Defending Against Side-Channel Attacks: DynamoREA

by

David Wen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author ....
Department of Electrical Engineering and Computer Science
August 22, 2011

Certified by..
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by ....
Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

Defending Against Side-Channel Attacks: DynamoREA

by

David Wen

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2011, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Modern computer architectures are prone to leak information about their applications through side-channels caused by micro-architectural side-effects. Through these side-channels, attackers can launch timing attacks by observing how long an application takes to execute and using this timing information to exfiltrate secrets from the application. Timing attacks are dangerous because they break mechanisms that are thought to be secure, such as sandboxing or cryptography. Cloud systems are especially vulnerable, as virtual machines that are thought to be completely isolated on the cloud are at risk of leaking information through side-channels to other virtual machines. DynamoREA is a software solution to protect applications from leaking information through micro-architectural side-channels. DynamoREA uses dynamic binary rewriting to transform application binaries at runtime so that they appear to an observer to be executing on a machine that is absent of micro-architectural side-effects and thus do not leak information through micro-architectural side-channels. A set of test applications and standard applications was used to confirm that DynamoREA does indeed prevent sensitive information from leaking through timing channels. DynamoREA is a promising start to using dynamic binary rewriting as a tool to defend against side-channel attacks.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Thesis Supervisor: Eran Tromer
Title: Co-Advisor

Acknowledgments

To my thesis supervisor, Saman Amarasinghe, I am thankful for providing me with the opportunity to work on this project and giving helpful feedback during the course of writing this thesis. I am grateful for my other thesis supervisor, Eran Tromer, who has been able to provide constant guidance in all phases of my M.Eng, even through his busy schedule multiple timezones away. I would also like to thank Qin Zhao for helping to integrate Umbra into DynamoREA and for having the answers to any questions I had regarding DynamoRIO. I would also like to acknowledge my mother, father, and brother for their support throughout the years, along with friends and teammates who have been able to keep me sane throughout my MIT experience.

Contents

List of Figures

Chapter 1

Introduction

In an ideal world, software programs are allowed access only to the information it needs, no more and no less. Furthermore, these programs are able to protect their private data from unauthorized access or modification from attackers who may want to read this data. In this ideal world, sensitive data is accessible only when permission to read is explicitly granted. Unfortunately, in the real world, this often is not the case. Even if a program takes measures to safeguard against unauthorized reading of its secrets, there are plenty of other ways it may unintentionally leak information about its secrets through computer architectural effects.

The confinement problem is the problem of ensuring that a borrowed program is incapable of stealing information from a borrower to pass on to the program's author without the borrower's permission. There are multiple channels in which information can pass through, including channels that are not intended for information transfer at all, such as file locks. For example, the borrowed program may read data and concurrently lock or unlock a file depending on whether it reads a 0 or a 1 bit. An outside program that tries to open this file will be able to deduce the data depending on whether or not opening the files succeeds. With this set-up, the borrowed program is able to communicate the borrower's information to the outside using indirect channels not normally used for communication. This particular attack where the author of the borrowed program writes mechanisms that reveal its user's secrets through indirect channels are called covert-channel attacks.

1.1 Covert-Channel Attack

In a covert-channel attack, an attacker attempts to exfiltrate a secret protected by security mechanisms like sandboxing or mandatory access control . The attacker sets up two processes, a trojan process and a spy process. In the covert-channel set up, the trojan process controls the application containing the secret and attempts to pass bits of information to the spy process on the same computer. The security mechanisms protecting the secret prevent the trojan process from communicating to the outside through trivial means, such as through the network or inter-process communication. However, the attacker can find other ways to communicate between the trojan and spy processes.

Shared access to the cache can be an example of a covert channel [12]. One example of a cache-based covert-channel attack involves the trojan process evicting the entire cache to transmit a '1' bit and leaving it intact to transmit a '0' bit. If the spy process is making memory accesses and timing how long it takes to retrieve data from memory, it can notice whenever the data it's retrieving is in cache or if it has been evicted, thus receiving bit information from the trojan process, despite security mechanisms designed to prevent such communication. Though the attacker never has direct access to the secret in the application, he can use the trojan process to relay information about the secret. In this case, the secret is revealed because of the micro-architectural effect of varied timings of memory accesses depending on the cache's state. If there were a machine where the cache evictions do not affect the timings of memory accesses, then this attack would fail.

1.2 Side-Channel Attack

Covert-channel attacks usually require the victim executing a program that the attacker authored to take advantage of indirect channels. However, certain attacks can extract secrets through indirect channels without the cooperation of a trojan process. These attacks are called side-channel attacks and can be more dangerous than covert-channel attacks since they can occur without the user running a program the attacker authored.

When handling data, a program may behave differently depending on its inputs. Timing variances from different inputs serve as side-channels and can leak information about the inputs. For example, some attacks take advantage of the fact that going through different branches of a conditional statement may take different time. In the RSA decryption algorithm, a modular exponentiation is calculated with the cipher as the base and the decryption key as the exponent. An algorithm can calculate this value by iterating through the bits of the decryption key and solving the value step by step. However, this algorithm treats the 0s and 1s in the key's binary representation differently and consequently has a different runtime depending on the conditional branch it takes per bit. By timing the RSA decryption, an attacker could take advantage of the timing discrepancies to make deductions on what the secret key could be. We refer to these type of attacks as code-based side-channel attacks, as the behavior of the code itself is the cause of leaking information through side channels.

Another example of a side-channel attack is the cache-based side-channel attack. An application may write to and read from the computers cache. The application runs faster if the data it needs is readily available in cache. If the data is not in the cache, the application needs to take time to fetch it from slower memory. An attacker can take advantage of this speed discrepancy by first filling the cache with his own data, then triggering a victim's application to execute, and then attempting to read his own data from cache. By timing his memory accesses, the attacker can deduce which of his own data had been evicted by the victim's application. This timing information can then be analyzed to reveal secret information, such as keys, about the victim's application. It has been demonstrated that attackers are able to use cache-based side-channel attacks to determine the secret key from an OpenSSL RSA decryption [4] [15] and AES encryption [17]. Again, if we had a machine where changing the state of the cache does not affect the time it takes to make memory accesses, cache-based side-channel attacks could be prevented.

In addition, non-determinism in concurrent programs is also a potential channel for leaking information. The choices a thread scheduler makes or variations in hardware performance can cause the same inputs to result in different outputs. These discrepancies caused by micro-architectural effects can reveal information about the inputs and leak secrets. We make the distinction between **internal non-determinism** and **external non-determinism**. The sources of non-determinism listed above are from within the computer system and we refer to this type of non-determinism as internal non-determinism. External non-determinism comes from outside the computer system, such as network latencies or the time it takes to wait for a user input.

1.3 Attack on Cloud Computing Services

Since cloud computing usually allows users to launch virtual machines on the same physical machine, cloud computing services may be suspect to side-channel attacks. Amazon provides a cloud computing service called Amazon Elastic Cloud Compute (EC2) [1] . Users are able to pay for a specified amount of computing power and launch instances to make EC2 do the computing work necessary. EC2 comes with the expectation that all instances are isolated from each other. Any two instances should be as separate as two different physical computers. However, in reality, Amazon uses virtualization and two instances may actually share the same hardware, even though they appear to be running on separate machines. As virtual machines share the same resources, side-channels may form and security vulnerabilities arise.

It has been demonstrated that it is possible for an EC2 user to systematically launch EC2 instances until he is sharing a computer with a target user's instance running on it [16] . In addition, communication channels between virtual machines have been implemented, showing that it is possible for an attacker to target a victim on an established cloud computing platform and set up a side-channel attack on that victim. Though an EC2 instance is supposed to be isolated from other instances, in practice it is possible to launch a separate instance onto the same machine and expose the instance's activity.

1.4 Motivation for Preventing Side-channel Attacks

Side-channel attacks are dangerous because it can break mechanisms that are thought to be secure through other security mechanisms, such as sandboxing or cryptography. No matter how secure an algorithm might be, it may be vulnerable if it does not protect its data from leaking through side-channels. AES encryption was thought to be algorithmically secure until it was discovered that attacking the system AES was running on instead of the AES ciphers themselves yielded enough information to deduce an AES key.

Side-channel attacks require that the attacker have access to the same computer the victim is using in order to observe and monitor these side channels. They are dangerous in environments where computers are shared and an attacker is free to share hardware with his targets. In cloud computing, an attackers virtual machine and a victims virtual machine could share the same physical machine. As services gradually migrate towards the cloud, preventing side-channel attacks is a significant area of concern.

1.5 Existing Countermeasures

To defend against side-channel attacks, countermeasures have been taken to try to ensure that the observations an attacker can make from a program (i.e. what is communicated by the program and when it is being communicated) are independent from the inputs of the program. For example, what the attacker can observe from triggering an RSA decryption must not depend on the private key.

1.5.1 Fuzzy Clock

The attacks that have been discussed rely on being able to time some action (e.g. how long it takes to access some memory) and learning bits of information based on the results of the timings. To disrupt such attacks, one can implement a low resolution computer clock, or a "fuzzy clock" [11], so that the attacker can not distinguish between different timing results, rendering analysis of some of the side channels ineffective. However, using a fuzzy clock can disturb legitimate programs that need to use an accurate clock. Also, if the clock's resolution isn't low enough, attackers would be able to average more samples to compensate for the fuzzier clock and still achieve the same results as before. Furthermore, even with the perfect fuzzy clock, a system would still be vulnerable to other side channels.

1.5.2 Security-Typed Language

Another countermeasure is to provide and use a security-typed language. Well-typed programs with these such languages provide a level of security depending on what the type system of the languages check.[2] [18]. Under these languages, it's possible to remove vulnerable timing variances and mitigate code-based side-channel attacks, such as in the conditional statement in the modular exponentiation algorithm mentioned above. However, security-typed languages do not apply well to proprietary and existing software, as changing languages can be complex, impractical, or impossible if the source is not available.

Security-typed languages can catch security issues in a program's code, but they are unable to do anything about side-channel vulnerabilities that can't be fixed from within the code. Even if a program is well-typed in a security-typed language, it does not guarantee that the program will not leak information through micro-architectural effects. It should be stressed that we are not trying to solve the same problems as security-typed languages. Our goals do not include fixing leaky code and saving programs from themselves. Rather, our mission is to make sure that securely-typed programs can execute securely without leaking any secrets through micro-architectural effects.

1.5.3 Enforcing Determinism

Various measures have been implemented to enforce determinism in multithreaded programs. CoreDet [7] uses a modified compiler and a new runtime system to execute multithreaded programs deterministically, alternating running threads in parallel and serial mode. DMP [5] proposes a hardware solution to enforce deterministic execution. dOS [8] borrows concepts from DMP and modifies the kernel to help enforce internal determinism. Kendo [14] uses a software approach and provides libraries to enforce determinism on multithreaded programs. Though security against side-channel attacks are usually not the primary goals for these determinism enforcers, many of their concepts are applicable to addressing vulnerabilities brought up by non-determinism. Like security-typed languages though, it is often difficult or impractical to apply these solutions to existing software, as these solutions either requires recompiling of programs, changes to the operating system, different hardware, or alterations in code.

1.5.4 DynamoREA

DynamoREA (Dynamic Runtime Enforcement of Abstraction) is our approach to mitigating side-channel attacks. DynamoREA is a software solution that enforces the abstraction of an ideal machine, a machine where applications run without micro-architectural effects that may leak information to outside observers. In this ideal machine, applications run deterministically and their behaviors are not affected by the activity of other processes that they do not directly interact with. If an attacker tries to observe an application in this ideal machine abstraction through software side channels, the attacker will learn nothing that can help him exfiltrate secrets. DynamoREA acts as a layer between the machine and the application so that to an observer, any application running on DynamoREA appears to be running on an ideal machine.

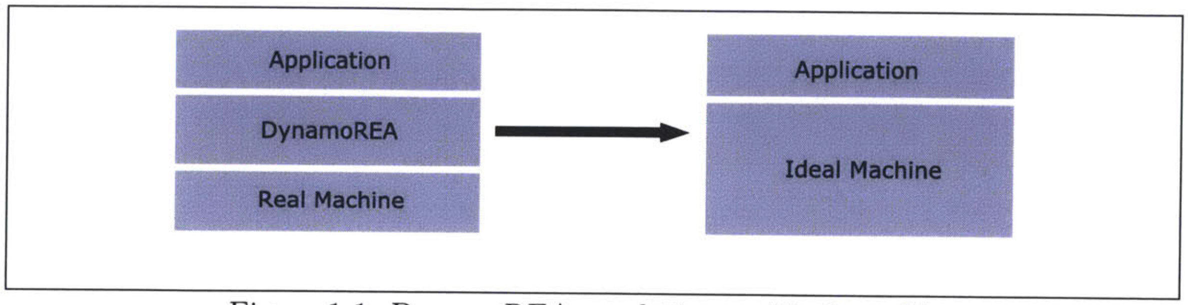

Figure 1-1: DynamoREA emulating an ideal machine

DynamoREA takes a unique approach to defend against side-channel attacks. We introduce and define the notion of an ideal machine that would be resistant to microarchitectural-based side-channel attacks. Unlike previous attempts to mitigate side-channel attacks or enforce determinism by modifying hardware, the compiler, the source code, or the kernel, DynamoREA uses dynamic binary rewriting to achieve its goal. This is both a new way to approach side-channel security and a new way to use dynamic binary rewriting tools.

In chapter 2, we will discuss the goals we want DynamoREA to accomplish, along with the theory and rationale behind its structure. We will also define the ideal machine abstraction that DynamoREA will enforce on applications and introduce the notion of ideal time. In chapter 3, we will describe how DynamoREA is implemented and reason out how our implementation enforces the ideal machine abstraction we defined. In chapter 4, we evaluate the performance and security of several applications executed on DynamoREA.

Chapter 2

Overview of DynamoREA: Theory and Rationale

2.1 Goals

Our goal is to provide a security system that defends against micro-architectural timing attacks by preventing any timing information from leaking through channels caused by micro-architecture effects. Our defense against these such side-channel attacks must be secure, generic, efficient, and extensible.

A **secure** defense is one that prevents applications from leaking sensitive timing information through micro-architectural effects. We need to provide a clear model of the behavior of an application and provide an argument why this behavior on our system is secure against micro-architectural timing attacks. To demonstrate security, we will provide a security invariant, argue that an application whose behavior maintains this invariant is secure, and then demonstrate that this invariant does indeed hold for applications running on our security system.

A **generic** defense should be able to be implemented on many standard platforms and not depend on any special platform properties. The security system should be usable on a wide variety of architectures and not rely on properties found on few systems to work properly.

An **efficient** defense is one that does not heavily impact performance. Even if a defense were secure and generic, it would still be impractical use if it hinders performance too much.

An **extensible** defense can be easily changed and deployed as newer attacks and vulnerabilities are discovered. As the defense changes, it should be easy to deploy the changes so that legacy code and commercial off-the-shelf code are also protected without modification.

As mentioned before, it's important to note that our defense against side-channel attacks is specifically targeting leaks through micro-architectural side effects. There are other channels that our solution does not address, such as code-based timing attacks discussed in section 1.5.2. We address security concerns stemming from micro-architectural effects, but assume that other side-channels are dealt with appropriately.

2.2 Ideal Machine Abstraction

Imagine a machine that does not have any micro-architectural artifacts. The number of CPU cycles required to execute each instruction does not depend on the state of the machine. For example, in this machine, an instruction will take the same number of cycles to execute regardless of the state of the cache. Multiple executions of a program on this machine would be indistinguishable from each other if we were monitoring their runtimes in terms of cycles. Timing attacks do not affect this machine since an attacker cannot gather useful timing results if every execution produces the exact same results.

We call this machine an *ideal machine*, since this machine would be impervious to micro-architecturally based timing attacks. On this ideal machine, the running time of a program depends only on its own activity and external events, not on the state of the machine or the activity of any other program on the machine. In reality, the typical machine is not ideal because most performance optimizations result in hardware resource sharing, which in turn creates micro-architectural artifacts. We can however strive to enforce an abstraction of an ideal machine that retains the performance optimizations of a real machine while eliminating the micro-architectural artifacts that usually result from those optimizations. While our physical machines are not ideal, our goal is to emulate an ideal machine and have every program look like it is actually running on an ideal machine. An external observer should not be able to tell that programs are in fact running on a real machine.

Though we are emulating this ideal machine on a real machine, we do not require that the ideal machine is executing at the same rate as the real machine it is being emulated on. That is, we do not specify that the time it takes to run a CPU cycle on the real machine has to be the time it takes to run a cycle on the emulated ideal machine. We introduce a wallclock constant, ω, that translates wallclock time to cycles on an ideal machine with (cycles / seconds) as units. The speed of the ideal machine that we will try to emulate depends on ω.

2.2.1 Autobiography

Since the running time of a program depends on its own activity in an ideal machine, we need some way of modeling what constitutes a program's activity. We introduce the notion of a program's *autobiography.* An autobiography provides this model and contains information about the program's activity, such as how many instructions have been executed, what types of instructions have been executed, and how many basic blocks have been instrumented. An autobiography of a program is built at runtime, as it may not be possible to predict a program's activity beforehand. As instructions and basic blocks are being executed, they are also being tracked and counted by the autobiography. The autobiography represents a program's own activity, from which we will derive how fast an ideal machine would execute this program and attempt to emulate this execution.

2.2.2 Holdback

There are certain points of a program's execution where the running time cannot be derived directly from the information provided in the autobiography. For instance, there are system calls that wait for particular events before completing and the amount of time the program needs to wait cannot be deduced from the autobiography. An application may have syscalls that wait for a reception of a network packet, a child thread to finish execution, or a user submitting input to the program before continuing. These events that an application might wait for are called *holdback events.* We define *holdback* as the number of cycles a machine takes to wait for a holdback event. *Real holdback* is the number of cycles a real machine had to wait for a holdback event. *Ideal holdback* is the number of cycles that an ideal machine would have needed to wait for a holdback event.

Like a real machine, an ideal machine must also spend time to wait for holdback events. A machine with no micro-architectural side effects is still subject to waiting for holdback events after all. Thus, if we wanted to emulate the behavior of an ideal machine, we must account for the holdback the ideal machine would experience. We can measure real holdback, but real holdback may not directly translate to ideal holdback since the ideal machine does not necessarily run at the same rate as a real machine. However, using the time, r, it takes to run a CPU cycle on a real machine and given the wallclock constant ω, we can introduce a constant of proportionality $\alpha = r\omega$ that translates real holdback to ideal holdback. If we measure real holdback to be h, the corresponding ideal machine's holdback would be αh.

There are two types of holdback events. A holdback event may come from outside of the program's own architecture, such as the cases of waiting for user input and waiting for incoming network packets. We call this type of event an *external holdback event.* To calculate ideal holdback in these cases, we measure real holdback and use α to translate the measurement into ideal holdback.

The other type of holdback event comes from within the program's own architecture. In the case of waiting for a child thread to finish executing, the length of the wait can be completely measured on the ideal machine if the child thread is not affected by external events. We call these holdback events *internal holdback events.* Internal holdback can be measured directly on the emulated ideal machine because it only depends on the program's own architecture, unlike external holdback which needs to be measured in the real machine and then translated to the ideal context.

The goal is to calculate how much total holdback an ideal machine would experience, the cumulative ideal holdback. If we measure H_i ideal machine cycles for internal holdback and H_r real machine cycles for external holdback, the cumulative ideal holdback would be $H_i + \alpha H_r$

2.2.3 Ideal Time

From a program's autobiography and cumulative holdback, we can calculate the running time of an application if it were running on an ideal machine. We call this running time on an ideal machine *ideal time*, or T_i, and it is measured in terms of CPU cycles on the ideal machine, or *ideal cycles.* Since in an ideal machine, a program's running time depends only on its own activity and holdback, it is important that the ideal time calculations depend only on a program's autobiography and holdback.

Given some point of a program's execution, let E_i be the number of cycles an ideal machine would execute, derived from solely the program's autobiography at that point, H_i be the ideal holdback calculated from internal holdback events, and H_r be the real holdback measured from external holdback events. The ideal time, or the amount of cycles we would expect an ideal machine to execute the program to that point, can be written as follows:

$$T_i = E_i + H_i + \alpha H_r$$

where α is the constant of proportionality relating holdback on a real machine to holdback on an ideal machine calculated from section 2.2.2.

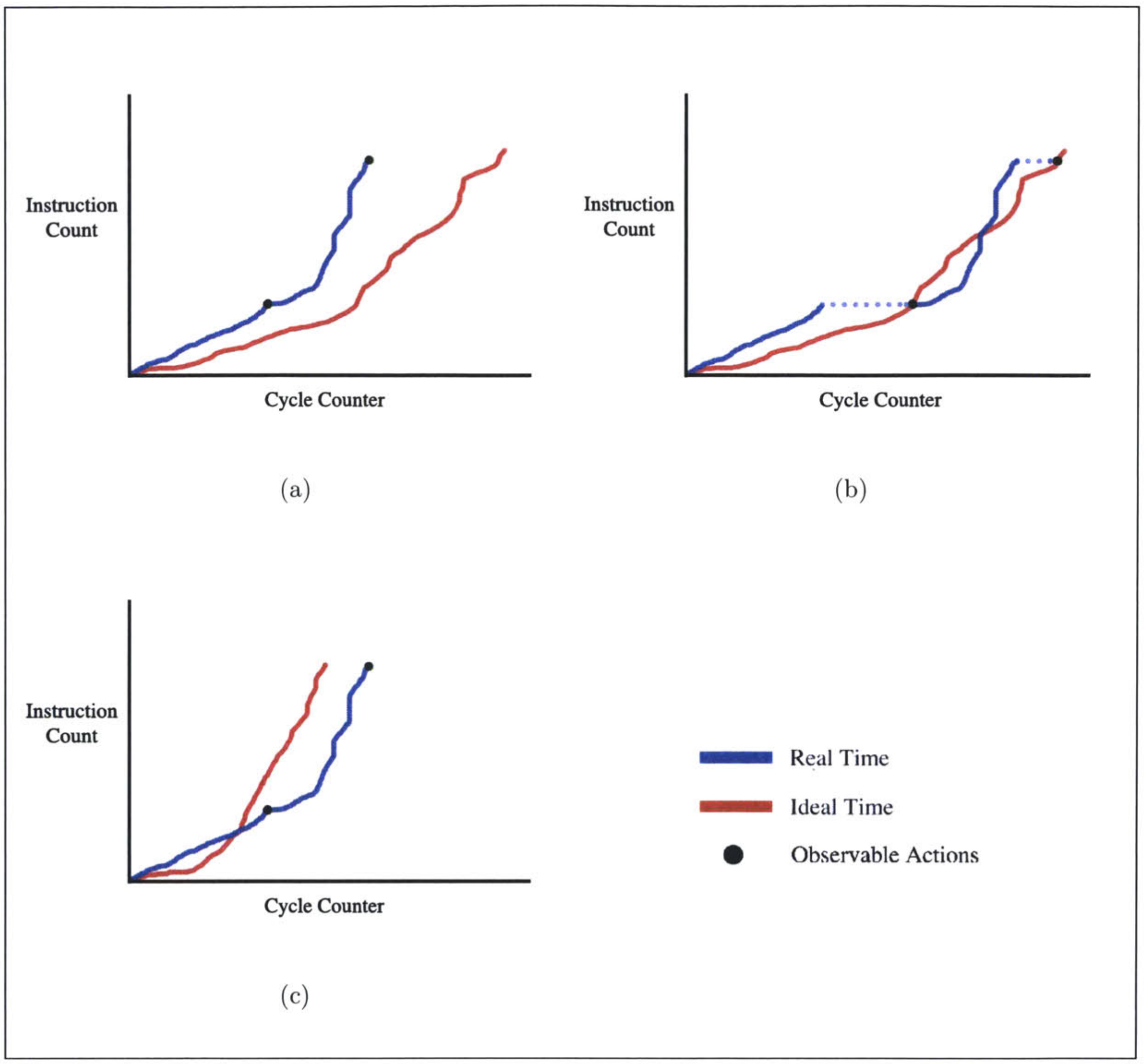

Figure 2-1: (a) Real time compared to ideal time (b) Real time delaying to match ideal time at observable actions. The dashed lines represent the program delaying right before executing the observable action. (c) An ideal machine abstraction violation: real time is greater than ideal time at observable actions

Our goal is to have every program look like it's running on an ideal machine, so at all points of a program's execution that can be observed by a hypothetical attacker, the program's actual count of executed cycles, or *real time*, must be equal to the program's ideal time. This maintains the illusion that the program is running on an ideal machine, otherwise known as enforcing the ideal machine abstraction.

Only at a program's observable actions do ideal time and real time need to be equal to enforce the ideal machine abstraction. At other periods of a program's execution, real time is allowed to differ from ideal time without violating the ideal machine abstraction (see Figure 2-1(b)). However, in order to ensure that real time and ideal time will be equal at observable actions, it is necessary that real time is less than or equal to ideal time right before observable actions. If real time is greater than ideal time, then there is no way of equalizing the times and enforcing the ideal machine abstraction at the observable actions (see Figure 2-1(c)). In contrast, if real time is less than ideal time, the program can delay right before the observable action and allow real time to "catch up" to ideal time before executing the observable action.

Because it is necessary that real time not be greater than ideal time right before observable actions, we have to be careful when calculating ideal time. In general, calculating ideal time requires assuming worst case program execution. If ideal time calculations do not take into account worst cases, such as no cache hits, then we run the risk that when worst cases do happen, real time surpasses ideal time and the ideal machine abstraction is violated. Ideal time calculations must be conservative enough so that even in the worst case, a program's real time will be no greater than ideal time at observable actions.

In addition, an ideal machine must be absent of internal non-determinism. Internal non-determinism is caused by micro-architectural effects and thus a machine without micro-architectural side-effects must be internally deterministic. We aim to emulate an ideal machine that executes observable actions at predictable times and is internally deterministic as a result of having no micro-architectural effects. An ideal machine however is still suspect to external non-determinism. An example of external non-determinism is holdback, since the amount of holdback experienced can depend on external effects like network latency. External non-determinism will still be present in an ideal machine, but problems arising from this type of determinism cannot be solved with an ideal machine.

Chapter 3

Design and Implementation of DynamoREA

3.1 Dynamic Binary Rewriting

One of our goals for DynamoREA is to have a defense that is easily deployed on existing code. It is impractical to manually rewrite every applications to be secure against side-channel attacks, however it is feasible if we could automatically rewrite the applications. Rewriting application sources is complex or impossible if source code is not available, but we can imagine being able to catch side-channel vulnerabilities in the application's binaries and doing something about them at that point.

Dynamic binary rewriting tools make it possible to modify an application's machine code after it has been compiled, which is useful when there is no access to source code. These tools are able to access and modify machine code instruction by instruction. Examples of popular dynamic binary rewriting tools include Pin[6] and Valgrind[13]. They are often used for analyzing performance or diagnosing problems in an application. However, for our case, dynamic binary rewriting tools will be useful to build a program's autobiography since autobiographies need to be built at runtime. Dynamic binary rewriting will also help with enforcing the ideal machine abstraction, for example inserting delays so that observed cycles equals ideal cycles at observable actions.

Dynamic binary rewriting is an effective method to provide a generic and efficient defense against side-channel attacks. Since dynamic binary rewriting works on compiled code, there is no need to recompile to reap the benefits of a dynamic binary rewriting solution. This also means that users can use a dynamic binary rewriting defense on applications even if they are not able to modify the sources. Being able to work at the instruction level also allows us to modify the behavior of applications to adhere to security invariants without too much overhead.

3.2 DynamoRIO

DynamoREA is built on DynamoRIO [3] (Dynamic Run-time Introspection and Optimization), a framework that allows for dynamic binary rewriting. Using this framework, a user writes a DynamoRIO client specifying how he wants the application to be transformed at runtime. DynamoRIO is able to modify blocks of instructions by transforming basic blocks the first time they are executed and using the transformed version of the basic block every time it is executed. DynamoRIO also provides hooks to modify the behavior of system calls, thread initialization and exit, and more. It provides the ability to transform an application binary to insert, remove, or replace instructions through the client code. Running any application through the DynamoRIO client modifies the instructions in the application binary and runs the modified application.

DynamoREA uses DynamoRIO to register events to help build the autobiography and maintain the ideal machine abstraction. As we go through the features of DynamoREA, we will note how DynamoRIO is used to implement these features.

3.3 Maintaining Autobiography

As discussed in section 2.2.1, we want to keep track of an autobiography, a model of a program's execution activity. From the autobiography, we calculate the ideal execution time, the number of cycles the ideal machine takes to execute the instructions in an application, disregarding holdback. The elements that we want to keep track of in the autobiography are the components in the instructions that contribute to ideal execution time.

The actual operation in each instruction contributes to the running time of the execution. Since different instruction operations take different time, the autobiography needs to keep track of the ideal time contribution of the operations in the application's instructions. For instance, an add operation is considerably cheaper than a divide operation, and we need to make sure we treat these operations differently when determining the instruction's execution time on an ideal machine. Using Agner Fog's table of instructions and cycles per instruction [9], DynamoREA can look up an estimate of how many ideal cycles an instruction should contribute based on the instruction's opcode.

Also, if an instruction accesses memory, it may take longer than a similar instruction that doesn't access memory, even if they have the same operations. As a result, we should also keep note of instructions that access memory. DynamoREA uses the number of memory accesses to calculate the number of cycles an ideal machine would use to make memory accesses in the application.

Also, it takes time to load a basic block to execute, so the number of basic blocks that have been encountered will also contribute to the running time of the application execution and should be tracked as well.

The autobiography keeps track of the following information about an application's history:

- **mem_instr -** Number of executed instructions that accessed memory
- **weighted_op_count -** Estimate on the number of cycles contributed by executing the operation of executed instructions
- **basic_blocks -** Number of basic blocks that have been executed
- **unique_basic_blocks -** Number of times a basic block needed to be instrumented for the first time

From these statistics, we will be able to produce an application's ideal execution time, how long it takes an ideal machine to execute the instructions that make up the application.

With the help of DynamoRIO, we can dynamically build an application's autobiography as the program is executing. Using DynamoRIO, we are able to inspect basic blocks at runtime before executing them. DynamoREA registers DynamoRIO's basic block event, which allows us to modify each basic block every time a new basic block is being executed in the application. This basic block event normally occurs on the first time a basic block is seen, as DynamoRIO keeps a cache of transformed basic blocks so that we do not need to change a basic block that has already been modified. The extra time DynamoRIO takes to instrument a basic block the first time it sees it is why we want to keep track of the number of unique_basic_blocks in the autobiography, as we need to factor in this time into the ideal execution time.

DynamoRIO's API allows DynamoREA to inspect instructions, determining whether or not its operands touch memory or not and what opcode the instruction has. As DynamoREA inspects a basic block for the first time, it counts how many memory instructions are in the block and computes the block's weighted op count. DynamoREA then inserts instructions to the application to update the autobiography with these values, in addition to incrementing the autobiography's basic block count. At any given point of an application's execution, the autobiography represents a summary of all the actions relevant to execution time that has been executed by the application to that point. It is from this snapshot that DynamoREA computes ideal execution time, the number of cycles an ideal machine would have had to use to get to that point of the application's execution according to the actions tracked in the autobiography. In section 3.5.1 we will discuss how exactly DynamoREA uses the autobiography statistics to produce an ideal execution time.

A *checkpoint* is defined as an observable action in the application's execution where real time needed to be delayed to ideal time. In addition to keeping track of the accumulated instructions, basic blocks, and operations statistics, the autobiography also keeps a snapshot of the accumulated statistics at the last checkpoint.

3.4 Timing Events

To maintain the ideal machine abstraction, DynamoREA needs to ensure that real time does not surpass ideal time at observable actions. Observable actions come in two types, internal and external. An action that is internally observable can be seen by other processes on the same machine. An externally observable action can be seen by observers outside of the machine, for example sending a packet through the network. Both observable actions need to be executed at the correct ideal time, but external observable actions also need to be executed at the correct wallclock time.

3.4.1 System Calls

System calls allow applications to communicate with the operating system and request services. Since system calls can affect or be affected by the results of other system calls from other processes, they are a main source of observable actions. Whenever a system call is made, DynamoREA needs to be especially careful to ensure that an observer cannot distinguish the behavior of the real machine from the behavior of an ideal machine.

DynamoRIO provides the ability to register for events immediately before and after system calls, allowing DynamoREA to take the appropriate actions immediately before or after each system call. DynamoRIO also allows us to inspect what type of syscall is being executed so that we may be able to treat different syscalls differently. In addition, DynamoRIO provides API for modifying syscalls, giving us the option to skip syscalls, modify the parameters of the syscall, or modify the result of the syscall.

3.4.2 Signals

A signal is a way for processes to directly communicate with another. Processes can receive signals from other processes which may interrupt its normal flow of execution and alter its behavior. Signals are observable actions and must be handled so that incoming and outgoing signals do not reveal that an application is executing outside of an ideal machine.

Like system calls, DynamoRIO also is able to register signal events, intercepting signals as an application receives them and allowing us to decide what to do with them before processing them.

3.4.3 Determinism

Race conditions may reveal information about how multiple processes interact with each other and also violate the ideal machine abstraction. Since an ideal machine has no micro-architectural effects, it is absent of internal non-determinism, the source of race conditions. Thus, DynamoREA needs to enforce internal determinism across multiple threads and processes so that every execution of an application given the same input will behave no differently from each other to an observer.

External non-determinism is still present in an ideal machine. While we will not try to remove external non-determinism, we need to ensure that the external non-determinism experienced on a real machine is translated appropriately to the external non-determinism an ideal machine would face.

3.4.4 Shared Memory

Shared memory allows two processes to pass information between each other. Like system calls and signals, shared memory writes are actions that are potentially observable by an outside observer who makes a shared memory read. Unlike system calls and signals, DynamoRIO does not come with a shared memory access detector, so we must find some method to determine when shared memory is being accessed and handle these accesses safely.

3.5 Computing Ideal Time

The ideal time is calculated strictly from information held in the autobiography of an application and the application's holdback (the total number of cycles spent waiting for external events). In this section, we will discuss how ideal time is derived from the autobiography and how ideal holdback is calculated.

3.5.1 Autobiography Ideal Cycles

The autobiography keeps track of exactly how many basic blocks have been executed, how many memory instructions have been executed, and a weighted op count for all instructions (an estimate of the number of CPU cycles used to execute all the logic operations). DynamoREA' uses the application's autobiography to calculate ideal execution time, how many cycles an ideal machine would take to execute the instructions and basic blocks tracked by the autobiography.

Since we need to maintain the ideal machine abstraction by ensuring that real time is less than ideal time at every observable action, to calculate ideal time as a function of the autobiography, we need to calculate upper bounds on the number of cycles spent on memory accesses, number of cycles spent on logic operations, and number of cycles loading basic blocks. By combining these components, DynamoREA calculates the upper bounds of cycles spent by the application according to its autobiography at every point during its execution. By using this upper bound as our ideal time calculation, we ensure that at every observable action, we never have real time exceeding ideal time and consequently we are able to maintain the ideal machine abstraction.

In practice, DynamoREA keeps track of ideal time by computing the number of ideal cycles since the last checkpoint. At each checkpoint, DynamoREA adds the ideal time at the previous checkpoint to the ideal time since the previous checkpoint to get the total ideal time at the current checkpoint.

3.5.2 Memory Access

When DynamoREA calculates the upper bound on the number of cycles spent on memory accesses, naively it could assume every memory access is a cache miss and the upper bound is simply the product of the number of memory accesses and the number of cycles spent on a cache miss. The number of actual cycles used for memory accesses would never exceed the number of ideal cycles computed in this approach. Assuming that memory access with a cache miss takes M_{worst} cycles, We can write the ideal time memory access function, f_{mem} in terms of the number of memory accesses since the last checkpoint, n_{mem} like so:

$$\text{naive } f_{mem}(n_{mem}) = M_{worst}n_{mem}$$

While this approach would satisfy the security requirement in that real time spent on memory accesses will never exceed the ideal time spent on memory accesses, its performance is quite poor. We improve on this memory access function by taking advantage of general cache behavior and relaxing the requirement that we must assume worst cases. Instead, we can aim to simulate a conservative average case of memory access behavior. In general, DynamoREA can expect initial memory accesses to take longer, due to cache misses on a cold cache, while later memory accesses should take less time on average after the cache is warmed up. To capture this behavior, we make the assumption that an application will take $n_{threshold}$ memory accesses before the cache warms up. During the first $n_{threshold}$ memory accesses, we assume that the application will take M_{worst} cycles for each access. Following the first $n_{threshold}$ accesses, memory accesses then take M_{avg} cycles, where M_{avg} is the average number of cycles spent on a memory access with a warm cache. In summary, our new ideal time memory access function is now:

$$\begin{aligned} \text{threshold } f_{mem}(n_{mem}) &= M_{worst}n_{mem} \text{ for } n_{mem} < n_{threshold} \\ &= M_{worst}n_{threshold} + M_{avg}(n_{mem} - n_{threshold}) \text{ for } n_{mem} \geq n_{threshold} \end{aligned}$$

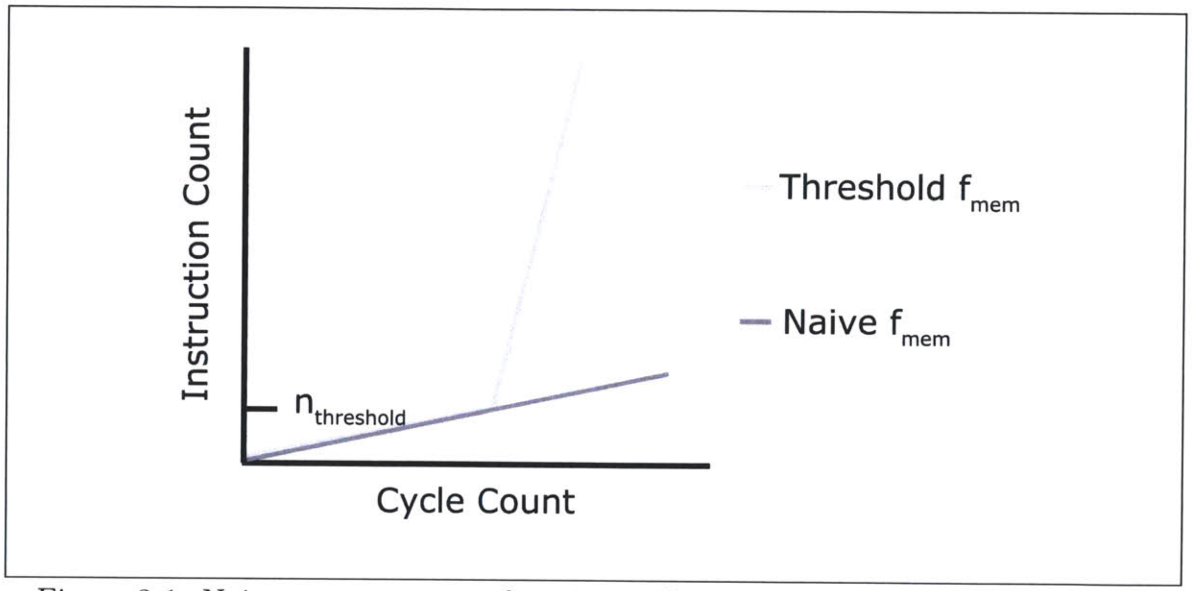

Figure 3-1: Naive memory access function and threshold memory access function

See figure 3-1 to see the two memory access functions compared to each other.

Recall that the memory access function is in terms of the number of memory accesses since the last checkpoint. This means at every checkpoint, DynamoREA assumes that we have a cold cache and memory accesses immediately after each checkpoint will take M_{worst} cycles. This is the intended behavior, since we can not assume that once the cache is warm that it will remain warm for the rest of the application's execution.

While this stepwise function an improvement, it is still only slightly less naive than the first function. In section 3.9.5, we revisit the memory access function and demonstrate how we can optimize it further by tailoring it to specific applications.

3.5.3 Logic Operations

DynamoREA contains a table with many ops and a cycle count associated with each op. As DynamoREA processes each instruction as the application executes, it looks up the instruction's op in the table and adds that op's associated cycle count to a running total in the autobiography, keeping track of a cumulative weighted op count. This weighted op count represents DynamoREA's estimate of the number of cycles used to execute the logic operation portion of all the instructions up to that point in the application's execution. If the op is not found in the table, then a default safe value will be used for that op's cycle count.

Instruction	Operands	Clock Cycles
MOV	r/m, r/m/i	1
POP	r	1
PUSH	m	2
LEA	r, m	1
ADD SUB AND OR XOR	r, r/i	1
ADD SUB AND OR XOR	r, m	2
ADD SUB AND OR XOR	m, r/i	3
CMP	r, r/i	1
CMP	m, r/i	2

Figure 3-2: Sample table of tested instructions and average number of cycles they consume. In the operands column, r = register, m = memory, i = immediate.

Not all ops can be easily associated with a single cost estimate. For example, the cost of string operations can depend on the length of the string. The cost can not be estimated by looking at just the op itself. Currently, DynamoREA assigns the default safe value for these such ops' cycle counts.

3.5.4 Other Delays

Since DynamoREA needs to instrument every new basic block that it sees, the ideal time function needs to account for the time it takes DynamoREA to instrument basic blocks. The autobiography keeps track of how many unique basic blocks DynamoREA has encountered. The delay is factored into the ideal time by multiplying the number of unique basic blocks encountered by an estimate of how long it takes for DynamoREA to instrument basic blocks.

In addition, DynamoREA also needs to account for the time it takes to load a basic block. For each basic block tracked by the autobiography, DynamoREA adds an estimate of how long it takes to load a basic block to the ideal time.

DynamoREA initialization also needs to factor into the ideal time function. A one-time initialization penalty is added from the very beginning to account for the initialization time.

3.6 Handling Timing Events

3.6.1 Handling System Calls

If we compute ideal time properly, then right before executing a system call, an application has a lower real time than ideal time. In most cases, DynamoREA needs to ensure that at the system call execution, real time must equal ideal time, and must delay the system call until real time has caught up to ideal time. Delaying is the default safe behavior to handle observable actions. Delaying is not always the most efficient way to handle some system calls however. In section 3.9, we will discuss a couple examples of system calls that can be handled more efficiently without the need for delaying.

3.6.2 Handling Signals

Signals are observable actions that are not found in the application's autobiography. They can break the ideal machine abstraction if an application receives a signal while its real time is greater than its ideal time(which is allowable as long as an observable action is not being executed), since DynamoREA will not be given a chance to delay so that the signal is handled at the correct ideal time.

To handle signals, we can take advantage of the fact that POSIX specs do not state that signals need to be handled immediately upon reception. We can implement a signal queue that accumulates signals upon reception. We may designate signal synchronizations at some regular interval, for example every million instructions. At the signal synchronizations, we can empty the signal queue and handle the accumulated signals. WIth this signal handler, signals are processed without breaking the ideal machine abstraction.

DynamoREA currently does not have this signal handler implemented, though it is a natural extension of the project for the future.

3.6.3 Handling Internal Non-Determinism

In the case of applications with multiple threads, if there are observable actions on separate threads, we must ensure that the order of execution of those observable actions must be the same every time we execute the application. Otherwise, the ideal machine abstraction is violated since separate runs look different from each other. The order of instructions executed outside of observable actions does not concern us, but every observable action is a **synchronization event** where DynamoREA must synchronize all the threads and choose which thread's observable action must be executed first in a deterministic fashion. We maintain a **deterministic logical clock**[14] for each thread to help us determine which observable action should go first. A deterministic logical clock tracks progress of a thread in a deterministic manner and is advanced similarly to ideal time.

To enforce determinism, we introduce the rule that an observable action will only be executed by a thread if the deterministic logical time of that thread is smaller than all of the other threads being synchronized. The deterministic logical time of every thread can be accessed globally so they can be examined to determine whether or not a thread should proceed. If a thread reaches an observable action but sees that its deterministic logical time is not the least of all threads, then it must wait until the lagging threads advance further before continuing its execution. Provided that we can maintain logical time and resolve synchronization events deterministically, we will be able to enforce determinism on multithreaded applications.

Deterministic Logical Time

Since we already have a deterministic means to track progress of a thread in ideal time, we will use ideal time as our basis for deterministic logical time for the most part. Unlike ideal time however, we do not need to be able to keep track of the precise deterministic logical time at every instruction; our resolution can be much lower. In DynamoREA, we update a thread's deterministic logical time approximately every 10000 ideal cycles or at every observable event, whichever happens first. Though the

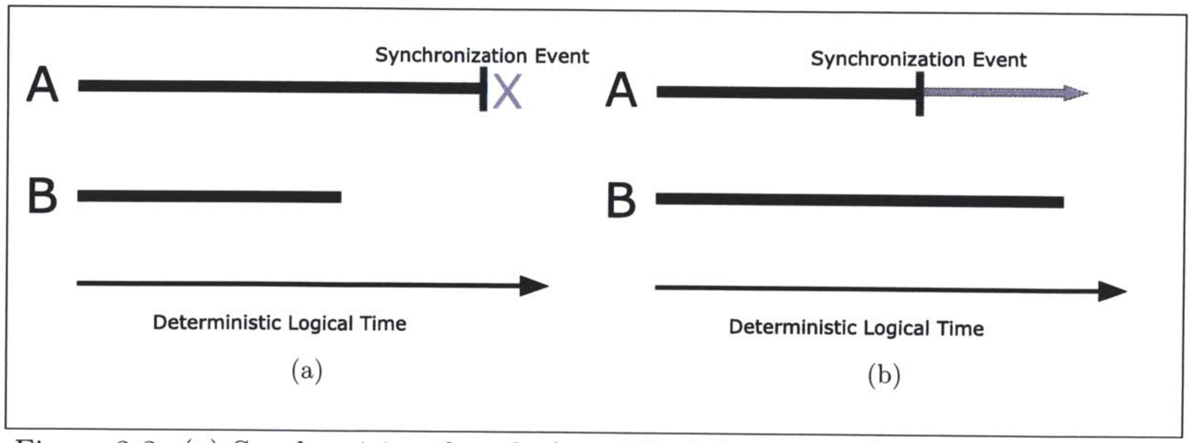

Figure 3-3: (a) Synchronizing threads A and B. A is at a synchronization event and must wait for B's deterministic logical clock to pass before it may continue. (b) A is at a synchronization event and is allowed to progress since its deterministic logical time is less than B's.

deterministic logical time and ideal time are similar, deterministic logical time is distinctly separate in that it can not be determined from an application's autobiography. There are cases where the deterministic logical clock is progressed separately from ideal time to resolve deadlocks as we will see in the following section. In these cases, the amount that the clock progresses depends on the behavior of other threads. As a result, the deterministic logical clock can not be derived directly from an application's autobiography.

Resolving Deadlock

Our method of providing deterministic multithreading of parallel applications uses a deterministic logical clock and an algorithm to decide which thread will proceed at synchronization points. When a synchronization point is reached and we have to decide which thread to progress, it chooses the thread with the smallest deterministic logical time. This helps ensure that the threads runs deterministically.

However, a deadlock can occur with this method. Suppose we have two threads, thread W and thread R. Thread W's next instruction is a `write()` system call and thread R's next instruction is a blocking `read()` system call that must wait for W's `write()` to finish. However, W's deterministic logical clock has progressed more than R's clock. By Kendo's thread progression rules, R must advance its deterministic logical time past W's time in order for W to continue. However, W must complete its `write()` in order for R to continue. Because of the blocking nature of R's `read()` call and our waiting rules, we have a deadlock.

To resolve deadlocks like these, we need to find a way to advance R's clock past W's deterministic logical time without skipping R's `read()` call. Once R's clock passes W's time, then W can continue with its `write()` call which in turn will allow R to execute its `read()` call.

In this case, we can resolve this deadlock by transforming the `read()` into a non-blocking `read()`. This new `read()` would fail as W's `write()` has not completed. On failure, R's thread would run a busy loop and repeatedly attempt another non-blocking `read()`. While R is in a busy loop, R's deterministic logical clock progresses. This pattern continues until R's clock progresses past W. At this point, it is W's turn to run and W executes the `write()`. Finally, R's non-blocking `read()` is successful and both threads continue, avoiding the deadlock problem.

This solution has a vulnerability if a user can observe what system calls are being executed by a multithreaded application. For example, `strace` and `truss` are system tools on Unix-like systems that monitors all the system calls used by an application. With our solution above, either of these tools would reveal how many `read()` calls have been executed. This information could reveal how many non-blocking `read()` it took for a thread to execute before a deadlock was resolved, which in turn would reveal how long a thread had to busy loop to resolve the deadlock. In our example above, this kind of information leaks how far behind thread R was from W and can violate the ideal machine abstraction.

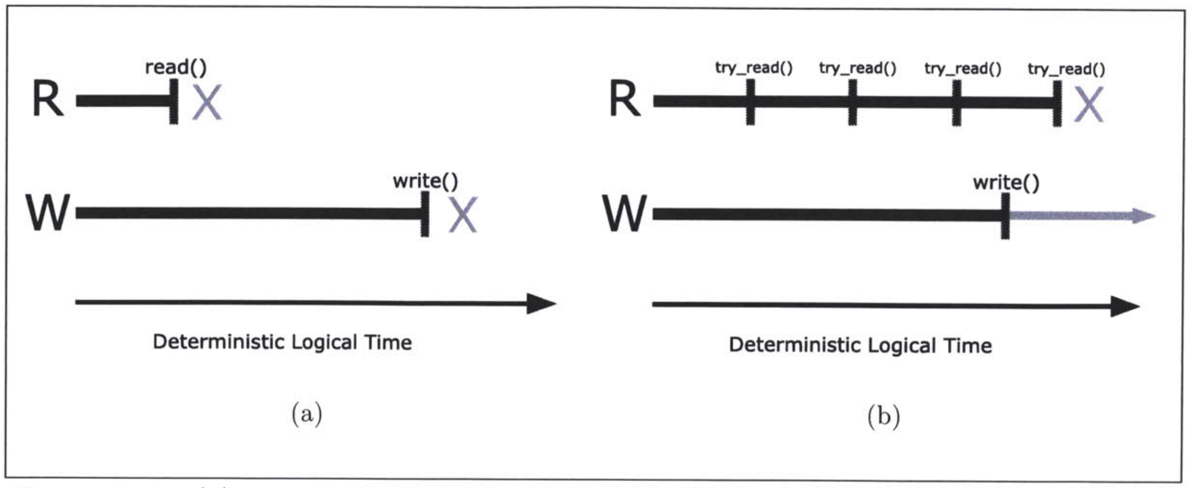

Figure 3-4: (a) Deadlock example. Thread R is blocked by a blocking read call that must wait for thread W's write to finish. Thread W is blocked because its deterministic logical clock is ahead of R's (b) Deadlock resolution. R's blocking read is transformed into a series of non-blocking reads until R's deterministic logical clock passes W's, freeing W's write and resolving the deadlock.

3.6.4 Handling Shared Memory

Umbra[19] is a memory shadowing framework that is an extension of DynamoRIO. With Umbra, we can determine when we are executing instructions that use shared memory. By using Umbra, we are able to include a hook to catch shared memory accesses and handle them appropriately. In our case, we treat a shared memory access as an observable action that needs to follow our determinism rules in the previous section. That is, a shared memory access instruction must not be executed until the deterministic logical time of every other thread has surpassed the deterministic logical time of the thread executing the shared memory access.

3.7 Holdback Ideal Time

As discussed in section 2.2.2, there are system calls that wait for events, which we called holdback events, before completing. We define the number of cycles spent waiting for a holdback event as holdback. If a system call is being blocked, waiting for a holdback event, the number of cycles waited for the holdback event gets added to the application's holdback. Real holdback is defined as the holdback experienced by a real machine, which we can measure. Ideal holdback represents how long an ideal machine would have spent waiting for holdback events. Real and ideal holdback differ by a factor constant, α, where *ideal holdback* = *real holdback* $\times\alpha$. An application's ideal time at a certain point of a program's execution is the sum of the cumulative ideal holdback up to that point and the total ideal execution time derived from the program's autobiography at that point.

3.7.1 Implementing Holdback

We use DynamoRIO's events before and after syscalls to measure holdback of system calls that wait on external events. During those DynamoRIO events, we examine the system call being caught and check if the system call qualifies as a holdback call against a pre-created table. If the system call is indeed a holdback call, we measure how many cycles was spent during the syscall by taking cycles measurements immediately before and after the syscall. If the syscall is waiting for an external holdback event, then those cycles measurements must be real cycles measurements, which will be converted to ideal cycles using α. Otherwise, if the syscall is waiting for an internal holdback event, then we are able to measure ideal holdback directly by making ideal cycles measurements in the syscall events. With these measurements, DynamoREA can determine how many cycles the holdback syscall had to wait before accepting the holdback event it was waiting on.

3.7.2 Applying Holdback

Once DynamoREA computes how much ideal holdback a holdback syscall has, it needs to ensure that the holdback gets applied to ideal time at the correct time. Holdback should be added once the ideal machine executes the holdback syscall, even though we start measuring holdback once the real machine encounters the holdback syscall. As a result, once we compute holdback for a particular holdback syscall, DynamoREA queues this value until the ideal machine surpasses the holdback syscall and only then starts adding holdback to the ideal time.

3.7.3 Observable and Unobservable Holdback Syscalls

There are two types of syscalls that may prompt holdback. *Queuing holdback syscalls* are able to accept queued holdback events that may have occurred before the syscall itself. An external observer may see the external events but will not be able to observe when the holdback events are actually accepted by the holdback syscall. *Non-queuing holdback syscalls* on the other hand only accept holdback events that occur after the call. Holdback events that occur before a non-queuing holdback syscall are rejected and an external observer is able to observe the holdback event failing. By triggering a stream of holdback events and observing when the events stop being rejected, an observer is able to figure out when the non-queuing holdback syscall is executed. Non-queuing holdback calls are thus an observable action and must be executed at the correct ideal time.

For non-queuing holdback syscalls, we need to ensure that we delay to ideal time before executing the system call since non-queuing holdback calls are observable actions. After the syscall is delayed does DynamoREA start measuring holdback.

Since queuing holdback syscalls are not observable, DynamoREA does not need to delay to ideal time before executing queuing holdback syscalls.

3.8 Ideal Machine

We have discussed how DynamoREA calculates ideal time as a function of an application's autobiography and holdback. We also discussed how DynamoREA detects and handles observable actions. In this section, we will define the behavior of the ideal machine we aim to emulate and make the claim that DynamoREA does indeed emulate this ideal machine.

3.8.1 Ideal Machine Model Without External Events

The ideal machine we want to emulate has no micro-architectural artifacts and is internally deterministic. Provided that the application does not interact with external events, the application will execute identically given identical inputs. This means given the same input, the application executes the same series of instructions, $[k_0, k_1, ..., k_n]$ at the same cycle count, $[t_0, t_1, ..., t_n]$ where cycle count refers to the number of cycles executed since the start of the application. We want to show that the ideal machine that DynamoREA emulates has these properties. First we will claim that the ideal machine that DynamoREA emulates executes the same series of instructions every time it runs an application that does not interact with external events. Then we will claim that DynamoREA assigns the same ideal time for each instruction at every application execution.

Claim: On a real machine, given a single threaded application that does not interact with external events, same inputs to this application will result in the same series of instructions, $[k_0, k_1, ..., k_n]$.

If the application does not interact with external events, the ith instruction executed by the application, or k_i, must depend solely on the input to the application and the previous instructions, or $[k_0, k_1, ..., k_{i-1}]$. The first instruction of an application thus depends solely on the input, since there are no previous instructions. As a result, given the same input, separate executions of the application must have the same first instruction. The next instruction now depends on the input and the first instruction. Since up to this point, separate executions would have the same input and first instruction, the next instruction must be the same as well if the application does not interact with external events. We can continue with this inductive reasoning and arrive at the conclusion that given the same input, an application that does not interact with external events executes the same series of instructions.

Though DynamoREA inserts instructions to maintain the autobiography, we make sure to save the state of the program, including arithmetic flags and register content, before executing DynamoREA's own instrumentation and to restore the state afterwards. As a result, the instructions DynamoREA adds to the application has no net effect on the series of application instructions.

Because a real machine executes the same instruction series as an ideal machine would, DynamoREA's emulation of an ideal machine would correctly execute the same series of instructions given the same input to an application that does not interact with external events.

Claim: Given a series of instructions of an application that does not interact with external events, the ideal time associated with each instruction is deterministic.

The ideal time calculated for an application that does not interact with external events is composed of:

- A constant DynamoREA initialization penalty
- A constant penalty for each basic block executed
- A constant penalty for each time a basic block is transformed by DynamoREA
- Ideal cycles contribution of accessing memory
- Ideal cycles contribution of executing each opcode
- Internal holdback

We want to show that the contributions of each of these factors to ideal time is deterministic given a series of instructions.

Initialization penalty. The initialization penalty is always applied at the beginning of the emulation of the ideal machine. It will always be present regardless of what instructions are being executed, so the initialization penalty is deterministic.

Basic block execution penalty. Given a series of instructions, a compiler determines how the instructions should be organized into basic blocks. DynamoREA executes compiled code, so basic blocks have already been organized and thus at any given instruction, the basic block penalty up to that instruction will be the same between separate program executions.

Basic block transformation penalty. In general, DynamoREA will only transform a basic block the first time it sees it. Since the organization of basic blocks is already established, the instructions where DynamoREA needs to apply the basic block transformation penalty are already decided. We do have to be careful, because

DynamoRIO keeps a basic block code cache of basic blocks that have already been transformed and transformed blocks could get evicted from this cache. If a transformed block does get evicted, the next time DynamoREA encounters this block, it needs to transform it once again. However, the size of the code cache and its eviction behavior is deterministic, so the basic block transformation behavior is also deterministic, resulting in the same basic block transformation penalties between separate program executions.

Memory access penalty. The number of ideal cycles spent on a memory access is a function of how many times the application accessed memory since the last observable action. Given a series of instructions, the locations of the observable actions are set and deterministic. For each instruction that accesses memory, the number of memory accesses since the last observable action is also deterministic. If the number of memory accesses since the last observable action remains the same for each memory access across separate program executions, then the ideal cycles contribution of each memory access will also be deterministic. As long as our memory access function is deterministic, the ideal time of memory accesses will also be deterministic.

Opcode penalty. Each opcode is assigned a constant representing how many cycles an ideal machine would take to execute that opcode. Given a series of instructions, the sequence of opcodes and the ideal time of each opcode will be deterministic.

Internal holdback. Holdback is always applied at the instruction where the holdback system call is being executed, thus the place where internal holdback is applied is deterministic. The amount of ideal holdback being applied depends on how many ideal cycles we wait before the holdback event occurs. Since the internal holdback event is not affected by external events, the ideal time between the holdback system call and the holdback event must be deterministic, since all other contributions to ideal time is deterministic as listed above. The instruction where holdback is applied and the amount of holdback applied are deterministic, so internal holdback's contribution to ideal time is deterministic as well.

Since every contribution to ideal time is deterministic given the instruction series of an application that does not interact with external events, the associated ideal time for each instruction will also be deterministic since it is a composition of deterministic values. Thus, as DynamoREA builds its autobiography and calculates internal holdback, the ideal time it calculates will correspond correctly to the number of cycles needed to execute the program on an ideal machine.

3.8.2 Ideal Machine Model With External Events

In the previous section, we discussed how DynamoREA correctly emulates an ideal machine on applications that depend on its own architecture and has no external events. For other applications that do react to the outside world, external holdback may need to be measured and applied correctly. An ideal machine is still suspect to external holdback and the amount of external holdback in our ideal machine model is translated from the external holdback experienced in the real machine. We need to ensure that DynamoREA applies the correct amount of external holdback at the correct instruction.

External holdback can be measured in terms of real cycles on the real machine. Since our ideal machine model's external holdback is derived from the external holdback experienced on the real machine, DynamoREA just needs to be able to convert the amount of external holdback in terms of real cycles to external holdback in terms of ideal cycles. The α constant, introduced in section 2.2.2, translates measurements of real cycles to ideal cycles. Computing α requires knowing the time it takes to run a CPU cycle on a real machine, which DynamoREA can measure during initialization. At initialization, DynamoREA computes α and is able to use it to correctly translate real holdback to ideal holdback for external events.

Like internal holdback syscalls, DynamoREA can keep track of which instruction contains an external holdback syscall and wait until the ideal machine it is modeling reaches that instruction before applying the holdback measured at that instruction. Consequently, the amount and timing of external holdback that DynamoREA applies corresponds correctly to the ideal machine model that we are using.

3.9 Efficiently Handling Timing Events

In the general case, when DynamoREA encounters a timing event such as a system call, DynamoREA must delay until real time catches up to ideal time before executing the timing event. However, there are certain timing events DynamoREA can be clever with and is able to maintain the ideal machine abstraction without the need to delay. In some cases, we can determine the result of a timing event just by inspecting the application's autobiography, allowing us to return this result without actually executing the timing event. In other cases, DynamoREA can recognize that the timing event does not have to be executed immediately and can defer its execution at a later convenient time, such as during a delay for another timing event. These alternate approaches to timing events improve DynamoREA's performance.

3.9.1 `getpid()`

`getpid()` returns the process id of the calling process. This is a case where DynamoREA can inspect what it already knows about the application to immediately produce the result of the system call. DynamoRIO keeps track of process ids internally and whenever `getpid()` is encountered, DynamoREA can use the DynamoRIO API to immediately return the correct process id. `getpid()` never has to be repeatedly executed, as we just return the correct result at once without needing to delay. `gettid()`, a system call that returns the thread id of the calling thread, can be handled similarly without needing to delay.

3.9.2 `RDTSC()`

The `RDTSC()` (Read Time-Stamp Counter) instruction can be found on all modern x86 processors, providing the user the number of executed CPU cycles since the last counter reset by saving the result into registers. `RDTSC()` is an observable timing event, as its result gives an indication of the real time of the application being run. Obtaining this information can violate the ideal machine abstraction since it reveals the actual runtime of the application instead of how it would run on an ideal machine.

Our default method of handling timing events would involve DynamoREA delaying the application until real time catches up to ideal time and then calling `RDTSC()`. Since real time will equal ideal time after the delay, DynamoREA will correctly maintain the ideal machine abstraction by returning the result of `RDTSC()` as if it were being called on an ideal machine.

`RDTSC()`, however, is a timing event that does not require delaying to maintain the ideal machine abstraction. The only requirement for `RDTSC()` to maintain the ideal machine abstraction is for it to behave as if it were being called on an ideal machine. While delaying until real time equals ideal time achieves this requirement, we can take advantage of the fact that we can use DynamoRIO to catch `RDTSC()` calls and are able to calculate ideal time at any instruction. We can compute the ideal time of the application at the `RDTSC()` call and alter its output according to this value, emulating the `RDTSC()` call on an ideal machine. From here, `RDTSC()` behaved correctly according to the ideal machine abstraction and the application can continue its execution without the need to delay.

```
rdtsc_handler(rdtsc_call):
  emulated_rdtsc_output = actual_rdtsc_output + ideal_time(autobio) - \
      get_real_time()
  set_rdtsc_output(rdtsc_call, emulated_rdtsc_output)
```

Figure 3-5: RDTSC pseudocode

3.9.3 sendto()

`sendto()` takes in a message and sends the message on a socket. Since outgoing packets are observable, this is certainly an observable timing event that we need to be careful with. If our default handler sees a `sendto()`, it would again delay until the application's real time is equal to its ideal time, at which point the `sendto()` would be executed. However, the POSIX specs on `sendto()` states that "successful completion of a call to `sendto()` does not guarantee delivery of the message". This condition gives DynamoREA flexibility on when `sendto()` actually needs to be executed.

DynamoREA can gain efficiency if we choose to pretend that `sendto()` completes successfully right when we see the call. In reality, we will actually defer `sendto()` to another process whose responsibility is to make sure the `sendto()` that we skipped over do actually get called at an appropriate time. We take advantage on the relaxed specification when `sendto()` needs to actually deliver its message to get around needing to delay.

```
queue Q

sendto_handler(sendto_call):
  Q.enqueue(sendto_call.parameters, ideal_time)
  skip_sendto_call

sendto_loop()
  while(true):
    if Q is not empty:
      [sendto_parameters, ideal_time] = Q.dequeue()
      delay_time = convert_cycles_to_time(ideal_time) - get_time()
      if delay_time > 0:
       delay(delay_time)
       sendto(sendto_parameters)
      else:
          catch_error
```

Figure 3-6: sendto pseudocode

3.9.4 `wait()`

One use of the `wait()` system call is to block a parent process from continuing until one of its children processes has completed its execution. Similar to `RDTSC()`, this is a call whose outcome can be predetermined by DynamoREA's knowledge of the application autobiography. In addition to keeping its own autobiography, a parent process also has access to the autobiography of any of its child processes. When a parent process sees a `wait()`, instead of delaying until it is an appropriate time to execute the wait system call, the parent process can deduce the outcome of the `wait()` based on the children's autobiographies.

If the ideal time of a parent process is i_p and the ideal time of its child process is i_c at the `wait()`, we can work out all the cases that DynamoREA might encounter when approaching a `wait()`. If we detect that the child process is dead and the child's ideal time is less than the parent's ideal time, then we can conclude that in an ideal machine, the child process would have exited before the parent process reaches its `wait()`. In this case, the parent can choose to skip over the `wait()`. If we detect that the child process is dead, but the child's ideal time is greater than the parent's ideal time, the parent just needs to delay until its ideal time matches the child's ideal time and then skip over the `wait()`. Note that in this case, though we delay, we are still able to skip over the `wait()` and avoid needing to delay to the parent's ideal time. In the case that we detect that the child process is still alive, then we have no choice but to defer to the default system call handler, meaning that we will need to delay to the parent's ideal time and then execute the `wait()`.

Some calls to wait specify a groupid and only continue when a thread with the specified groupid has completed its execution. Unfortunately, because a thread may modify its own groupid without notifying its parent, DynamoREA is unable to keep track of thread groupids reliably and thus is unable to make the same optimization on these types of wait calls.

The `wait()` is also subject to the determinism deadlock resolution mechanism shown that was applied to `read()` and `write()` before. The wait handler first determines if the call needs to be made. If it decides that the `wait()` does indeed need to be executed, DynamoREA then must ensure that the deterministic logical time of the thread calling `wait()` is the least of all threads. To resolve potential deadlocks, DynamoREA may need to transform the `wait()` into a non-blocking `wait()` and loop it to help progress all the threads.

In addition, `wait()` is a holdback syscall. If the child it is waiting on reacts externally to the system, then it is an external holdback syscall. Otherwise, `wait()` is an internal holdback syscall. DynamoREA would also need to properly measure the holdback for the `wait()`. This holdback measurement and determinism mechanism from earlier is abstracted away into the execute_wait_call function in figure 3-7.

```
wait_handler(wait_call):
  if waiting_for_child_to_exit:
    for child in children:
      if child.exited:
        if child.ideal_time < ideal_time:
          skip_wait_call
        else:
          if child.ideal_time < ideal_time:
            delay_time = convert_cycles_to_time(child.ideal_time) - \
               get_time()
            if delay_time > 0:
              delay(delay_time)
              skip_wait_call
            else:
              catch_error
          else:
            delay_time = convert_cycles_to_time(ideal_time) - \
               get_time()
            if delay_time > 0:
              delay(delay_time)
              execute_wait_call
            else:
              catch_error
    if no_children_exited:
      execute_wait_call
```

Figure 3-7: wait pseudocode

3.9.5 Optimizing Ideal Cycles

To attain strong security and maintain the abstraction of an ideal machine, ideal time must not ever get ahead of real time at observable moments. Yet, to attain strong performance, it is important that ideal time does not lag behind real time by too much. The challenge here is to find an ideal time function that provides both strong performance and security for all applications.

We find that we can optimize the memory accesses portion of the ideal time function. Like before, we want a memory delay function that takes in the number of memory instructions since the last checkpoint, n_{mem}, and outputs an upper bound on the number of cycles spent on the memory accessing portion of these instructions.

Before, we had a stepwise function that assumed cache misses for the first $n_{threshold}$ memory accesses and average cache behavior for the rest of the memory accesses until the next checkpoint. Our improved function should have a derivative of ∞ as n_{mem} goes to 0, as we make the assumption that initial memory accesses may take a long time. As n_{mem} goes to ∞, we assume that the cache has warmed up and the derivative should stabilize at some constant C, representing the average time to make a memory access on a warm cache. The derivative of this function goes from infinity to C but it should do so gradually, allowing plenty of leeway for the cache to warm up to be safe. A function that matches this behavior is $f(n_{mem}) = A(\log[n_{mem}+1]) + Cn_{mem}$, where A and C are parameters that define the exact shape of the memory delay curve.

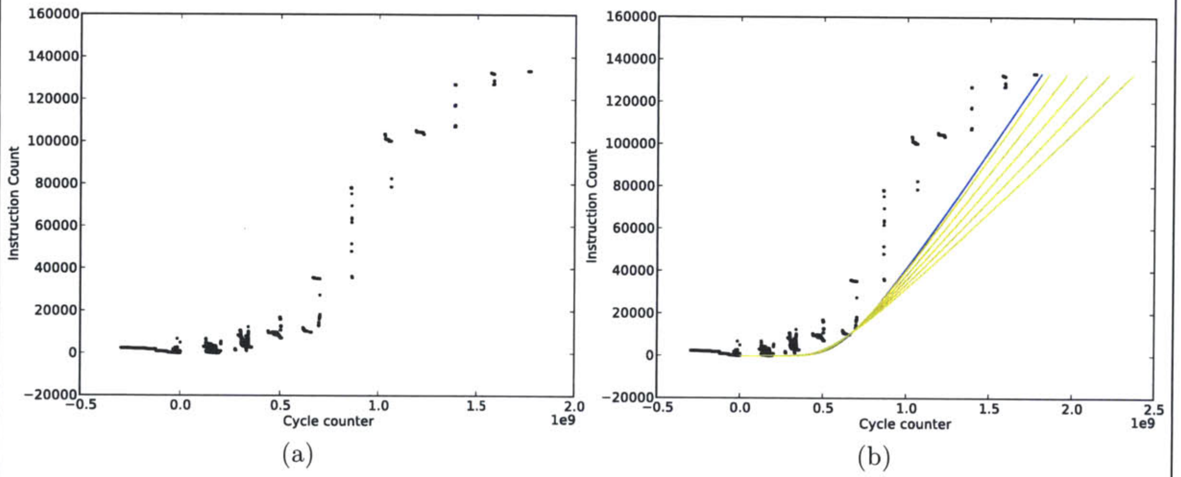

Figure 3-8: (a) Scatter plot of memory delay curves of an `ls` call (b) Iterations of finding the best fit curve. The best fit curve is colored in blue.

Different applications have different memory access patterns, so we would not be able to fit the ideal time curve tightly if we use the same parameters for the memory delay curve for all applications. Instead we wrote a profiler to generate the best parameters for each particular application. The profiler traces application execution over many repetitions, gathering many real time curve segments from checkpoint to checkpoint. Once we have all the data of how many real cycles and instructions has been executed since the last checkpoint at all times, we need to strip the real time contributions from basic block transformation, holdback, and weighted opcounts away. By subtracting these non-memory penalties from our real time curves, we get a set of curves that represents the relationships between memory access cycles and instructions since the last checkpoint. The last step is to fit our memory access function onto these set of curves.

The profiler searches for the best values of A and C that fits the memory delay curve by trying values until they converge towards a solution that minimizes an error heuristic. Once the profiler determines values for A, and C, it stores these values in file, assigning them to the application being profiled. When the profiled application is run under DynamoREA, its associated parameters are loaded during initialization and its particular unique memory delay function is used when calculating ideal time.

Note that since this new memory access function is still deterministic in terms of number of memory accesses since the last observable action, using this memory access function will still result in deterministic contribution to ideal time and maintains the correctness of our ideal machine model.

Chapter 4

Evaluation

4.1 Security

4.1.1 Security Invariant

For an application to be secure, its execution must not violate the ideal machine abstraction. To an external observer, the application appears to be running on an ideal machine with no micro-architectural effects. In other words, every observable action of an application must occur at the same time at every execution, no matter what the state of the computer is. At every observable action, the real time must not exceed the ideal time so that it is possible to delay our program so that the observable action is executed at the correct ideal time.

Another aspect of maintaining the ideal machine abstraction is determinism. Internal non-determinism is a micro-architectural effect so an ideal machine is completely absent of internal non-determinism. The variances resulting from non-determinism can leak secret information, thus DynamoREA must ensure that multithreaded applications are executed deterministically.

4.1.2 Ideal Machine Abstraction

This section presents graph representations of how various applications executed on DynamoREA. Applications vary from test applications written to highlight certain behaviors we wanted to evaluate to standard applications that are commonly found. Each graph presents three lines representing instructions executed to cycle count. The red line represents ideal time, the progression of the application as if it were executed on an ideal machine. The blue line represents real time, the actual progression of the application on the real machine. The yellow line represents compute time, an approximation of what the progression of the application would look like if we did not delay at every observable action.

In each of these graphs, the key attribute we want to be looking for is making sure that the ideal time and observed time are equal when observable actions are executed. In the common case, this means that we can see the blue line delay at the instruction count where an observable action is being executed until it "catches up" to the red line.

mem-arith

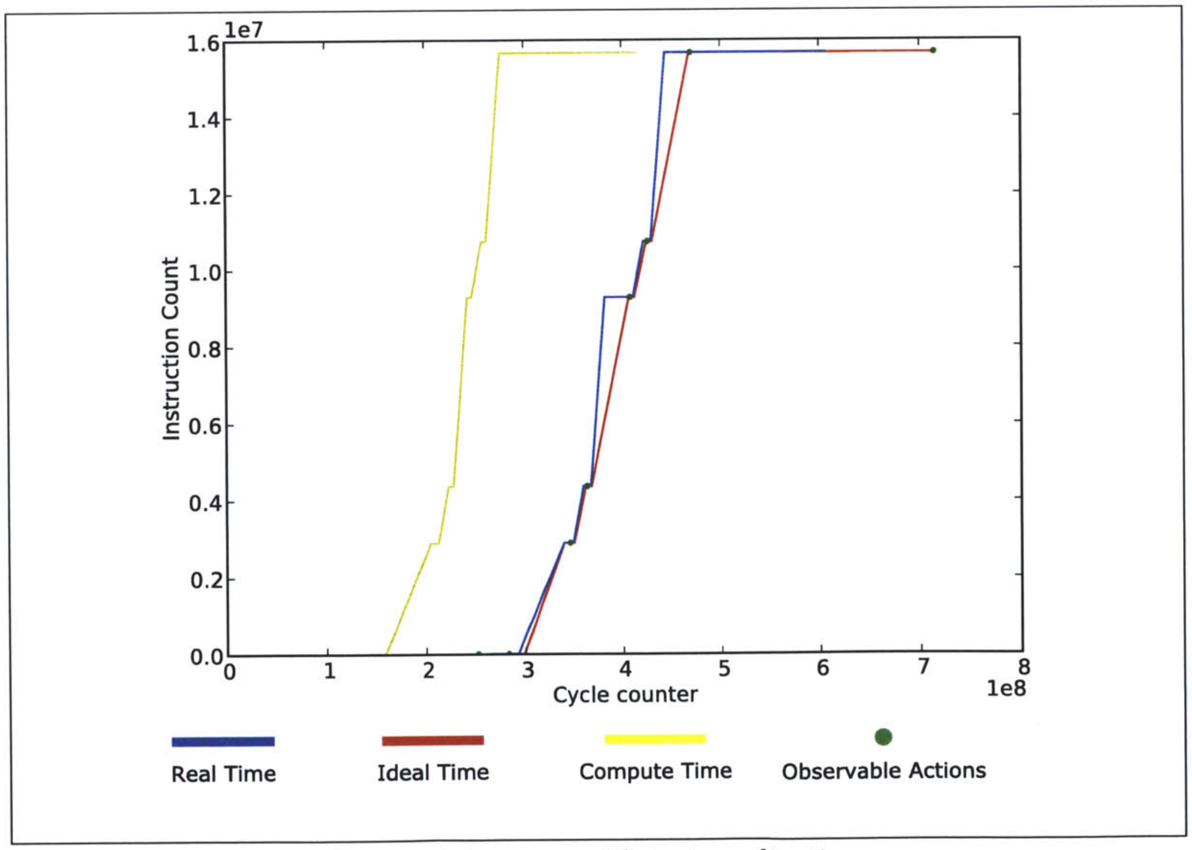

Figure 4-1: mem-arith test application

The mem-arith test application first allocates 4 MB for a character array and then initializes all the values of the array to bring the array into the cache. The application then makes four loops; the first and third loops access the elements of the array with a 64 byte stride and the second and fourth loop executes a series of arithmetic operations. After initializing the array and each loop, a write call is made to represent an observable action. Here we can see the flat regions in the real time line, representing the times where an observable action is being made and DynamoREA is delaying so that the observable action is executed at the corresponding ideal time.

aes

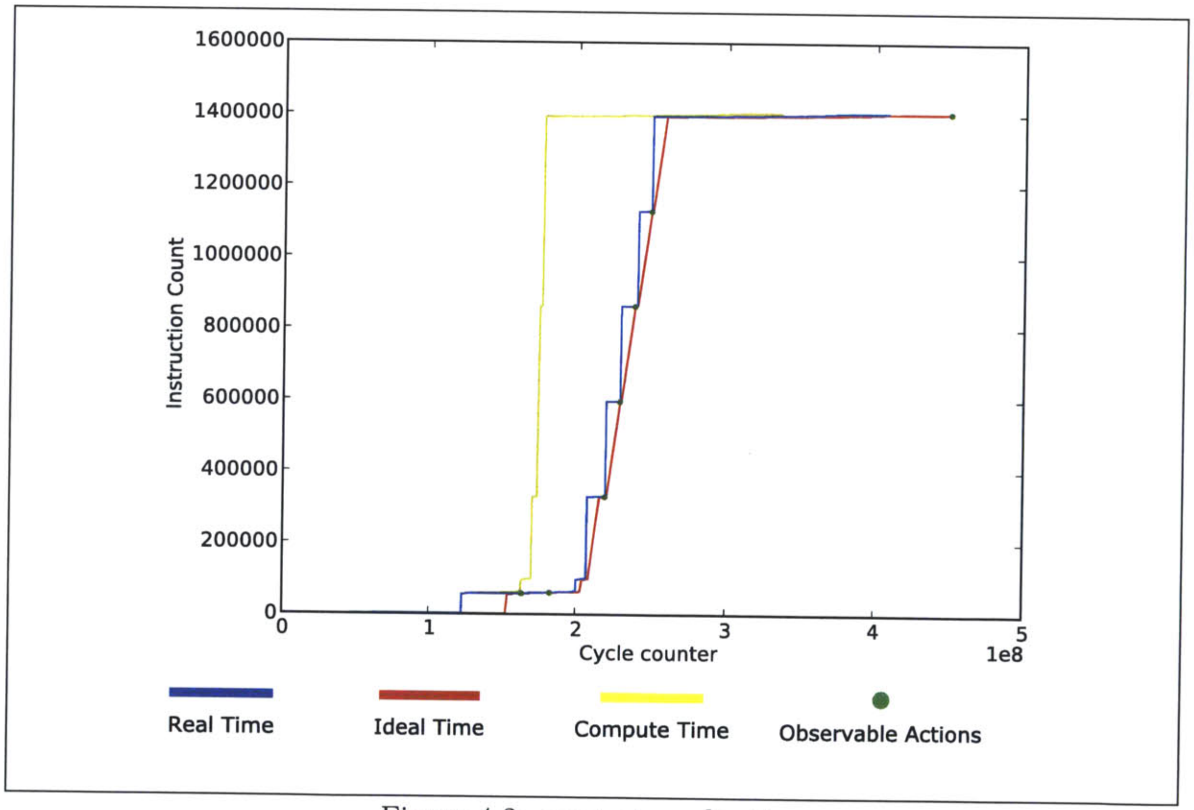

Figure 4-2: aes test application

The aes test application first initializes a 6 KB message, generates an encryption key, and then executes an observable action. The application then executes a series of five encryptions using the generated key and message, executing an observable action between each encryption.

wget

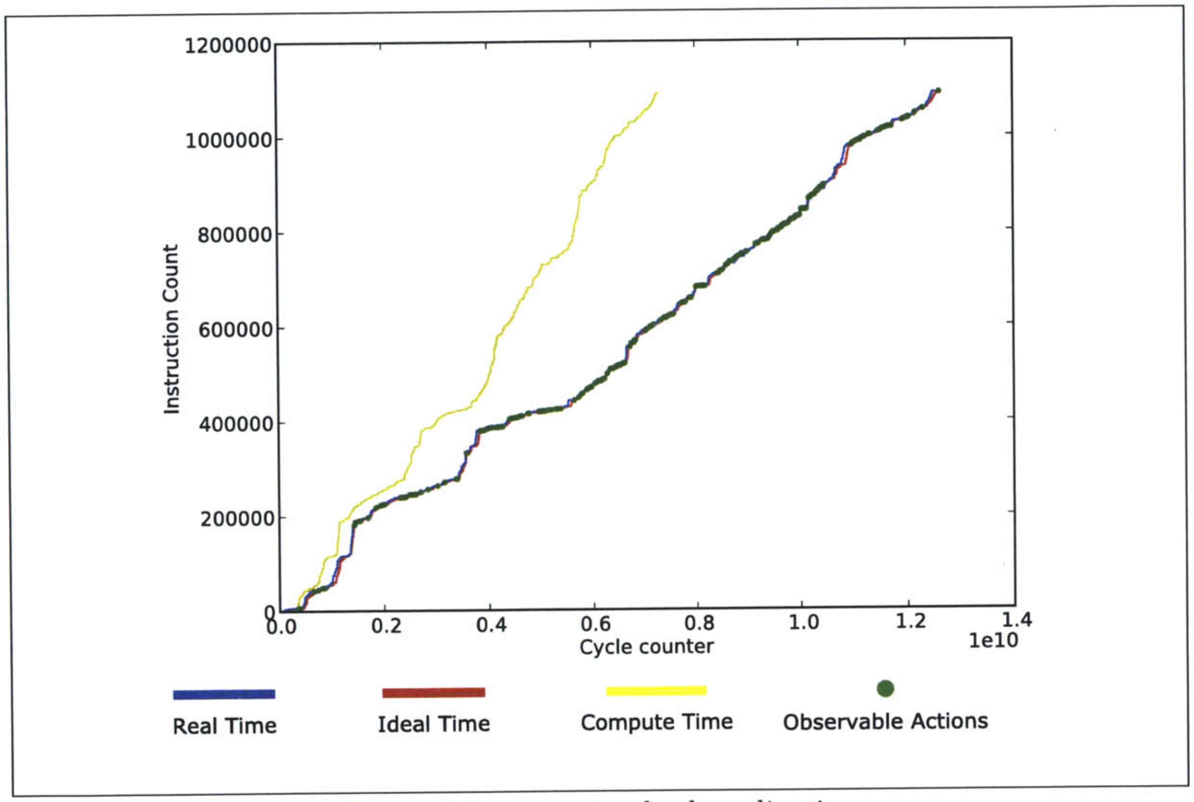

Figure 4-3: wget standard application

wget is a standard application that retrieves content from the web. This graph represents the behavior of wget on DynamoREA while fetching a single page from the web.

rsa

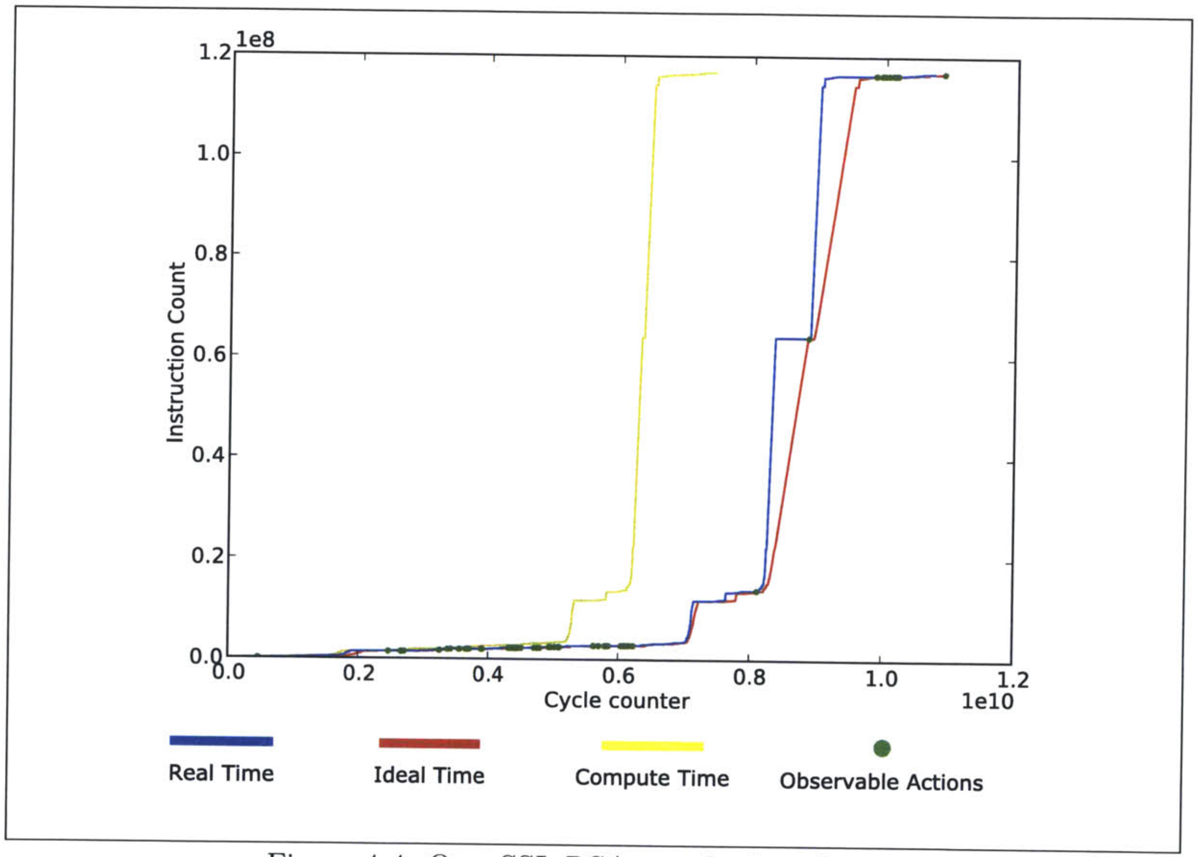

Figure 4-4: OpenSSL RSA standard application

This graph represents an RSA decryption using a 4096-bit private key, using the OpenSSL `rsautl` binary.

cache

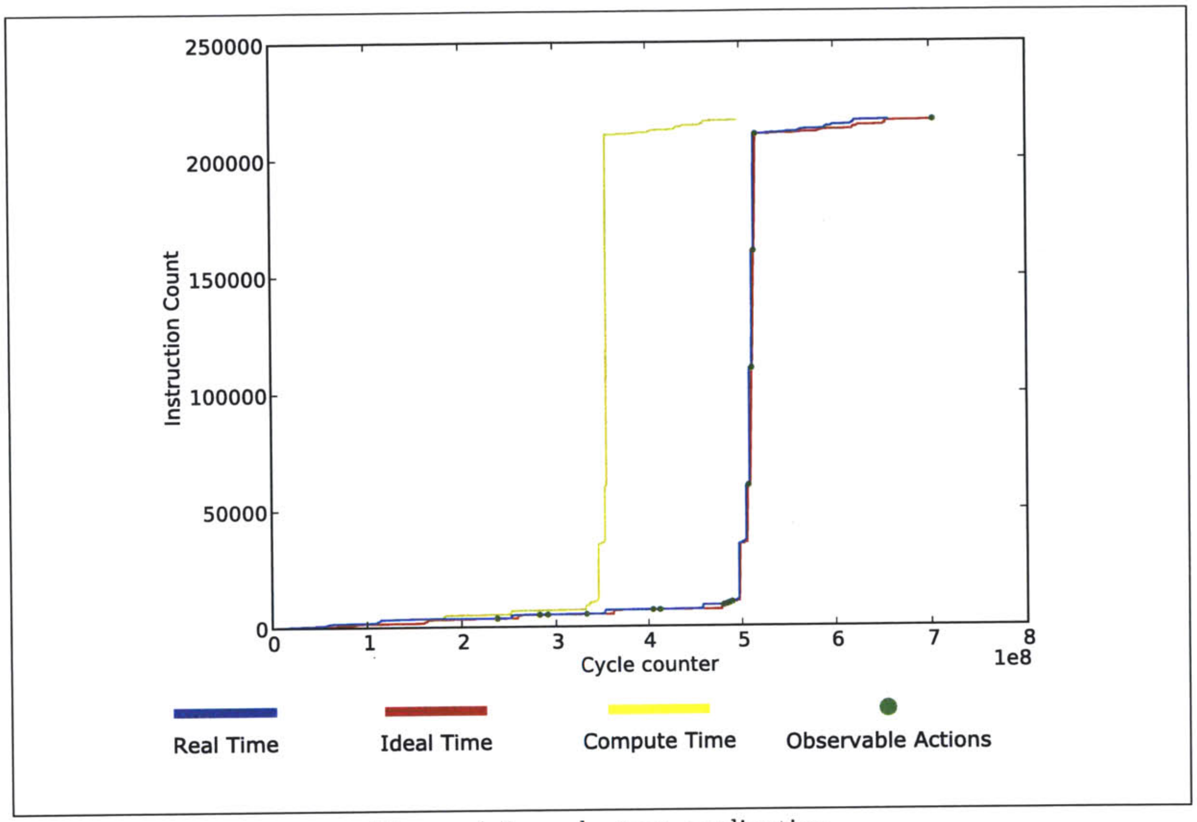

Figure 4-5: cache test application

The cache test application initializes two arrays of characters: a "small" 1 KB array and a "large" 12 MB array. The application then probes the two arrays by accessing every element in the arrays in a pseudo-random order. A round consists of randomly completely probing the small array 25000 times and randomly completely probing the large array 5 times. The application executes an observable action in between each round.

4.1.3 Determinism

01 Test

In the first variant of the 01 test, a process spawns two threads with one thread repeatedly printing '1' and the other thread repeatedly printing '0'. When executed without DynamoREA, the resulting print contains 1s and 0s listed in a random order.

When executed with DynamoREA and its determinism enforcing mechanism, the resulting print contains 1s and 0s alternating perfectly.

The second variant of the 01 test involves a process that forks, resulting in two processes where one repeatedly prints '1' and the other repeatedly prints '0'. The same result is achieved as above: the results are non-deterministic when executing the test off of DynamoREA but deterministic when executing with DynamoREA.

`racey` Results

`racey`[10] is a stress test for deterministic execution that produces a signature that is very sensitive to the order of unsynchronized data races. `racey` takes in a parameter specifying how many threads should be spawned. Executing the `racey` test off of DynamoREA with two interleaving threads 500 times produced 456 different signatures. When running on DynamoREA however, `racey` with two threads produces the same signature in each of the 500 executions. However, DynamoREA was unable to produce a consistent signature with `racey` with more than two threads.

4.1.4 Shared Memory

mtaddmult

In the mtaddmult test, shared memory is allocated to contain an integer, which is then manipulated by two threads. One thread executes multiplication operations on the integer and the other thread executes addition operations on the integer. Shared memory accesses must be deterministic to preserve security. When executed without DynamoREA, the resulting integer after all the operations are executed varies from execution to execution. When executed with DynamoREA and using the shared memory detection provided by Umbra, the resulting integer is the same across all runs, demonstrating that shared memory accesses are deterministic under DynamoREA.

The graph also demonstrates an example of the ideal machine abstraction being preserved with multiple threads. The parent thread spawns the child threads at the correct ideal time and in each of the child threads, real time does not surpass ideal time at observable actions.

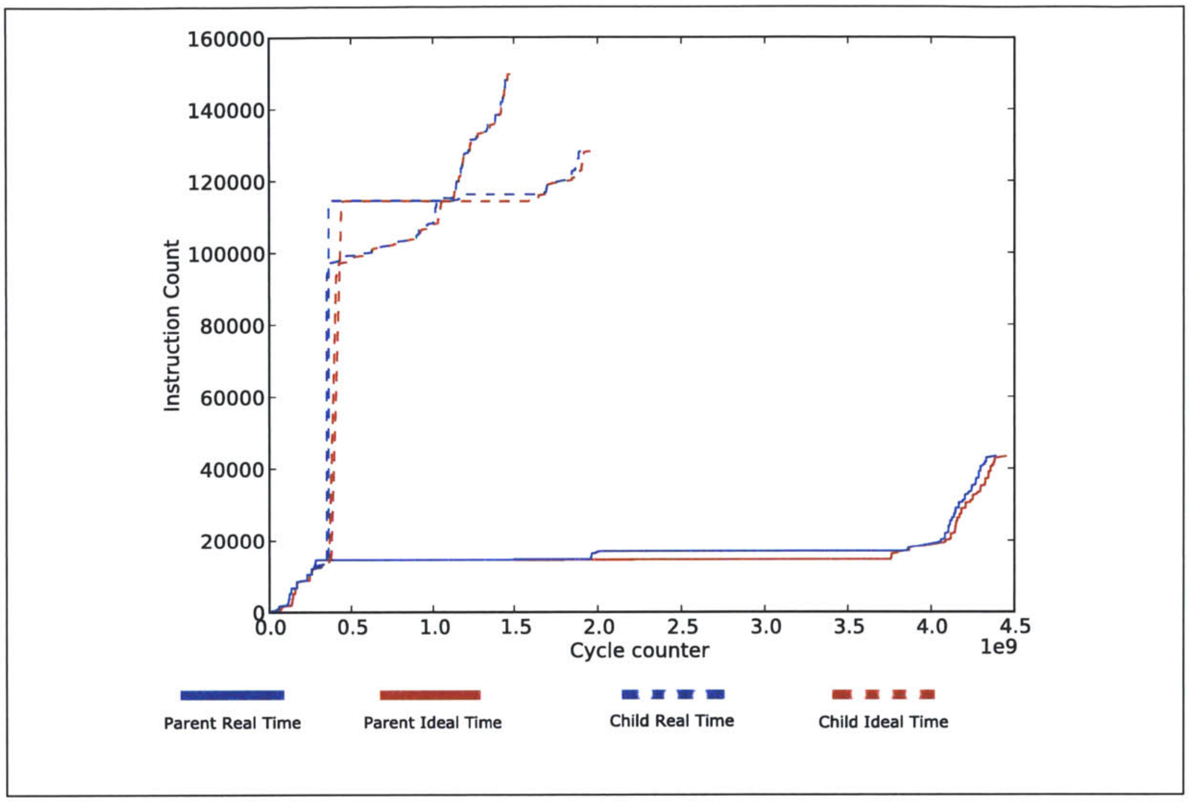

Figure 4-6: mtaddmult test application. The parent thread's real and ideal times are shown as filled lines. The child threads' real and ideal times are shown as dashed lines.

Chapter 5

Conclusion

With DynamoREA, we have shown that it is possible and practical to use dynamic binary rewriting as a tool to defend against microarchitectural-based timing attacks. This is done by using DynamoRIO to emulate an ideal machine. Applications' timing behaviors are transformed so that to an observer, it looks like they are being executed on a machine with no microarchitectural side effects. We demonstrated that observable actions of applications running on DynamoREA have deterministic timing behavior, meaning that observers will not be able to extract any useful timing information from repeatedly observing application timing behavior. In addition, applications under DynamoREA have demonstrated determinism across multiple threads and processes, meaning attackers will not be able to observe artifacts caused by non-determinism to gather information about an application's secret. Though performance has room to improve, we have laid out some outlines and examples on how DynamoREA can find shortcuts on handling certain types of observable actions that make performance gains.

Bibliography

[1] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`.

[2] J. Agat. Transforming out timing leaks. In *Proceedings 27th Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.

[3] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, MIT, 2004.

[4] David Brumley and Dan Boneh. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[5] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. *DMP: Deterministic Shared Memory Multiprocessing*, 2009.

[6] Chi-Keung Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.*, 2005.

[7] T. Bergan et al. Coredet: A compiler and runtime system for deterministic multithreaded execution. *15th ASP-LOS*, Mar. 2010.

[8] T. Bergan et al. Deterministic process groups in dos. *9th OSDI*, Oct. 2010.

[9] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `http://www.agner.org/optimize/`, 2011.

[10] Mark D Hill and Min Xu. Racey: A stress test for deterministic execution. `http://pages.cs.wisc.edu/~markhill/racey.html`.

[11] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE Computer Society Symposium in Security and Privacy*, pages 8–20. IEEE, 1991.

[12] Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*, pages 52–61. IEEE, 1992.

[13] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[14] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[15] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[16] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds, 2009.

[17] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology, available online*, 2009.

[18] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, 2003. IEEE Comp. Soc. Press.

[19] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *International Symposium on Code Generation and Optimization (CGO '10)*, April 2010.