# Architectural Support for Automated Software Attack Detection, Recovery, and Prevention

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

***Abstract*—Attacks on software systems are an increasingly serious problem from an economic and security standpoint. Many techniques have been proposed ranging from simple compiler modifications to full-scale re-engineering of computer systems architecture aimed at attack detection. Traditional techniques ignore the arguably more important problem of graceful recovery. Without recovery, even a successful attack detection can become an effective Denial-of-Service. We propose an architectural approach to attack detection and recovery called *rollback and huddle* that monitors a program's execution with a lightweight attack-detection module while continuously checkpointing the system state. In the case of an attack, the program state is rolled back to a time before the attack occurred and an additional module is loaded to identify the source of the attack, repair the original vulnerability, and prevent future attacks. The simple hardware modules work alongside a standard computer architecture and aid in attack detection, checkpoint creation, and attack recovery. Experimental results show minimal runtime overhead and resource utilization.**

## I. Introduction

Significant financial damage has been caused by Trojans, worms, and other varieties of malware. At the root of most malware is a software vulnerability such as a overflowable buffer. In extreme cases, an attacker can gain complete control of a system over the network without having physical access to the user's machine. In the traditional software cycle, patches are released retroactively for vulnerable software. Not only does this miss the initial damage caused by an exploit, but it can be difficult to ensure compliance in applying the patches to the software. In the end, this methodology leads to a back-and-forth struggle between software vendors and attackers.

Researchers have attempted to detect and prevent attacks against vulnerable software with varied approaches, ranging from simple compiler modifications to fundamental changes in processor architecture. While many of these schemes have shown to be effective at detecting attacks, they commonly terminate execution of the vulnerable program to prevent further progress of the attacks. By terminating execution of the attacked service, an effective Denial-of-Service (DoS) is created. While this is sufficient in some situations, a deployed embedded system lacking the ability and feasibility of direct human interaction, requires an effective recovery scheme in order to harden itself from attacks.

Our work attempts to bridge the gap between the traditional software patching model and current protective schemes. In our approach, *rollback and huddle*, we combine attack detection and system checkpointing to create an environment that avoids system down-time in the case of an attack and instantly patches vulnerable software, thus preventing future attacks. Figure 1 illustrates the basic concepts of our system. A lightweight security mechanism continuously operates with minimal performance overhead and periodic checkpoints are recorded during execution that allow a program's state to be gracefully recovered. If this initial scheme detects that an attack has occurred, the program is "rolled back" to a previous state and the software is inspected for vulnerabilities. Vulnerable instruction sequences are replaced by secure equivalents (the "huddle" part), effectively patching the compromised program and preventing repeat attacks.

As will be explained in Section IV, 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 Lightweight Protection Unit (LPU), which performs function-level verification. A Heavyweight Protection Unit (HPU) stores a number of generic instruction sequences which it uses to create specific patches for vulnerable programs after rollback. Our 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 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 and their effectiveness against real world vulnerabilities. Finally, the paper is concluded in Section 6 with a look toward planned future efforts in this project.

## II. Related Work

Work related to our approach falls into two distinct categories: attack detection schemes in the security domain and checkpoint and rollback schemes in the fault tolerance and

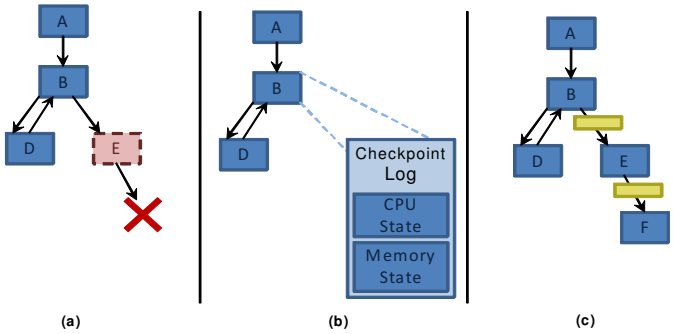


Fig. 1. (a) Typical schemes cause an anomaly in function E to halt the program.(b) Our approach rolls back the program to a safe state. (c) It restarts execution with extra safety checks in place.

software debugging domain. Our approach meshes ideas from both domains to provide a more resilient protection.

### A. Attack Detection and Prevention

**Encrypted Encryption:** Several approaches focus on architectural support for encrypted execution and storage. [10] introduces eXecute-Only Memory, or XOM, which provides a mechanism for cryptographic separation of instruction and data-memory space. Yang et al. [25] introduce a more efficient implementation of XOM by moving the encryption process off of the critical path. PointGuard [3] is a compiler-based approach that encrypts all pointers in memory, later improved by Tuck et al. [21] by providing hardware support to reduce overhead and by increasing the complexity of encryption to extend security features. An in-depth discussion of the benefits and limitations of encrypted execution platforms can be found in [27].

**Instruction and Data Flow Enforcement:** A common theme in preventing software attacks is the use of a modified or secondary stack to enforce a security policy. For example, SmashGuard [14] uses additional hardware functionality to intercept function calls and returns to manage its own hardware stack. In [15], the Return Address Stack (RAS) is modified to provide a software-transparent defense against buffer overflows. While these stack-based approaches have been shown to be insufficient to protect against all attacks [23], the low performance overheads and modest hardware requirements make hardware stack protection an ideal candidate for a lightweight protection scheme in our approach. Software techniques such as StackGuard [4] and CFI [1] can also be effective but their performance overheads tend to be considerably higher.

Many software attacks originate in data from spurious sources which disrupt program flow. Minos [5] is an architectural approach that tags input from untrusted sources with an integrity bit that guards program flow accordingly. A similar approach is presented in [20] where the operating system is modified to tag I/O from spurious sources. Program Shepherding [9] dynamically instruments programs to enforce security policies which preserve the intended flow of the program. A related family of approaches exist that rely on static analysis to enforce program flow [7], [8], [17], [22]. While our approach shares little in common with these static schemes, we follow a similar security model for enforcing program flow.

### B. Checkpoint 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 caused by an attack. While the concept of checkpoint and rollback began in the domains of fault tolerance and debugging, it is receiving more attention in the software security and attack recovery domains.

**Fault Tolerance and Software Debugging:** Our checkpointing scheme is loosely based on FDR [24] which itself can be traced back to SafetyNet [19]. SafetyNet was designed to handle faults in shared-memory multiprocessor systems by using hardware checkpoint logs in the processor's cache as well as main memory to log writes. FDR expanded the logging found in SafetyNet to include system I/O, DMA transfers and memory races, and added LZ77 hardware for compression. Also stemming from FDR is BugNet [13] which logs program execution so that it can be deterministically replayed off-line once a bug in encountered. ReVive [16] is an approach to multi-processor fault tolerance, where in addition to logging, memory is supplemented with parity information to recover from a loss of data. Recent innovations in this area have focused on replay efficiency [11], [12].

**Security and Attack Recovery:** ExecRecorder [6] is a Virtual Machine-based checkpointing scheme with a log-based recovery mechanism in conjunction with an intrusion-detection system and post-attack analysis tools. Like our approach, replay is signaled by an attack-detection mechanism, but where ExecRecorder operates at the Virtual Machine layer and uses off-line replaying for attack analysis, our approach works with the native architecture and uses on-line rollback.

DIRA [18] is an approach similar to our idea of heavyweight protection. In DIRA, execution is checkpointed and rolled back after attack detection attempting to identify the source of the attack and repair itself. However, it suffers from substantial run-time overhead in many benchmarks. In our approach, we avoid the additional overhead by implementing the identification and repair mechanisms only after an attack has been detected.

## III. Conceptual Approach

In this section, we present a high-level overview of our approach consisting of three parts: Lightweight Protection Unit (LPU), Hardware Checkpoint Unit (HCU) and Heavyweight Protection Unit (HPU). Figure 2 shows the interaction between traditional hardware units and our augmentation while later sections cover implementation details in more depth.

### A. Lightweight Protection

Initially, a lightweight attack detection mechanism monitors program execution. This is not dependent on one specific

method for attack detection and in practice many of the schemes mentioned in Section II-A could provide its basis. Our emphasis during this phase is on minimizing performance overhead and protecting against the most common forms of attacks. Considering this, we will focus on a lightweight protection mechanism similar to StackGuard [4] and its evolutionary replacement, PointGuard [3]. At process creation time, a random key is generated and subsequently used to encrypt and decrypt function return addresses during each function call and return. Even if an attacker overwrites the return address, the decrypted value of the overwritten return address cannot be predicted. This will cause the function to crash when it tries to return to a random location. In our approach, we use this as a sign that an attack has occurred signaling the system to go into rollback mode.

### B. Checkpoint and Rollback

In order to recover from the initial attack, execution is checkpointed so that the compromised program can revert its state to a point in time before the attack occurred. A checkpoint is the precise system state including the CPU and memory state at a specific point in time. A checkpoint interval is the execution time between checkpoints. Longer checkpoint intervals lead to greater rollback lengths, but a finer granularity of error can be corrected with shorter, more "precise" checkpoint intervals. The balance between the two is implementation specific and our scheme follows that of SafetyNet [19].

Both the CPU state and memory need to be saved to construct an accurate system snapshot. At the beginning of the checkpoint is the CPU state consisting of register values and the program counter (PC). Memory writes cause the value being overwritten to be logged the first time that a memory address is written during a checkpoint interval so that the state of the program can be restored to what it was at the beginning of the checkpoint interval. The current checkpoint interval ends when the memory log is filled.

Rollback, the inverse of checkpointing, occurs when the lightweight detection scheme detects an attack. During rollback, the damage of the attack is nullified and the program restored to a safe state before the attack occurred. Rolling back a single checkpoint interval is performed by writing back the memory values in the checkpoint log. Subsequent checkpoint intervals are rolled back iteratively until the log containing the compromised return address has been rolled back. Finally, the register values of the CPU are restored to the CPU state saved in the last rolled back checkpoint log.

In our approach, each process is monitored and checkpointed separately so that a compromised process will not disrupt other processes. This is in contrast to many checkpoint and rollback recovery mechanisms that do full-system recording of all the active processes and system I/O which are later rolled back together. A full-system recording model is appropriate for debugging purposes but may be overzealous when considering application attack and rollback scenarios.

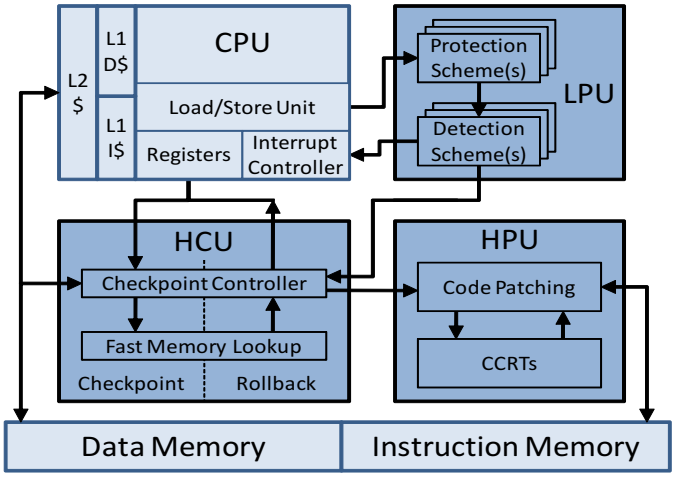


Fig. 2. Architectural overview

### C. Heavyweight Monitoring

Once a program has been rolled back to a safe state, the attack can be repeated if no changes are made to the executing program. This leads us to the final phase of our approach: *heavyweight monitoring*.

Many software vulnerabilities follow similar patterns such as iterating over a buffer without bounds checking or using unsafe library functions such as `strcpy` that do not consider source and destination sizes. Heavyweight monitoring identifies the root of the attack and replaces the vulnerable instruction sequence with a safe version that may, for instance, perform bounds checking before the buffer is written.

Code replacement is performed through *Configurable Code Replacement Templates* (CCRTs) which are parameterized patch templates used to create a specific patch. Modifying a program's binary to incorporate these patches is a non-trivial task. Safe instruction sequences are larger than their unsafe counterparts and cannot simply replace existing instructions. Instead, the patch is appended to the end of the program's binary and the unsafe instruction sequence is replaced with a jump to the safe sequence. Once the safe sequence is executed, the program jumps back to the instruction following the unsafe sequence and resumes execution.

### D. Limitations

Currently the issue of gracefully handling I/O with regards to checkpointing and rollback is an open problem due to the on-line nature of replay in our scheme. Approaches that attempt to provide off-line deterministic replay of execution must log I/O. However, the goal of our checkpointing scheme is not to provide deterministic replay; instead, rollback is used to nullify the damage of an attack. Presently, we follow the latter model and choose not to log I/O directly, but further studies into the effects of logging and rolling back I/O are left for future work.

Our approach assumes a single-core processor with a write-through memory model. Accommodating a write-back cache requires additional logging at the cache level, similar to what is

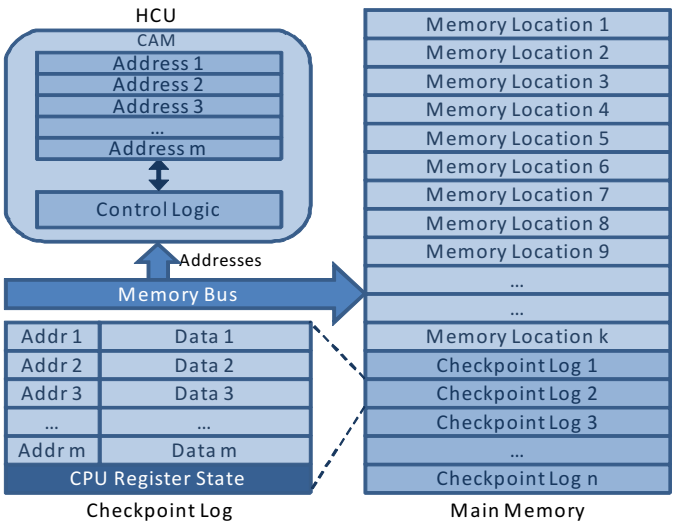


Fig. 3. The CAM within the HCU contains $m$ entries for the $m$ current checkpoint memory log entries and $n$ memory logs are stored in main memory.

used for logging main memory. Likewise, supporting a multi-core processor requires additional logging mechanisms to store memory race conditions between cores. Both of these cases are handled in [19] and [24] but their application to our research is left for future work.

## IV. Architectural Implementation

In this section, we outline the implementation details of the additional architectural features that enable rollback and huddle. Hardware modules include a Lightweight Protection Unit (LPU), Hardware Checkpoint Unit (HCU), and Heavyweight Protection Unit (HPU). These additional hardware features are non-intrusive and integrate into a standard computer architecture. Our hardware is designed to minimize the performance impact during standard execution, as well as minimize the total storage footprint required for checkpointing, both of which aid in the constant resource usage constraints of embedded systems.

### A. Lightweight Protection Unit

The Lightweight Protection Unit (LPU) is based on the design introduced in [21] which implements a hardware version of PointGuard [3]. Two instructions are added to the ISA: an encrypting store and decrypting load that XOR the random key generated at process creation with the return address being stored or loaded. This approach requires recompilation or binary modification of the target application to augment `call` instructions with the encrypting store and `return` instructions with the decrypting load instruction. Meanwhile each time the LPU decrypts a return address, it stores the encrypted address until the next decryption. If an attacker attempts to overwrite the return address, the program returns to a random location since the unencrypted return address is unencrypted to a garbage value. When this happens, an exception is thrown, the system is stalled and the LPU passes the original encrypted address to the HCU so that it can determine how far to rollback execution. This process is detailed in Section IV-B.

In this approach, an attacker cannot read the LPU's key directly, however, if an adversary is able to exploit a "read attack" addressed in [21] it is trivial to deduce the key and modify the attack accordingly. We do not defend against such attacks, but a more secure encryption could be substituted for the XOR in order to provide a better defense at the cost of increased runtime overhead.

TABLE I
Simulator Configuration

| | |
|---|---|
| Functional Units | 2 ALU + 2 FPU<br>2 Load / Store Units |
| Pipeline | 64-bit, 11 stages |
| ROB | 128 entries |
| L1 I-Cache | 32 KB, 4-way set associative<br>1 cycle latency |
| L1 D-Cache | 16 KB, 4-way set associative<br>1 cycle latency |
| L2 Cache | 256 KB, 16-way set associative<br>6 cycle latency |
| L3 Cache | 4 MB, 32-way set associative<br>16 cycle latency |
| Cache Block Size | 64 bytes (8 words) |
| Main Memory | 140 cycle latency |

### B. Hardware Checkpoint Unit

The Hardware Checkpoint Unit (HCU) provides the system with checkpoint and rollback capabilities. 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 a modified memory controller. It is designed to optimize the relatively common case of logging memory addresses.

Figure 3 shows the HCU with a high-level view of how logging works. The HCU itself consists of a basic logic controller and internal memory. The HCU logs memory writes for the current checkpoint interval while keeping past checkpoints in a circular buffer of reserved memory such that new checkpoints overwrite the oldest checkpoint. The internal memory, a Content Addressable Memory (CAM), must be large enough to store each address of the current memory log. It is used to store the memory addresses that have been logged during the current interval and is flushed when a new checkpoint is created. The CAM can quickly check if an address has been logged during the current checkpoint interval for each write to main memory. In this way, the address is stored in both the memory log for rollback and the CAM for quick access.

The HCU contains simple control logic to keep track of the current checkpoint log. When the memory log is filled, the end of the current checkpoint interval, the HCU stalls the CPU, flushes the CAM and finds the location of the next checkpoint log in memory. The fixed size of checkpoint logs allows the HCU to find the next checkpoint log without explicitly storing its location. The controller saves the register state of the CPU

TABLE II
MAXIMUM CYCLES (MILLIONS) ROLLED BACK

| Entries<br>Checkpoints | 512<br>8 | 256<br>16 | 128<br>32 | 64<br>64 | 32<br>128 |
|---|---|---|---|---|---|
| gzip | 3.913 | 3.807 | 3.752 | 3.683 | 3.540 |
| vpr | 198.0 | 1.130 | 0.824 | 0.638 | 0.534 |
| gcc | 11.68 | 6.980 | 6.176 | 5.008 | 4.070 |
| mcf | 0.961 | 0.953 | 0.943 | 0.938 | 0.820 |
| crafty | 12.71 | 10.94 | 7.737 | 3.328 | 0.846 |
| eon | 269.5 | 52.39 | 19.50 | 1.122 | 0.599 |
| bzip2 | 25.01 | 24.40 | 23.66 | 21.76 | 18.23 |
| twolf | 1.527 | 1.462 | 1.366 | 1.227 | 1.106 |
| perlbmk | 12.40 | 10.12 | 6.241 | 1.396 | 1.146 |
| parser | 10.21 | 9.063 | 6.912 | 4.588 | 2.763 |
| gap | 5.927 | 5.070 | 3.805 | 2.286 | 1.342 |
| vortex | 9.986 | 6.457 | 3.198 | 2.818 | 1.980 |

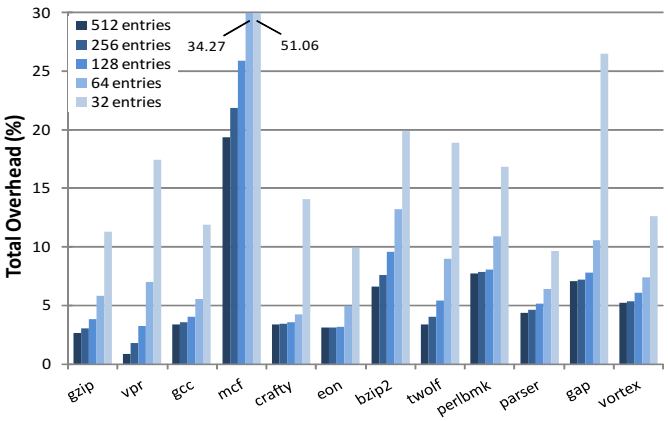


Fig. 4. Total overhead

in the current checkpoint log and then allows execution to resume.

Various memory parameters can be tuned for a variety of metrics. First, the memory log granularity can be as fine as the byte level leading to greater storage overhead or as coarse as the memory-page level leading to lesser log utilization. Two realistic choices are logging at the word level or at the cache-block level. Low log utilization can lead to more frequent checkpoints and thus decrease performance due to excess checkpoint creation. We choose to focus on logging memory at the cache-block level in order to keep the CAM in the HCU at a reasonable size.

Secondly, the size of each checkpoint log and the number of checkpoint logs can vary. A larger memory log with more entries will allow for a longer checkpoint interval and decrease the performance penalty due to checkpoint creation. But, not only does the larger memory usage incur a greater cost, it also reduces the precision of rolling back to a particular point.

When an attack is detected, the HCU enters *rollback mode*. The LPU sends the HCU the encrypted value of the last return address that was used. Starting with the current checkpoint log, the HCU writes back the memory values from the memory logs continuing until it finds the failed return address. When the HCU completes the memory write-back process, it restores the CPU register state to that of the final checkpoint log. 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.

### C. Heavyweight Protection Unit

The final piece of hardware, the Heavyweight Protection Unit (HPU), is used to patch a compromised program after it has been rolled back. The HPU is connected to the system's main memory bus and must have write access to a program's instruction space so that it can perform the additional instrumentation required to repair an attack.

The HPU is composed of a logic controller and a small amount of storage for the lightweight CCRTs, which are used to patch unsafe code sequences as described in Section III-C. Preliminary CCRT designs consist of only 5-10 instructions and take up 30-100 bytes of storage apiece. Although we are still investigating all of the possibilities for CCRTs, a mere 128 KB of HPU storage would be sufficient for storing approximately 1,000 different CCRTs. Potentially, the HPU could be extended to include additional CCRTs after deployment, but this extra functionality is left for future work.

Once rollback has taken place, the HPU receives a signal from the HCU that the program must be instrumented. The HPU scans the program's code segment for unsafe instruction sequences, identifies the vulnerable instructions, infers the size of the buffer being written and constructs a specific patch from a CCRT. The HPU's patch remains in place permanently or until a proper patch is deployed.

## V. EVALUATION

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 [26], a cycle-accurate 64-bit x86 simulator. Simulation was performed using PTLsim's out-of-order core model with realistic branch prediction and cache behaviors. The simulator configuration is shown in Table I.

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 and 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 to log memory writes at the cache-block level with a conservative memory model that transfers 8 bytes 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 location before the current logging transaction is complete, it must stall the pipeline until logging is finished.

We used the SPECint subset of the SPEC2000 benchmarks suite to simulate 200 million instructions on our modified architecture. As an initial investigation into the performance impact of our approach, we compared the total overhead,

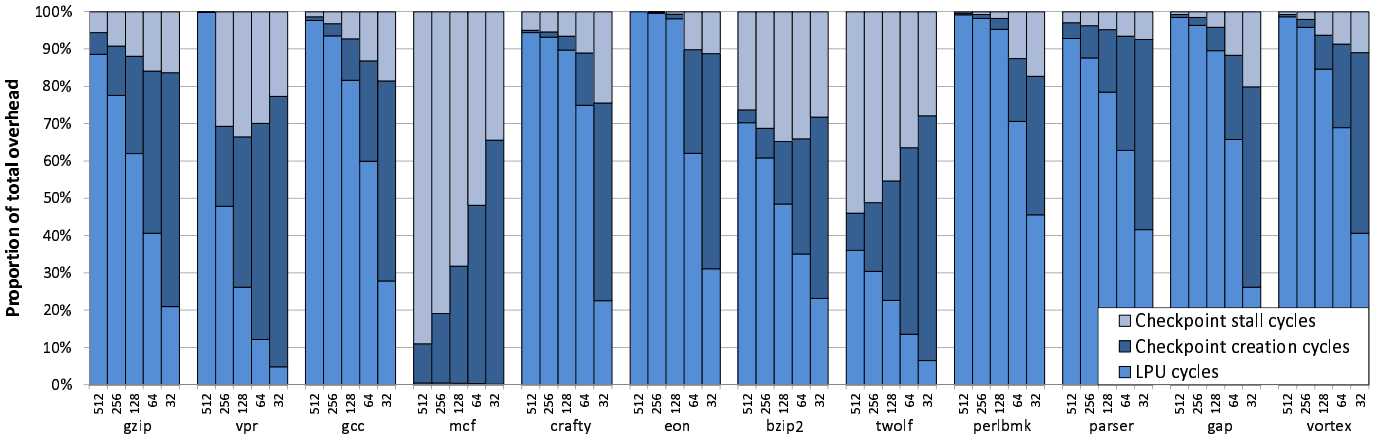


Fig. 5. Breakdown of overhead sources for different HCU configurations.

maximum rollback distances and detection effectiveness for various configurations of the HCU.

### A. Performance Overhead

Figure 4 shows the total overhead for a number of HCU configurations. We varied the size of the memory logs from 512 entries down to 32 entries where each entry consists of a cache block containing eight 64-bit words, along with a 64-bit address identifier. The size of the HCU was also kept constant at 4096 total memory log entries across all configurations which requires approximately 295 KB of total storage. This meant that fewer checkpoints were held in storage for configurations with larger individual checkpoints. Overhead during the lightweight monitoring phase of our approach comes from three sources: lightweight protection, checkpoint creation, and HCU busy stall cycles. The overhead related to the LPU is constant across all configurations since 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 logs 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 V-B we provide analysis into how each of these sources contributed to the total overhead experienced.

As expected, Figure 4 shows that the larger memory log configurations perform better than the smaller memory log configurations. The 32-entry configuration performed dramatically worse than the other four configurations. Across many benchmarks, the performance impact of log size begins to level off at 128 entries with a 9.43% average performance impact across all benchmarks and all configurations. If we remove the poor performance of the 32-entry configuration, that average drops to 7.20%. Excluding both the 32-entry and 64-entry configurations, all benchmarks stayed below the 10% mark and many of them did not exceed 5% overhead. The one exception was `mcf`, which experienced a 19% slowdown for its 512-entry configuration and more than 50% slowdown with the 32-entry configuration. The unusually large amount of overhead was due to its tendency to write to many unique memory locations which stressed the HCU. Running `mcf` for 200 million cycles, the 512-entry configuration created 19,189 checkpoints. Many more than the others benchmarks with the closest, `twolf`, creating 3,247 checkpoints.

### B. Overhead Breakdown

Figure 5 shows a breakdown of how each of the overhead sources contributed to the total overhead in each of the SPECint benchmarks for the various HCU configurations. As mentioned in Section V-A, LPU overhead is constant for a given benchmark regardless of HCU configuration because the 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 HCU-related overhead due to the increased logging frequency. Conversely, configurations with larger memory log sizes have a larger proportion of overhead due to the LPU.

Looking at the 512-entry configuration, overhead related to checkpoint creation is negligible compared to LPU overhead, contributing 4% and 81% respectively. HCU stalling is also kept to a minimum in this configuration, accounting for 15% of the slowdown. The effects of LPU stalling are negligible across all benchmarks, with the exceptions of `mcf`, `bzip2`, and `twolf`. The `mcf` benchmark is the clear outlier, with over 90% of its slowdown resulting from HCU stalling. This shows that the HCU simply cannot keep up with the large volume of unique memory writes and the subsequent logging load. It should be noted that the total overhead across all benchmarks for this configuration was approximately 5%, which means that the LPU did not negatively affect performance as much as this figure may imply.

The 128-entry configuration in Figure 5, shows that the LPU still accounts for the largest portion of overhead at 65%, but checkpoint creation begins to have a more noticeable effect, accounting for 19% of the overhead. HCU stalling remains at 16% when averaged across all benchmarks. Once again `mcf` suffers the most from HCU stalling, but more benchmarks

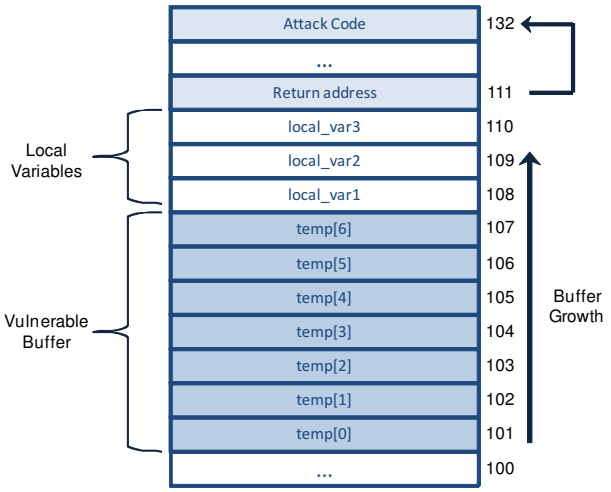


Fig. 6. Stack with a buffer overflow vulnerability

are noticeably affected by HCU stalling than in the larger configurations.

The 32-entry configuration shows dramatically different results than the 512-entry configuration in Figure 5. With the smaller checkpoint log size, slowdown is dominated by checkpoint creation, accounting for 56% of the total overhead, while LPU overhead falls to only 24% for this configuration. Similar to the larger configurations, HCU stalling accounted for 20% of the overhead. Although the overall impact of HCU stalling is similar when averaged across all benchmarks,the distribution of that average has nearly evened out across all benchmarks. This contrasts with the results of the 512-entry configuration, where HCU stalling was isolated to a small number of benchmarks. When comparing the results of Figure 4 and Figure 5, it becomes clear that the poor overall performance of the smaller configurations is due to the HCU and the increase in checkpoint creation frequency.

### C. Rollback Distance

Despite storing the same number of total checkpoints in every configuration, many of the benchmarks are very sensitive to changes in the HCU configuration. Table II compares the maximum rollback distances for the same five HCU configurations used in Section V-A. We observed a large amount of variation in the maximum rollback distances between benchmarks. For instance, the 512-entry configuration was able to rollback over 250 million cycles of the `eon` benchmark, but could rollback just under 1 million cycles of the `mcf` benchmark. Also, the 32-entry configuration of the `eon` benchmark can only rollback 1% of the cycles of the 256-entry. In each case, the sensitivity stemmed from an even distribution of memory writes and a constant rate of checkpoint creations.

### D. Attack Analysis

The stack segment for a function `F` is shown in Figure 6. 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.

TABLE III
VULNERABLE PROGRAM ROLLBACK CYCLES

| Application | Cycles to detection (thousand) | Cycles to overflow (thousand) | Buffer size (bytes) | Program Recovery Possible? |
|---|---|---|---|---|
| villistextum | 6,745 | 6,130 | 32,768 | Yes |
| rngtonetools | 3,640 | 95 | 1,024 | Yes |
| csv2xml | 1,841 | 119 | 1,000 | Yes |
| 2fax | 728 | 3 | 256 | Yes |
| bsb2ppm | 665 | 86 | 1,024 | Yes |
| jpegtoavi | 433 | 99 | 4,096 | Yes |
| o3read | 305 | 92 | 1,024 | Yes |

We tested our approach by using PTLsim to simulate our hardware against eight vulnerabilities found in real-world applications in order to evaluate its effectiveness. All of the exploits contain an buffer overflow vulnerability and are found in open source programs written for a Unix environment as published in US-CERT Cyber Security Bulletin SB04-357 [2] The results are summarized in Table III.

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 return address or the point of attack-detection. Table III shows the results of these tests where the cycle count ranged from 304,864 in `o3read` to 6,745,763 in `villistextum`. We then verified that our approach would be able to rollback to the point of malicious input. Using a 32-entry HCU with a maximum of 128 checkpoint logs the HCU contained sufficient information to rollback to the time the input was received in all of the tested vulnerabilities.

After verifying that our approach was able to recover from each attack, we attempted to quantify the time required to overflow the vulnerable buffers. 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 Figure 6 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 III. The time required to overflow a buffer is related to the size of the buffer as well as the additional computations taking place during the vulnerable loop. The buffer in `2fax` is only 256 bytes and 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.

## VI. Conclusions and Future Work

We have presented a new approach to security that puts equal emphasis on attack detection and recovery. By combining detection and recovery our approach provides a higher level of reliability by restoring the system to a working order instead of terminating on detection. Since the heavyweight protection scheme is only used once an attack has occurred, our scheme is lighter on resource usage making it a candidate for embedded systems. Our approach was tested against real-world buffer overflow vulnerabilities and provided promising initial results, but we see several avenues for future work to build upon the foundations presently laid.

We designed the checkpointing scheme so that it requires a very small storage footprint for each process that is logged. 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. Checkpointing multiple processes independently will require small modifications to the OS so that our checkpointing hardware can be made aware of context switches and swap CAM entries.

We utilized a simple lightweight protection scheme that only protects against the most basic of software attacks. We chose this scheme for its simplicity and relatively low overhead characteristics. Future implementations of "rollback and huddle" could explore the use of more complex detection schemes and investigate the tradeoffs involved in making such decisions. Finally, we intend on further exploring the CCRTs since we believe that there exists many more possibilities in their design and implementation.

## VII. Acknowledgments

This work is supported in part by the NSF under grant CNS-0831041 and by the AFOSR under grant FA9550-09-1-0194.

## References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, Nov. 2005.

[2] CERT. US-CERT cyber security bulletin sb04-357. available at http://www.us-cert.gov/, Dec. 2004.

[3] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium (SSYM)*, pages 91–104, Aug. 2003.

[4] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium (SSYM)*, pages 63–78, Jan. 1998.

[5] 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.

[6] D. 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 *Proceedings of the 1st Workshop Architectural and System Support for Improving Software Dependability (ASID)*, pages 66–71, Oct. 2006.

[7] 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, May 2003.

[8] 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.

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

[10] 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)*, pages 168–177, Nov. 2000.

[11] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 289–300, June 2008.

[12] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: Abstractions and software-hardware interface for hardware-assisted deterministic multiprocessor replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[13] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 284–295, June 2005.

[14] 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.

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

[16] M. Prvulovic, Z. Zhangzy, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 111–122, May 2002.

[17] 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, May 2001.

[18] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2005.

[19] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint / recovery. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 123–134, May 2002.

[20] 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 (ASPLOS)*, pages 84–96, Oct. 2004.

[21] 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.

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

[23] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, pages 149–162, Feb. 2003.

[24] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 122–133, June 2003.

[25] 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.

[26] 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.

[27] J. Zambreno, D. Honbo, A. Choudhary, R. Simha, and B. Narahari. High-performance software protection using reconfigurable architectures. *Proceedings of the IEEE*, 94(2):419–431, Feb. 2006.