, we observe that the instruction counting versions of Qemu (qemu–icount and qemu–icount–capture) perform nearly as fast as vanilla Qemu. Secondly, we notice that the instruction counting version of Qemu performs better than our own version which disables block chaining. This is logical, as block chaining is a serious performance optimization in Qemu.

Furthermore, we observe that capturing Qemu's execution, both with the instruction counting version as well was with our own capturing mechanism, is faster than performing Argos natively. This means we can use the architecture as described in section 5.2 to speed up Argos' execution.

Network load

The second test we have performed simulates real–world usage of the virtual machine as a webserver. We have set up our virtual machine to function as a webserver using Apache 2.2.8, serving static content. We have measured the time required to transfer 400 files of 200Kb each. The results are shown below

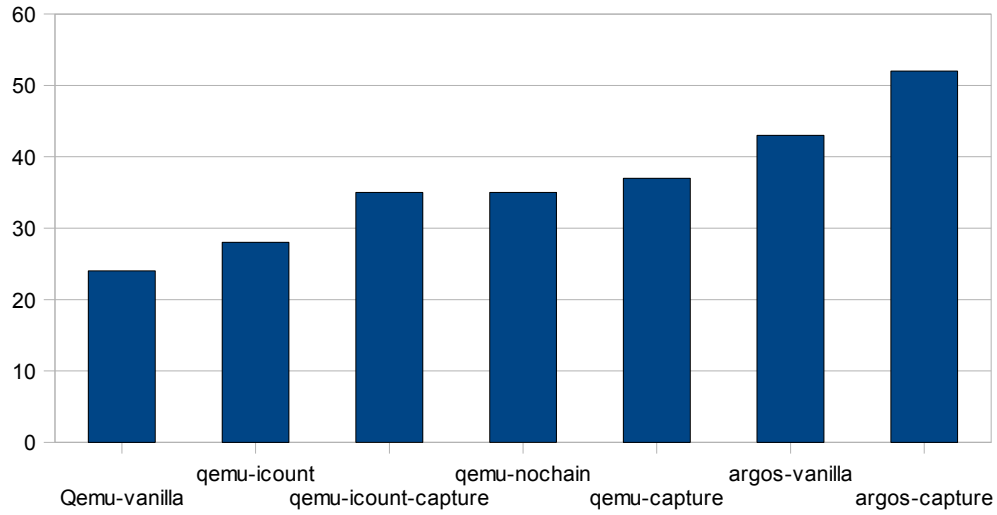

Illustration 2: Time in seconds needed to transfer 400 files of 200Kb from various Qemu and Argos versions. (lower is better)

This graph shows that the various capturing versions of Qemu all perform faster than Argos natively. Once again, we observe that the instruction counting version of Qemu is slightly faster than our own version, although the difference is barely noticable. This is not surprising, as the block chaining is an optimization that affects CPU execution, and this test benchmarks network IO instead of CPU speed.

6.2 Replay speed

While replaying a captured virtual machine's execution, the virtual machine executes the same instructions as during the capture phase. However, the systemcalls performed by the virtual machine, to read data from the network or from the virtual machine image file, do not need to be performed again, as the results are already known and stored in the logfile. This can lead to a speedup.

It turns out that replaying the long term execution of the machine is indeed faster during playback. The exact speedup is dependent on the amount of blocking systemcalls made during the capture phase. If the capturing machine spends a lot of time waiting for network input or user input, replaying the execution will benefit from the results already being available.

As described in Section 5, we aim to facilitate playback in an enhanced virtual machine which adds extra instrumentation to analyze the virtual machine's execution in greater detail. It should be noted, that in such a scenario, the secondary virtual machine which replays the execution executes the full virtual machine environment, just like the capturing virtual machine, in addition to the extra instrumentation.

We have measured to what extend the slowdown imposed by this extra instrumentation may be offset by the speed increase gained from not having to make the actual systemcalls during replay. In order to do this, we have set up an environment where we capture a virtual machine's execution in Qemu, and replay it in the Argos virtual machine. For completeness, we also replay the execution in an unmodified Qemu image.

We measure the number of instructions executed by the virtual machines execute per second. Note that since replay is deterministic, the number of executed instructions can be used to pinpoint where in the execution a virtual machine is, and the number of seconds needed to reach that point is thus a metric for the speed of the virtual machine.

We have tested replay speed in two scenarios, similar to the scenarios used for testing capture overhead. First. we have run a CPU–intensive test, again using the NBench benchmarking tool. Second. we have simulated real–world server usage by setting the virtual machine up as a webserver. We have plotted the number of instructions executed The results are shown below:

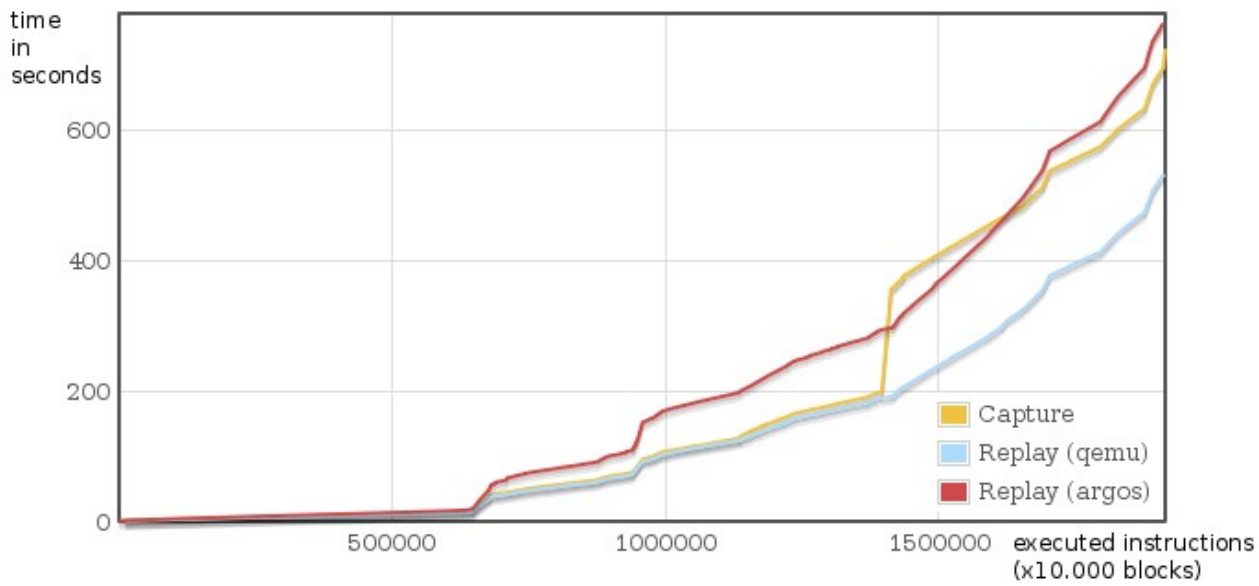

Illustration 3: replay speed for CPU intensive execution

Replay speed for CPU intensive execution

This graph shows time taken per number of executed instructions. A very steep curve means that the machine takes much time for few instructions, which means that the machine is relatively idle. A lower line is better as it indicates that less time is needed for the same number of instructions.

Any line segment above the yellow line indicates the replaying virtual machine is slower than the capturing machine during that time. Any segment below the yellow line means the replaying virtual machine is faster than during the capture run.

The graph shows a virtual machine is booting for the first 200 seconds (~90.000 instructions), is then idle until a little under 400 seconds (~14.000 instructions), and executes the NBench benchmark.

This graph shows that replaying in the Qemu virtual machine is faster than the capture run. Furthermore, this graph shows that when Argos is used as the replaying machine, it will lag behind when the machine is under heavy load, such as while booting or while performing the NBench test. However, we see that it will quickly catch up when the machine is idle.

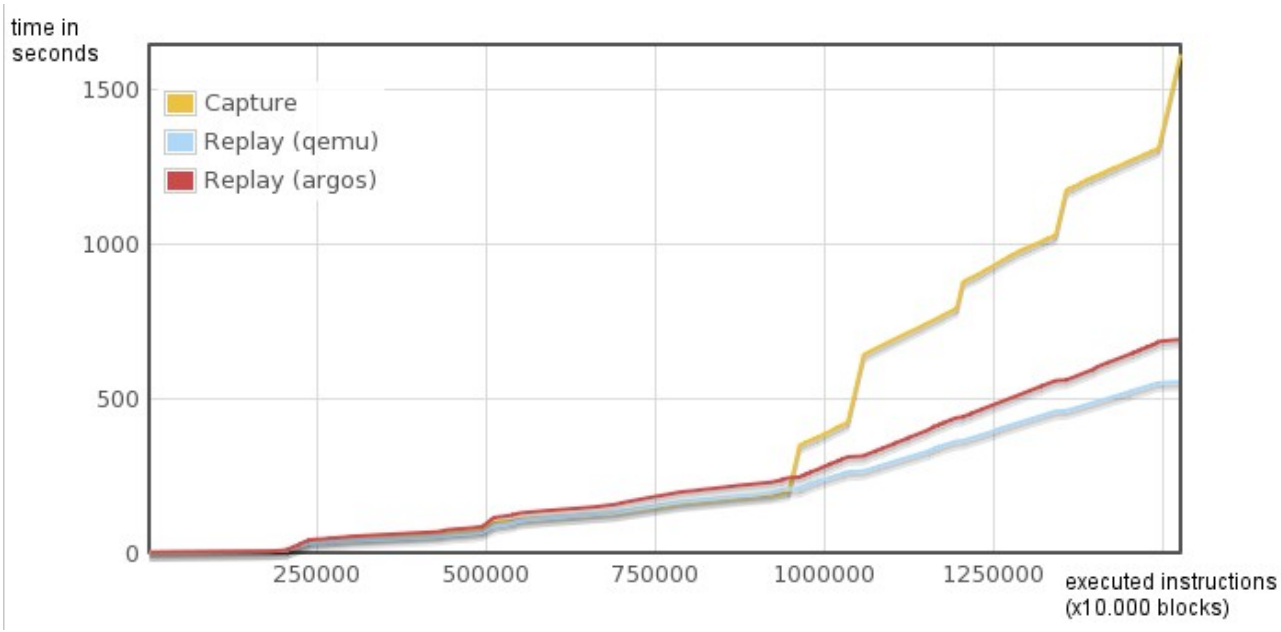

Illustration 4: Replay speed for IO-based operation

Replay speed for network based operations

This graph shows time taken per number of executed instructions. A very steep curve means that the machine takes much time for few instructions, which means that the machine is relatively idle. A lower line is better as it indicates that less time is needed for the same number of instructions.

Any line segment above the yellow line indicates the replaying virtual machine is slower than the capturing machine during that time. Any segment below the yellow line means the replaying virtual machine is faster than during the capture run.

This graph shows the execution of a virtual machine in the role of a webserver. We can see that the virtual machine is idle in between bursts. We see 4 bursts, one short around 100.000 instructions, and three longer ones around 110.000, 125.000 and 135.000. We also observe that both Qemu and Argos as the replaying virtual machine is able to replay the execution much faster.

Those results indicate that using a virtual machine with additional instrumentation, such as Argos, can be used to analyse the execution of a virtual machine executing in real time.

6.3 Log space requirements

Apart from speed, another metric of the performance of our system is the amount of required disk space to record the execution of the running virtual machine. We record the results of all system calls made by the capturing virtual machine, so the size of our log file scales with the amount of calls made by Qemu or Argos.

Our observation is that both Qemy and Argos generate 64Kb of logging information per second when idle. This amounts to 225 Mb/hour of execution. This space is needed for the numerous `select()` and `read()` calls Argos makes. Future research should reveil to which extend this size can be reduced.

If the virtual machine performs much network IO, the log file size will contain all data that is read. This amount will be added to this baseline number of 225 Mb/hour of execution.

6.4 Limitations

During the design and development of Argos–replay, we have focussed first and foremost on developing a system that would be capable of deterministically replaying the machine. Simplicity and "getting it to work" have been prioritized over performance and efficiency. This choice has mostly been made due to time constraints. In this section, we describe areas of our project where we feel improvements are possible.

Missing feature: Using snapshots to seek through executions

The Argos–replay mechanism is capable of capturing the long–term execution of a virtual machine. When replaying this execution, we have to go through all steps of the execution. This means that in the case of a machine that has been running for a few days, the replaying will take a few days to run as well.

Although this limitation is not severe in simulated environments where capture runs are limited to few minutes each, it may render the system unusable to replay the steps that lead to the exploitation of a machine that has been running for days.

This problem can be solved by periodically taking a snapshot, or checkpoint, of the virtual machine. A snapshot, as described in libckpt [15] essentially is a dump of the internal state of the machine which can be restored to resume execution from that point onward.

Using snapshots, one can effectively seeking through the virtual machine execution without having to execute all steps in between.

Periodically taking a checkpoint of the virtual machine has the added benefit that it can contain the size of the logfile – if one is merely interested in only replaying the last steps of a virtual machine, it is possible to cleanse the logfile of all events prior to the most recent snapshot. By doing this, replaying could start at the latest snapshot and replay from there.

Qemu has a snapshotting feature built–in, which should be usable to achieve this bit of functionality. However, its behavior changed in between release 0.8 and 0.9, and at the moment, for the current version of Qemu, it is nonfunctional. This issue also affects the current version of Argos, which is based on this Qemu–release. We have decided to postpone implementing snapshotting until Qemu solves the issues currently present.

Performance optimizations

As shown before, the capture mechanism slows down Argos considerably. Although the benefit our our architecture offers is that multiple instrumentations can be run in parallel, without perturbing the execution of the virtual machine, we still aim to increase the performance of the capturing mechanism.

Block chaining

The biggest improvement in performance can be gained by re–enabling block chaining, as shown in section 6.1. Under normal operation, the signal handler is invoked every 10ms, interrupting the control flow. The signal handler performs actions such as reading user input and updating the screen in–place. Furthermore, a method invoked in the signal handler breaks the chain of currently executing translated blocks, and sets a flag to indicate that the signal handler was triggered. The result of this is that once the signal handler is finished, the currently executing chain will finish after executing the current block. Because the flag is set, Qemu knows to dispatch control to the routine that reads IO from the virtual IO devices instead of translating and executing the next block. For more details, see sections 2.6 and 4.3.

The important observation is that the signal handler interrupts the chain of blocks in two ways. First, the signal handler itself executes at any point in execution, which can be in the middle of an executing block. Second, the signal handler breaks the chain in order to divert control to the routines that perform the rest of the actions that need to be performed periodically. This interruption occurs at block granularity, meaning that the current block is always finished entirely.

Further research should indicate to which extend it is possible to adjust the block chaining mechanism, so that a chain can be broken after a known number of chains, without an asynchronous event triggering this.

Reducing disk usage

As described above, logging the virtual machine's execution requires at least 64Kb per second of execution. Although this is not a problem for testing purposes, it might make capturing longterm execution problematic. There are a number of options to reduce the disk space required.

The first option is to compress the data as it is generated. To this end, we could employ the gzip library to compress data as it is written out. The drawback of this solution is that our replaying mechanism is already CPU intensive, and compressing data on the fly would only make this worse.

Another option we have is to take a closer look at the arguments of the systemcalls that Argos makes, and only store data that is absolutely needed. The most likely candidate for this is the `select()` system call, which takes 60% of the diskspace.

7. Conclusions

We have presented Argos–replay, an enhancement to the Argos virtual machine which provides a record/replay functionality. we have shown that replaying an execution is faster than its capture phase by up to a factor 2–3. This speedup is explained by the fact that during replay, actual system calls that request data do not need to be made as the data is already available from the capture log.

We have also described a two–tier architecture where a lightweight capturing machine stores all nondeterministic events that occur in a virtual machine to a log. This log is read in real time by a secondary machine which replays the virtual machine's execution in parallel with the capturing machine.

The secondary machines may be equipped with additional instrumentation to examine the execution of the machine in greater detail. We have shown that the slowdown imposed by this extra instrumentation can be offset by the speedup achieved by replaying.

It is also possible to run multiple secondary machines each with their own additional instrumentation, effectively parallelizing the additional checks performed to the virtual machine's execution.

REFERENCES AND LITERATURE

[1] tcpdump; a tool for network monitoring and data acquisition; `http://www.tcpdump.org`

[2] tcpreplay; Pcap editing and replay tools for *nix. `http://tcpreplay.synfin.org`

[3] Bit-twiste; a pcap-based Ethernet packet generator; `http://bittwist.sourceforge.net`

[4] Tomahawk; a tool to bidirectionally replay saved tcpdump logs. `http://www.tomahawktesttool.org`

[5] A. Turner. Flowreplay design notes. `http://www.synfin.net/papers/flowreplay.pdf`

[6] M. Bailey, E. Cooke, F. Jahanian, D. Watson, J.Nazario. The Blaster Worm: Then and Now. In *IEEE Security & Privacy*, Vol. 3, No. 04, pp. 26-31, Jul/Aug 2005.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical conference.* April 2005.

[8] W. Cui, V. Paxson, N. C. Weaver and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *proceedings of the 13th Annual Network and Distributed System Security Symposium.* February 2006

[9] J. Chow, T. Garfinkel, and P.M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *proceedings of the Usenix Annual Technical Conference 2008*

[10] G.W. Dunlap, S.T. King, S. Cinar, M. Basrai and P.M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceeding of the 2002 Symposium on Operating Systems Design and Implementation*. December 2002

[11] G.W. Dunlap. Execution Replay for Intrusion Analysis. *Ph.D. Dissertatio*n. 2006.

[12] C. Leita, K. Mermoud, M. Dacier, ScriptGen: an automated script generation tool for honeyd. I*n 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005

[13] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. In *J Mol biol 48.* 1970.

[14] J. Newsome, D. Brumley, J. Frankin and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In *Proceedings of the 13th ACM conference on Computer and communications security*. 2006.

[15] J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX Winter Tech. Conf.,* New Orleans, LA, USA. January 1995.

[16] G. Portokalidis and A. Slowinska and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *proceedings of the ACM SIGOPS EUROSYS'* . 2006

[17] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the International symposium on Automated Analysis-driven Debugging (AADEBUG).* September 2005.

[18] F. Shafique, K. Po, A. Goel. Correlating multi-session attacks via replay. In *Proceedings of the 2nd conference on Hot Topics in System Dependability, vol 2.* 2006.

[19] A. Slowinska and H. Bos. Prospector: Accurate analysis of heap and stack overflows by means of age stamps. Technical Report IR-CS-031. Vrije Universiteit Amsterdam, January 2007.

[20] M. Xu, V Malyugin, J. Sheldon, G. Venkitachalam and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS.* 2007