Des Autom Embed Syst
DOI 10.1007/s10617-008-9020-4

Automated software attack recovery using rollback and huddle

Jesse Sathre · Joseph Zambreno

Received: 5 March 2008 / Accepted: 16 April 2008

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.

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

J. Sathre · J. Zambreno (✉)
Dept. of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA
e-mail: zambreno@iastate.edu

J. Sathre
e-mail: jsathre@iastate.edu

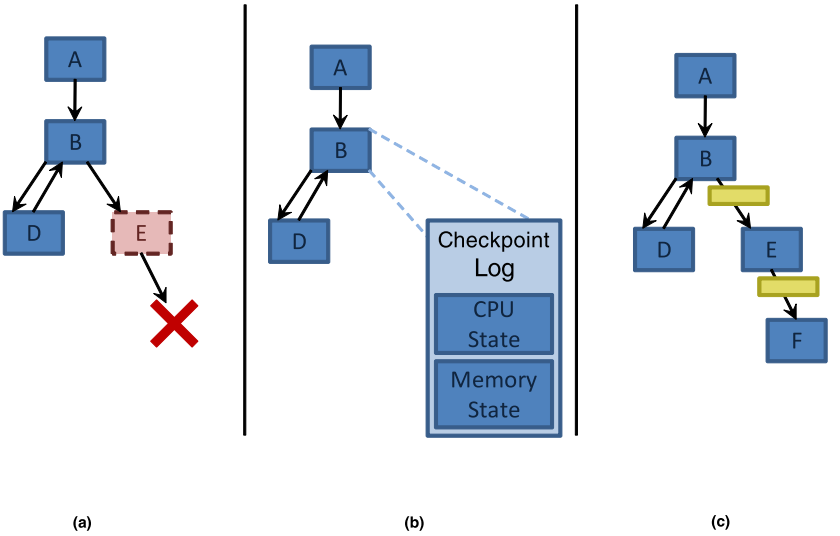

Fig. 1 (**a**) An anomaly is detected in function E. Typical protection schemes cause the program to halt execution. (**b**) In our approach, the program is rolled back to a safe state by utilizing checkpoint logs. (**c**) The program restarts execution with extra safety checks in place

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 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 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. 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 Sect. 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 7.8% performance overhead to a variety of embedded benchmarks, and we are able to successfully rollback from a number of real-world vulnerabilities using only 300 KB of additional storage.

The remainder of this paper is organized as follows. In Sect. 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 Sect. 3 we outline the architectural features of our approach. In Sect. 4 we present experimental results exploring the performance tradeoffs of these features. Finally, the paper is concluded in Sect. 6 with a look toward future planned efforts in this project.

2 Related work

Work related to our approach falls into two broad categories: attack detection and checkpointing/rollback. Our approach combines these two general fields of study by applying the concept of checkpointing and rollback to the domain of application security.

2.1 Attack detection

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 [16], 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 prohibitive performance impact of XOM was later improved in [32] by using a "one-time pad" encryption scheme to reduce encryption latencies. AEGIS secure coprocessor [26] provides an implementation of physical random functions that can be used for assigning unique keys to an individual processor. The IBM 4758 secure coprocessor [10] 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.

A number of attack detection schemes work at the compiler level. Stackguard [6] is a compiler patch which writes a "canary" value on the stack to verify that the return address has not been tampered. Wilander and Kamkar [30] showed that Stackguard fails to detect many forms of buffer overflow attacks. Pointguard [7] is an extension of Stackguard that encrypts all pointers stored in memory. A hardware implementation of Pointguard can be found in [28]. The hardware version of Pointguard greatly reduces the performance overhead of the original Pointguard approach by introducing specialized instructions to carry out pointer encryption and decryption.

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 [18] 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 [19], where the Return Address Stack (RAS) of a speculative processor is modified, increasing stack security and resulting in less than a 1% performance overhead. In [5], the authors use hardware-based dynamic instruction rewriting to maintain a "shadow" stack of return addresses. 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 [30].

The general threat model that we consider is similar to that found in [1] and other works [3, 8, 15, 25] that check the integrity of program flow and call-graphs formed during program execution. In [1], program flow integrity is considered at the basic block level. As will be explained in Sect. 3, our lightweight protection scheme expands this granularity to the function call scope in order to incur less overhead. A separate approach [11, 29] 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 [13, 21], which studies the patterns of a normally executing program to form its acceptable call graph. Hybrid examples exist as well [12].

2.2 Checkpoint and rollback

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 [31] which itself can be traced back to the scheme found in [24]. 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 multiprocessor systems. Other examples of hardware-assisted checkpointing and rollback can be found in [20, 27]. 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).

Recently, there have been some approaches which apply checkpointing and rollback to the domain of software security. One example is ExecRecorder [9]. ExecRecorder is a log-based recovery mechanism designed to work with an intrusion-detection system, similar to our approach. However, ExecRecorder works at the Virtual Machine layer, and it uses its logs as a means for off-line analysis as opposed to on-line rollback. Another example is DIRA [23] which uses 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. However, it 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 [22].

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 *heavy-weight 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 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 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 [31] and [24]. 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 paper 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 [24]. 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 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

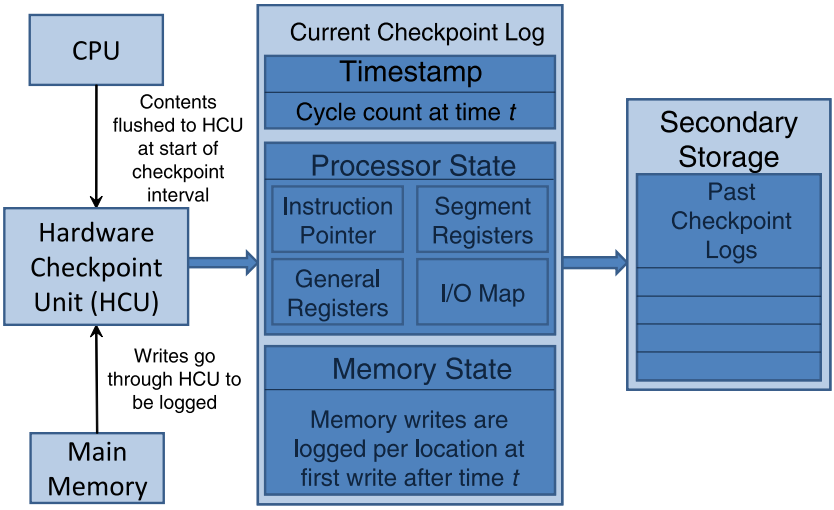

Fig. 2 Contents of a typical checkpoint log. Checkpoint logs are identified by a timestamp and contain the necessary information to restore a program's state

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. 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 [31] and [24] 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

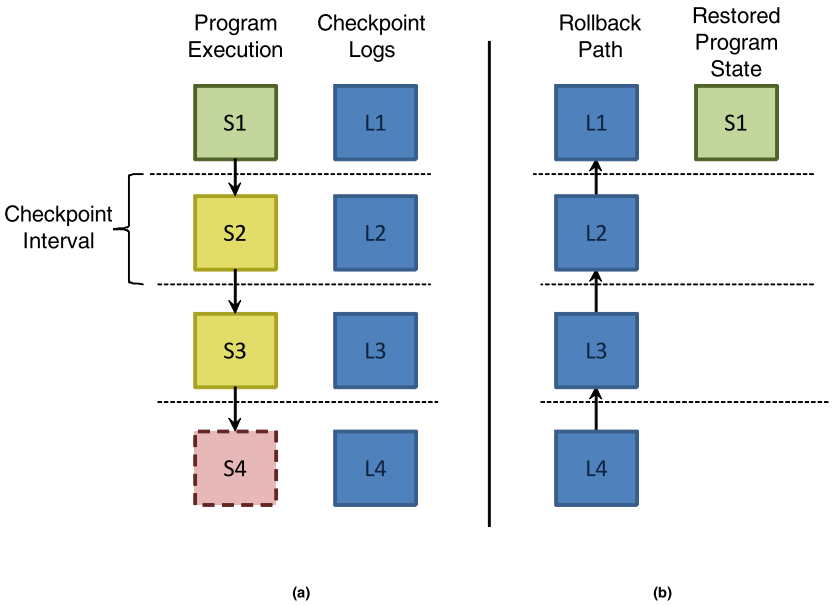

Fig. 3 (**a**) As a program executes, its state is checkpointed at normal intervals until an anomaly is detected at S4. (**b**) The checkpoint logs are cycled through until the system is restored to a safe point at S1

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.

3.3 Detection and rollback

Program rollback occurs when the lightweight protection scheme has signaled that an attack has taken place. In Fig. 3, 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 Fig. 3. 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 the safe state, S1.

One method for determining how far to rewind execution is given in Fig. 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

```
ROLLBACK_DISTANCE(t_attack, n_attack, N_cp, C) {
    distance = 0;
    extra = 2^(n_attack − 1);
    for each checkpoint c ∈ C do {
        distance = distance + 1;
        if (t_c < t_attack) then {
            break;
        }
    }
    return (MIN(distance + extra, N_cp));
}
```

Fig. 4 An algorithm for determining rollback distance

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 [25]. 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 nth 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

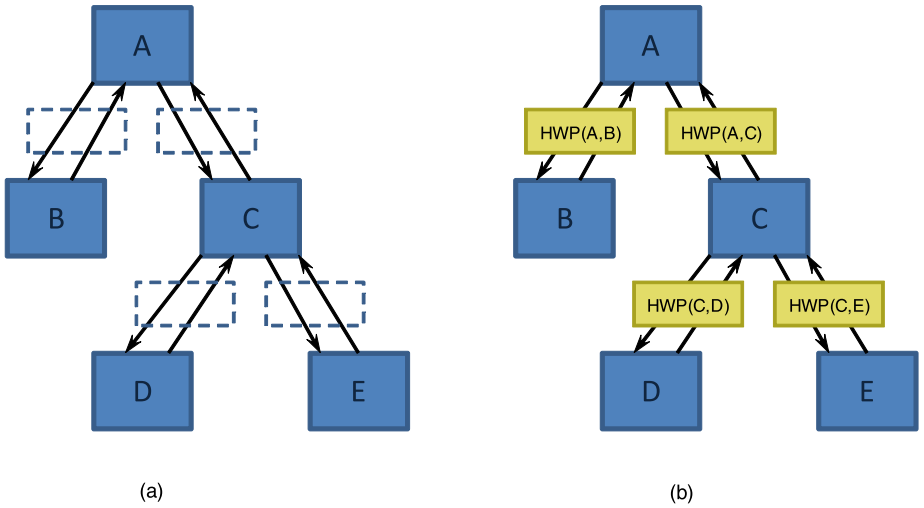

Fig. 5 (**a**) Placeholder NOP instructions are inserted by the compiler. (**b**) The program is further for re-execution in the "Heavyweight Monitoring" state

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. This concept of heavyweight monitoring is illustrated in Fig. 5.

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 [17]. 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 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 added security.

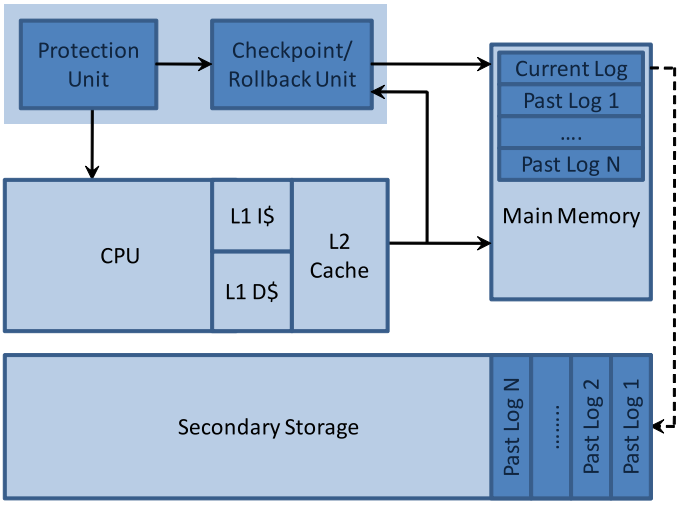

Fig. 6 High-level architecture view with protection and checkpoint/rollback hardware

4 Architectural features

Our approach can make use of several architectural features to operate efficiently. Figure 6 shows how our supplemental hardware fits into existing architectures. Additional hardware modules include a Hardware Checkpoint Unit (HCU) and a Lightweight Protection Unit (LPU). Each module is made up of a small amount of logic and internal storage. Not shown in Fig. 6 is the Heavyweight Protection Unit (HPU), which depending on the required functionality may be more tightly coupled with the target processor. These additional modules can be incorporated into existing architectures and commercial systems with minimal changes.

4.1 Hardware checkpoint unit

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(CAM) alongside some standard memory. The CAM is used to store a table containing 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 current and past checkpoint logs.

The current checkpoint log, as well as a fixed number of past checkpoints, are stored in a reserved section of main memory. Storing the past checkpoints allows for the program to be rolled back beyond just the current checkpoint. It is important for the current log to reside in main memory so that it can be quickly accessed during the actual logging process. In memory-constrained situations it is possible to offload the past checkpoints to secondary storage, as the increased access latency is a non-issue in that case. However, as will be shown in Sect. 5 our approach is capable of recovering from a number of real-world attacks utilizing a total storage capacity of only 300 KB. If all checkpoint logs are stored in main memory, they can be viewed as a circular buffer where the oldest checkpoint is simply overwritten by the new checkpoint log. This simplifies checkpoint creation and minimizes latency as

checkpoint creation becomes only a matter of updating the pointer to the current checkpoint log.

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. The HCU continues snooping the memory bus while the current value is being logged. If the current address is requested from main memory, or if another location must be logged, the HCU stalls the CPU until the current transaction is complete. This can be avoided by adding some additional storage to the HCU to buffer a number of requests, and treating the HCU as an additional cache level.

Memory location granularity can vary by implementation. Logging memory writes on a per-word basis creates a large amount of storage overhead to keep track of the location specifier. A possible 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 will increase run-time overhead. In Sect. 5 we will investigate the performance effects of memory granularity choice.

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 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.

Once the current checkpoint log is filled, the HCU stalls the CPU so that it can perform the necessary steps for checkpoint creation. First the HCU must find the location of its next checkpoint in main memory. If all of the reserved checkpoint storage is utilized, the oldest checkpoint log is overwritten. Next, the CPU's register state is stored in the current checkpoint log and the current timestamp is recorded. Finally, the HCU clears its content-addressable memory and execution resumes.

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 process is complete. Once a checkpoint has been rolled back, it is no longer needed and can be discarded. In this case, the log is simply overwritten when the next checkpoint is brought into main memory.

4.2 Lightweight protection unit

Located adjacent to the HCU and CPU, the LPU consists of its own storage and a controller. The LPU is shown in Fig. 7. To implement the lightweight scheme described in Sect. 3.1, 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.

Original Instruction Sequence

```
  call foo    ; call foo sub-routine
  inc  eax    ; next instruction in program


foo:
  ret         ; return to calling routine
```

Instrumented Instruction Sequence

```
  hw_push, <ret_addr>   ; move the return address to HW stack
  call  foo             ; call foo sub-routine
  inc   eax             ; next instruction in program

foo:
  hw_pop                ; pop HW stack and compare to this address
  ret                   ; return to calling routine
```

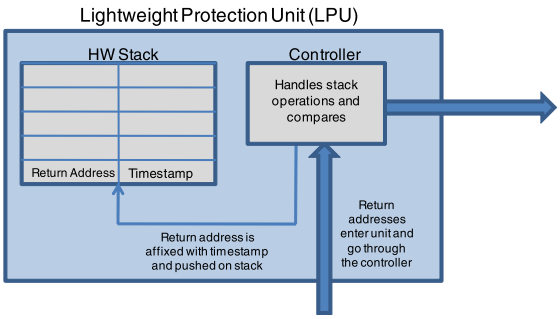

Fig. 7 Operation of the LPU with code instrumented to make use of the secondary hardware stack

For a program to take advantage of the LPU, it must be compiled with advance knowledge of the LPU location. An example of instrumented code is shown in Fig. 7. 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.

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 phase 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 NOP instructions in the original program, the instructions could already be in place in an encrypted form. Once rollback takes place, 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 execution environment similar to XOM [16]. 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.

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 PTLSim Classic [33], 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 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 memory heirarchy was configured as follows:

- L1 Instruction Cache: 32 KB, 4-way set associative, 1 cycle latency
- L1 Data Cache: 16 KB, 4-way set associative, 1 cylce 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.

A write-through policy was used for data writes so that any cache writes were automatically propagated through the memory hierarchy. 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 applications for performance evaluation were six benchmarks chosen from the MiBench embedded benchmark suite [14]. The benchmarks were selected based on their memory access patterns, as well as frequency of function calls to highlight the effects of the HCU and LPU components, respectively. As an initial investigation into the performance impact of our approach, we compared the total overhead and maximum rollback distances for various configurations of the HCU. Finally, we tested the rollback capabilities of our approach against a number of real software vulnerabilities.

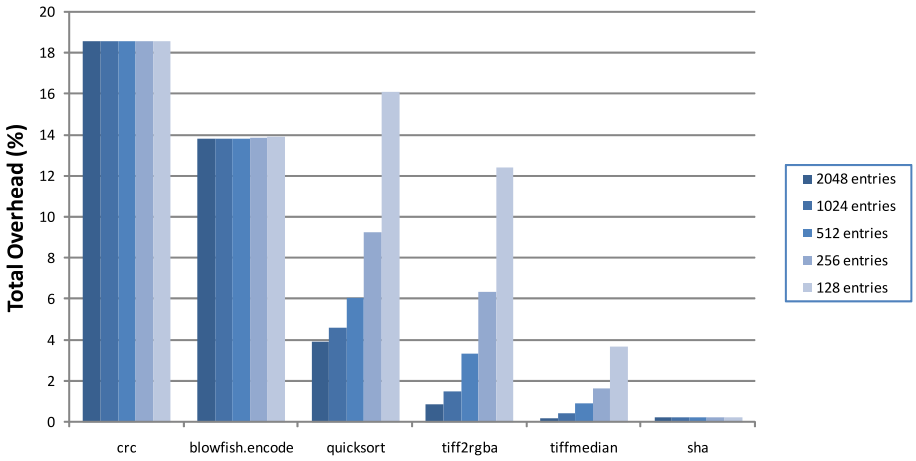

Fig. 8 Total overhead as a function of memory log size

5.1 Performance overhead

Figure 8 shows the total overhead for five different memory log sizes. We varied the log sizes from 128 to 2048 entries, while keeping the total number of stored logs constant at 64. In these tests, we chose to log memory on a per-word basis, so each entry consists of an address, and the corresponding data for that address. The total performance overhead is the sum of the checkpointing scheme and the lightweight protection scheme. The percentage overhead is based on the cycle difference between our approach, and an unmodified system. The overhead related to the lightweight detection mechanism is based on the number of function calls, so it is completely benchmark dependent. Because of this, the overhead of lightweight detection is constant across all configurations for a given benchmark.

Figure 8 shows that overhead related to the checkpointing scheme can be minimized by utilizing larger checkpoint logs. Once the overhead of the lightweight protection scheme is factored out, performance generally scales proportionally to the log size used. Our results show that primary performance bottleneck is benchmark dependent. The `crc` and `blowfish.encode` benchmarks exhibit the largest average performance degradation at 18.5% and 13.8%, respectively. Both programs utilize frequent, short function calls with minimal writes to memory, so they are most affected by the lightweight protection scheme. At the other extreme are `tiff2rgba` and `tiffmedian`, who are most affected by the checkpointing scheme. The effects of varying the memory log size is the most evident in these benchmarks. The `quicksort` benchmark is not bottlenecked by one component in particular, and `sha` showed negligible performance penalties across all configurations. The average overhead across all benchmarks and all configurations was found to be 7.8%.

Figure 9 shows the effect that memory log granularity can have on the total overhead. Once again, the total overhead percentage was based on the cycle difference between our approach and an unmodified system. In one configuration we logged on a per-cache-line basis, and in the other we logged on a per-word basis. Each cache line in the simulation environment contains 8 words, so we chose 128 entries and 1024 entries for our per-line and per-word configurations, respectively. These values allow for the data portions of each configuration to be of equal size; however, the per-word configuration will require some extra storage for the increased number of addresses to store. The effects of granularity are minimal

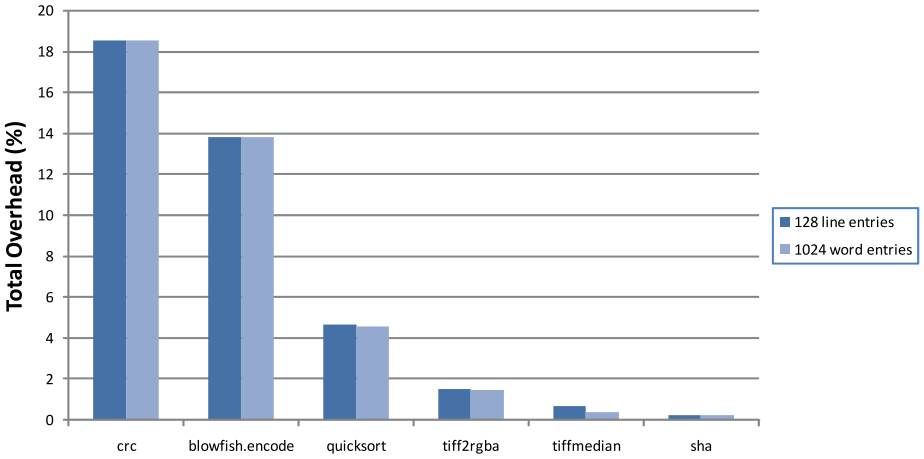

Fig. 9 Effects of granularity choice on performance overhead

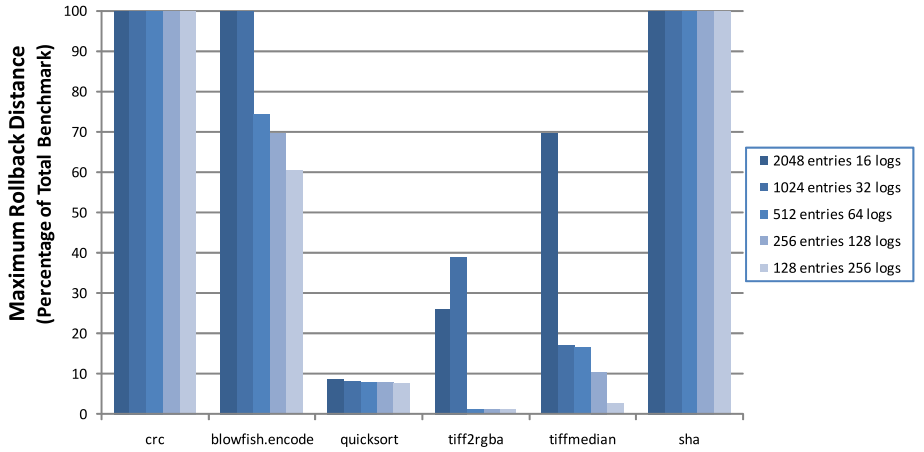

Fig. 10 Rollback distance as a function of log size

in these benchmarks. This is a sign that most of the benchmarks used contained a significant amount of spacial locality in their memory access patterns. The `tiffmedian` benchmark was the only benchmark to show more than a neglible difference in overhead. Although the final overhead difference was less than 1%, the per-line configuration required more than twice the number of checkpoints of the per-word configuration for `tiffmedian`.

5.2 Rollback distance

Figure 10 shows how varying the log size can affect the maximum rollback distance. The maximum rollback distance was calculated by taking the timestamp difference of the newest checkpoint log and the oldest checkpoint log. Once again we logged memory writes on a per-word basis. The total storage was kept constant at 300 KB using 64-bit addresses and 64-bit data words. To keep storage space constant, as the memory log increases in size, the

Table 1 Required cycle rollback of vulnerable programs

Application	Version	Cycles from input to detection	Recoverable?
villistextum	2.6.6	6,745,763	Yes
ringtonetools	2.22	3,640,082	Yes
mplayer	1.0pre5	2,890,044	Yes
csv2xml	0.5.1	1,841,252	Yes
2fax	3.0.4	727,625	Yes
bsb2ppm	0.0.6	665,687	Yes
jpegtoavi	1.5	432,591	Yes
o3read	0.0.3	304,964	Yes

total number of stored logs is reduced. The results are based on the percentage of the total benchmark cycles that were able to be rolled back. The `crc` and `sha` benchmarks made very few memory writes, so we were able to log their entire runs without reusing any checkpoint logs. Across the remaining benchmarks, the general trend is that a smaller number of large checkpoint logs leads to a longer achievable rollback distance. The single exception to this rule was `tiff2rgba`, which achieved the longest rollback distance by utilizing 1024-entry logs as opposed to the 2048-entry logs. The `tiffmedian` benchmark showed the most dramatic difference between configurations with a drop-off from 70% down to 3% for the 2048-entry and 128-entry configurations respectively. This equated to a difference of over 100 million cycles. Maximum rollback distances for the various benchmarks and configurations ranged from 720,000 cycles all the way to 107 million cycles with an average value of 36 million cycles across all benchmarks and configurations.

5.3 Real-world vulnerabilities

To verify the effectiveness of our checkpointing and rollback mechanisms we tested our approach against eight vulnerabilities found in real-world applications. The programs along with the vulnerable versions are listed in Table 1. All of the programs are open source and designed to be run in a Unix environment. Originally discovered as part of a class project [2], details for each exploit can be found in US-CERT Cyber Security Bulletin SB04-357 [4]. Each of the programs contain overflowable buffers at various points in their execution, which an attacker can use to run arbitrary code on the victim's machine.

Initially, we investigated the number of cycles which would need to be rolled back to recover from an attack. Each of the vulnerable programs was fed malicious input designed to overflow its vulnerable buffer. Within our simulation environment we measured the cycle count from the point of input to the point that the program attempted to use the malicious return address. The results of these tests are shown in Table 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 of 128 entries per log because this it was shown in our maximum rollback tests that this set-up was the least effective of all the tested configurations. Once again the vulnerable programs were given malicious input. In every case once the program attempted to return to the malicious return address, the HCU contained sufficient information to roll back to the time that input was received.

What can we conclude from all of these tests? One general observation is that a system's use should be taken into consideration when designed to be used with our approach. While the configurations that we tested yielded acceptable results for general use, there are situations which are open to optimizations for specialized uses. For instance, a system which will have frequent, short function calls (similar to `crc`) will benefit from a lightweight protection scheme with less function call and return instrumentation. Another example is considering a system's memory access patterns when choosing logging granularity. Program's with considerable spatial locality will benefit from the lower storage requirements of a coarser granularity, whereas programs with poor locality (such as `tiffmedian`) benefit from a finer granularity.

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. We have shown that our system can be integrated into existing architectures with minimal changes to accommodate the additional hardware and features introduced by our approach. Initial experimental results indicate that the overall impact to performance is minimal (less than 8% on average). We have also shown that it is possible to recover from a number of real-world software attacks using no more than 300 KB of additional storage.

The focus of this paper has been on introducing the concepts and tradeoffs involved in designing an attack-recoverable system, with much of the focus on the LPU and the checkpointing schemes. In the future we plan on expanding the idea of heavyweight monitoring and developing new architectural features to aid in this process. Refining and optimizing our checkpointing scheme to accommodate multi-threaded environments is also left for future work. Finally, we intend to explore how different types of protective mechanisms can fit into our system. We feel that this recoverable software model combined with the idea of additional software checks could one day lead to the automatic identification and patching of software vulnerabilities.

References

1. Abadi M, Budiu M, Erlingsson U, Ligatti J (2005) Control-flow integrity: Principles, implementations, and applications. In: Proceedings of the ACM conference on computer and communications security (CCS), pp 340–353
2. Bernstein DJ (2004) Unix security holes. Available at http://cr.yp.to/2004-494.html
3. Castro M, Costa M, Harris T (2006) Securing software by enforcing data-flow integrity. In: Proceedings of the workshop on architecture and system support for improving software dependability (ASID), pp 42–51
4. CERT (2004) US-CERT cyber security bulletin sb04-357. Available at http://www.us-cert.gov/cas/bulletins/SB04-357.html
5. Corliss M, Lewis EC, Roth A (2005) Using DISE to protect return addresses from attack. Comput Archit News 33(1):65–72
6. Cowan C, Pu C, Maier D, Hinton H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q (1998) Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the USENIX security symposium, pp 63–78
7. Cowan C, Beattie S, Johansen J, Wagle P (2003) Pointguard: Protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the USENIX security symposium, pp 91–104
8. Crandall J, Wu SF, Chong F (2006) Minos: Architectural support for protecting control data. ACM Trans Archit Code Optim 3(4):359–389

9. de Oliveira DAS, Crandall JR, Wassermann G, Wu SF, Su Z, Chong FT (2006) 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, pp 66–71
10. Dyer J, Lindemann M, Perez R, Sailer R, van Doorn L, Smith S, Weingart S (2001) Building the IBM 4758 secure coprocessor. Computer 34(10):57–66
11. Feng H, Kolesnikov O, Fogla P, Lee W, Gong W (2003) Anomaly detection using call stack information. In: Proceedings of the IEEE symposium on security and privacy, pp 62–75
12. Gao D, Reiter MK, Song D (2004) Gray-box extraction of execution graphs for anomaly detection. In: Proceedings of the ACM conference on computer and communications security (CCS), pp 318–329
13. Ghosh A, O'Connor T, McGraw G (1998) An automated approach for identifying potential vulnerabilities in software. In: Proceedings of the IEEE symposium on security and privacy, pp 104–114
14. Guthaus M, Ringenberg J, Ernst D, Austin T, Mudge T, Brown R (2001) MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings of the international workshop on workload characterization (WWC), pp 3–14
15. Kiriansky VL (2003) Secure execution environment via program shepherding. Master's thesis, Massachusetts Institute of Technology
16. Lie D, Thekkath C, Mitchell M, Lincoln P, Boneh D, Mitchell J, Horowitz M (2000) 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), pp 168–177
17. Necula G, McPeak S, Weimer W (2002) CCured: Type-safe retrofitting of legacy code. In: Proceedings of the ACM symposium on principles of programming languages (POPL), pp 128–139
18. Ozdoganoglu H, Vijaykumar T, Brodley C, Jalote A, Kuperman B (2003) 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
19. Park Y-J, Zhang Z, Lee G (2006) Microarchitectural protection against stack-based buffer overflow attacks. IEEE Micro 26(4):62–71
20. Prvulovic M, Zhangzy Z, Torrellas J (2002) ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: Proceedings of the international symposium on computer architecture (ISCA), pp 111–122
21. Sekar R, Bendre M, Dhurjati D, Bollineni P (2001) A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the IEEE symposium on security and privacy, pp 144–155
22. Shi W, Lee H-HS, Falk L, Ghosh M (2006) An integrated framework for dependable and revivable architectures using multicore processors. SIGARCH Comput Archit News 34(2):102–113
23. Smirnov A, cker Chiueh T (2005) Dira: Automatic detection, identification, and repair of control-hijacking attacks. In: Proceedings of the 12th annual network and distributed system security symposium
24. Sorin D, Martin M, Hill M, Wood D (2002) SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: Proceedings of international symposium on computer architecture (ISCA), pp 123–134
25. Suh GE, Lee JW, Zhang D, Devadas S (2004) Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th international conference on architectural support for programming languages and operating systems, pp 84–96
26. Suh GE, O'Donnell C, Sachdev I, Devadas S (2005) Design and implementation of the AEGIS single-chip secure processor using physical random functions. In: Proceedings of the international symposium on computer architecture (ISCA), pp 25–36
27. Teodorescu R, Torrellas J (2005) Prototyping architectural support for program rollback using FPGAs. In: Proceedings of the international symposium on field-programmable custom computing machines (FCCM), pp 23–32
28. Tuck N, Calder B, Varghese G (2004) Hardware and binary modification support for code pointer protection from buffer overflow. In: Proceedings of the international symposium on microarchitecture (MICRO), pp 209–220
29. Wagner D, Dean D (2001) Intrusion detection via static analysis. In: Proceedings of the IEEE symposium on security and privacy, p 156
30. Wilander J, Kamkar M (2003) A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the network and distributed system security symposium, pp 149–162
31. Xu M, Bodik R, Hill M (2003) A flight data recorder for enabling full-system multiprocessor deterministic replay. Comput Archit News 31(2):122–135
32. Yang J, Zhang Y, Gao L (2003) Fast secure processor for inhibiting software piracy and tampering. In: Proceedings of the 36th international symposium on microarchitecture (MICRO), pp 351–360
33. Yourst M (2007) Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: Proceedings of the IEEE symposium on performance analysis of systems and software (ISPASS), pp 23–34