An Architectural Framework for Providing Reliability and Security Support

N. Nakka, Z. Kalbarczyk, R. K. Iyer
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Email: {nakka, kalbar, iyer}@crhc.uiuc.edu

J. Xu
Department of Computer Science
North Carolina State University
Email: {junxu@csc.ncsu.edu}

Abstract. *This paper explores hardware-implemented error-detection and security mechanisms embedded as modules in a hardware-level framework called the Reliability and Security Engine (RSE), which is implemented as an integral part of a modern microprocessor. The RSE interacts with the processor through an input/output interface. The CHECK instruction, a special extension of the instruction set architecture of the processor, is the interface of the application with the RSE. The detection mechanisms described here in detail are: (1) the Memory Layout Randomization (MLR) module, which randomizes the memory layout of a process in order to foil attackers who assume a fixed system layout, (2) the Data Dependency Tracking (DDT) module, which tracks the dependencies among threads of a process and maintains checkpoints of shared memory pages in order to rollback the threads when an offending (potentially malicious) thread is terminated, and (3) the Instruction Checker module (ICM), which checks an instruction for its validity or the control-flow of the program just as the instruction enters the pipeline for execution. Performance simulations for the studied modules indicate low overhead of the proposed solutions.*

1. Introduction

A variety of specific processor-level mechanisms have been proposed for error detection and recovery (e.g., duplication, instruction-retry, control-flow checking, cache-based recovery) on one hand, and security protection (e.g., preventing buffer-overflow, tamper-resistant hardware) on the other. Each technique is based on custom augmentation of the processor; additionally, most methods do not provide application specific support. Processor-based mechanisms are important since, from a security perspective they can be less susceptible to attacks and, from the reliability perspective can minimize error propagation and hence enable robust recovery. Application-aware error detection performed with hardware is known to minimize fault propagation, and many security mechanisms need to be application-aware.

This paper demonstrates a common framework to provide a variety of application-aware techniques for error-detection, masking of security vulnerabilities and recovery under one umbrella, in a uniform, low overhead manner. To illustrate our approach, we describe hardware-implemented error-detection and security mechanisms embedded as modules in the hardware-level framework, called the Reliability and Security Engine (RSE), which is implemented as an integral part of a superscalar microprocessor.

Security is provided by the Memory Layout Randomization (MLR) module, which randomly relocates key memory regions (e.g., the start address of the application stack) in a process' address space to break the fixed memory layout assumed by an attacker and thus defeat a large class of security attacks (about 60% of attacks reported by CERT [1]). Recovery is supported by the Data Dependency Tracking (DDT) module, which tracks the runtime data dependency among threads in a multithreaded process and uses this information to recover a multithreaded application from a thread crash due to malicious (security attacks) faults. Reliability is achieved by introducing the Instruction Checker Module (ICM), which checks instruction validity or the control-flow of the program as the instruction enters the pipeline for execution, providing coverage for the multiple bit errors in instruction while it is being transferred from memory to the dispatch stage of the pipeline. The operating system and/or application are aware of these mechanisms and can be instrumented using compiler support to insert a special CHECK for invoking the modules. The instruction set architecture of the core processor is extended to include and support these CHECK instructions.

2. Related Work

Earlier work focused on providing software-based solutions [6][8][21][22]. Hardware-implemented error-detection mechanisms are proposed to support control-flow checking by means of watchdogs [10] or embedded signature monitoring for checking data-access faults [11]. Other hardware-based techniques (1) use duplication, as in G4[18] and G5[19], or hardware redundancy, as in DIVA [20], or (2) use existing spare hardware to re-execute the instruction and compare the results [12]-[14].

Most previous work on processor-level security focuses on providing tamper-resistance and cryptography. For example, the IBM 4758 Secure Coprocessor [5] provides both physical tamper-resistance and hardware support for encryption algorithms. Independently, [21] and [23] propose to enhance the existing processor pipeline to detect stack-based buffer overflow attacks.

Cache-based error recovery algorithms in single- or multiprocessor systems [25][26] are usually implemented by modifying the cache replacement or cache coherence protocol in order to save copies of dirty cache lines when they have to be replaced. ReVive [24] changes the memory directory controller in shared-memory multiprocessor machines to support rollback recovery. In ReVive, internode memory communication is intercepted at the memory directory controller to achieve memory-based checkpointing, logging, and distributed parity maintenance.

Expansions to the pipelines of microprocessors have also been proposed to perform concurrent error checking. Wilken and Kong [11] propose a hardware technique meant specifically for control-flow checking using embedded signature-monitoring for a RISC architecture. The signature monitor is on-chip and executes in parallel with the pipeline. It performs exception handling by saving and restoring the signatures by means of special instructions.

3. RSE Framework

The Reliability and Security Engine (RSE) is implemented as an integral part of the processor, residing on the same die. Hardware modules providing error-detection and security services are embedded in the RSE and execute in parallel with the core pipeline. Special CHECK instructions, inserted into a target application by a compiler, provide the interface between the RSE and the application. As instructions are fetched from the pipeline, the CHECK instructions are forwarded to the RSE to invoke the security and reliability hardware checker modules.

Figure 1 depicts the RSE framework in the context of a superscalar processor (similar to the DLX processor described in [27]) simulated using an augmented *sim-outorder* SimpleScalar [3] simulator. The left side of Figure 1 shows the details of a DLX-like superscalar pipeline structure together with the architectural parameters used in the simulation. The RSE framework is shown on the right side (shaded rectangle). The RSE can be broadly divided into two parts: the *input interface* and the *internal hardware.*

3.1. Input Interface of the Framework

The modules in the RSE interface with the execution pipeline through the common input interface. This interface consists of a set of input queues, each of which contains the outputs of a particular pipeline stage. Additional fan-outs from the pipeline stage outputs of the core processor provide input to the framework. The set of RSE input queues includes:

- *Fetch_Out* delivers the currently fetched instructions.
- *Regfile_Data* delivers the operands of an instruction.
- *Execute_Out* delivers results of ALU operations or address generation.
- *Memory_Out* provides data values loaded by the pipeline from memory during the current cycle.
- *Commit_Out* indicates the instructions that are committed or squashed by the pipeline; used to remove the data corresponding to these instructions from the input queues.

Two additional inputs, *Mem* and *Mem_Rdy*, are used by the MAU (Memory Access Unit in the RSE) to access memory in response to requests from the hardware modules.[1] The number of entries in each input queue is equal to the number of entries in the re-order buffer in the pipeline. For the sake of clarity, Figure 1 only shows the register and the multiplexer (MUX) driving its input for a single entry (the i^{th} one) in a queue. The signals going across the boundary of the framework and the pipeline are shown as dashed lines.

The modules are connected to the appropriate input queues to obtain necessary data, e.g., the MLR module is connected to the *Fetch_Out* and the *Commit_Out* queues, as shown in Figure 1. Assuming a 32-bit processor with a re-order buffer size of 16 entries, an estimate of the hardware overhead due to the input queues and input MUXes is approximately 2560 flip-flops and 12,800 gates[2]. The delays due to the framework are discussed in Table 3. Clearly, there will be additional delays due to the increased wiring placement and routing, which need to be addressed via circuit design techniques.

The interface can handle speculative execution by the pipeline. When the pipeline is flushed in case of a misprediction or mispeculation, the instructions being squashed are passed to the RSE through the *Commit_Out* input. The RSE uses this information to flush the input queues and the instruction output queue to reflect these changes. As will be seen, no speculative state is maintained in the RSE modules.

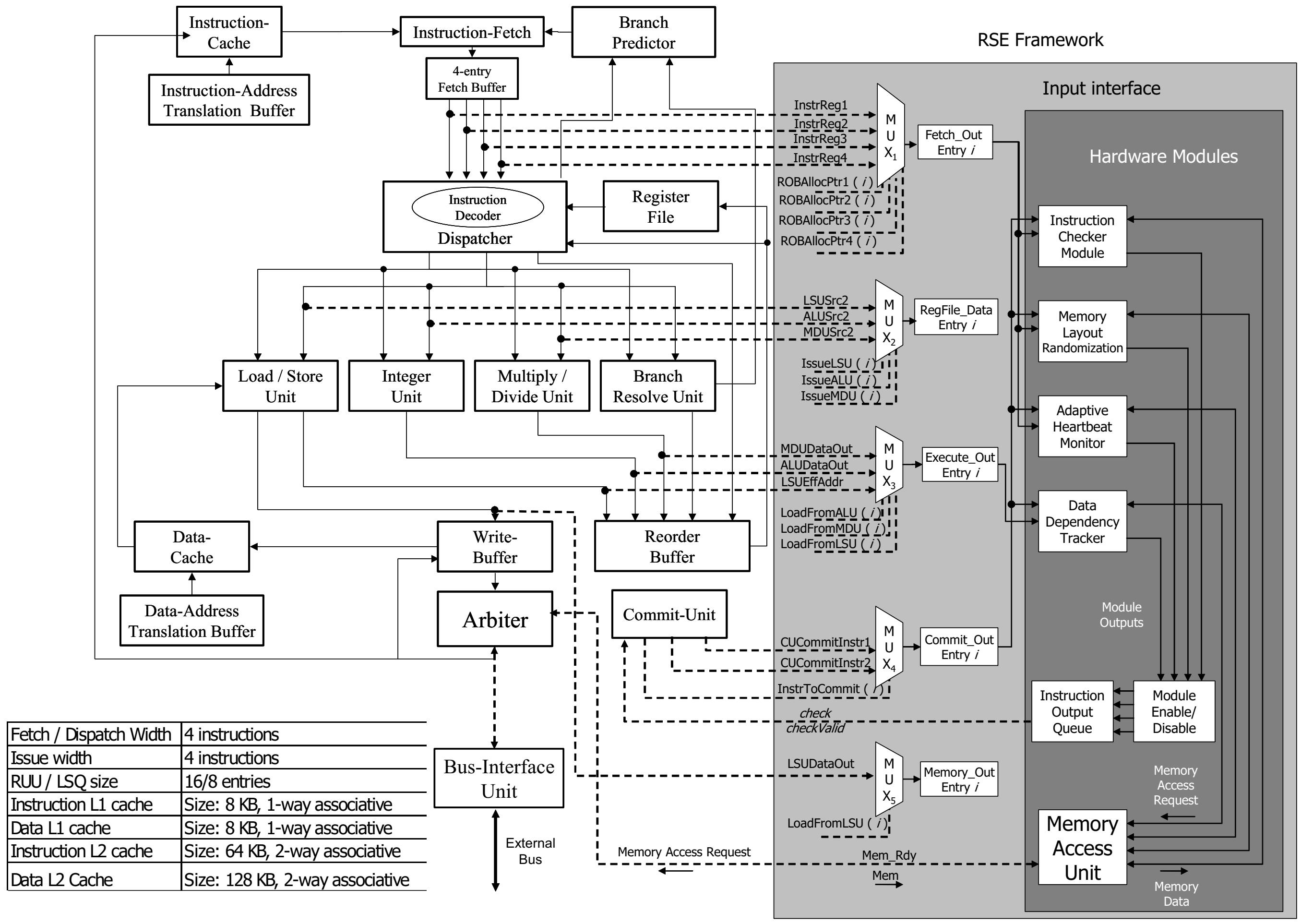
An illustration of instruction execution. A module in the RSE can operate in one of two modes: synchronous and asynchronous. In *synchronous* mode, the pipeline can commit only when the check executed by the module completes; this mode is used when error-detection is performed concurrent with instruction execution. In *asynchronous* mode, the pipeline commits an instruction and sends a signal to the RSE. On receiving this signal, the RSE module collects permanent state used for checking and recovery; this mode is used when the module performs operations such as dependency tracking and checkpointing, while minimally interfering with the pipeline. Together, the two modes cover a broad range of security and reliability checks. Figure 2(a) shows the execution of instructions in the synchronous case. The “CHK INST_CHECK” in the instruction stream notifies the instruction checker module in the framework that the following instruction “beq Label X” is to be checked. The module fetches a redundant copy of the instruction, compares the two copies and writes the results of the instruction to the pipeline. If the module has detected no error, the pipeline commits the instruction. Figure 2(b) depicts the instruction execution in the asynchronous mode. In this case the “ld C” instruction is encountered by the data dependency tracker module in the framework, and using the effective address calculated by the pipeline, the module tracks whether this read operation makes the current thread dependent upon another thread (details of the algorithm are explained in Section 4.2.1). When the pipeline commits the instruction and sends a signal to the RSE, the module logs the collected dependency information as permanent state.

3.2. Internal Organization of RSE

Figure 1 also shows the internal organization of the RSE (rectangle shaded dark gray). It is divided into three main parts:

[1] These inputs are different from the *Memory_Out* input, which receives the output of the memory stage of the pipeline (data loaded by the processor from memory).

[2] #flip flops for input queues = $(5 \times 16 \times 32) = 2560$; 2 inputs need 4to1 MUXes, 2 need 2to1 MUXes and 1 needs a 3to1MUX. The total number of gates = $(2 \times 6 + 2 \times 4 + 1 \times 5) \times 32 \times 16 = 25 \times 32 \times 16 = 12800$. (32-bits per input queue entry, 16 entries per input queue).



Fetch / Dispatch Width	4 instructions
Issue width	4 instructions
RUU / LSQ size	16/8 entries
Instruction L1 cache	Size: 8 KB, 1-way associative
Data L1 cache	Size: 8 KB, 1-way associative
Instruction L2 cache	Size: 64 KB, 2-way associative
Data L2 Cache	Size: 128 KB, 2-way associative

Figure 1: Organization of the RSE Framework (in the context of the simulated superscalar processor derived from the SuperScalar DLX [27] processor)

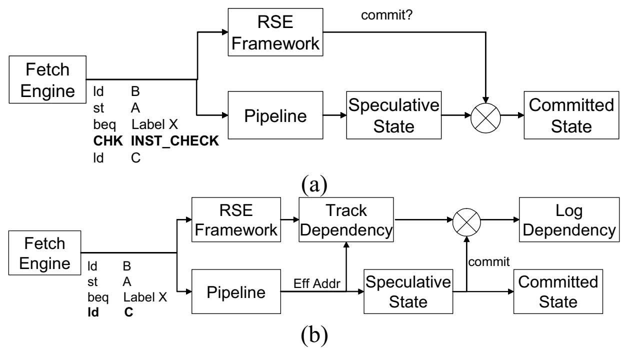


Figure 2: Instruction Execution Scenario (a) synchronous mode; (b) asynchronous mode

(1) the memory access unit, which handles the memory access requests for the modules, (2) the instruction output queue used by the modules to communicate results back to the pipeline, and (3) the hardware modules, which implement application-aware error-detection and security mechanisms.

Memory Access Unit (MAU). This hardware unit provides memory access for RSE modules (e.g., control flow checking) and thus eliminates the need for a bus interface unit in each module, which would incur a high overhead.

A memory access request is placed by a module in a memory request queue and serviced by the MAU in a cyclic order. The MAU shares the bus interface unit with the main processor pipeline (Figure 1). The requests from the MAU and the main pipeline are arbitrated upon, giving the main pipeline the higher priority[3]. This requires a small modification to the arbitration logic (Arbiter in Figure 1) for bus access to include the MAU as an additional driver. The frequency of memory accesses by the processor is generally low due to high hit rates for the on-chip level 1 and level 2 caches. Thus, even though the arbitration logic incurs a performance overhead whenever the processor accesses memory, the overall impact on the processor's performance, as shown in Section 5.2, is small. It is clearly less than in the case where the access is performed through the processor cache. Additionally, via this technique, the memory accesses from the framework do not pollute the cache with data that is irrelevant to the application execution in the pipeline.

[3] While in modern CPUs the processor can directly access memory, bypassing the cache, to store the result in a register, this approach is not a feasible one because (a) we do not want to load the processor with additional memory accesses, and (b) the data needs to arrive to the module and not to the processor.

Instruction Output Queue (IOQ). An entry in the instruction output queue is allocated for each instruction (including the CHECK instructions) at the time the instruction is forwarded to the *Fetch_Out* queue in the framework. This happens simultaneously with the instruction being dispatched in the pipeline. The queue contains the following fields for each entry: (i) *check* holds the result from the check performed by a module, and (ii) *checkValid* is used by the main pipeline to determine whether the data in the *check* field is valid. Together, these two fields constitute the check bits of the IOQ. Interpretation of the *check* and *checkValid* bits is described in Table 1. For a CHECK instruction, these bits (*checkValid* and *check*) are initially set to '00'. For all other instructions, they are set to '10'. This allows the pipeline to commit the non-CHECK instructions as usual. Irrespective of its functionality, each module has a hardware mechanism to scan the *Fetch_Out* queue in the framework to acquire any CHECK instruction intended for the module.

In the *synchronous* mode, the module sets the *checkValid* bit when it completes execution. The pipeline reads the check bits and takes the appropriate action (commit or flush). If the instruction in the pipeline is ready to commit and the module has not yet completed execution, the pipeline waits until the execution is completed in the module and the *checkValid* bit is set. In the asynchronous mode, the module sets the *checkValid* bit of a CHECK instruction entry to '1' immediately after it scans the *Fetch_Out* queue and acquires the CHECK instruction intended for it. The pipeline commits the instruction as usual and sends a signal to the RSE (*Commit_Out*). The module executes independent of the pipeline, and on receiving the commit signal from the pipeline, logs the permanent state that it is collecting. Thus the modules are either stateless or independent of the speculative state of the pipeline.

Observe that for a '10' combination of the check bits (*checkValid* and *check* fields) the pipeline commits the instruction as usual. We take advantage of this observation to introduce an enable/disable mechanism for modules (*Module Enable/Disable* unit in Figure 1). When a module needs to be disabled, instead of the module output, a constant '1' is written to the *checkValid* field and a constant '0' to the *check* field.

Hardware modules. The RSE hardware modules implementing security support and error-detection mechanisms are embedded in the framework and execute in parallel with the execution of the instruction stream in the main execution pipeline. Figure 1 shows the Instruction Checker Module (ICM), the Memory Layout Randomization (MLR) module, the Data Dependency Tracker (DDT) module, and the Adaptive Heartbeat Monitor module. (See Section 4 for detailed description of the modules[4]).

[4] Due to space limitations the Adaptive Heartbeat Module is not discussed in this paper and can be found in [29]

3.3. Application Interface: The CHECK Instruction

The application interfaces with the engine using special instructions called *CHECK* instructions. This requires an extension of the instruction set architecture of the processor. The format of the CHECK instruction includes:

- *Opcode* – contains the opcode *CHK*.
- *Module#* – specifies the module that performs the check.
- *BLK/NBLK* – specifies whether the instruction is blocking or nonblocking. Blocking CHECK instructions are used for synchronous operation, whereas nonblocking checks are used for asynchronous mode of operation.
- *Config Options* – The encoding of these bits is specific to the module. They are used to (i) invoke a hardware module and (ii) provide additional parameters.
- *Operation* – defines the specific operation to be performed by the module.
- *Parameter* – specifies any parameters required for the operation.

When the main pipeline encounters CHECK instructions, it considers them as NOP instructions in all stages of the pipeline, except in the *retirement* or *commit* stage.

Table 1: *CheckValid* and *Check* Fields in IOQ

		check Field	
		0	1
CheckValid Field	0	1. IOQ entry is free. 2. IOQ entry is allocated, contains a CHECK instruction, and its execution in the module is not yet complete. 3. The pipeline may stall if the instruction is ready for commit.	N/A
	1	1. IOQ entry is allocated and contains a non-CHECK instruction. 2. IOQ entry is allocated, contains a CHECK instruction, and *no error is detected* from the execution of the CHECK instruction. 3. The pipeline commits the instruction as usual.	1. IOQ entry is allocated, contains a CHECK instruction, and *an error is detected* by a module. 2. The pipeline is flushed.

3.4. Self-Checking Mechanisms in RSE

The framework employs a self-checking mechanism to detect its own errors and decouple from the pipeline. The self-checking is based on monitoring the transitions on the *check* and *checkValid* bits (as discussed in Table 1). Four error scenarios are considered (see Table 2).

To handle error scenarios outlined in Table 2, a mechanism for self-checking of the RSE framework is proposed. *A watchdog* monitors the transitions occurring in the output bits of each IOQ entry. Both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions are monitored. If a $0 \rightarrow 1$ transition does not occur in the *checkValid* field for a specified number of cycles, a module executing the CHECK instruction in this entry does not make progress or it has a *stuck-at-0* fault in the *check* field. If a $1 \rightarrow 0$ transition does not occur in the *checkValid* field for a specified number of cycles, a *stuck-at-1* fault is assumed in the *checkValid* or *check* field. In addition, a *counter* is maintained for each entry in the IOQ to monitor the $0 \rightarrow 1$ transition (i.e., an error indication) in the *check* field of that entry. If the transition occurs more than a threshold number of times within a specified interval of time (set equal to the watchdog timeout interval), then the module is declared erroneous.

If the self-checking mechanism declares an error, the framework is decoupled from the main pipeline. This can be achieved by switching to a safe mode where the *checkValid* and *check* bits are set to '1' and '0' respectively. This allows the pipeline to always commit the instruction.

Table 2: Error Scenarios of the RSE Framework

Error Scenario	Possible Implications
Module does not make progress	The module does not produce any result. Instruction in the pipeline can wait forever.
False Alarm – the module always declares an error	The pipeline is flushed and starts execution repeatedly at the same CHECK instruction; application does not make progress. A burst of errors that *set* the *check* field in the IOQ entry can also cause this error scenario.
False Negative – the module always declares there is no error	Application proceeds with execution and effectively is not receiving any protection from the module. A burst of errors that *reset* the *check* field in the IOQ entry can also cause this error scenario.
The output bit (*Check* or *Check-Valid*) in the IOQ entry is stuck-at-1 or stuck-at-0	Stuck-at-0 of *checkValid* – equivalent to the scenario where a module does not progress. Stuck-at-1 of *checkValid* – application executes without waiting for the result from the module. Stuck-at-0 of *check* – equivalent to the scenario where the module produces a false negative. Stuck-at-1 of *check* – the pipeline is repeatedly flushed for recovery.

4. Description of Modules in RSE

The following sections discuss issues in designing the RSE framework (Table 3) and present a detailed description and experimental evaluation of example hardware modules embedded in the framework.

4.1. Memory Layout Randomization Module

In our earlier study, that implemented a software-based approach, the Transparent Runtime Randomization (TRR) [22] was proposed to transparently randomize the memory layout of an application process in order to defeat a large class of security attacks (e.g., buffer overflow, format string, etc.). In this study, we implement and integrate a hardware Memory Layout Randomization (MLR) module into the RSE framework. Hardware implementation of the randomization algorithm has several advantages: (1) The RSE implementation can potentially be used by different operating systems, while our software implementation was always system-specific. Although the memory layouts on Windows and Unix/Linux-based systems are quite different, we propose a simple function call to enable uniform implementation of the MLR across different operating systems. (2) A hardware implementation is more difficult to tamper with than an implementation that is entirely in software. For example, a malicious user can exploit operating system vulnerabilities and compromise a software solution. (3) The RSE implementation can speed up the algorithm, especially useful in re-randomizing for long-running applications.

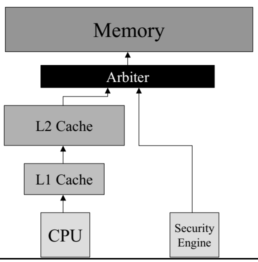
Implementation of the MLR. The randomization task is split between the program loader and the MLR module. For ease of implementation, we provide a portable library. The only change is for a loader to call a special library function. When invoked, the special library function performs the following actions (described in Figure 3): (1) A special header is assembled, which includes information for the position-independent regions (Instruction I0 in Figure 3). The library function extracts information from different parts (the locations are system-specific) of the executable and places them in the special header, which consists of the locations and sizes of text and data segments, and the starting address of stack and shared libraries. (2) The header is passed (via a CHECK instruction) to the MLR module, which performs the randomization operations and writes the results to memory locations specified by the loader. (3) The output produced by the MLR module is used to complete the loading process.

Figure 3 shows the code section within the library function that performs the randomization operations. The code section is a mix of RSE CHECK instructions (tagged with *chk*) and library function instructions (in shadowed boxes). Instructions I1 through I3 perform the randomization of the position-independent regions such as stack, heap, and shared libraries. The library first provides the location of the executable header to the MLR and then issues I2 to request the randomization operation. During execution of I2, MLR reads and parses information from the header and computes the randomized address values for position-independent regions by adding the value from the clock cycle counter. The results are written to predefined memory locations, randomized shared library base, randomized stack segment base, and randomized heap segment base. The library then executes a series of instructions (pointed to by I3) to create a random mapping for these regions.

Randomization of position dependent regions is discussed in the context of the global offset table (GOT), employed on Linux to keep the function pointers used for dynamic library function calls. The equivalent structure on Windows is called IAT (Import Address Table). The randomization of the GOT is performed by instructions I4 through I11. It involves the following steps: (1) executing a series of instructions at I4, the library allocates space for the new GOT at a random memory location, (2) executing CHECK instructions I5 and I6, the library passes the address and size information to the module. CHECK instruction I7 requests the module to copy the old GOT to the new location. The module first copies (without any software intervention) the GOT entries to the internal GOT buffer (shown in Figure 3(B)), and then back to the new location in the memory. (3) Instructions I8 through I11 perform the rewriting of the program's procedural linkage table (PLT) so that the entries refer to the new GOT[5]. First, I8 provides the size and address information of the PLT to the MLR. The library then grants write permission to the PLT section, after which it issues I10 for the MLR to perform the

[5] Each PLT entry is an indirect jump to a library function through an entry in the GOT. Thus, rewriting the PLT involves replacing the address value in the indirect jump pointing to the old GOT to the corresponding address in the new GOT.

Table 3: Issues in the Design of the Framework

Issue	Description	Proposed Solution, Rationale, Pros and Cons
Overhead introduced in memory hierarchy	The Memory Access Unit (MAU) serves memory requests on behalf of all the modules in the framework. It uses the same memory interface circuitry as the main pipeline.	*Solution:* the memory access of the main pipeline and MAU are arbitrated upon, giving a higher priority to the main pipeline in case of a conflict. *Rationale:* choose between (i) accessing L1 Cache; (ii) accessing L2 Cache directly and (iii) accessing main memory directly. L1 Cache to processor path is used very frequently by the processor; any delay introduced in this path due to the arbiter will be very prominent (Amdahl's law). Cache hit rate is very high for both the L1 and L2 Caches; frequency of accessing the memory to L2 Cache path is relatively low. Arbiter in this path would have a much lesser affect on the pipeline execution delay.  Memory Arbiter L2 Cache L1 Cache CPU Security Engine
Fan-out from the pipeline stages	The extra fan-out from each stage introduces a delay in the pipeline's critical path.	*Solution:* the output from the pipeline stage is latched into a register and then provided to the circuit in the framework. *Rationale:* Using an intermediate latch provides constant additional loading to the pipeline outputs, irrespective of the type and number of modules accesses the input. *Pros and Cons:* a constant loading to the pipeline outputs leads to a nearly uniform increase in execution time for each stage; an increased delay in the execution time for each stage increases the clock cycle time; information passed by pipeline is available to the framework only after a delay of one cycle. Additional overhead of wires to get outputs. Complexity in locating outputs for each stage in a modern superscalar out-of-order processor.
Context switch in the processor	When a context switch occurs in the system, the framework may be in the process of executing a CHECK instruction.	A context switch is automatically handled. Before executing a context switch, the processor waits until all the instructions in the reservation station have completed execution and committed. Thus, all the CHECK instructions in the framework must have been completed and their results forwarded to the processor before the system call for context switch begins execution.
	The framework may contain process-specific state.	Currently considered CHECKs are stateless; thus, the framework state does not need to be saved in case of a context switch.
Calculation of delays	Pipeline to input queue entry delay.	The multiplexer feeding the input queue is a combinational circuit with 3 gate-levels and can be evaluated in a single cycle.
	Module to output queue delay.	Each module can write the output of execution of the CHECK instruction to the corresponding entry in the IOQ. This requires a broadcast mechanism, incurring a delay of 1 cycle.
	Module input queue scanning delay.	Each module scan the *Fetch_Out* input queue to scan for CHECK instructions intended for that module. This would incur a delay of 1 cycle.

actual PLT rewriting. The module copies PLT entries into the PLT buffer, rewrites them to reflect the new location of the GOT, and writes back to the PLT in the application code segment (see Figure 3(B)). Finally, the library restores the write permission on the PLT.

4.2. The Data Dependency Tracking Module

The Memory Layout Randomization module discussed in the previous section essentially converts a security attack into a program crash. While this prevents a program from being hijacked, the loss of service due to the attack is nevertheless undesirable. This is especially true if the server is multithreaded and the entire thread pool has to be killed due to the crash of one faulty thread.

In this section, we describe a hardware module to minimize the impact of a single malicious thread in a multithreaded application. The Data Dependency Tracker (DDT) is proposed for tracking the data dependency among threads and for saving their respective memory pages to provide the information needed to recover the healthy surviving threads. While there are similarities to traditional checkpointing schemes, the difference is in not maintaining a thread's execution state. The current DDT implementation targets applications whose inter-thread dependencies are due to operations on main memory data structures. For example, in the case of a multithreaded Apache web server, threads independently serve web requests, and dependency occurs only when two threads read from and write to the same memory page.

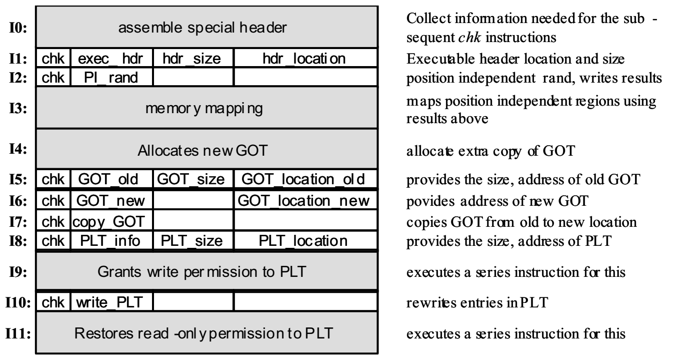

I0: assemble special header
Collect information needed for the sub - sequent *chk* instructions
I1: chk exec_hdr hdr_size hdr_location
Executable header location and size
I2: chk PI_rand
position independent rand, writes results
I3: memory mapping
maps position independent regions using results above
I4: Allocates new GOT
allocate extra copy of GOT
I5: chk GOT_old GOT_size GOT_location_old
provides the size, address of old GOT
I6: chk GOT_new GOT_location_new
povides address of new GOT
I7: chk copy_GOT
copies GOT from old to new location
I8: chk PLT_info PLT_size PLT_location
provides the size, address of PLT
I9: Grants write permission to PLT
executes a series instruction for this
I10: chk write_PLT
rewrites entries in PLT
I11: Restores read -only permission to PLT
executes a series instruction for this

Figure 3: MLR module instruction sequence

4.2.1. Data Dependency Tracking

The problem, from a security perspective, is to determine which surviving threads are healthy. A thread is *healthy* in this context if and only if the thread is not dependent on the faulty thread. A thread t_1 is said to *depend* on t_2, if t_1 consumes data produced by t_2, directly or indirectly. A thread t is healthy if it does not consume any data produced by the faulty thread. A thread remains healthy if there is no future dependency on the faulty thread. To guarantee a healthy thread's future independence of the faulty thread, we must undo changes made by the faulty thread. In the context of the RSE, we propose a page-based mechanism for tracking data dependency among different threads in a process[6].

Tracking mechanism. The proposed technique is similar to the copy-on-write mechanism implemented in many operating systems for memory page sharing between processes. The copy-on-write mechanism is implemented by setting a writable page to be read-only and having the processor MMU raise exception when writing is detected at the page. We employ a similar approach to support recovery of memory pages shared by threads in the same virtual memory address space. Each memory page has a *read-owner* and a *write-owner*, denoted by the owner thread IDs. Let t' and t'' be the read- and write-owners of a page p respectively. When a thread t reads the page p ($t' \neq t$), t becomes the read-owner of p and a data dependency $t'' \rightarrow t$ is defined and entered into a Data Dependency Matrix (DDM). Note that t'' is the producer in the dependency relationship because it last wrote to the page p. Thread t is the consumer because it reads information produced by t''. The DDM is an $N \times N$ matrix, where N is the number of threads in the process. Each entry (x, y) in the matrix is one bit, which when set to 1, indicates that thread y is data-dependent on thread x. Note that the dependency relation is transitive but not symmetric. In order to recover a page updated by a faulty thread, it is important to save a copy of the memory page prior to its update by a thread that is not the current write-owner of the page. A block diagram for the DDT module is shown in Figure 4. The DDT tracks and logs thread dependency in parallel with the execution of memory access instructions in the main pipeline and minimizes the performance impact[7].

The DDT receives memory access instructions from the instruction fetch stage (*Fetch_Out*). The target memory address for a load or store instruction is received from the instruction execution stage (*Execute_Out*), where the effective address is computed and converted to a unique *PageID*. The DDT maintains two main structures for tracking data dependencies: the page status table (PST) and the data dependency matrix (DDM). Due to memory access locality, only a small number of "hot" pages need to be kept in the PST at any given time, and an LRU replacement policy can be used to improve cache performance. An entry in the PST is the tuple (*PageID, write-owner, read-owner*), where *PageID* is a unique tag for a memory page, and *read-owner* and *write-owner* are the thread IDs of the read- and write-owners of the memory page. Whenever a dependency is detected, the corresponding bit in the dependency matrix is set to 1. The operations in the module are done transparent to the pipeline execution, and hence the instruction in the pipeline can commit without waiting for the module to complete. In other words, the module can lag behind the pipeline in completing the logging of the dependencies.

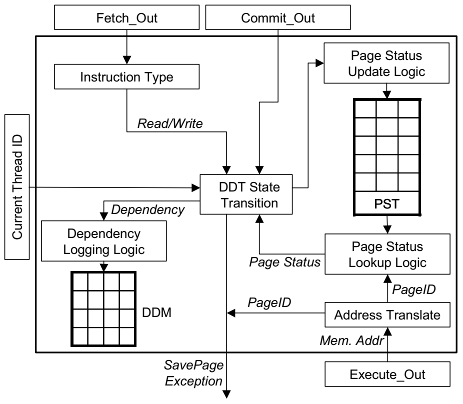


Figure 4: Block Diagram of DDT Module

4.2.2. Issues in DDT Design

Target applications. The DDT mechanism targets applications whose interthread dependencies are mainly due to operations on main memory data structures. The memory-based tracking scheme can also be extended to cover these applications, since all file I/O operations must go through memory buffers.

Garbage collection. The current scheme saves memory page snapshots in the main memory. A natural question to ask is how overflow of the memory buffer can be handled. A solution is to have a kernel thread (independent of the target application threads) that asynchronously flushes the memory pages on the disk. For long-running applications, however, it is possible that even the disk space will be used up. For this, we can either (a) periodically restart the application and remove all previously saved memory pages; or (b) handle the space overflow using a garbage collection algorithm that periodically removes pages from the buffer using a time-based threshold. In order to keep the application state consistent in case of recovery, we keep history information for deleted pages. When any of the deleted pages is needed for recovery, the recovery algorithm terminates the entire process (i.e., all threads) due to insufficient information.

Recovery algorithm. Although DDT tracks thread dependencies and collects information needed for recovery, it does not perform the actual recovery operations. System software performs recovery by retrieving information stored in PST and DDM through a special *size query* and *retrieval* check instruction. The information provided by the DDT is used to recover memory pages and resume execution of surviving threads without execution rollback. The approach leverages existing checkpointing and recovery algorithms for both distributed and multithreaded applications [9][15]. To minimize application-wide impact of the faulty thread t_f, we identify (using information stored in DDM) and terminate all threads that are data-dependent on t_f. The memory updates due to t_f and its dependent threads are undone so that they do not impact the future execution of the healthy threads in the process. Due to space limitation, we skip discussion of specifics of the recovery algorithm. The details can be found in [28].

[6] Software mechanisms for detecting data dependency, such as debugger-based tracing, can detect word-level fine grain data accesses by different threads through single stepping, but incurs prohibitively high overhead. Also, the current thread-based checkpointing work does not quite address the issue.

[7] The page-based memory dependency-tracking algorithm can also be implemented by changing the structure of the processor memory management unit's TLB (translation look-aside buffer). The problem with the TLB-based approach is that TLB is usually on the critical path for memory access, and the added structural and functional complexity may slow down memory access and the performance of the pipeline.

4.3. Instruction Checker Module

The Instruction Checker Module (ICM) preemptively checks for errors in an instruction just at the time the instruction is dispatched, by comparing the binary of the instruction in the pipeline with a redundant copy of the instruction fetched from memory. Thus it provides coverage for multiple-bit errors in the instruction from the time the instruction is fetched from memory to when it is dispatched in the pipeline. The instruction checked can be a control flow, load/store or a critical code section of the application. Thus this approach is complementary to several architectural fault tolerance approaches that effectively employ redundancy in the pipeline in space or in time [12]-[14], [20]. Instructions following a CHECK instruction (*checked* instructions) with *module#* field set to ICM are checked by the ICM. The program is initially statically parsed, and all the checked instructions are stored in a separate chunk of memory (*CheckerMemory*) in a contiguous manner.

Implementation of ICM. The ICM is designed and implemented as a pipeline. The individual tasks of the ICM form three independent pipeline stages: (1) ICM_IDLE scans *Fetch_Out* queue for CHECK instructions and posts a memory request to the MAU for a redundant copy of the instruction, (2) ICM_MEMREQ waits for the redundant copy and the two copies are sent to the comparison stage, and (3) ICM_COMP – The two copies of the instruction are compared and the result is written to the instruction output queue (*check* and *checkValid* fields).

ICM cache. To reduce the performance overhead of accessing the *CheckerMemory*, a dedicated cache *(Icm_Cache)* is employed within the ICM. The replacement policy used for the ICM cache is a least-recently-used (LRU) algorithm implemented using a stack. The contiguous placement of checked instructions in the checker memory makes it easy to fetch multiple checked instructions in a single fetch to provide spatial locality.

Figure 5 depicts the timeline for the execution of the ICM. The time scale is shown to the left. The processor fetches instructions from the memory into its internal cache. Instructions are fetched from the cache into the fetch buffer for execution at time *t*. The instruction is renamed and allocated in the reorder buffer at time *t+1*. Instructions are forwarded from the reservation station to the RSE and arrive at the RSE fetch queue at time *t+2*. At this point, the instruction has a unique identifier—the reorder buffer entry number—by which it is addressed throughout its lifetime in the pipeline. This enables the RSE and the pipeline to use the same unique identifier for communication of inputs and outputs of an instruction. The ICM detects the presence of a CHECK instruction in the fetch buffer and stores the following instruction (checked instruction) from the fetch buffer. This is the copy of the instruction executing in the pipeline. The ICM places a request for a redundant copy of the same instruction to the ICM cache. In the case of ICM cache hit, the redundant copy of the instruction is available, and the two copies are sent to the comparator at time *t+3*. Results are written to the instruction output queue at time *t+4* and are available to the commit stage of the pipeline at time *t+5*. In the case of a miss in the ICM cache, a memory request is placed to fetch the missing instruction. The data received from memory is placed in an internal buffer and then stored to the ICM cache.

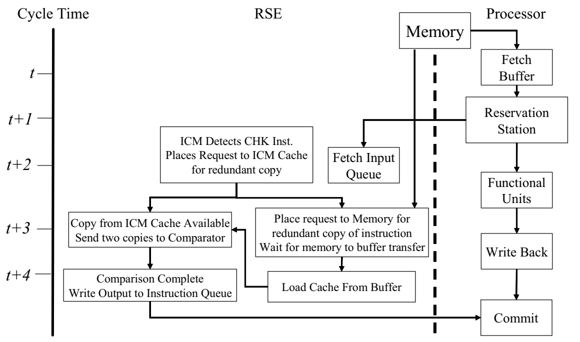

Cycle Time
RSE
Processor
Memory
t
t+1
t+2
t+3
t+4
Fetch Buffer
Reservation Station
ICM Detects CHK Inst. Places Request to ICM Cache for redundant copy
Fetch Input Queue
Functional Units
Copy from ICM Cache Available Send two copies to Comparator
Place request to Memory for redundant copy of instruction Wait for memory to buffer transfer
Write Back
Comparison Complete Write Output to Instruction Queue
Load Cache From Buffer
Commit

Figure 5: Timeline for ICM Execution

5. Experimental Setup and Results

The experiments to evaluate the Reliability and Security Engine can be classified into the following three categories:

1. *Framework Overhead.* Measure the overhead incurred by the processor due to the presence of the framework without any modules instantiated.
2. *Cache Performance Overhead.* Measure performance impact of CHECK instructions (now part of the application stream) inserted into the application by the compiler.
3. *Performance Overhead Due to Modules.* Measure performance overhead of individual RSE modules.

5.1. Simulation Setup

The *sim-outorder* processor performance simulator of the SimpleScalar Tool Set [3] has been augmented to simulate the RSE with embedded hardware modules. *sim-outorder* simulates an out-of-order pipelined processor. Currently, CHECK instructions are embedded at runtime, not at compile time. When an instruction is fetched, the simulator determines (either by decoding the instruction or by monitoring the fetched instruction address) whether the instruction has to be checked and, if so, inserts a CHECK instruction before it into the instruction stream. The experiments were performed by running the SPEC200-INT *vpr* (Placement and Routing) and the *kMeans* application, which implements the K-Means clustering algorithm.

Cache overhead simulation. We measure the impact on cache performance by rewriting the code segment of the process. NOP instructions are inserted wherever a CHECK instruction has to be placed and the code is executed on the baseline simulator. A fetch for an instruction from the simulator is redirected to the new code segment. The NOP instructions are also fetched into the I-cache, simulating the effect of the CHECK instructions.

5.2. Results

Table 4 presents the experimental results for the benchmark applications. Row 2 shows the number of cycles (in millions) for the execution of the benchmark on the baseline simulator. Row 3 shows the number of cycles with the simulator augmented with the framework but with no modules instantiated. Row 4 shows the number of cycles with the simulator augmented with the framework, including the ICM module. The benchmark is instrumented to check all control-flow instructions. The subsequent rows, 5 and 6, give the percentage overheads for the above two configurations. The next set of rows present the results for the cache performance overhead. The number of accesses and miss rates for the il1 and il2 caches are tabulated.

The framework overhead is attributed to delay of the memory arbiter, which is introduced for arbitration of memory access between the cache and the RSE Manager. Memory access is pipelined and in the baseline case, the memory access latency for the first chunk is 18 cycles; each of the subsequent chunks arrives 2 cycles after the previous one. The memory access latency for the first chunk and the inter-chunk latency have been changed to 19 and 3 cycles, respectively. In arriving at these numbers, delay due to the arbiter has been assumed to be 1 cycle. The framework overhead is seen to be 3.47% for vpr Placement, 3.64% for vpr route, and 4.99% for kMeans, averaging to 4.03%.

The ICM has been simulated with an *ICM_Cache* size of 256 and a replacement size of 8 least-recently-used entries. The overhead due to the ICM (combined with that due to the framework) is 11% for vpr placement, 7.7% for the vpr route, and 5.44% for kMeans. The average overhead is 8.1%. Preemptive checking, which protects against uncontrolled crashes, is expensive [6]. In this context, this overhead is small.

5.3. Performance and Hardware Overhead of MLR Module

The random relocation of the position-independent regions, i.e., stack and heap, has been implemented in our simulator. The random relocation of the GOT and shared libraries requires support for executable format with dynamically loaded objects from the simulator.

The penalty for position independent regions is fixed and was found to be 56 cycles. The position dependent part of the algorithm (GOT randomization) is only applicable to dynamically linked executables. The *SimpleScalar* simulator does not support dynamic linking. Given the goal of the evaluation, i.e., to measure the time it takes for RSE to do the GOT/PLT randomization and to compare that to a pure software implementation, we use a method equivalent to a real dynamic-linking implementation. The proposed approach embeds the dynamic linking mechanism and the randomization algorithm inside a target application, creating an application private dynamic loader. The approach avoids making significant changes in a third-party simulator but allows us to evaluate the performance of the algorithms in a realistic way.

Table 5 shows the performance results (in #cycles) from using the approach outlined. The *TRR* columns show the result for the pure software implementation of the TRR algorithm, while the *RSE* columns show the results for the MLR module. The performance improvement from the TRR version to the RSE version is about 20-30% over the different numbers of GOT entries. It should be noted that the goal is to randomize the memory layout of the process each time it is loaded; hence, performance overhead is incurred only during the initialization of the process when the segments of the process are set up. No overhead is incurred during the normal execution of the process.

Table 4: Framework Evaluation Results

Benchmark		VPR-Place	VPR-Route	kMeans
# Cycles (in millions)	Baseline	32.91	6.92	0.26
	Framework	34.05	7.17	0.27
	Framework + ICM	36.54	7.45	0.27
Framework % Overhead		3.47%	3.64%	4.99%
Framework + ICM % Overhead		11.04%	7.73%	5.44%
#il1.accesses (in millions)	Baseline	40.46	11.71	0.29
	With CHECKs	49.64	14.77	0.34
il1.missrate	Baseline	5.24	1.16	0.06
	With CHECKs	6.01	1.75	0.06
#il2.accesses (in millions)	Baseline	21.22	1.36	0.02
	With CHECKs	29.82	2.58	0.02
il2.missrate	Baseline	4.06	5.65	1.06
	With CHECKs	1.45	4.31	1.04

The hardware required for the MLR module for randomization of position-independent regions (from Figure 3) is 24 word-length registers, 4 adders, and a 4KB memory block for holding the executable header (4KB will accommodate any header for any operating system and file format). Also, the hardware requirements for the randomization of the position-dependent regions are a 4KB memory block to hold the GOT table, a second 4KB memory block to hold the PLT Table, 5 adders, and 2 word-length registers.

5.4. Performance Evaluation of the DDT

To implement the Data Dependency Tracking module, the simulator is augmented to enable execution of multithreaded applications with networking capabilities. The performance overhead of the DDT is measured using a multithreaded network server, depicted in Figure 6, which shows an example scenario involving five threads. Here t_2 writes to page *p1*, t_1 subsequently reads *p1*, which causes the dependency $t_2 \rightarrow t_1$. Similarly, $t_1 \rightarrow t_0$ because t_0 reads the page *p2* written by t_1, and $t_0 \rightarrow t_1$ because they are related by page *p3*. Let t_2 be the faulty thread. When t_2 crashes at the checkmark (x), t_0 and t_1 are dependent on t_2. The recovery algorithm will terminate t_0, t_1 and t_2, undo any memory updates by these three threads, and continue executing the remaining two threads, t_3 and t_4 from where they are last suspended by the scheduler. The recovery line in this case is only for the two surviving threads.

We vary the number of threads and measure the time for the server to handle 100 requests. The measurements are shown in Figure 7. The bottom two curves show the execution time with and without DDT support.

Table 5: Performance of the MLR module

GOT Entries	#Cycles			#Instructions		
	TRR	RSE	Gain	TRR	RSE	Gain
128	11165	9139	18%	9597	6295	34%
256	13099	10018	24%	12808	6242	51%
512	17037	11907	30%	19311	6217	68%
1024	24580	18131	26%	32245	6095	81%

In both cases, the running time decreases as more threads are added to the pool, and it stabilizes when four or more threads are in the system. Adding more threads allows exploiting more I/O parallelism in thread execution (initial decrease in execution time). After the number of active threads reaches a threshold (four in our simulation) beyond which no more parallelism can be exploited, the thread scheduling overhead comes into play, and addition of new threads does not change the execution time. The performance overhead of DDT, as compared with the baseline case, while initially low, climbs to about 7-8% after most of the thread parallelism is exploited. This overhead is mainly attributed to saving memory pages. The top curve shows the number of memory pages saved during the execution. This number is small initially and increases when more threads are added, since more instances of sharing are possible when threads compete for jobs.

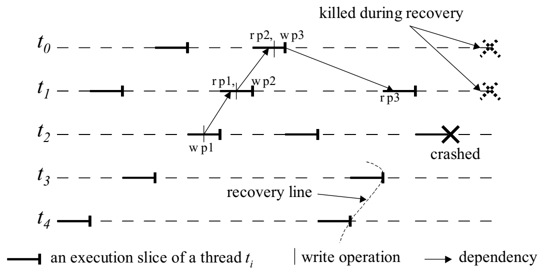


Figure 6: DDT Example

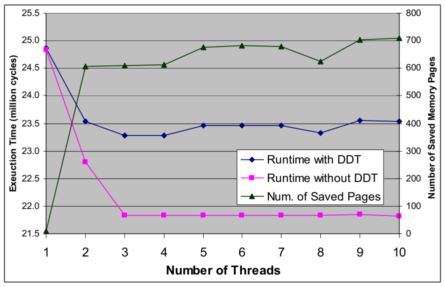


Figure 7: Performance Evaluation for DDT

6. Conclusions

This paper introduces RSE, an architectural framework for providing security and reliability support without sacrificing performance. The framework: (i) hosts hardware modules that provide reliability and security services, and (ii) implements input/output interface between the modules and the main pipeline and between the modules and the executing software (operating system and application). The paper describes in detail the RSE architecture, discuses hardware issues (e.g., implications to the main pipeline) in designing the framework, and then introduces hardware modules for security and reliability services. Simulation is used to evaluate the performance of the proposed solution. The performance results indicate that hardware solutions provide low overhead mechanisms for protecting against a broad range of security vulnerabilities and accidental error. Further work will focus on incorporating new security and reliability hardware support and translating the simulated architecture into a real hardware implementation.

Acknowledgements

This work was supported in part by Gigascale Systems Research Center (GSRC) and in part by NSF grant ACI-0121658 ITR/AP. We thank F. Baker and T. O' Neill for their careful reading of this manuscript.

References

[1] CERT/CC. CERT Advisories. http://www.cert.org/advisories/.
[2] A. Baratloo et al, "Transparent Run-Time Defense Against Stack Smashing Attacks," in Proc. of USENIX, June 2000.
[3] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0," TR-CS-1342, Univ of Wisconsin, June 1997.
[4] C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in Proc. 7th USENIX Security Conference, 1998.
[5] J. Dyer et al, "Building the IBM 4758 Secure Coprocessor," IEEE Computer, 34(10), Oct. 2001.
[6] S. Bagchi et al., "A framework for database audit and control flow checking for a wireless telephone network controller," in Proc. of DSN-2001.
[7] Z. Kalbarczyk et al. "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," IEEE Transactions on Parallel and Distributed Systems, 10(6), June 1999.
[8] M. Cukier et al. "AQuA: An Adaptive Architecture That Provides Dependable Distributed Objects," in Proc SRDS-17, Oct. 1998.
[9] M. Kasbekar, et. al, "Selective Checkpointing and Rollbacks in Multi-Threaded Object-Oriented Environment," IEEE Transactions on Reliability, 48(4), 1999.
[10] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," IEEE Transactions on Computers, 37(2), 1988.
[11] K. Wilken, T. Kong, "Concurrent Detection of Software and Hardware Data-Access Faults," IEEE Transactions on Computers, 46(4), April 1997.
[12] T. N. Vijaykumar, Irith Pomeranz, Karl Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," in Proc. ISCA-29, May 2002.
[13] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in Proc. ISCA-27, June 2000.
[14] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," in Proc. FTCS-29, 1999.
[15] E. N. Elnozahy, et al, "A Survey of Rollback-Recovery Protocols in Message Passing Systems," TR CMU-CS-96-144, Aug. 1996.
[16] Aleph One. Smashing the Stack for Fun and Profit. Phrack Magazine, 49(7), Nov. 1996.
[17] OpenWall Project. Linux Kernel Patch from the OpenWall Project.
[18] L. Spainhower and T.A. Gregg. "G4: A Fault Tolerant CMOS Mainframe," in Proc. FTCS-28, 1998.
[19] M. Mueller et al. **"**RAS strategy for IBM S/390 G5 and G6**,"** IBM Jnl of Research and Development, 43(5/6) pp. 875-888
[20] C. Weaver and T. Austin, "A Fault Tolerant Approach to Microprocessor Design," in Proc. DSN 2001, pp. 411-420.
[21] J. Xu, et al., "Compiler and Architecture Support for Defense against Buffer Overflow Attacks," in EASY Workshop 2002.
[22] J. Xu, Z. Kalbarczyk, R. Iyer, "Transparent Runtime Randomization for Security," in Proc. SRDS-22, Oct. 2003.
[23] Ruby B. Lee et al., "Enlisting Hardware Architecture to Thwart Malicious Code Injection," in Proc SPC, March 2003.
[24] M. Prvulovic, Z. Zhang, J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," in Proc. ISCA-29, May 2002.
[25] D. B. Hunt and P. N. Morinos. "A general purpose cache-aided error recovery (CRER) technique," in Proc. FTCS-17, 1987.
[26] Bob Janssens and W. Kent Fucks. "The Performance of Cache-Based Error Recovery in Multiprocessors," IEEE Transaction on Parallel and Distributed Systems. 5(10), Oct 1994.
[27] H. Eveking. "SuperScalar DLX Documentation." http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/DlxPdf.zip
[28] J. Xu, "Software and Hardware Mechanisms for Masking Security Vulnerabilities," Ph.D. Thesis University of Illinois, 2003.
[29] N. Nakka, Z. Kalbarczyk, R. K. Iyer, J. Xu, "An Architectural Framework for Providing Reliability and Security Support". http://www.crhc.uiuc.edu/~nakka/DSN2004_RSE.pdf