2008

Architectural support for secure and survivable embedded software

Jesse Sathre
Iowa State University

Architectural support for secure and survivable embedded software

by

Jesse Sathre

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Zhao Zhang
Thomas Daniels

Iowa State University

Ames, Iowa

2008

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

First, I would like to thank my adviser Dr. Joseph Zambreno for funding me during my stay at Iowa State, and keeping me motivated and busy. I would also like to thank the rest my committee members, Dr. Zhao Zhang and Dr. Thomas Daniels. I want to express thanks to Dr. Akhilesh Tyagi for convincing me to attend Iowa State University. Finally, I would like to thank my friends and family for showing me support in all of my decisions.

ABSTRACT

Attacks against vulnerable software have become a serious problem for industry and end-users alike. There have been many techniques proposed to combat these attacks which range from compiler modifications to additional architectural features. Most of these techniques focus on attack detection, while ignoring the problem of how to gracefully recover from such attacks. In this thesis we propose an architectural approach to attack detection *and recovery* which we call *rollback and huddle*. In our approach, a lightweight attack-detection module monitors a program's execution as its state is continuously checkpointed. In the case of an attack, the program state is rolled back to a time before the attack occurred, and an additional HW/SW module is loaded to gain extra insight into the attack and possibly repair the original vulnerability. Our approach is based on the observation that the vast majority of a program's execution can be trusted. Therefore, we aim to minimize the performance overhead during "normal" execution. Once an attack has been detected, the system is put into a "high alert" mode where a larger performance overhead is tolerated to make use of more complex techniques and avoid system down-time. We introduce simple hardware modules that work alongside a standard computer architecture, and aid in attack detection, checkpoint creation, and attack recovery. Our experimental results show that this approach can be achieved with minimal run-time overhead and resource utilization.

CHAPTER 1. Introduction

One of the key problems facing the computer industry today is ensuring the integrity of end-user executables and data. Most programs are written in low-level languages that allow developers to write efficient code. However, this efficiency often comes at the cost of security features commonly found in high-level languages such as bounds checking on arrays, type checking of pointers, and protection of individual data elements. These language shortcomings can be compensated for at the application level with proper programming techniques, but many times they are neglected – both accidentally through programming errors and deliberately to produce more efficient code. Software patches can fix specific vulnerabilities, but they act retroactively after a vulnerability has been found and potentially exploited.

Recent research has introduced several compiler and architectural approaches which are aimed at detecting general classes of attacks against vulnerable software. One issue is that many of these current software protection approaches consider the detection stage as the logical endpoint of their scheme, trapping an attack and then ceding control to a supervising process. In some cases this is the preferred course of action. However, program termination can effectively be viewed as a successful Denial-of-Service (DoS) attack. If the exploited vulnerability were in a high-value application (a Web server for an e-commerce site, for instance), an application restart required by any failed attack could lead to a loss of revenue. In summary, we see a need for a software protection approach that can handle a general class of attacks while also continuing to execute gracefully once an attack is detected.

In this thesis we introduce a novel approach to software security which we call *rollback and huddle*. Figure 1.1 shows a conceptual view of our approach. We make use of software rollback to achieve a graceful recovery of compromised programs. Software rollback is not a new idea,

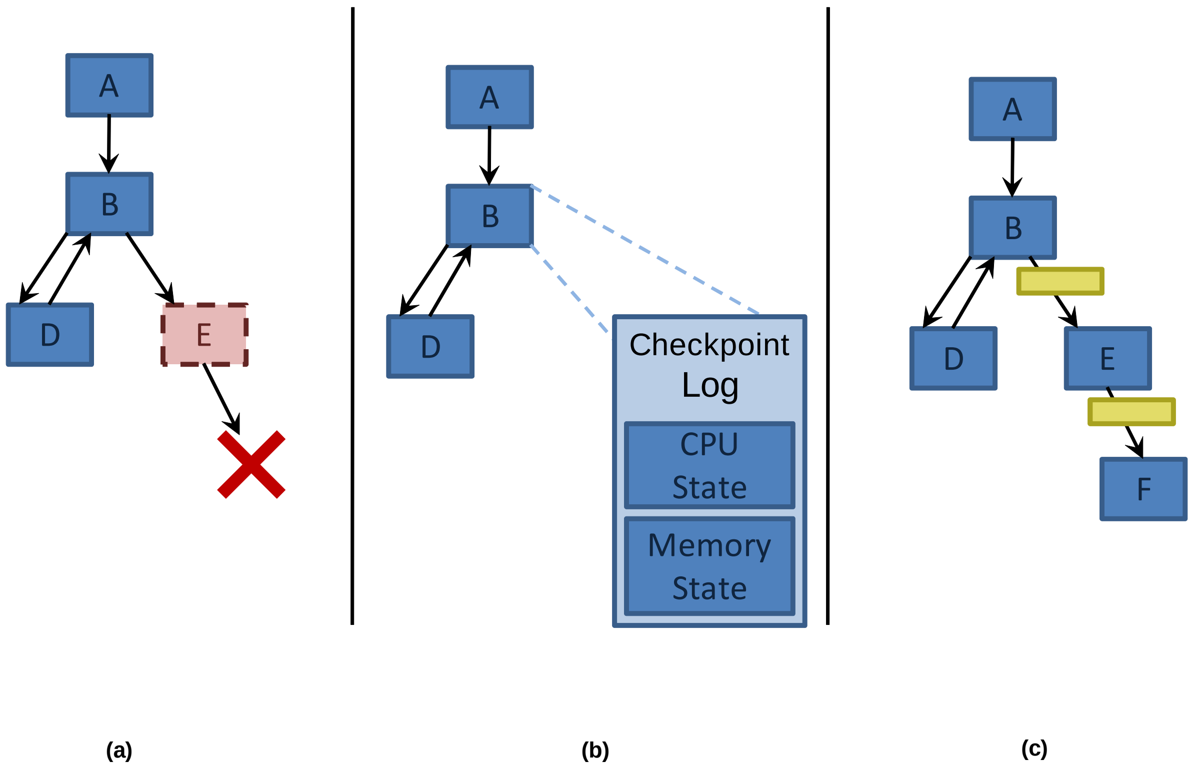

Figure 1.1 (a) An anomaly is detected in function E. Typical protection schemes cause the program to halt execution. (b) In our approach, the program is rolled back to a safe state by utilizing checkpoint logs. (c) The program restarts execution with extra safety checks in place.

but applying it to the domain of software security is a new application of the concept. In our approach, a lightweight security mechanism continuously operates with minimal performance overhead. If this initial scheme detects that an attack has occurred, the program is "rolled back" to a previous point in time. This rollback operation is made possible through the recording of periodic checkpoints during execution that allow a program's state to be recovered. After rollback, the vulnerable portion of the program is further instrumented with stricter security policies which may monitor control flow and memory accesses at a finer granularity. Once identified as exploitable, a section of instructions could in theory be patched in real-time. After these new policies are in place execution resumes from the state of the safe checkpoint.

As will be explained in Chapter 4, we introduce some non-intrusive architectural features to support our proposed approach. A Hardware Checkpoint Unit (HCU) snoops off-chip memory accesses in order to log checkpoints and perform rollback operations. Initial continuous security monitoring is accomplished through the use of a Lightweight Protection Unit (LPU), which enforces a basic security policy. A Heavyweight Protection Unit (HPU) is placed inside the memory fetch path, but acts only as a pass-through until a more secured mode of execution is requested after a rollback. Our architectural simulation results show that the lightweight monitoring and continuous checkpointing add an average of less than 10% performance overhead to a variety of benchmarks.

The remainder of this thesis is organized as follows. In Chapter 2 we provide an overview of related research in the fields of hardware-supported checkpointing and software protection. Chapter 3 introduces the concept of *rollback and huddle.* In Chapter 4 we provide an example implementation of our architectural features, and in Chapter 5 we present our benchmark results for the example architecture. In Chapter 6 we perform analysis using real software vulnerabilities and test the rollback capabilities of our approach. Finally, the thesis is concluded in Chapter 7 with a discussion of what was accomplished with this project, as well as an exploration into possible future work related to this research.

CHAPTER 2. Related Work

Work related to our approach falls into two distinct categories. The first category consists of attack detection schemes in the security domain. The second category is checkpointing and rollback which follows work in the fields of fault tolerance and software debugging.

2.1 Attack Detection

The computing research literature is filled with various approaches to detecting and preventing software-level attacks. One aspect that makes our work unique is that we focus on system recovery after the initial point of detection.

Data Encryption: Several approaches focus on providing architectural support for encrypted execution and storage. In [24], the authors introduce the concept of eXecute-Only Memory, or XOM, which provides a mechanism for cryptographic separation of instruction and data-memory space. Yang et al. [48] present a more efficient implementation of XOM by moving the encryption process off of the critical path. In [52] Zhuang et al. make optimizations to memory encryption techniques at the compiler level by grouping secure memory transactions so that encryption takes place over a block of data. The approach in [35] uses speculation to decrease the latency of memory encryption. PointGuard [11] is a compiler-based approach that encrypts all pointers in memory. Tuck et al. [44] improve upon PointGuard by providing hardware support which reduces overhead and extends its security features by increasing the complexity of the encryption algorithm.

Hardware Stack Protection: Due to the prevalence of stack-related software attacks, one common theme found across a variety of approaches is the use of a secondary stack to enforce a security policy. An early implementation of a return address stack can be found in [25]. One notable example is SmashGuard [30] which adds hardware functionality to intercept function calls and returns in order to manage its own hardware stack. Modern speculative processors contain their own Return Address Stack (RAS) which is used to predict return addresses. In [31], the RAS is modified to enforce a security policy which results in less than a 1% performance overhead. StackGhost [17] takes advantage of the sliding register window of the SPARC architecture to protect the stack. Although using a secondary stack for return addresses detects many common attacks, it does not detect all forms of buffer overflow attacks as is shown in [46]. However, its low overhead and modest hardware requirements make hardware stack protection an ideal candidate for a lightweight protection scheme in our approach.

Software Stack Protection: Software techniques to protect the stack exist as well. While these techniques can be effective at preventing attacks, they often have prohibitively high overhead. StackGuard [12] is an approach that detects stack-based attacks by writing a "canary value" on the stack after every return address. Any changes made to the canary value indicate that an attack has taken place. StackGuard does not protect against all forms attack, and its protection was shown to be easily bypassed in [8]. CFI [1] expands on StackGuard by checking the integrity of all control flow branches. CFI was proved to be effective against all buffer overflow attacks found in [46]. CFI's additional security comes at the cost of additional execution overhead (45% in the worst case).

Data Tracking and Program Flow Enforcement: Many software attacks originate in data from spurious sources which disrupt program flow. Minos [13] is an architectural approach that tags data with an integrity bit. Input from untrusted sources are tagged as such and not allowed to modify the program flow. A similar approach is presented in [40] where the OS is modified to tag I/O from spurious sources. These approaches require non-trivial modifications to existing hardware in order to add tag bits to all system storage. Program Shepherding [23] dynamically instruments programs to enforce security policies which preserve the intended flow of the executing program. While Program Shepherding does not require the additional tag bits of [13] and [40], it must be run on a specialized platform that is capable of instrumenting program binary. A related family of approaches exist that rely on static analysis to enforce program flow [16, 45, 19, 34]. The scheme introduced in [50] uses additional hardware to enforce a similar security policy at a finer granularity. While our approach shares little in common with the execution of these schemes, we follow a similar security model of enforcing program flow.

Physically Secure Architectures: While our approach focuses on guarding against software-based attacks, a number of techniques exist that aim to protect against an adversary who has physical access to a machine. Side-channel attacks [22] are one example of a physical attack. In a side-channel attack, the adversary attempts to learn about a program's behavior by observing the physical timing of a chip [33, 7] and its microarchitectural features [2, 3]. The AEGIS secure coprocessor [41] provides an implementation of physical random functions that can be used for assigning unique keys to an individual processor. The IBM 4758 secure coprocessor [15] is an earlier example of an architectural approach to software security. An adversary with physical access to the memory bus of a system can attempt to reverse engineer the program by observing memory access patterns. Goldreich et al. [20] introduce the concept of *oblivious memory*. Oblivious memory hides the actual pattern of memory accesses by accessing memory in the same repeated order regardless of which addresses are actually needed by the program. HIDE [51] is another approach that attempts to obfuscate the addresses that are sent over the memory bus. A number of approaches exist[6, 18] that use Merkle hash trees [26] to check that the contents of memory have not been physically tampered by an adversary.

2.2 Checkpointing and Rollback

Attack recovery is made possible in our scheme by checkpointing execution and rolling back to a previous state. Once execution is logged, it can be replayed off-line for the purposes of debugging or rolled back and re-executed to recover from errors. While the concept of checkpointing and rollback began in the domains of fault tolerance and debugging [4], it has began receiving more attention in the domain of software security and attack recovery.

Fault Tolerance and Software Debugging: Our checkpointing scheme is loosely based on FDR [47] which itself can be traced back to SafetyNet in [39]. SafetyNet was designed to handle faults in shared-memory multiprocessor systems. It contains hardware checkpoint log buffers in the processor's cache as well as main memory to log writes to each location. FDR expands the logging found in SafetyNet to include system I/O, DMA transfers, and memory races. It also adds LZ77 hardware to compress the logs.

Another scheme based on [47] is BugNet [27]. Program execution is logged so that it can be deterministically replayed off-line once a bug in encountered. It is unique because it logs memory reads as opposed memory writes. BugNet is similar to our checkpointing scheme because it is not a full-system recorder, but logs programs at the application level.

ReVive [32] is an approach to multi-processor fault tolerance. Processor state and memory writes are logged similar to other approaches [47, 39]. In addition to the logging aspect, memory is supplemented with parity information to recover from a loss of data. Revive works with off-the-shelf components and only requires a modified cache-coherence directory controller. An FPGA-based approach is presented in [43]. Similar to our approach, once the buggy application has been rolled back, an attempt is made to instrument the program's binary so that the bug does not occur again. SWITCH [42] is an approach that builds on the work presented in [43], and puts more emphasis on the checkpointing features of the architecture. Sidiroglou et al. [37] also present a similar scheme that uses a speculative state of execution to recover from faults.

Security and Attack Recovery: ExecRecorder [14] is a Virtual Machine-based checkpointing scheme which shares some similarities with our approach. ExecRecorder is a log-based recovery mechanism designed to work with an intrusion-detection system and post-attack analysis tools. Like our approach, replay is signaled by an attack-detection mechanism. However, our approach differs on a number of key points. First, ExecRecorder operates at the Virtual Machine layer, whereas our approach works with the native architecture. The other key difference is that ExecRecorder performs off-line replaying in order to analyze the source of an attack. Our approach performs an on-line rollback without terminating program execution.

DIRA [38] is an approach that is similar to our idea of heavyweight protection. In DIRA, execution is checkpointed and rolled back upon attack detection. It attempts to identify the source of the attack and repair itself in certain situations. DIRA suffers from substantial run-time overhead in many benchmarks. In our approach, we avoid most of this additional overhead by implementing the additional identification and repair mechanisms after an attack has been detected. Another example of attack recovery can be found in [36].

CHAPTER 3. Rollback and Huddle

Our approach features multiple execution phases in order to achieve the goal of attack detection and recovery. The first phase is *lightweight monitoring* which utilizes a low-overhead, relatively simple attack detection mechanism designed to detect many common forms of attacks. The second phase is the *continuous checkpointing* that occurs as a program executes. These checkpoints act as system "snapshots" for a given instance of time. The third phase is *rollback*. Rollback occurs after an attack has been detected. In this stage, the executing program is restored to a point in time before the attack took place. The final phase is *heavyweight monitoring*. Simply rolling back to a previous point and restarting execution would not be sufficient to overcome an attacker who is aware of the checkpointing and rollback mechanism. To prevent a replay of the original attack, the application is instrumented to enforce a more robust security policy.

3.1 Lightweight Monitoring

The first phase of our approach is lightweight monitoring of the executing program. The purpose of lightweight monitoring is to detect most attacks while minimizing the run-time performance overhead. This is accomplished through the use of a low-overhead attack-detection mechanism. Our approach is not limited to one specific type of attack detection scheme. In this chapter we provide one applicable example, but in practice the attack detection mechanism can be tailored to a given system based on the perceived threat model.

Our example lightweight detection scheme uses a secondary stack to store return addresses. When a function is called, its return address is pushed onto the stack along with a timestamp of when the function call took place. When the callee function returns, the value on the top of the stack is compared to the return address that is requested by the processor. If the called address is not found in the secondary stack, the detection unit signals that an attack has occurred. The timestamp of the return address that did not match is used to indicate the time of the attack.

This lightweight protection scheme detects attacks at the function level. It verifies that a called function eventually returns to the calling function. For instance, if function A calls function B, the lightweight protection scheme checks that B returns to A. This prevents malicious code from being called in B and returning to A. In that case, an attack is detected, execution is temporarily halted, and the system is signaled to begin the rollback process.

With all detection schemes there exists a tradeoff between security and performance. Detection schemes with a very fine granularity of analysis can provide a more secure environment, but this extra security comes at the cost of increased performance overhead. As a practical matter, for our lightweight scheme we put more emphasis on minimizing performance overhead while still detecting the most common types of attacks.

3.2 Continuous Checkpointing

System state is saved by adapting aspects of the approaches found in [47] and [39]. As a program executes, its state is saved in logs kept separate from the rest of memory. These checkpoints allow a program to be rolled back to that point in time if an attack is detected. The period of time between checkpoints is called the *checkpoint interval.*

In this thesis we make the simplifying assumption that the application is running on a single-core processor. To expand our approach to work in a multi-core system, a mechanism for logging memory race conditions is necessary, similar to what is found in [39]. Because we are not interested in strict deterministic replay, rather restoring a system's state we also choose not to directly log I/O transactions. Instead, I/O is logged indirectly by capturing its effects on the system through memory and processor state logging. I/O behaviors may change upon rollback and re-execution depending upon how far the program is rolled back and how the program is modified for re-execution.

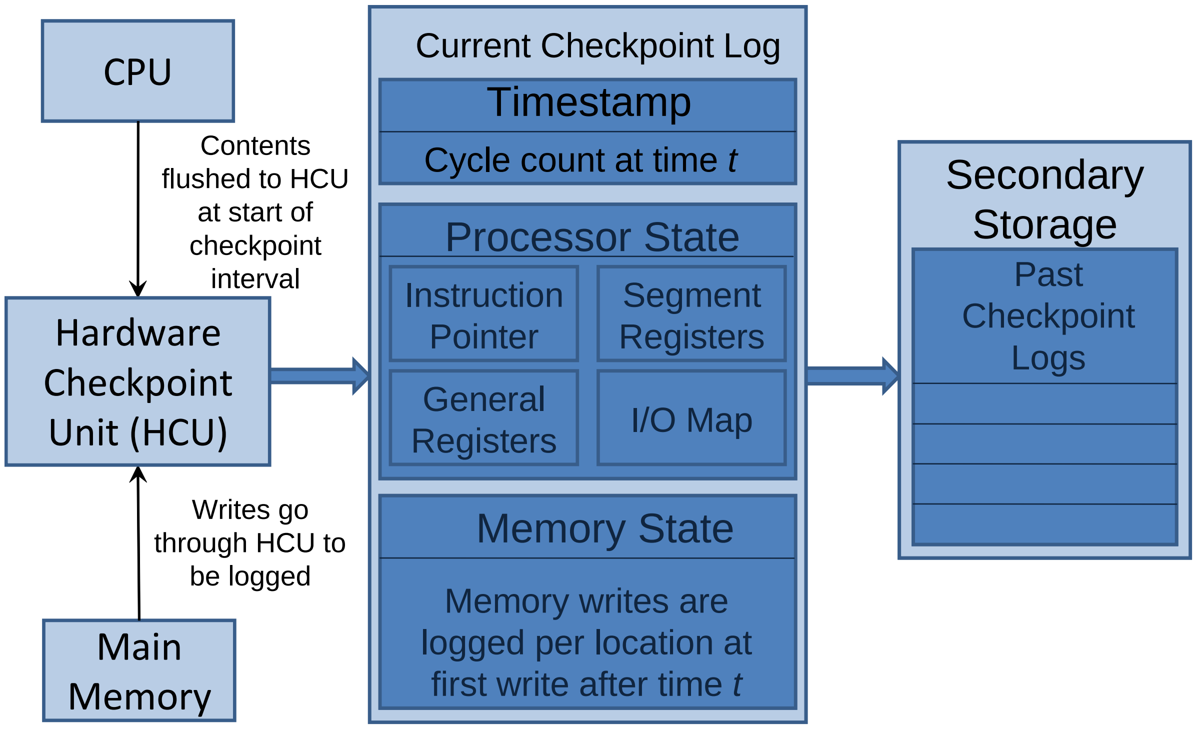

Figure 3.1 Contents of a typical checkpoint log. Checkpoint logs are identified by a timestamp and contain the necessary information to restore a program's state.

Figure 3.1 shows a detailed view of a checkpoint log in our implementation. The logs consist of three pieces: timestamp, memory state, and processor state. The timestamp is the identifier for a particular log. The timestamp is the cycle count in which the checkpoint interval began, and it is recorded immediately when a new checkpoint is created. Memory state is recorded on a per-location basis when writes occur. The current value along with a location identifier are copied into the checkpoint log before the value is overwritten. It is sufficient to record the value of a given memory location on the first write to that location and ignore subsequent writes until the next checkpoint interval begins. Location granularity can vary between implementations. Practical choices include logging memory writes at a per-word or per-cache-block level. We will explore the tradeoffs involved in granularity choice in later chapters. Because our goal is to restore the state of the program at the beginning of the checkpoint interval, we only need to log the initial value for each interval. Processor state consists of the instruction pointer, register values, and I/O map. Like the timestamp, processor state is immediately logged at the beginning of the checkpoint interval.

The policy for determining the length of a checkpoint interval can vary from system to system depending on the application. There are a number of tradeoffs in policy choice which need to be taken into consideration. The policy used in [47] and [39] is to checkpoint after a fixed number of cycles. The main benefit of a *fixed-cycle* policy is that it guarantees a regular checkpoint schedule. The primary drawback is that the memory logs will be of varying sizes because there will not be a fixed number writes between checkpoints. This type of policy is best for applications which make very little use of the memory log buffers. These can include applications with few memory writes, and applications whose writes are to a limited range of addresses. The checkpointing policy which we choose to focus on in this paper is a *fixed-write* policy. In a fixed-write policy, the length of checkpoints is determined by memory write frequency and locality. A fixed size is chosen for the memory log buffers and a new checkpoint begins when the memory log buffer of the current checkpoint is filled. This maximizes the time lengths of checkpoints and minimizes wasted storage because a memory buffer is guaranteed to be full at the end of a checkpoint interval. A fixed-write policy also guarantees a fixed storage size for each checkpoint which simplifies checkpoint storage allocation. This policy is optimal for applications which make many memory writes and make extensive use of the memory log buffers. Alternatively, the two checkpointing policies can be combined such that fixed-size memory buffers are used in conjunction with a cycle threshold, and a new checkpoint interval begins when either of the two conditions are met. This will guarantee regular checkpointing in applications with few memory writes, but it may also lead to wasted storage due to the unused portions of the memory log buffers.

Regardless of checkpointing policy, when a checkpoint interval comes to an end the CPU is temporarily halted, and a new checkpoint log is created. Once the new checkpoint log is created, the processor's state is stored in the new checkpoint log and a timestamp is recorded. When this process is complete, execution resumes and memory writes are recorded in the newly created checkpoint log.

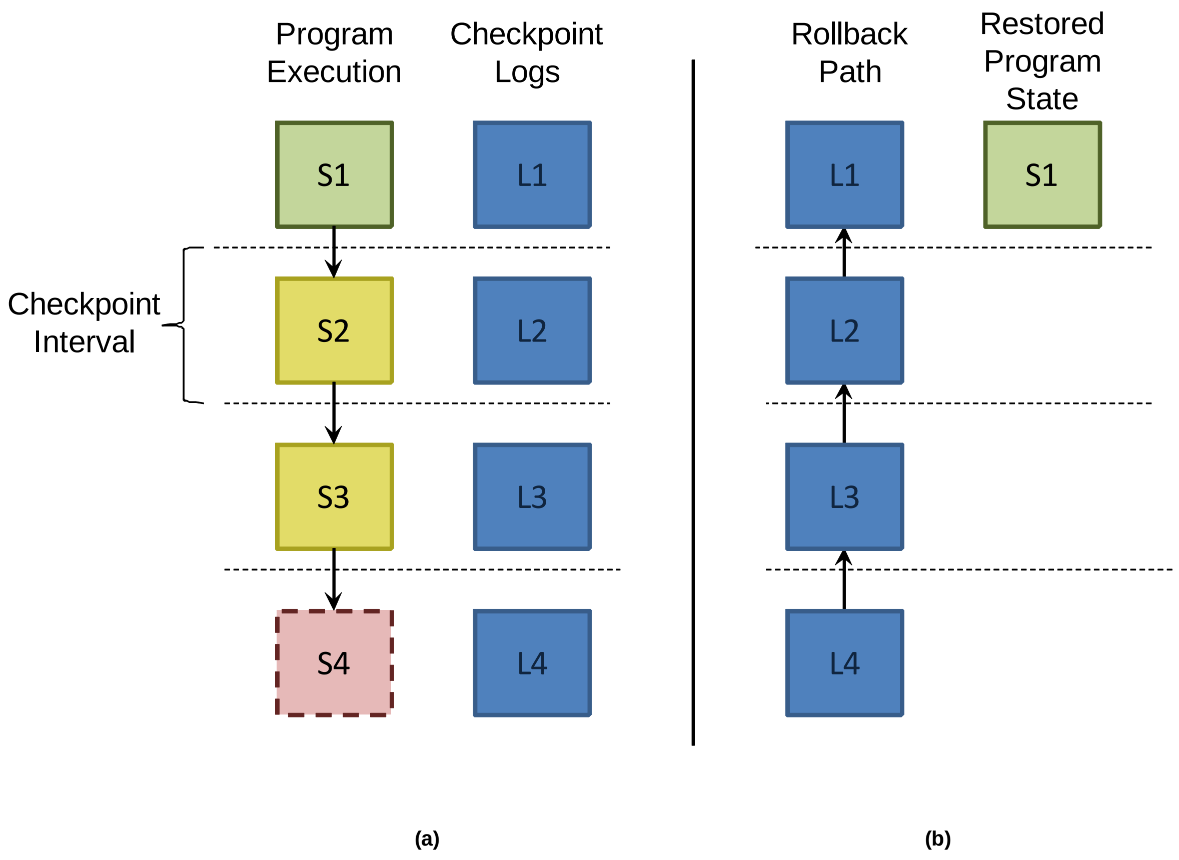

Figure 3.2 (a) As a program executes, its state is checkpointed at normal intervals until an anomaly is detected at S4. (b) The checkpoint logs are cycled through until the system is restored to a safe point at S1.

3.3 Detection and Rollback

Program rollback occurs when the lightweight protection scheme has signaled that an attack has taken place. In Figure 3.2, a program's normal execution is illustrated on the left. As time goes on, its state changes (S1–S4). These states are captured in checkpoint logs (L1–L4) by means of the checkpointing scheme that is in place. During state S4, the lightweight protection scheme has detected that an attack has occurred. At this point execution is stalled and the rollback process begins. The rollback process is illustrated on the right side of Figure 3.2. Rollback begins by restoring the state captured by the most recent log, L4. The rollback process continues to traverse through the earlier logs (L3–L1) until the program is restored to

```
$ROLLBACK\_DISTANCE(t_{attack}, n_{attack}, N_{cp}, C)$ {
    $distance = 0$;
    $extra = 2^{(n_{attack}-1)}$;
    **for each** checkpoint $c \in C$ **do** {
        $distance = distance + 1$;
        **if** $(t_c < t_{attack})$ **then** {
            **break**;
        }
    }
    **return** $(MIN(distance + extra, N_{cp}))$;
}
```

Figure 3.3 An algorithm for determining rollback distance.

the safe state, S1.

One method for determining how far to rewind execution is given in Figure 3.3. As was previously described, when an attack is detected, the detection mechanism associates a timestamp with the event. This timestamp is an estimate of when the attack occurred. If this is the first attack, we simply rollback to the first checkpoint before the time of the attack. This is not sufficient if the vulnerability that led to the attack took place at an even earlier point. The vulnerability could be repeatedly exploited causing an endless loop of rollbacks. To prevent repeated attacks we keep track of how many times a rollback has happened. An extra rollback distance is determined by an exponential function of the number of times that we have already rolled back. More complex schemes are possible that make use of dynamic information flow tracking [40]. This is left for future work.

When rollback begins, program execution is temporarily halted and the on-chip caches are flushed. The first stage of program rollback is rolling back to the beginning of the current checkpoint interval. This is accomplished by writing back all of the values in the current checkpoint log. Writing back the values of the current checkpoint log effectively "rewinds" all of the memory writes that have occurred during the current checkpoint interval. Once the current checkpoint is rolled back, the next checkpoint is loaded from secondary storage. This process continues until n checkpoints have been rolled back, where n is the number returned by algorithm *ROLLBACK_DISTANCE*. At this point the processor is restored to the state stored in the n^{th} checkpoint, and execution resumes from the rollback point in a heavyweight monitoring mode.

In our example system, the secondary stack of the LPU must also be rolled back because its state is tied to the state of the executing program. The timestamp of the top element of the stack must be no later than the time to which the program is rolling back. If the LPU is not rolled back with the program, its contents will become corrupted and result in the incorrect identification of attacks (false positives). In order for the LPU's state to remain consistent, its top element should be the first return address with a timestamp less than that of the program's rolled back state. Once the timestamp of the program's rolled back state is known, this can be accomplished by simply popping elements off of the LPU's stack until a sufficiently early timestamp is found.

If an attack takes place before the earliest checkpoint log, execution is immediately terminated. While this scenario is not ideal for our approach, this behavior is no worse than a standard detection scheme that employs no recovery mechanism. It is possible that an application that terminates in this fashion can be restarted with a heavyweight monitoring scheme in place to prevent repeated attacks.

3.4 Heavyweight Monitoring

Once a program has been rolled back to a safe state after an attack, an adversary could simply repeat the attack if no changes are made to the executing program. This observation is what leads to the final phase of our approach: *heavyweight monitoring.* One possible implementation of heavyweight monitoring is shown in Figure 3.4. Before execution resumes from the point of rollback, the program's binary is further instrumented with additional safety checks in key locations to prevent a repeated attack.

While modifying a program's binary executable at run-time is a non-trivial task, our approach avoids this issue by adding "phantom" instructions at initial compile-time. Conceptually, the compiler identifies potential locations of security vulnerabilities, and adds additional

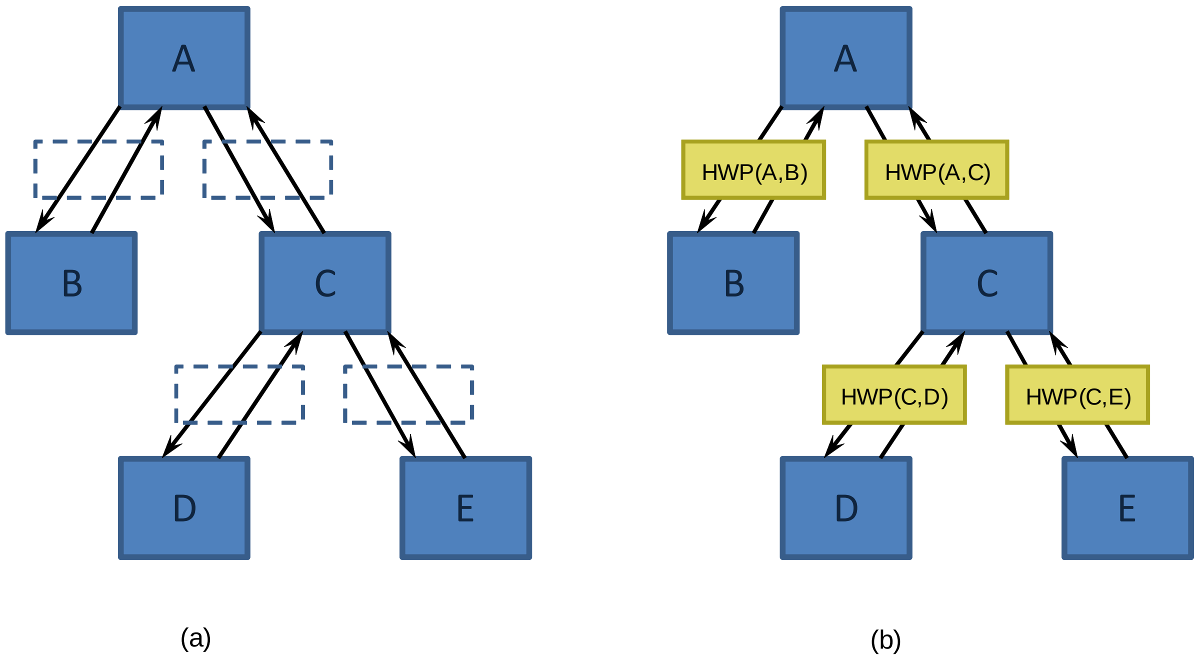

Figure 3.4 (a)Placeholder NOP instructions are inserted by the compiler (b)The program is further for re-execution in the "Heavyweight Monitoring" state

NOP instructions to serve as placeholders for a future live patch. These NOPs are preceded by a jump instruction in order to minimize the performance overhead.

Once an attack is detected, the NOP instructions – along with the preceding jump – can be replaced with instructions to patch the vulnerable code or to enable some other monitoring mechanism. These new instructions are stored in a protected region of memory until the point of execution replay. This provides an additional burden to an attacker looking to break these heavyweight protection mechanisms.

Adding security features such as bounds-checking on arrays and type-checking to pointers can add considerable overhead [28]. Our approach works under the assumption that program degradation is better than program termination. With heavyweight monitoring we are willing to make a compromise on pure execution speed in favor of increased security and ensured stability.

Different possibilities exist for the length of time that heavyweight monitoring should last. One possible policy would be allowing secure execution to last until program termination. This

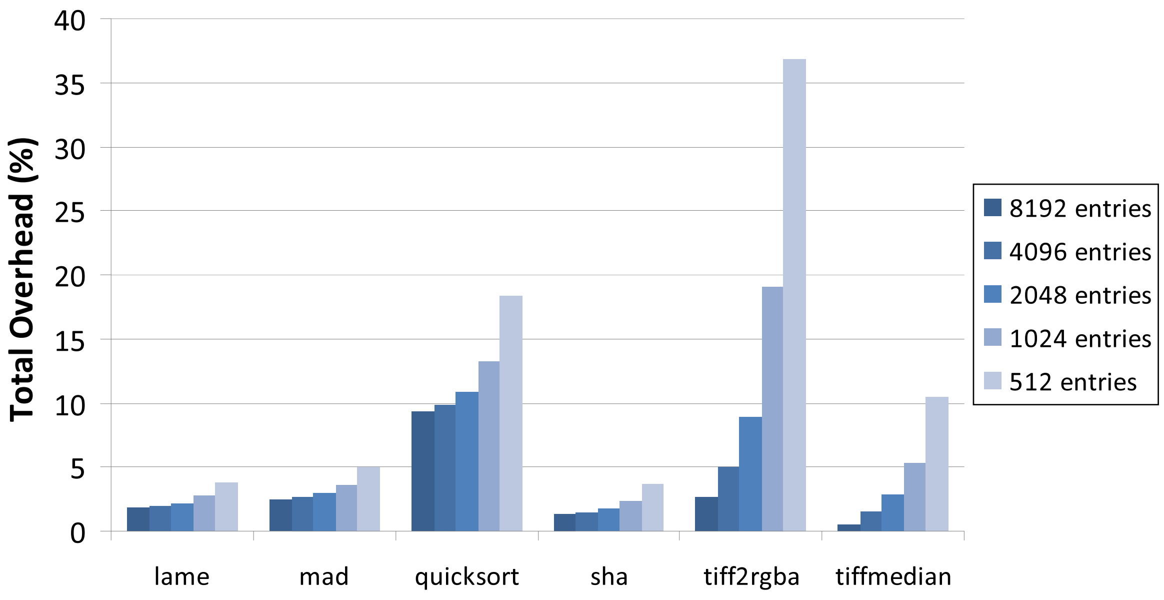

Figure 3.5 Total overhead as a function of memory log size.

policy offers maximum monitoring, but at the cost of an increase in overhead. A second policy would be to execute in secure mode until the point of the original attack. This has the benefit of minimal overhead, but it might open the application up to a second type of attack. A third possibility is a mix between the two: executing in secure mode past the point of the original attack, but eventually lifting the additional security.

3.5 Initial Evaluation

In order to evaluate the performance overhead of our proposed approach, we incorporated a behavioral model of the LPU and HCU modules into the SimpleScalar/ARM toolset [9], a series of architectural simulators for the ARM ISA. We assumed a 4 cycle delay for accessing the LPU on each function call and return, as well as a 200 cycle delay at the HCU for storing the processor state and managing the checkpoint data structure at the beginning of a new interval. The input for evaluation were six benchmarks from the MiBench embedded benchmark suite [21]. As an initial investigation into the performance impact of our approach, we compared the total overhead, checkpoint frequency, and maximum rollback distance for various

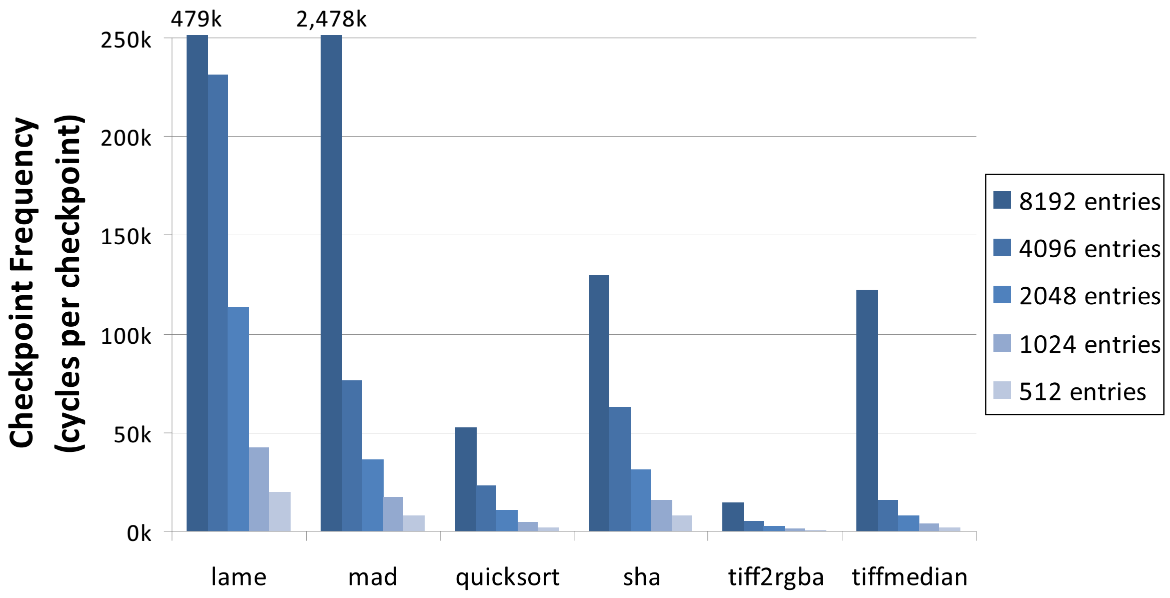

Figure 3.6 Checkpoint frequency as a function of log size.

configurations of the HCU. In each of these HCU configurations, we are logging memory on a per-word basis.

3.5.1 Overhead

Figure 3.5 shows the total overhead of our approach for five different checkpoint log sizes. We varied the log sizes from 512 to 8192 entries, while keeping the total number of stored logs constant at 64. The total performance overhead is the sum of the lightweight protection scheme and the checkpointing scheme. The overhead of the lightweight protection scheme is based on the number of function calls, so it is completely benchmark dependent. Because of this, the overhead of the lightweight scheme is constant across all configurations for a given benchmark. Figure 3.5 shows that allowing for larger checkpoint logs results in less overhead due to checkpointing. In all benchmarks but `quicksort`, the total overhead was less than 5% for log sizes of 4096 entries and 8192 entries, and less than 10% for all benchmarks with these two configurations. The average across all benchmarks and configurations was 6.41%. When the 512-entry configurations of `quicksort` and `tiff2rgba` are removed, that number drops to

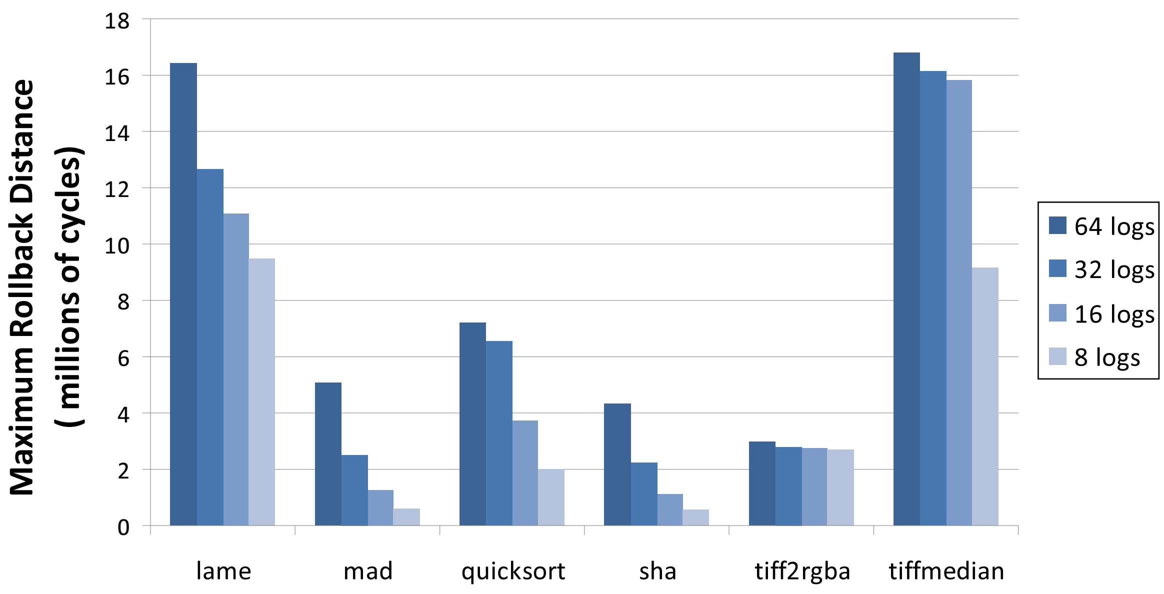

Figure 3.7 Rollback distance as a function of total stored logs.

4.87%.

3.5.2 Checkpoint Frequency

Figure 3.6 shows the checkpointing frequency of the same five configurations that were used to measure total overhead. The checkpointing frequency of a configuration was generally proportional to the memory log size of the checkpoints. The average checkpoint interval varied greatly between benchmarks. Configurations with log sizes smaller than 2048 entries showed considerable overhead in benchmarks with frequent checkpoints. This is especially apparent when comparing the `tiff2rgba` benchmark in Figures 3.5 and 3.6. A log size of 8192 entries results in an overhead of 2.6% while reducing the log size to 512 entries results in a 36.9% overhead.

3.5.3 Rollback Distance

Figure 3.7 shows the maximum number of cycles that can be rolled back for a given number of logs. The maximum rollback distance was calculated by taking the timestamp difference of the newest checkpoint log and the oldest checkpoint log. We kept the log size constant at 4096 entries while varying the total number of logs kept on record. This experiment demonstrates that the maximum rollback distance does not always have a proportional relationship with the total number of logs on record. The `mad`, `quicksort`, and `sha` benchmarks generally show a relationship proportional to log number, while `lame`, `tiff2rgba`, and `tiffmedian` do not. The reason for this is that memory writes do not always happen at regular intervals in all programs. If a program executes for a long period of time with very few memory writes, its state will not need to be checkpointed as frequently. Results indicate that `tiffmedian` exhibited phase behavior with a large number of writes that spanned between 8 and 16 logs.

CHAPTER 4. Architectural Implementation

In this Chapter we will outline a possible implementation of the additional architectural features that enable rollback and huddle. Hardware modules include a Lightweight Protection Unit (LPU), Hardware Checkpoint Unit (HCU), and a Heavyweight Protection Unit (HPU). These additional hardware features are non-intrusive, and integrate into a standard computer architecture consisting of a CPU, main memory, and optional secondary storage. Our hardware is designed to minimize the performance impact during standard execution, as well as minimize the total storage footprint required for checkpointing.

4.1 Lightweight Protection Unit

For this implementation of the LPU we consider a design that differs slightly from the return address stack approach introduced in Chapter 3. Once again our design will protect software at the function-call level. The LPU in this example is based on the design introduced in [44] which implements a hardware version of PointGuard [11]. In this approach two instructions are added to the ISA: an encrypted store and decrypted load. At process creation time, the LPU will create a random key. When a return address is stored, it is encrypted by XORing with the random key in the LPU. When the return address is used, it is decrypted by XORing it with the LPU's key.

This approach requires recompilation or binary modification of the target application. During recompilation, `call` instructions are augmented with the encrypted store instruction. Likewise, `return` instructions are augmented with the decrypted load instruction. If an attacker attempts to overwrite the return address, the program will return to a random location due to the XORing occurring during decryption. When this happens, an exception is thrown and the system is stalled so that rollback can occur. When the LPU decrypts an a return address, it stores the encrypted address until the next decryption. If an attack occurs, it passes the original encrypted address to the HCU so that it can determine how far to rollback execution. This process is detailed in Section 4.2.

In this approach, an attacker cannot read the LPU's key directly, however if an adversary has access to an unmodified version of the executing program, and the attacker is able to read the modified return address it is trivial to deduce the key and modify the attack accordingly. These "read attacks" are addressed in the original work [44], however we do not defend against such attacks due to the difficulty in mounting such an attack and the additional overhead required to protect against them. This approach can also be extended to protect all code pointers like the original PointGuard implementation [11], but at the cost of increased runtime overhead. A full PointGuard implementation could be an ideal candidate for a heavyweight protection scheme.

4.2 Hardware Checkpoint Unit

The Hardware Checkpoint Unit (HCU) provides the system with checkpointing and rollback capabilities. The HCU handles both checkpointing and rollback duties because rollback is essentially just the inverse of checkpointing. The HCU requires direct access to the main memory bus, and implementation can take the form of either a standalone module plugged into the system, or alternatively as a modified memory controller. It is designed to optimize the relatively common case of logging memory addresses.

Figure 4.1 shows the HCU, and gives a high-level of view of the logging process. The HCU itself consists of a basic logic controller, and some internal storage. Its internal storage is a Content Addressable Memory (CAM) used to store the memory addresses that have been logged during the current interval. Each time a write to main memory occurs the HCU must check whether it has been logged. Utilizing a CAM allows for fast lookups during memory writes. The CAM must be large enough to store each of the addresses of the current memory log. The contents of the CAM are flushed when a new checkpoint is created. The physical memory logs are stored in a reserved section of main memory. The HCU logs memory writes for the current checkpoint interval, but it also keeps a number of past checkpoints on record. The reserved memory can be viewed as a circular buffer. When a new checkpoint is created, the oldest checkpoint on record is overwritten. Each entry of the memory logs consist of data, as well as an address specifier. It should be noted that the address is stored in both the memory log and the CAM. The addresses stored in the CAM are only used for the purpose of quickly checking whether the location has been logged, while the addresses stored in the memory log are used for addressing the memory writebacks during the rollback process.

Configuration of the HCU is open to a number of optimizations, particularly in the area of memory log parameters. The first parameter is memory log granularity. Memory log granularity is the definition of what constitutes a unique memory address to be logged. Memory can be logged as fine as the byte level, or as coarse as the memory-page level. Two realistic choices are logging at the word level and logging at the cache-block level. As memory log granularity becomes finer the storage overhead becomes greater, but log utilization increases as well.

Figure 4.2 provides two examples illustrating the tradeoffs in memory logging granularity. In the first example, a program makes eight writes to the same cache block. The word logging policy requires a 100% storage overhead to log the addresses. The cache-block logging policy will log the cache block on the first write, and the seven subsequent writes will already logged. Once again there is 100% log utilization, but this policy only requires a 12% storage overhead for logging the addresses, assuming an eight word cache block. In the second example, a program makes eight writes, but this time each of the writes are to different cache blocks. The word logging policy once again requires a 100% storage overhead for logging the addresses, and it also retains 100% log utilization. In the cache-block logging scheme, each of the writes will be considered unique and require a separate logging transaction. The cache-block logging scheme retains the 12% storage overhead for address logging, but log utilization drops to 12%, as shown in Figure 4.2 by the light-colored boxes. Low log utilization can lead to more frequent checkpoints, and thus decrease performance due to excess checkpoint creation. In Chapter 5 we choose to focus on a cache-block logging approach.

The second memory log parameter to consider is the total size of the memory logs. A larger memory log with more entries will allow for longer checkpoint intervals, and decrease the performance penalty due to checkpoint creation. We can vary not only size of the current memory log, but also the total number of checkpoints that we keep in storage. Although longer checkpoint intervals can reduce total performance overhead, it also makes it more difficult to rollback to a specific point in a program's execution because of the long duration of checkpoint intervals. In Chapter 5 we utilize a total memory log storage of 295 KB. Configurations range from 8 memory logs with 512 cache-line entries to 128 logs with 32 cache-line entries per log. We were able to achieve rollback distances in the millions of cycles range with these configurations.

Finally, the HCU contains some simple logic to carry out its checkpointing and rollback duties. The controller must keep track of the current checkpoint log. When the memory log is filled, it is the end of the current checkpoint interval and the HCU stalls the CPU. The controller flushes the CAM, and finds the location of the next checkpoint log in memory. The controller saves the register state of the CPU in the current checkpoint log and then allows execution to resume.

This particular implementation of the HCU assumes that the CPU is using a write-through caching policy. We choose not to log the CPU cache directly because all writes will propagate through the memory hierarchy in a write-through system. To enable logging in a system using a write-back cache the HCU would need to flush the caches at the end of the checkpoint interval and log any writes that have not been logged yet. A certain portion of the memory log must be left free to store the outstanding writes remaining in the cache.

The HCU is also used for rollback. When an attack is detected, the HCU enters *rollback mode*. The LPU sends the HCU the last return address that was used. Starting with the current checkpoint log, the HCU writes back the memory values stored in the memory logs. The HCU continues rolling back checkpoint logs until it finds the failed return address. The HCU continues writing back the log that contains the return address. If this is not the first rollback, additional checkpoints may need to be rolled back. When the HCU completes the memory write-back process, it restores the CPU register state to the state stored in the final checkpoint log to which it rolled back. If all of the checkpoint logs are exhausted without finding the return address, the HCU throws an exception and the system terminates execution of the compromised program.

4.3 Heavyweight Protection Unit

The final piece of hardware added in our approach is known as the Heavyweight Protection Unit (HPU). The HPU is used in the *heavyweight monitoring mode* of our approach. The HPU must have write access to a program's instruction space so that it can facilitate and accelerate the additional instrumentation required for heavyweight monitoring. Implementation details of the HPU will vary depending on the intended functionality and goals of heavyweight monitoring. We will outline a couple design possibilities for hardware which could be implemented to aid in this process.

One possibility is using the HPU as protected storage for a number of instruction templates which are used to further instrument the original code and replace the NOP phantom instructions. The instructions contained within the HPU would be templates to correct general classes of attacks. One example would be code to check that the bounds of an array are not overrun. Once an attack has occurred and the program has been rolled back, this HPU implementation would take a number of parameters including the attack type and specifics about the attack such as the size of the array being overrun. The HPU would use these parameters to select the correct code template and tailor the instructions for the situation at-hand.

Another possibility for the HPU is for it to act as an accelerator for encrypted execution. In this case, instead of inserting NOP instructions in the original program, the instructions could already be in place in an encrypted form. Once rollback takes place, the the HPU would identify and decrypt the encrypted instructions so that they can be executed upon rollback and re-execution. Using the HPU as an accelerated encryption engine could also be extended to create an encrypted-execution environment similar to XOM [24]. Instead of decrypting placeholder instructions upon rollback, the HPU would encrypt the entire instruction space.

As instructions are loaded from memory they would be fed through the HPU and decrypted before they are delivered to the CPU and executed.

Systems which frequently interact with a network could make use of an HPU configured as a hardware firewall. If the system was compromised by spurious input from the network, the program (or even the HPU itself), could identify the IP from which the input originated. Upon rollback and re-execution, the HPU could filter requests and block inputs from blacklisted IP addresses.

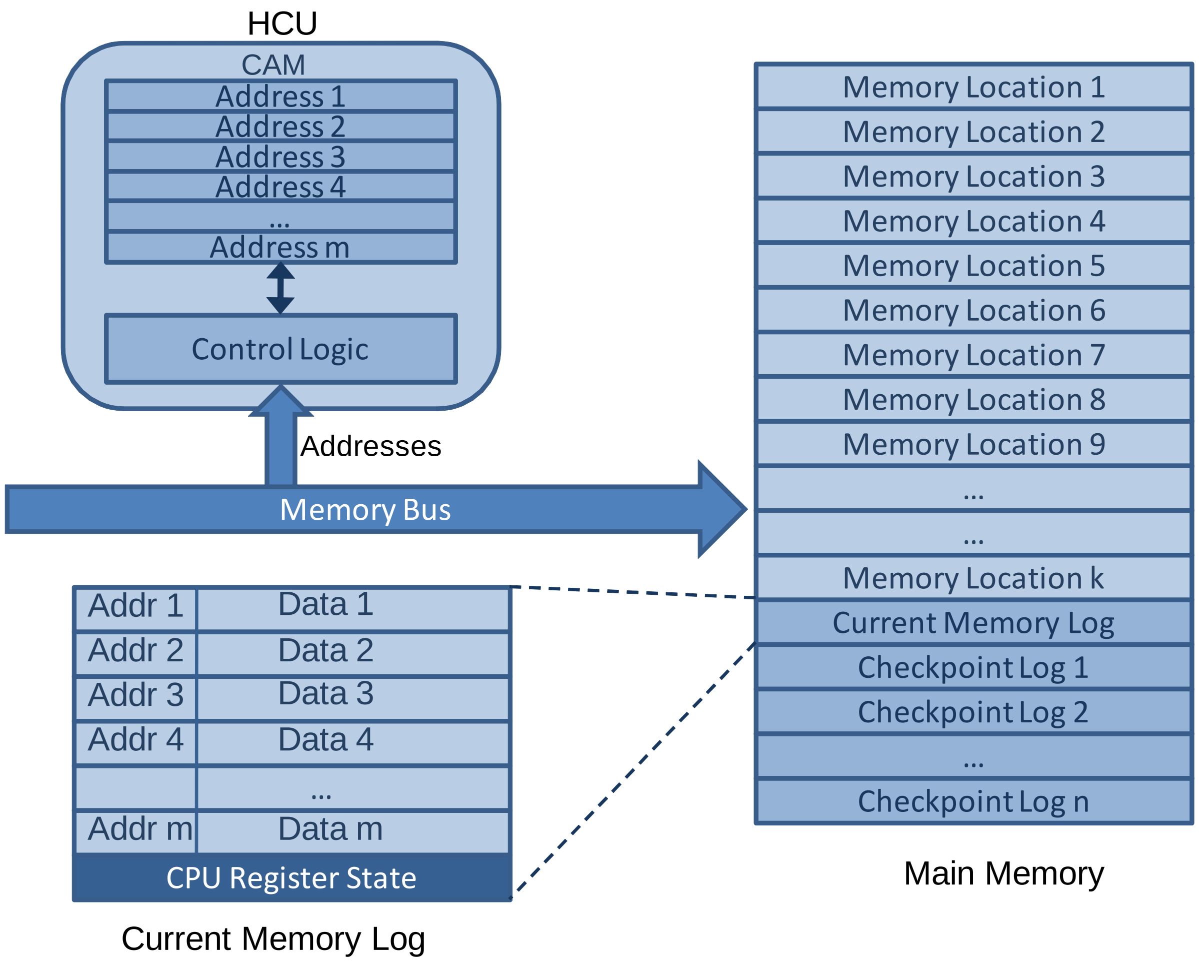

Figure 4.1 The CAM within the HCU contains m entries corresponding to the m memory log entries of the current checkpoint log. A total of n memory logs are stored in main memory. The HCU snoops addresses passing over the memory bus, and the controller looks for the addresses in the CAM. If an address is not found, data is copied into the checkpoint log before it is overwritten.

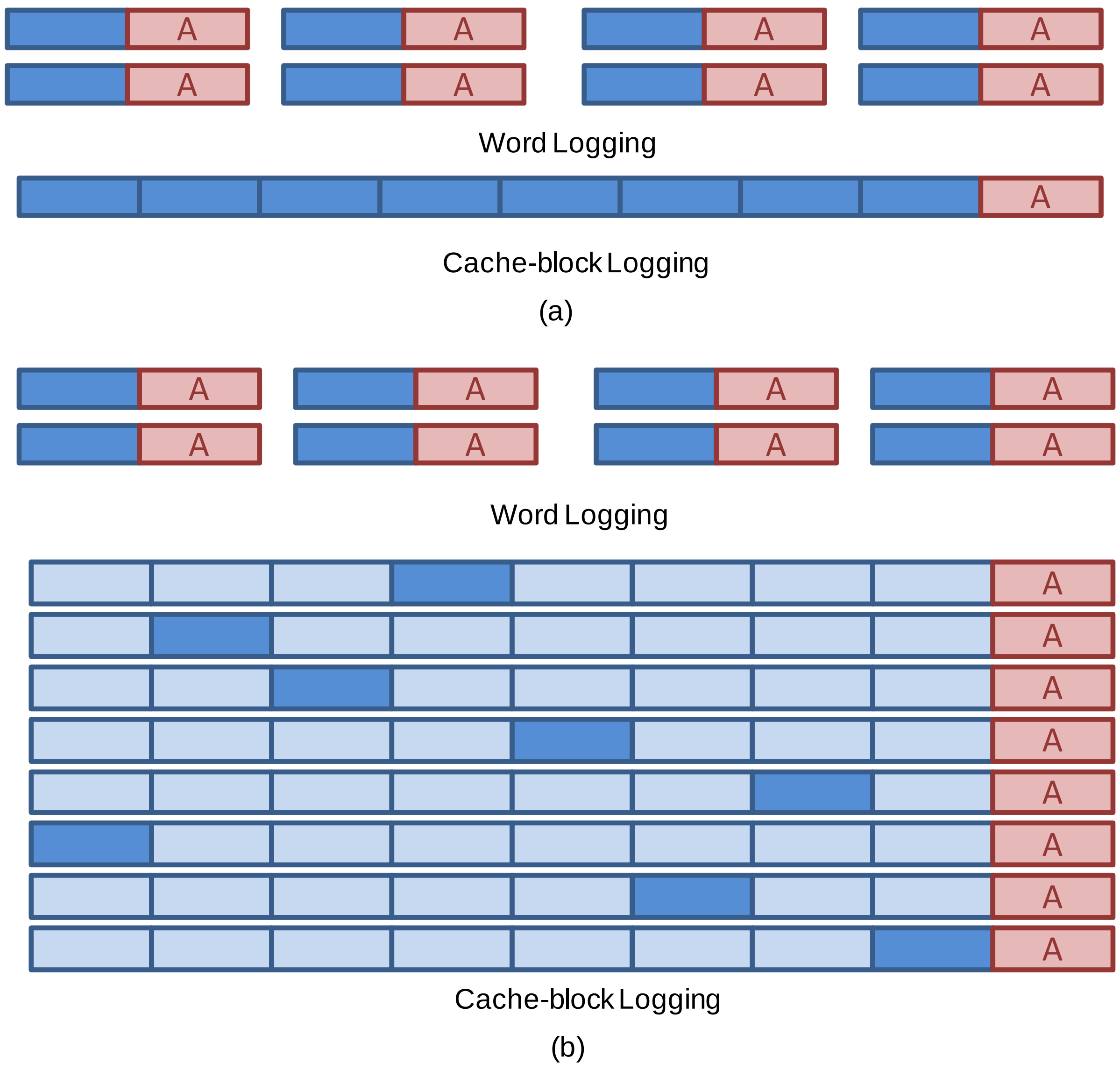

Figure 4.2 Effects of logging granularity. (a) A series of 8 writes to a single cache block are logged. Both policies achieve 100% log utilization, but word logging requires more space for storing addresses. (b) A series of 8 writes to 8 different cache blocks are logged. The cache-block scheme logs a number of words that were not actually written.

CHAPTER 5. Benchmarks and Experiments

In order to evaluate the performance overhead of our proposed approach, we incorporated a behavioral model of the LPU and HCU modules into PTLSim Classic [49], a cycle-accurate x86 simulator. Simulation was performed using PTLSim's out-of-order core model which simulates a single-threaded application with realistic branch prediction and cache behaviors. PTLSim's processor core models a 64-bit processor and uses a combination of features found in AMD Athlon 64, Intel Pentium 4, and Intel Core 2 Duo processors. The simulator configuration is shown in Table 5.1:

A write-through policy was used for data writes so that any cache writes were automatically propagated through the memory hierarchy. We assumed a 1 cycle delay for accessing the LPU on each function call and return, as well as a 200 cycle delay at the HCU for storing the processor state and managing the checkpoint data structure at the beginning of a new interval. The HCU was configured log memory writes at the cache-block level. We assumed a memory model that transfers 8 bytes (1 word) each cycle. Therefore, each logging transaction takes 9 cycles to complete (8 words per line, plus the address). If the HCU needs to log another address before the current logging transaction is complete, it must stall the pipeline until it completes.

PTLSim comes with patches that can be applied to the SPEC CPU2000 benchmarks suite. The PTLSim patches do not modify the semantic behaviors of the SPEC benchmarks, the patches simply insert function calls to the simulator engine within each of the benchmarks so that simulation begins at the start of the main benchmark loops. We used a patched version of the SPECint subset of the SPEC2000 benchmarks suite, and simulated 200 million instructions on our modified architecture. As an initial investigation into the performance

Table 5.1 Simulator configuration

Operating Mode	64-bit Out of Order
Functional Units	2 ALU 2 FPU 2 Store Units 2 Load Units
Pipeline	11 stages
ROB	128 entries
L1 Instruction Cache	32 KB, 4-way set associative, 1 cycle latency
L1 Data Cache	16 KB, 4-way set associative, 1 cycle latency
L2 Cache	256 KB, 16-way set associative, 6 cycle latency
L3 Cache	4 MB, 32-way set associative, 16 cycle latency
Main Memory	140 cycle latency

impact of our approach, we compared the total overhead and maximum rollback distances for various configurations of the HCU.

5.1 Performance Overhead

Figure 5.1 shows the total overhead for a number of HCU configurations. We varied the size of the memory log from 512 entries down to 32 entries. Each entry consists of a cache block containing 8 64-bit words, along with a 64-bit address identifier. We kept the total size of the HCU constant across all configurations at 4096 total entries, which requires approximately 295 KB of total storage. Overhead during the lightweight monitoring phase of our approach comes from three sources: the LPU, checkpoint creation, and stall cycles when the HCU is busy. The overhead related to the LPU is constant across all configurations because it is directly related to the function call-return pairs in a given program. Changing the size of the memory log affects the other two sources of overhead because smaller memory log lead to more frequent checkpoints. Decreasing the size of the memory logs also puts additional stress on the HCU and leads to increased number of stall cycles. In Section 5.2 we provide analysis into how each of these sources contribute to the total overhead.

Looking at Figure 5.1, as expected the larger memory log configurations perform better than the smaller memory log configurations. The 32-entry configuration performed dramat-

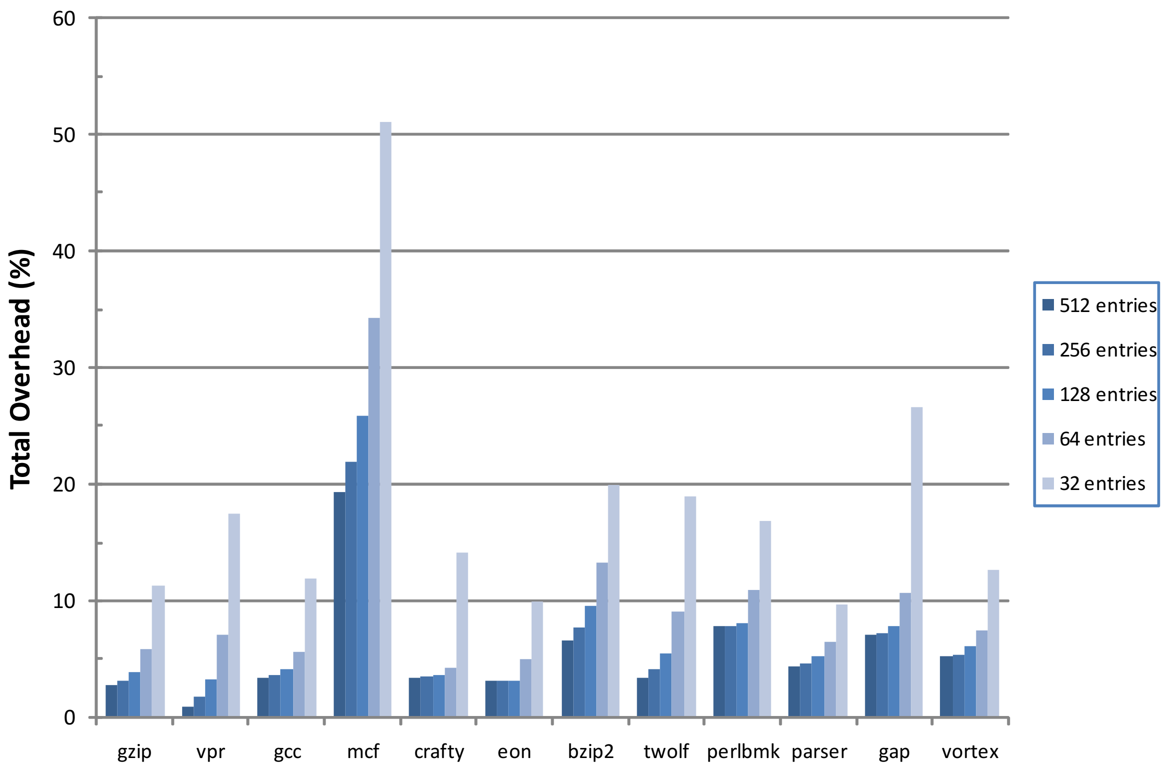

Figure 5.1 Total overhead of LPU and checkpointing schemes.

ically worse than the other four configurations. Across many benchmarks the performance impact of log size begins to level off at 128 entries. The average performance impact across all benchmarks and all configurations was 9.43%. If we remove the poor performance of the 32-entry configuration, that average drops to 7.20%. Excluding the 32-entry and 64-entry configurations, all benchmarks stayed below the 10% mark, and many of them did not 5% overhead. The one exception was `mcf`, which experienced a 19% slowdown for its 512-entry configuration, and more than 50% with the 32-entry configuration. The unusually large amount of overhead was due to its tendency to write to many unique memory locations which put a lot of stress on the HCU. While running for 200 million cycles, the 512-entry configuration created 19,189 checkpoints. The closest benchmark to this number was `twolf` which created 3,247 checkpoints.

5.2 Overhead Breakdown

Figure 5.2 shows a breakdown of how each of the overhead sources contributed to the total overhead in each of the SPECint benchmarks for the 128-entry HCU configuration. As mentioned in Section 5.1, LPU overhead is constant for a given benchmark regardless of HCU configuration because LPU overhead is based on the number of function call-return pairs, which does not change. Configurations with smaller memory log sizes have a larger proportion of overhead come from checkpoint creation and stall cycles due to the associated logging frequency. Conversely, configurations with larger memory log sizes have a larger proportion of overhead related to the LPU.

Figure 5.2 shows that the LPU is the largest source of overhead when compared across all benchmarks, accounting for approximately 65% of total overhead. Checkpoint creation and HCU stalling account for approximately 19% and 16%, respectively. The benchmark `mcf` suffers from tremendous checkpoint-related overhead. This shows that the HCU simply cannot keep up with the large volume of unique memory writes and the subsequent logging load. The benchmarks `vpr`, `twolf`, and `bzip2` show similar checkpoint-related slowdowns, though not as extreme as `mcf`. When comparing with Figure 5.1, the total overheads of these three benchmarks were significantly lower than `mcf`. Although HCU stalling accounted for the smallest proportion of overhead, its effect is not negligible. Future HCU designs could possibly incorporate some type of caching abilities so that the hardware can handle a number of outstanding logging operations. This could potentially eliminate overhead related to HCU stalling.

5.3 Rollback Distance

Table 5.2 compares the maximum rollback distances for the same five HCU configurations used in Section 5.1. We observed a large amount of variation in the maximum rollback distances between benchmarks. The 512-entry configuration was able to roll back over 250 million cycles of the `eon` benchmark, but the same 512-entry configuration could not roll back more than 1 million cycles of the `mcf` benchmark. Configurations of 64 entries could generally rollback

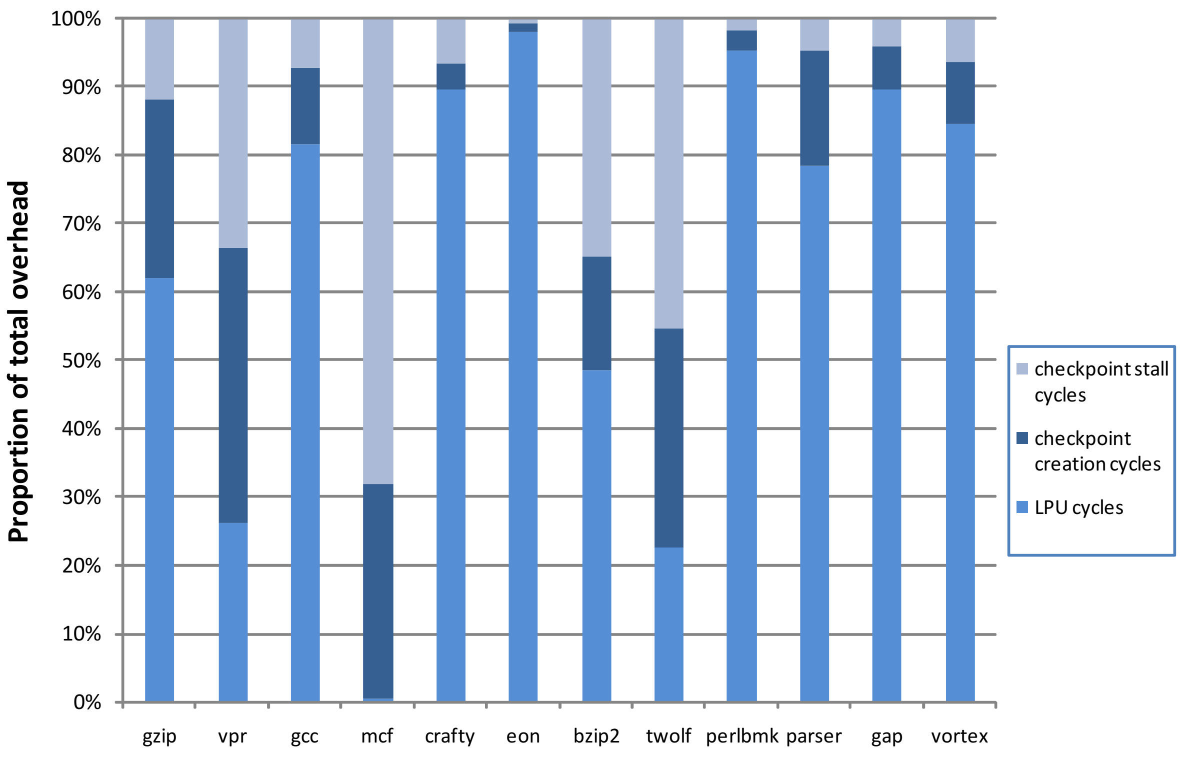

Figure 5.2 Breakdown of overhead sources.

anywhere from 1 million to 10 million cycles, with some exceptions. The 32-entry configuration performed much worse. The 32-entry failed to rollback more than 1 million cycles in four benchmarks, and it was only able to rollback more than 2 million cycles in three benchmarks.

Figure 5.3 shows the relative impact that different HCU configurations had on a given benchmark. The graphs for each benchmark are normalized to the number of cycles rolled back by the 512-entry configuration, which achieved the largest maximum rollback distance in every benchmark. The `vpr` and `eon` benchmarks are normalized to their respective 256-entry configuration results because the 512-entry configuration was capable of rolling back the entire benchmark runs for those two programs, which skewed their results.

Despite storing the same number of total memory log entries in every configuration, many of the benchmarks are very sensitive to changes in the HCU configuration. The `eon` benchmark is especially sensitive to HCU configuration. The 32-entry configuration can only rollback 1% of

Table 5.2 Maximum cycles rolled back for different HCU configurations

Memory Log Size Total Checkpoints	512 entries 8	256 entries 16	128 entries 32	64 entries 64	32 entries 128
gzip	3,912,582	3,807,096	3,751,585	3,682,519	3,539,779
vpr	197,973,063	1,130,098	824,390	638,215	533,719
gcc	11,676,040	6,979,757	6,176,254	5,007,841	4,069,716
mcf	960,964	953,047	943,423	937,900	819,897
crafty	12,710,404	10,941,094	7,736,916	3,327,726	845,634
eon	269,482,558	52,384,716	19,492,186	1,121,911	598,838
bzip2	25,006,574	24,395,576	23,655,060	21,762,244	18,233,093
twolf	1,527,168	1,461,540	1,366,358	1,227,178	1,105,523
perlbmk	12,398,191	10,115,520	6,241,233	1,396,125	1,145,623
parser	10,206,897	9,062,926	6,912,091	4,588,033	2,763,116
gap	5,926,917	5,070,217	3,804,918	2,285,961	1,341,548
vortex	9,985,679	6,457,392	3,197,701	2,817,878	1,980,220

the cycles that the 256-entry configuration can rollback. A number of other benchmarks show the 32-entry configuration rolling back less than 30% as far as the 512-entry counterpart. The benchmarks that were sensitive to the HCU configuration were those with an even distribution of memory writes and a constant rate of checkpoint creation. Some benchmarks showed less sensitivity to changes in the HCU configuration. The `gzip` benchmark is one such benchmark. This indicates that there was most likely some stretch of execution with very few writes that all of the configurations were able to rollback.

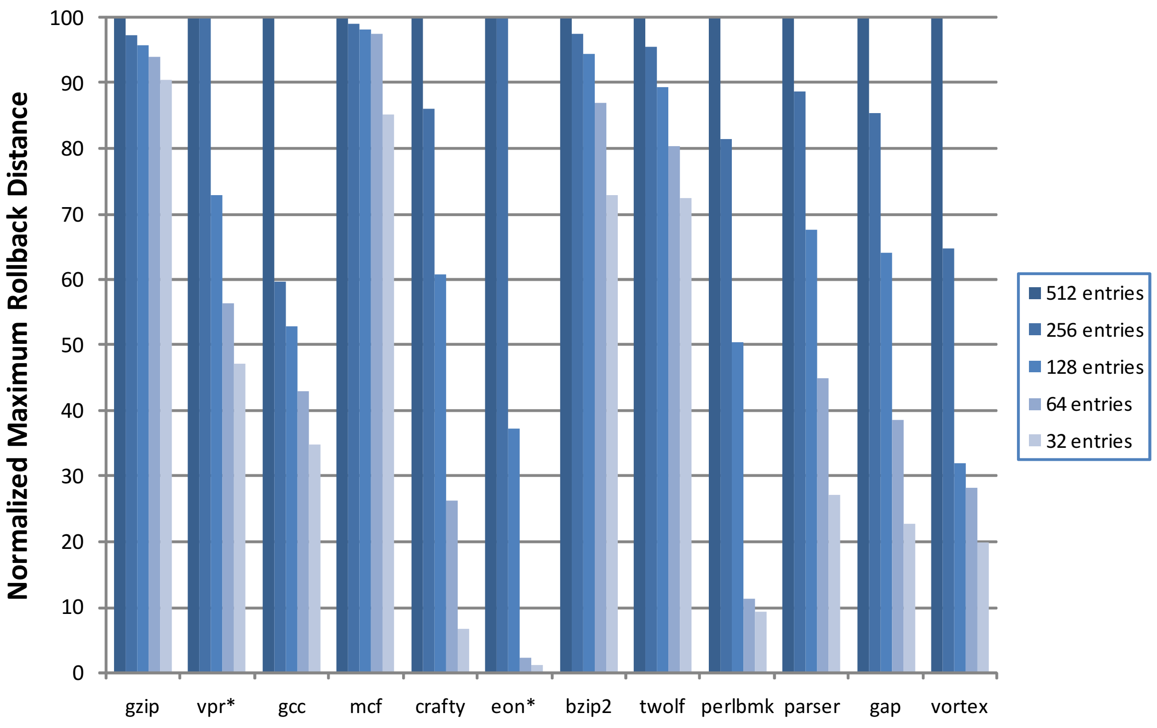

Figure 5.3 Maximum Rollback distance normalized to rollback distance of the 512-entry configurations. The `vpr` and `eon` benchmarks are normalized to the 256-entry configurations.

CHAPTER 6. Attack Analysis

A common form of software-based attacks is known as "Stack Smashing" [29], and it is an example of a buffer overflow attack. The stack segment for function `F` is shown in Figure 6.1. It contains an array `temp`, a number of local variables, and the function's return address. As `temp` is written, it grows toward the top of the stack. If enough data is written into `temp`, it is possible to write past its intended bounds and eventually replace the function's return address. If an attacker knows that a program contains a vulnerable buffer, it is possible to craft a specially-formulated input so that the return address is overwritten with a specific value to return to the attacker's malicious code.

To verify the effectiveness of our checkpointing and rollback mechanisms we tested our approach against eight vulnerabilities found in real-world applications. The program names and vulnerable are listed in Table 6.1. All of the programs are open source and written to be run in a Unix environment. Originally discovered as part of a class project [5], details for each exploit can be found in US-CERT Cyber Security Bulletin SB04-357 [10]. Each of the programs contain an overflowable buffer that an attacker can use to run arbitrary code on the victim's machine. We simulated the programs using the PTLSim [49] Classic out-of-order core model.

6.1 Attack Time

Initially, we investigated the number of cycles that need to be rolled back to recover from an attack. Each of the vulnerable programs was given malicious input designed to overflow its vulnerable buffer. Within our simulation environment we measured the cycle count from the point that input was given until the point that the program attempted to use the malicious

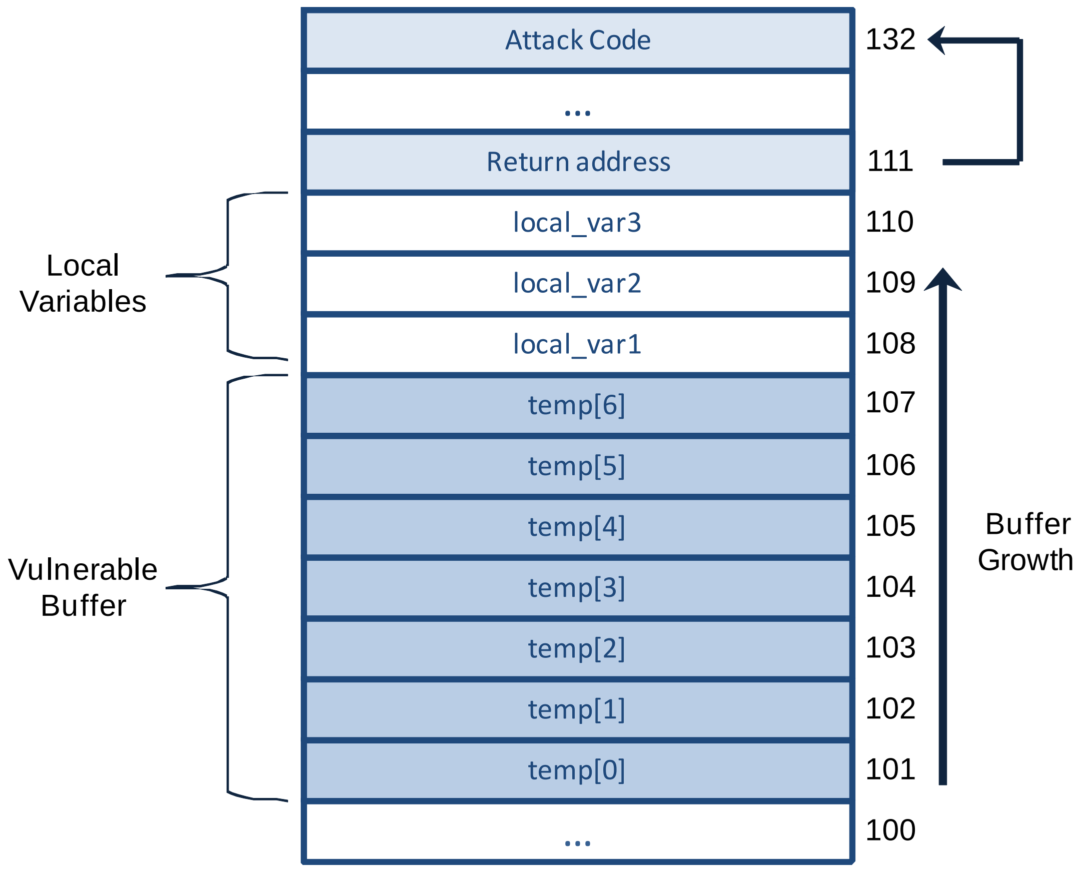

Figure 6.1 A vulnerable buffer, `temp` is placed on the stack. An attacker is able to write past its intended bounds and replace the return address so that the function returns to attack code.

return address. This is the point of attack-detection for any of the protection schemes used in this thesis. The results of these tests are shown in Table 6.1. Cycle counts ranged from 304,864 in `o3read` all the way to 6,745,763 in `villistextum`.

Once we collected general information about the attacks, we verified that our approach would actually be able to rollback to the point of input. We configured the HCU to use 256 memory logs consisting of 128 memory-word entries per log. Once again the vulnerable programs were given malicious inputs. In each case, when the program attempted to return to the malicious return address, the HCU contained sufficient information to rollback to the time that input was received.

Table 6.1 Required cycle rollback of vulnerable programs

Application	Version	Cycles from input to detection	Cycles to overflow buffer	Size of buffer (bytes)	Recoverable?
villistextum	2.6.6	6,745,763	6,129,857	32,768	Yes
ringtonetools	2.22	3,640,082	94,828	1,024	Yes
csv2xml	0.5.1	1,841,252	119,261	1,000	Yes
2fax	3.0.4	727,625	3,016	256	Yes
bsb2ppm	0.0.6	665,687	86,108	1,024	Yes
jpegtoavi	1.5	432,591	99,465	4,096	Yes
o3read	0.0.3	304,964	92,206	1,024	Yes

6.2 Overflow Time

After verifying that our approach was able to recover from each attack, we attempted to quantify the time required to overflow the vulnerable buffer. We measured the cycle count from the first write to the vulnerable buffer until the first write past the intended bounds of the buffer. Using 6.1 as an example, this would be the cycle count required to write from *address 101* (`temp[0]`) to *address 108* (`local_var1`). These times are shown in Table 6.1. One of two causes was at the root of each of these vulnerabilities. In some programs the vulnerabilities were the result of calling library functions such as `scanf` that copy any amount of data into a buffer, regardless of the source and destination sizes. The other vulnerabilities were caused by blindly incrementing a pointer during a loop without checking the intended buffer boundary. The time required to overflow the buffer is related to the size of the buffer as well as the additional computations taking place during the loop. The buffer in `2fax` is only 256 bytes, and the there are no computations other than the data copying. At the other extreme is `villistextum` which has a 32,768-byte buffer, and many additional computations are performed during each loop iteration.

CHAPTER 7. Conclusions and Future Work

In this thesis we have presented a new approach to software security that puts equal emphasis on attack detection and attack recovery. We have also investigated a number of timing characteristics in a set of real-world buffer overflow vulnerabilities. In this work, we have focused on the initial lightweight monitoring mode of execution, consisting of a lightweight detection mechanism operating alongside a checkpointing scheme. Although our initial results are promising, we see several avenues for future work to build upon the foundations laid in this thesis.

We designed the checkpointing scheme so that it requires a very small storage footprint for each process that is logged. The small storage footprint will enable many processes to be simultaneously and independently checkpointed. For instance, the HCU configurations in our benchmarks utilize approximately 300 KB of storage per process. Using a similar configuration, a system could manage over 100 processes and only require 30 MB of total checkpoint storage. This is a very small memory footprint considering many modern computers contain more than 2 GB of system memory. Checkpointing multiple processes independently will require small modifications to the OS, so that our checkpointing hardware can be made aware that context switches are occurring. Our checkpointing hardware will also need to be extended so that it can swap out its CAM entries when context switches occur. Checkpointing multiple processes independently will be an important feature because if one process is compromised, we do not want to disrupt all of the healthy processes by rolling back the entire system.

The work in this thesis utilized a simple lightweight protection scheme that only protected against the most basic of software attacks. We chose this scheme for its simplicity and relatively low overhead characteristics. Future implementations of our approach could explore the use of more complex detection schemes and investigate the tradeoffs involved in making such decisions. The area of our work that leaves the most room for future research is the idea of heavyweight monitoring. We have suggested the basis for a number of possible implementations, but we have left the architectural details of such features for future work. Effective implementations of heavyweight monitoring could one day lead to self-patching software. Developing self-patching software will require a study of the structural characteristics that make up software bugs/vulnerabilities. Those characteristics and their fixes will need to be generalized so that they can be automatically identified and applied by our hardware module.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, Nov. 2005.

[2] O. Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture (CSAW)*, pages 11–18, 2007.

[3] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 312–320, 2007.

[4] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 194–206, 1991.

[5] D. J. Bernstein. Unix security holes. available at http://cr.yp.to/2004-494.html, Dec. 2004.

[6] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 90–99, 1991.

[7] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium (SSYM)*, pages 1–13, 2003.

[8] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[9] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[10] CERT. US-CERT cyber security bulletin sb04-357. available at http://www.us-cert.gov/cas/bulletins/SB04-357.html, Dec. 2004.

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 91–104, Aug. 2003.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, PerryWagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, pages 63–78, Jan. 1998.

[13] J. Crandall, S. F. Wu, and F. Chong. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):359–389, Dec. 2006.

[14] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, Oct. 2006.

[15] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.

[16] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–75, 2003.

[17] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 5–5, 2001.

[18] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, page 295, 2003.

[19] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, May 1998.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[21] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC)*, pages 3–14, Dec. 2001.

[22] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2,3):141–158, 2000.

[23] V. L. Kiriansky. Secure execution environment via program shepherding. Master's thesis, Massachusetts Institute of Technology, 2003.

[24] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, Nov. 2000.

[25] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee. A processor architecture defense against buffer overflow attacks. In *Proceedings of the International Conference on Information Technology*, pages 243–250, Aug. 2003.

[26] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology (CRYPTO)*, pages 218–238, 1989.

[27] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 284–295, June 2005.

[28] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, May 2002.

[29] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.

[30] H. Ozdoganoglu, T. Vijaykumar, C. Brodley, A. Jalote, and B. Kuperman. SmashGuard: A hardware solution to prevent security attacks on the function return address. Technical Report TR-ECE 03-13, School of Electrical and Computer Engineering, Purdue University, Nov. 2003.

[31] Y.-J. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE Micro*, 26(4):62–71, 2006.

[32] M. Prvulovic, Z. Zhangzy, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 111–122, May 2002.

[33] W. Schindler. A timing attack against rsa with the chinese remainder theorem. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 109–124, 2000.

[34] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[35] W. Shi and H.-H. Lee. Accelerating memory decryption and authentication with frequent value prediction. In *Proceedings of the 4th International Conference on Computing Frontiers (CF)*, pages 35–45, May 2007.

[36] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. *SIGARCH Comput. Archit. News*, 34(2):102–113, 2006.

[37] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 149–161, Apr. 2005.

[38] A. Smirnov and T. cker Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[39] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint / recovery. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 123–134, May 2002.

[40] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 84–96, Oct. 2004.

[41] G. E. Suh, C. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2005.

[42] R. Teodorescu, J. Nakano, and J. Torrellas. Swich: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro Magazine*, 26:28–40, Sept. 2006.

[43] R. Teodorescu and J. Torrellas. Prototyping architectural support for program rollback using FPGAs. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 23–32, Apr. 2005.

[44] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 209–220, Dec. 2004.

[45] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 156, 2001.

[46] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*, pages 149–162, Feb. 2003.

[47] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. *Computer Architecture News*, 31(2):122–135, 2003.

[48] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 351–360, Dec. 2003.

[49] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34, Apr. 2007.

[50] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous path detection with hardware support. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 43–54, 2005.

[51] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pages 72–84, Oct. 2004.

[52] X. Zhuang, T. Zhang, and S. Pande. Compiler optimizations to reduce security overhead. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 346–357, 2006.