Reliability and Security: From Measurements to Design

Ravi K. Iyer
Karthik Pattabiraman, Weining Gu,
Giampaolo Saggese, Zbigniew Kalbarczyk
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Supported by: NSF, SRC, DARPA, SUN, IBM, HP

http://www.crhc.uiuc.edu/DEPEND
Crash Latency Distributions for
(Linux on Pentium P4 and PowerPC G4)

Early detection of kernel stack overflow on PPC major contributor to reduced crash latency
Crash Severity: Linux Kernel on Pentium

- Significant percentage (33%) of errors that alters the control path have no effect
  - Inherent redundancy in the code
- The most severe crashes are due to reversing the condition of a branch instruction
- The most severe crashes require a complete reformatting of the file system on the disk
  - Can take nearly an hour to recover the system
  - Profound impact on availability
  - To achieve 5NINES of availability (5 minutes/yr downtime) one can effort one such failure in 12 years
Crash Causes:
Linux on PowerPC G4 & Pentium 4

- NULL Pointer: NULL pointer de-reference;
- Bad Paging: Bad paging (except NULL pointer)
- General Protection Fault: Exceeding segment limit;
- Kernel Panic: Operating system detects an error;
- Invalid TSS: Selector, or code segment outside limit;
- Bounds Trap: Bounds checking error.

- Bad Area: Bad paging including NULL pointer;
- Stack Overflow: Stack pointer of a process out of range
- Machine Check: Errors on the processor-local bus;
- Alignment: Load/store operands not word-aligned;
- Bus Error: Protection faults;
- Bad trap: Unknown exceptions.
Breakdown of Vulnerabilities (*Bugtraq*)

- **Access Validation Error**: an operation on an object outside its access domain.
- **Atomicity Error**: code terminated with data only partially modified as part of a defined operation.
- **Boundary Condition Error**: an overflow of a static-sized data structure: a classic buffer overflow condition.
- **Configuration Error**: a system utility installed with incorrect setup parameters.
- **Environment Error**: an interaction in a specific environment between functionally correct modules.
- **Failure to Handle Exceptional Conditions**: system failure to handle an exceptional condition generated by a functional module, device, or user input.
- **Input Validation Error**: failure to recognize syntactically incorrect input.
- **Race Condition Error**: an error during a timing window between two operations.
- **Serialization Error**: inadequate or improper serialization of operations.
- **Design Error** and **Origin Validation Error**: Not defined.

*Bugtraq* database included 5925 reports on software related vulnerabilities (as of Nov.30 2002)
Observations from Vulnerability Analysis

• Exploiting a vulnerability involves multiple vulnerable operations on several objects.

• Exploits must pass through multiple elementary activities, each providing an opportunity for performing a security check.

• For each elementary activity, the vulnerability data and corresponding code inspections allow us to define a predicate, which if violated, naturally results in a security vulnerability.
Example: FSM Model for the *Sendmail* Vulnerability

**Operation 1:**
Write integer *i* to *tTvect*[x]

- **pFSM₁**
  - Get text strings *str_x* and *str_i*
  - (integer represented by *str_x*) > 2³¹
    - Convert *str_i* and *str_x* to integer *i* and *x*
  - (integer represented by *str_x*) ≤ 2³¹

- **pFSM₂**
  - *x* < 0 or *x* > 100
  - 0 ≤ *x* ≤ 100

- **pFSM₃**
  - *addr_setuid* changed
  - *addr_setuid* unchanged

- **Function pointer is corrupted**
  - Executing *MCode*

**Operation 2:**
Manipulate the function pointer

- Load the function pointer
- *addr_setuid* changed
- *addr_setuid* unchanged
- Execute code referred by *addr_setuid*

- Executing *MCode*
Some Lessons Learned

- Extracted common characteristics of a class of security vulnerabilities
- Developed an FSM methodology to model vulnerabilities.
- Only three pFSM types were required. Enforced reasoning indicate opportunities for security checking.
- Most vulnerabilities are in the interface between applications and library functions
- **Question:** Can we develop *Vulnerability-Masking* schemes based on the observed characteristics
Challenges: Understanding Failure Data

• **Expectation is that transients will increase**
  – Shrinking device size $\rightarrow$ Increased transient error rate
    • More error checking that is closer to processor needed

• **System level impact of increase in transients**
  – Increased error propagation $\rightarrow$ near-coincident (correlated) errors
  – More latent errors
  – Question: What are the corresponding high level fault models?

• **Current recovery techniques oriented towards single isolated errors**

• **Recovery of correlated (or latent) errors is complex and adds significantly to unavailability**
Challenges: Understanding Attack Data

• Analysis of data (from Bugtraq) on security attacks to:
  – identify vulnerabilities and to classify the attacks according to attacks causes
  – understand potential inconsistencies in application/system specifications resulting in security vulnerabilities of an actual application/system implementation

• Measurement-based models depicting the attack process

• Software (e.g., compiler-based) and hardware (e.g., processor embedded) vulnerability masking/prevention techniques
What is Needed?

• **Application aware detection mechanisms**
  – generic fault-tolerance and security techniques, targeting a particular fault/attack-model provide limited coverage
  – application cannot selectively take advantage of mechanisms, which best meet the needs

• Extract application properties that can be used as an indicator of correct behavior

• Exploit the knowledge of such properties to derive efficient error detection
  – application-specific checks can complement the coverage provided by generic techniques

• Assess the benefits (tradeoffs) of software or hardware implementaion
Application Aware Checking in Software: ARMOR Self-checking Middleware

• Adaptive Reconfigurable Mobile Objects of Reliability
  – Processes composed of replaceable software modules.
  – Provide error detection, recovery and security services to user applications.

• ARMORs Hierarchy form runtime environment:
  – System management, detection, and recovery services distributed across ARMOR processes.
  – ARMORs resilient to their own failures.
ARMOR Self-checking Middleware: “Embedded Solution”

- Modular design of ARMOR processes around elements lends itself well to small footprint solutions.
- Special versions of elements optimized for memory and performance requirements.
- Specialized microkernel:
  - Remove support for inter-ARMOR communication through regular messaging infrastructure
  - Static configuration of elements; no need to dynamically add/remove elements
Application Aware Checking in Hardware: Reliability and Security Engine

For Input Interface;
Queue Size = 16
32-bit regs = 80;
Gate Count = 12800

Reliability and Security Engine

- 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.
- FPGA implementation as an integral part of a superscalar microprocessor
- Hardware-implemented error-detection and security mechanisms embedded as FPGA modules in the framework
- The framework serves two purposes
  - Hosts hardware modules that provide reliability and security services, and
  - Implements interface of the modules with the main pipeline and the executing software (OS and application)
TRUSTED ILLIAC

COMBINING HIGH PERFORMANCE WITH APPLICATION-AWARE RELIABILITY AND SECURITY
HTTP://WWW.CSL.UIUC.EDU
Goal: Application-Aware Trusted Computing

- Create a large, demonstrably-trustworthy, enterprise computing platform
  - Application aware reliability and security
  - Reconfigurable
  - High performance
  - Easy programming

- Support for
  - Enterprise computing with seamless extension across wireline-wireless domains
  - Significant number of applications that co-exist and share the HW/SW resources

- State of the Art: Provide HW and SW with a one-size-fits-all approach
  - Creating a trustworthy environment is complex, expensive to implement and difficult to validate
Application Aware Trusted Computing

• Applications-specific level of reliability and security provided in a transparent manner, while delivering optimal performance

• Customized levels of trust (specified by the application)
  – enforced via an integrated approach involving
    • re-programmable hardware,
    • compiler methods to extract security and reliability properties
    • configurable OS and middleware

• Scale from few nodes to large networked systems
• Enable inclusion of ad-hoc wireless nodes
Application-Aware Checking: An Example

On-core approach – processor, framework, and modules part of the same core.

A Reliability and Security Engine (RSE)
- Reconfigurable processor-level hardware framework
- Provides HW modules for reliability and security
- Modules and framework interface configured to meet application demands

Assertion-Based Checking
- Automatic generation and software/hardware implementation of error detectors

Application-Failure Mitigation Driver
- Crash detection
- Hang detection
- OS driver
- OS kernel
- OS level error detection/recovery
- Application-transparent OS-level checkpointing
- OS health monitoring
• Seamless integration of hardware accelerators into the Linux software stack
• Compiler supported deep program analysis and transformations to generate CPU code, hardware library stubs and synthesized components
• OS resource management
Validation Framework

• An integral part of the Trusted ILLIAC
• Quantitative assessment of alternative designs and system solutions
• Provides tools for
  – Analytical models (e.g., MOBIUS)
  – Simulation (e.g., RINSE)
  – Experimental validation (e.g., NFTAPE)
    • Fault/error injection
    • Attack generation
  – Monitoring
  – Measurement
• Crucial in making design decisions, which require understanding tradeoffs such as cost (in terms of complexity and overhead) versus efficiency of proposed mechanisms.
Trusted *ILLIAC*: The Broader Context

- New experience in system building: reliable and secure processing architectures, smart compilers combined with configurable OS and hardware
- Pushing the boundaries in customizable trusted computing technologies
- Enable university, industry, and government collaboration
- Train the next generation of students and professionals
- (See next slide)
Example: Trusted ILLIAC Node