Rollback and Huddle: Architectural Support for Trustworthy Application Replay

Jesse Sathre Joseph Zambreno
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, USA
{jsathre, zambreno}@iastate.edu

ABSTRACT

While research into building robust and survivable networks has steadily intensified in recent years, similar efforts at the application level and below have focused primarily on attack discovery, ignoring the larger issue of how to gracefully recover from an intrusion at that level. Our work attempts to bridge this inherent gap between theory and practice through the introduction of a new architectural technique, which we call *rollback and huddle*. Inspired by concepts made popular in the world of software debug, we propose the inclusion of extra on-chip hardware for the efficient storage and tracing of execution contexts. Upon the detection of some software protection violation, the application is restarted at the last known safe checkpoint (the *rollback* part). During this deterministic replay, an additional hw/sw module is then loaded that can increase the level of system monitoring, log more detailed information about any future attack source, and potentially institute a live patch of the vulnerable part of the software executable (the *huddle* part). Our experimental results show that this approach could have a practical impact on modern computing system architectures, by allowing for the inclusion of low-overhead software security features while at the same time incorporating an ability to gracefully recover from attack.

Categories and Subject Descriptors: B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance; D.4.6 [*Operating Systems*]: Security and Protection

General Terms: Security, Design, Reliability

Keywords: attack detection, checkpoint and rollback, buffer overflows, hardware support

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.

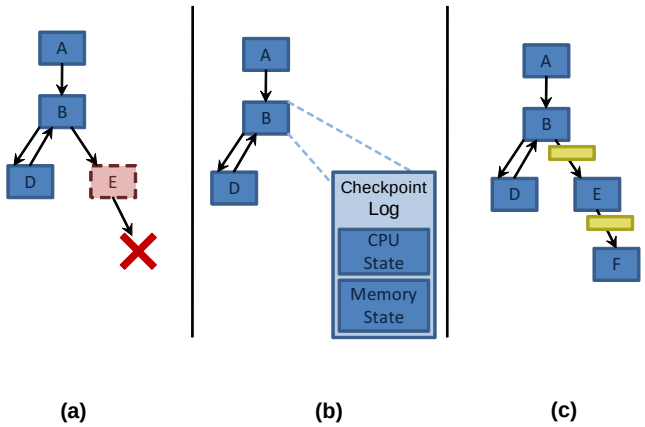

Figure 1: (a) An anomaly is detected in function E. Typical protection schemes cause the program to halt execution. (b) In our approach, the program is rolled back to a safe state by utilizing checkpoint logs. (c) The program restarts execution with extra safety checks in place.

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 paper we introduce a novel approach to software security which we call *rollback and huddle*. Figure 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, but applying it to the domain of soft-

ware 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. Once rolled back, 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 Section 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 sits as a memory mapped peripheral. 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 only 6.4% performance overhead to a variety of embedded benchmarks.

The remainder of this paper is organized as follows. In Section 2 we provide an overview of related research in the fields of hardware-supported checkpointing and software protection. Section 3 describes our conceptual approach in more detail. In Section 4 we outline the architectural features of our approach, and in Section 5 we present experimental results detailing the performance overhead of these features. Finally, the paper is concluded in Section 6 with a look toward future planned efforts in this project.

2. RELATED WORK

The computing research literature is filled with various approaches to detecting and preventing software-level attacks. Several of these focus on providing architectural support for encrypted execution and storage. In [9], the authors introduce the concept of eXecute-Only Memory, or XOM, which provides a mechanism for cryptographic separation of instruction and data-memory space. The AEGIS secure coprocessor [16] provides an implementation of physical random functions that can be used for assigning unique keys to an individual processor. The IBM 4758 secure coprocessor [4] is an earlier example of an architectural approach to software security. One aspect that makes our work unique is that we focus on system recovery after the initial point of detection.

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. SmashGuard [11] is one notable example, which adds hardware functionality to intercept function calls and returns in order to manage its own hardware stack. Another example can be found in [12], where the Return Address Stack (RAS) of a speculative processor is modified, increasing stack security and resulting in less than a 1% performance overhead. 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 [19].

The general threat model that we consider is similar to that found in [1] and other works that check the integrity of program flow and call-graphs formed during program execution. In [1, 3], program flow integrity is considered at the basic block level. As will be explained in Section 3, our lightweight protection scheme expands this granularity to the function call scope in order to incur less overhead. A separate approach [5, 18] for enforcing program flow, called *whitebox training*, involves analyzing source code to determine an acceptable pattern of function and system calls. A variation of whitebox training is *blackbox* training [7, 14], which studies the patterns of a normally executing program to form its acceptable call graph. Hybrid examples exist as well [6].

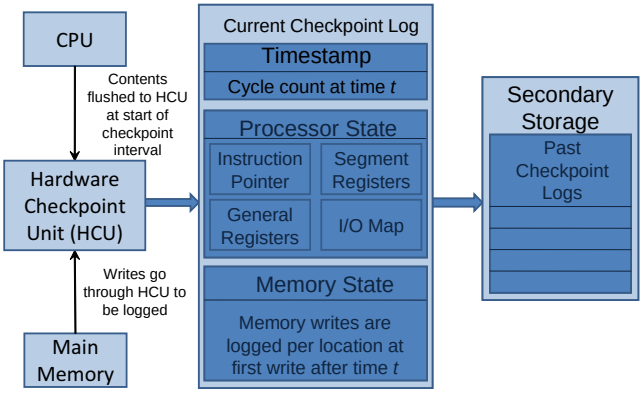

Figure 2: Contents of a typical checkpoint log. Checkpoint logs are identified by a timestamp and contain the necessary information to restore a program's state.

Checkpointing and rolling back program execution is not a new concept, although it has received some recent attention from the computer architecture community who have applied it toward fault tolerance and debugging. Our checkpointing scheme, as will be described in the following section, is loosely based on [20] which itself can be traced back to the scheme found in [15]. The former was aimed at creating a debugging environment for crashed programs while the goal of the latter was to achieve fault tolerance in shared memory multi-processor systems. Other examples of hardware-assisted checkpointing and rollback can be found in [13, 17]. What sets our approach apart from these is the domain in which we apply our checkpointing (software security), and the goal for which we roll back (to obtain a more trusted state).

3. CONCEPTUAL APPROACH

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

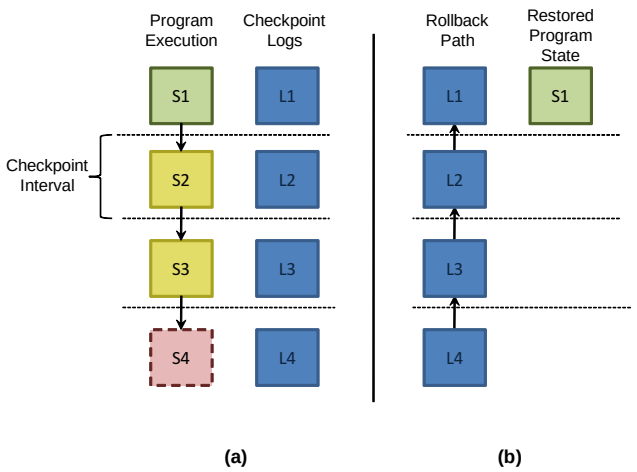

Figure 3: (a) As a program executes, its state is checkpointed at normal intervals until an anomaly is detected at S4. (b) The checkpoint logs are cycled through until the system is restored to a safe point at S1.

attack-detection mechanism. Our approach is not limited to one specific type of attack detection scheme. In this paper 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 two addresses do not match, 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 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 [20] and [15]. 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*.

Figure 2 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

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

Figure 4: An algorithm for determining rollback distance.

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. Because our goal is to restore the state of the program to the beginning of the checkpoint interval, we only need to log the initial value for each checkpoint 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.

Checkpoint logs are a fixed size. A new checkpoint interval begins once the memory state buffer of a checkpoint log is filled. When this occurs, the current checkpoint log is offloaded to secondary storage. A new checkpoint log is created by saving the processor state and recording the timestamp. The length of checkpoint intervals are determined by the frequency of memory writes.

An alternative policy would be to store a checkpoint after a fixed number of CPU cycles. Checkpointing based on CPU cycles would allow for more regularly spaced checkpoint intervals, especially in programs that make relatively few memory writes. The downside of this approach is that the constant size of checkpoint logs can no longer be guaranteed.

3.3 Detection and Rollback

Program rollback occurs when the lightweight protection scheme has signaled that an attack has taken place (S4 in Figure 3). The final goal of the rollback stage is to restore the program's state to a "safe" point before the attack occurred and allow re-execution in a secure state (S1). This is made possible by utilizing the checkpoint logs saved during normal execution (L1–L4).

One method for determining how far to rewind execution is described in Figure 4. 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.

When rollback begins, program execution is temporarily halted

and the on-chip caches are flushed. The first stage of program rollback involves 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 "undoes" 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. At this point execution resumes from the rollback point in a heavyweight monitoring mode.

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 with no recovery mechanisms.

3.4 Heavyweight Monitoring

Once a program has 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. 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 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 [10]. We make 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 to allow secure execution to last until program termination. This policy offers maximum security, but at the cost of an increased 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 another 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 added security.

4. ARCHITECTURAL FEATURES

Our approach can make use of several architectural features to operate efficiently. Minimal changes to conventional computing platforms are required to accommodate the hardware of our approach. Figure 5 shows how our supplemental hardware fits in with existing architectures. Added hardware includes a Hardware

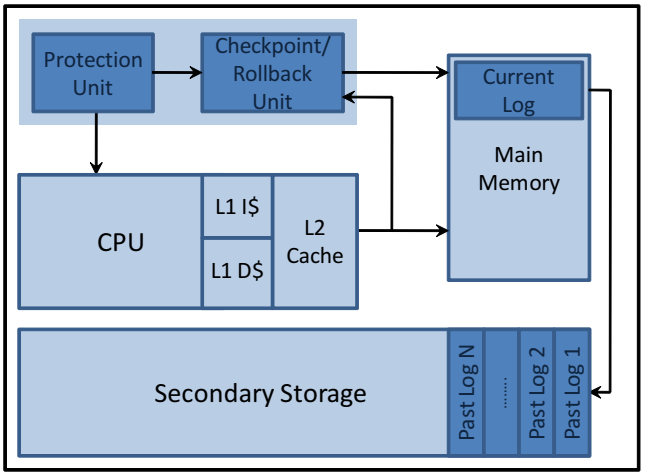

Figure 5: High-level architecture view with protection and checkpoint / rollback hardware.

Checkpoint Unit (HCU) and a Lightweight Protection Unit (LPU), each with a small amount of logic and internal storage. Not shown in Figure 5 is the Heavyweight Protection Unit (HPU), which depending on the required functionality may be more tightly coupled with the target processor.

The HCU has direct access to main memory as well as the CPU/main memory bus. The HCU contains a small amount of content-addressable memory alongside some standard memory. The content-addressable memory is used to store the addresses of memory locations that have been written during the current checkpoint interval. Content-addressable memory is preferred in this situation to allow for quick look-ups on memory writes. The standard memory is used to store pointers to the old checkpoint logs that have been offloaded to secondary storage. The current checkpoint log resides in a reserved section of main memory. A fixed number of past checkpoints are also retained to allow rollback beyond the current checkpoint interval. These past checkpoints can stay in main memory or be offloaded into secondary storage depending on the size of the checkpoint logs and availability of main memory.

The HCU snoops the bus on the data path between the CPU and main memory. When the CPU writes data values to main memory, the HCU checks its table to see whether that memory location has been logged for the current checkpoint interval. If it has not, the existing value in memory is copied to the current checkpoint log before the new value is written. Once the current checkpoint buffer is filled, it is offloaded to secondary storage. A pointer to the checkpoint log is retained in the HCU to allow for retrieval in the case of a rollback.

Memory location granularity can vary by implementation. Logging memory on a per-byte basis creates a large amount of storage overhead to keep track of the location specifier. An optimization would be to log memory writes at the cache-block level. Increasing the size of what constitutes a unique location will decrease the storage overhead required to identify each location. However, there is a tradeoff in selecting the proper size. Very large location sizes coupled with a lack of write locality will create very frequent checkpoints and as a result it will increase run-time overhead.

This scheme for logging memory assumes a write-through cache policy is in place. This guarantees that the memory log contains all writes to cache. If a write-back policy is used, additional checkpointing will be needed to log the cache state. Cache would be

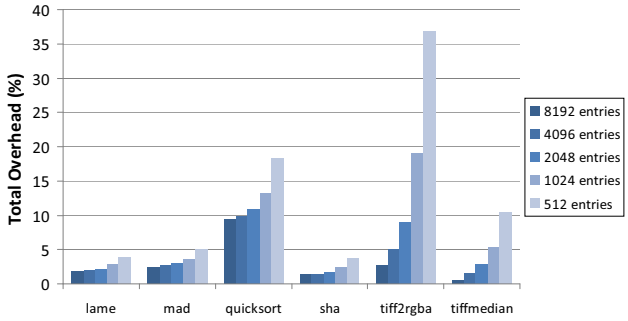

Figure 6: Total overhead as a function of memory log size.

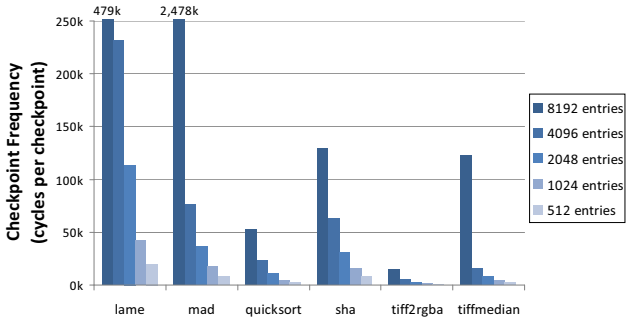

Figure 7: Checkpoint frequency as a function of log size.

logged similarly to how main memory is currently logged using the HCU. The pre-existing cache values would be logged on the first write of the checkpoint interval.

Program rollback also makes use of the HCU. The HCU cycles through and reads each value in the current checkpoint log, writing the values back into the specified memory locations. Once the current checkpoint log has been rolled back, the next checkpoint log is loaded into the storage reserved for the current checkpoint log. This process continues until the rollback is complete. Once a checkpoint has been rolled back it is no longer needed and can be discarded. In this case, it is simply overwritten when the next checkpoint is brought into main memory.

Located adjacent to the HCU and CPU, the LPU consists of its own storage and a controller. To implement the lightweight scheme described in the previous section, the storage of the LPU acts as a secondary stack to hold return addresses on function calls. When a return address enters the LPU on a function call, the controller appends it with a timestamp, and then this data is pushed onto the HW stack contained within the LPU. The timestamps appended to the return addresses are meant to identify when an attack takes place. When a return address enters the LPU on a function return, the controller pops the top of the stack and compares its value to the return address entering the LPU. If a mismatch occurs, the LPU sends an interrupt to the CPU so that execution is halted and rollback begins.

For a program to take advantage of the LPU it must be compiled with express knowledge of the LPU location. Our modified `gcc` compiler adds instructions to push the expected return address onto the LPU's hardware stack before function calls. On function returns, our compiler adds instructions to pop a value off of the LPU's stack and verify that the address on the top of the stack matches the address being returned to. We do not introduce new instructions to accomplish the LPU operations. Instead, the LPU push and pop operations are accomplished through memory-mapped writes. One address, *&PUSH*, is used to specify a push operation while a second address, *&POP*, is used to specify a pop operation. If the LPU receives a write at the *&PUSH* address, the controller pushes the written value onto its internal stack. If the LPU receives a write at the *&POP* address, the controller pops the top value off of its stack and compares it to the value written to the *&POP* location. If there is a mismatch, the LPU sends an interrupt to the CPU and signals for a rollback to begin.

5. EXPERIMENTAL RESULTS

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 [2], 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 [8]. As an initial investigation into the performance impact of our approach, we compared the total overhead, checkpoint frequency, and maximum rollback distance for various configurations of the HCU.

Figure 6 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 dependant. Because of this, the overhead of the lightweight scheme is constant across all configurations for a given benchmark. Figure 6 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 4.87%.

Figure 7 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 6 and 7. 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.

Figure 8 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

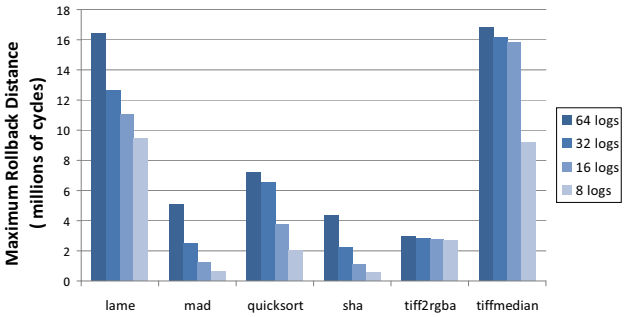

Figure 8: Rollback distance as a function of total stored logs.

to be checkpointed as frequently.

One general observation is that the checkpoint frequency has a large impact on the total performance overhead. This frequency, in turn, is a function of an application's data locality and write access frequency. Our architecture can restore program state from upwards of tens of millions of cycles. This provides hope that even complex program flow attacks can be thwarted.

6. CONCLUSIONS

In this paper we have presented a new approach to security at the application level that can detect as well as gracefully recover from attacks. Initial experimental results show that these goals can be achieved with minimal run-time performance overhead (less than 7% on average). While this paper focused mainly on lightweight monitoring and the LPU, in the future we are looking to develop new architectural features that may enable heavyweight monitoring using the proposed HPU. We also intend to explore new protective mechanisms that fit within our scheme. Finally, we hope to prototype our architecture in FPGA hardware in order to perform real-time experiments.

7. REFERENCES

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, pages 318–329, Oct. 2004.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.