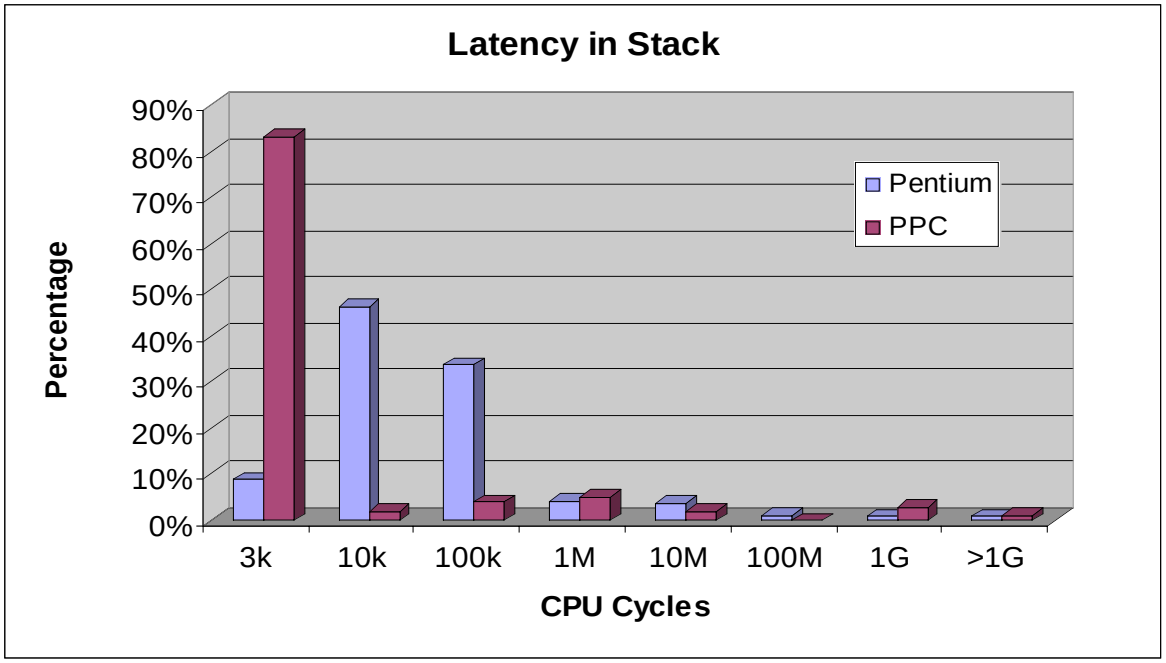
Application Aware Detection: The Trusted ILLIAC Approach

Ravi K. Iyer
(Joint work with Wen Mei Hwu, Klara Narhrstedt, Z. Kalbarczyk, William Sanders
Coordinated Science Laboratory and
The Information Trust Institute
University of Illinois at Urbana-Champaign

http://www.csl.uiuc.edu

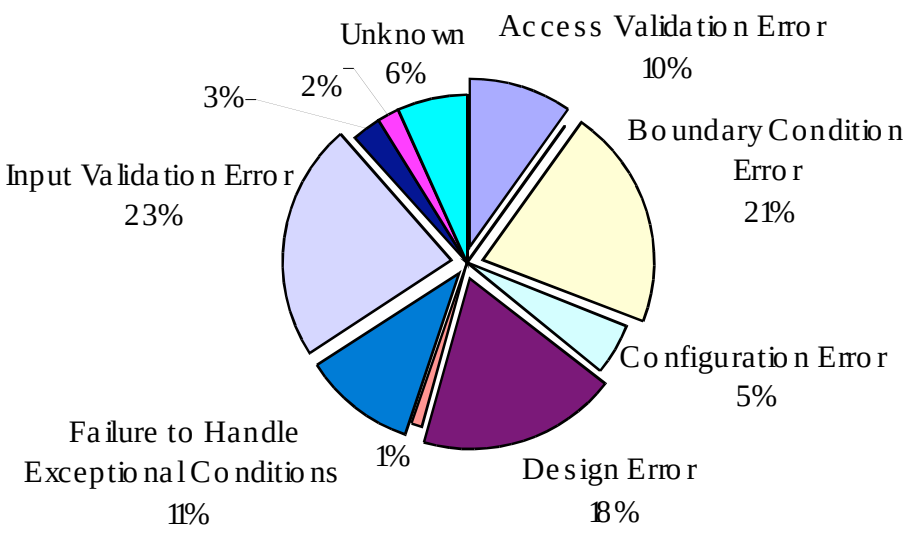
ILLINOIS

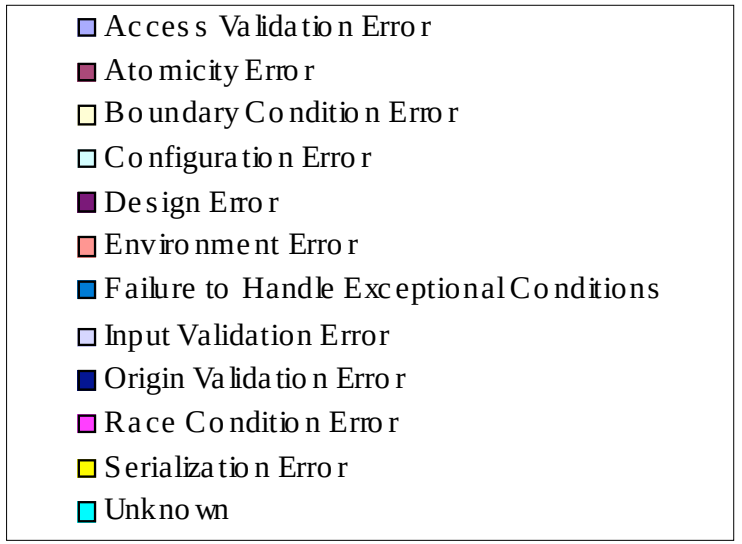
Crash Latency Distributions for
(Linux on Pentium P4 and PowerPC G4)



Early detection of kernel stack overflow on PPC major contributor to reduced crash latency

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.

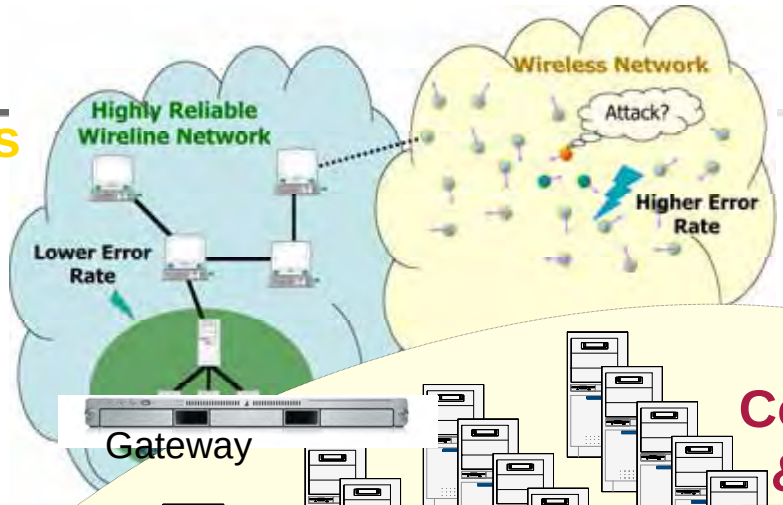
Bugtraq database included 5925 reports on software related vulnerabilities

ILLINOIS

Trusted *ILLIAC*: Application Domains

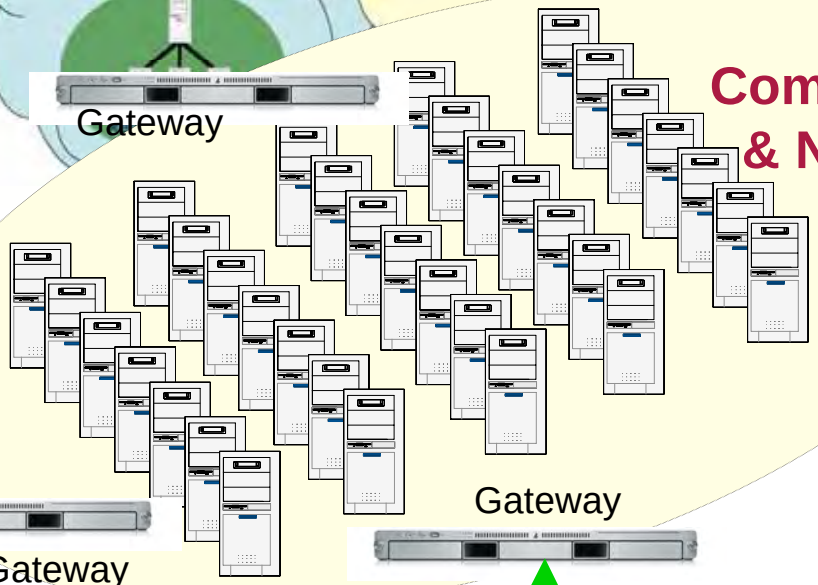
Heterogeneous User and Information Base

- Multilevel/hierarchical information abstraction
- Integrated analysis and visualization
  - automated and with human in the loop

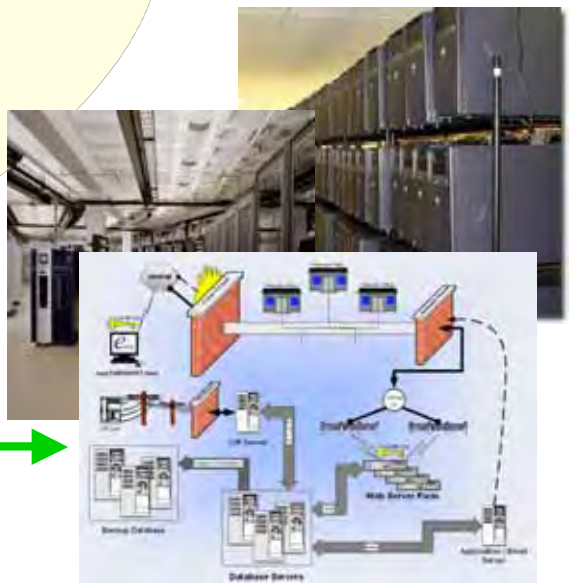


- Wireline and wireless users
- Multiple computing platforms
- Distributed communications
- Heterogeneous information
- Variable level of trust

Computing & Network Fabric



Human Machine Interface



- Data Centers: High fidelity, reliable and secure data mining, processing and communication engines
- Data caching and authentication of information

Trusted Information Management

ILLINOIS

TRUSTED *ILLIAC*

APPLICATION AWARE RELIABLE AND SECURE COMPUTING

Goal: Application-Centric Trusted Computing

- Create a large, demonstrably-trustworthy, computing platform
  - Application centric reliability and security
  - Reconfigurable; High performance
- Support for
  - Enterprise computing with seamless extension across wireline-wireless domains
  - Applications: Services, Client specified level of privacy and security
- Educate a new generation of students
- Underlying Research Support: NSF, HP, AMD, IBM, Intel, GSRC. Illinois, Commerce Dept..

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,
    - New compiler methods to extract security and reliability properties
    - Run time framework to enforce diversity
    - configurable OS and middleware
- Scale from few nodes to large networked systems
- Enable inclusion of ad-hoc wireless nodes

Checks

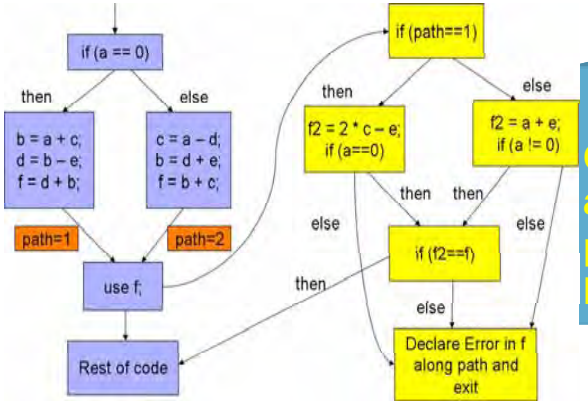
Reliability

Identify critical variables and their location within a program to place detectors for best coverage

Security

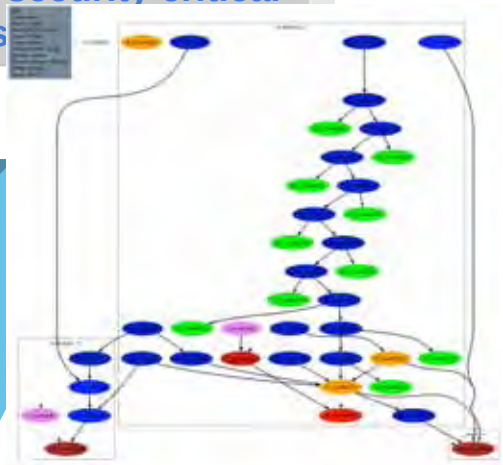
- Construct dynamic dependence graph of the program via profiling
- Apply heuristics, e.g., *fanouts* metric, to identify critical variables

- Use knowledge of the application semantics to identify security critical variables, e.g., a pass



Static program analysis

Compute backward slice of program variable along each program path starting from the program point at which the critical variable is located

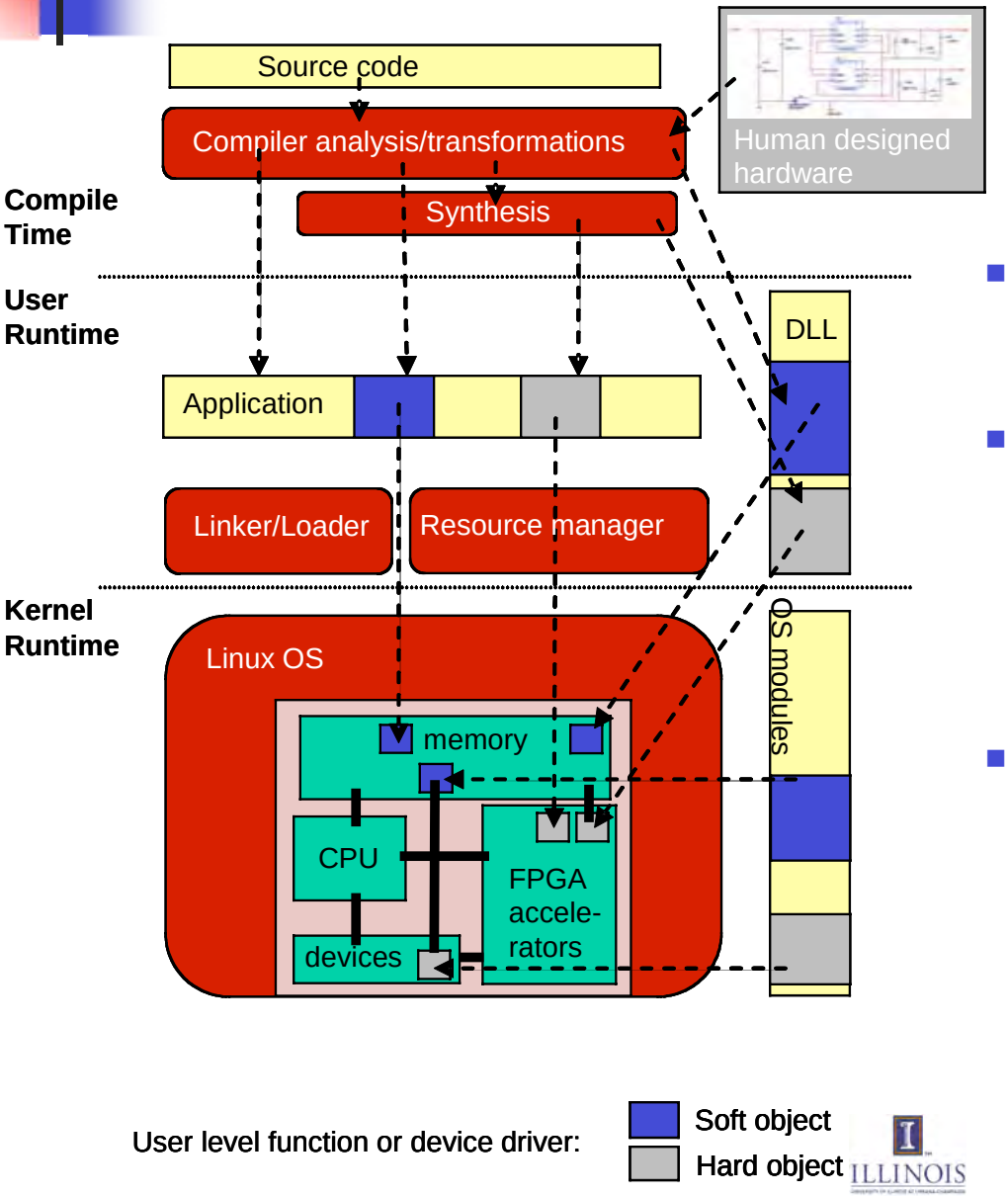


- Generate correctness checks for data values in critical program locations
- Check encoded as path optimized sequence of instructions

- Generate checks to verify that the value is produced by a legitimate set of instructions
- Check encoded as the set of objects to which an instruction is allowed to write

- Runtime checking
- Implementation in software or programmable hardware

Hardware/Software Execution Model



- 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

ILLINOIS

Model-Driven Trust Management

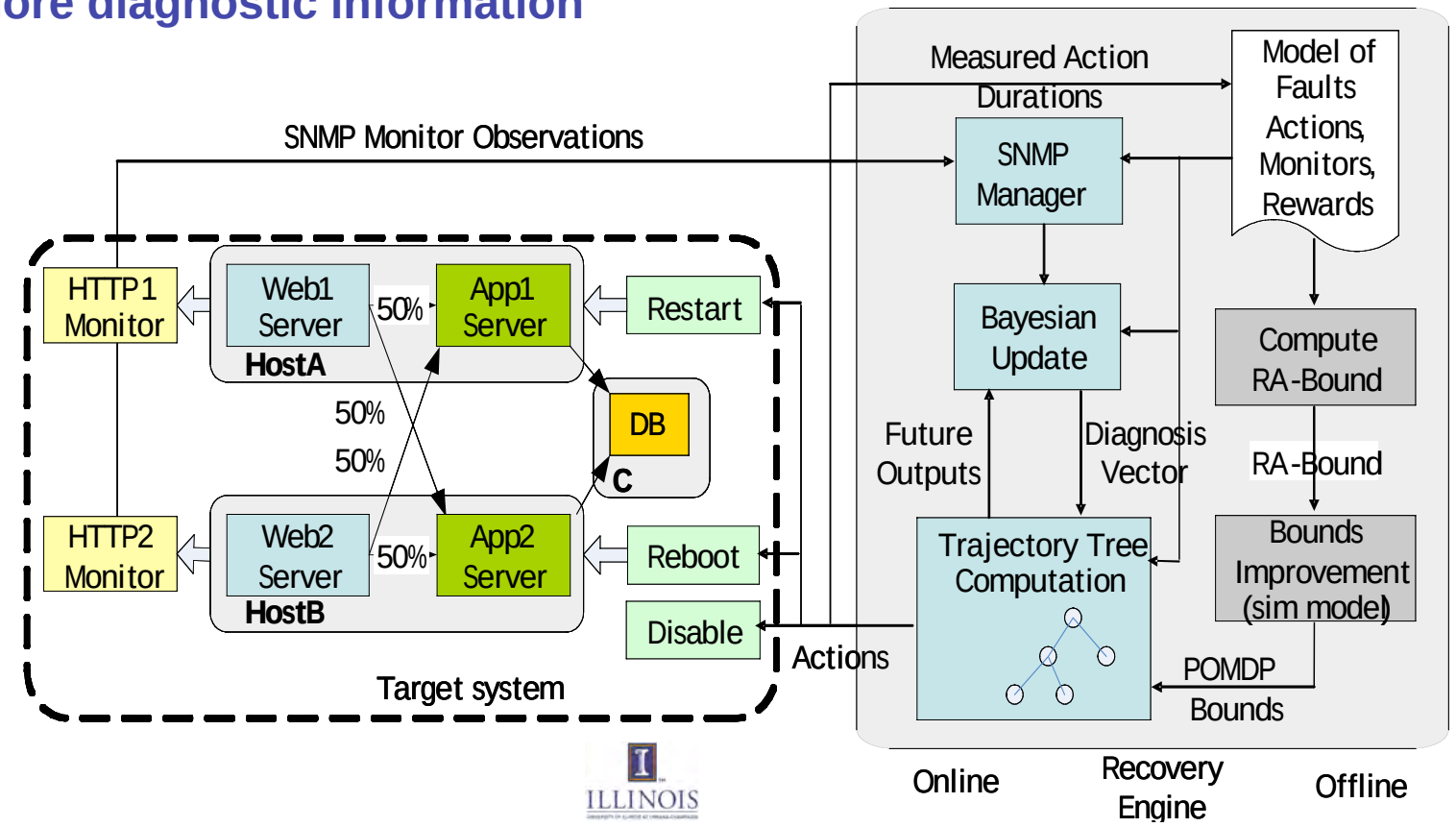
Preserving system health using adaptive recovery

… *when* the precise cause of failure is unknown

- Monitoring in one layer, fault in another
- Poor localization, false positives and negatives

… *when* several recovery options are available

- Restart or fail-over of component, host, entire system
- Get more diagnostic information

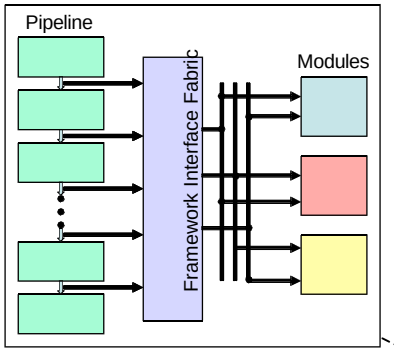


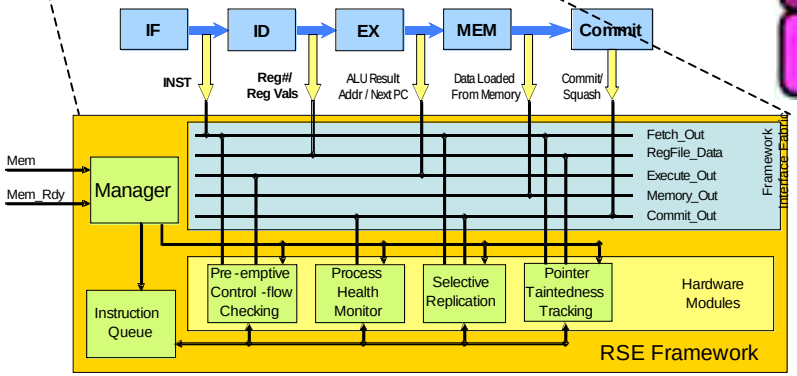
Validation Framework

- 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
  - Run-time monitoring and Diagnosis
  - Measurement and Benchmarking
- Crucial in making design decisions, which require understanding tradeoffs such as cost (in terms of complexity and overhead) versus efficiency of proposed mechanisms.

Application-Aware Checking: An Example

Processor, framework, and modules part on the same core.



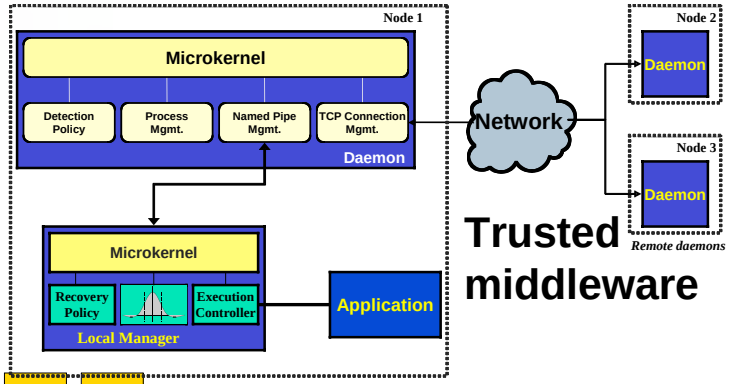


Reliability and Security Engine

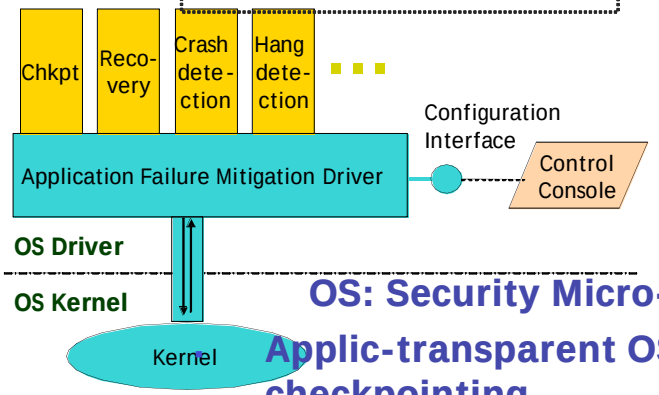
- **Reconfigurable processor-level hardware framework**
- **Provides HW modules for reliability and security**
- **Modules and framework interface**

Assertion-Based Checking

- **Automatic generation and software/hardware implementation of error detectors**



Trusted middleware



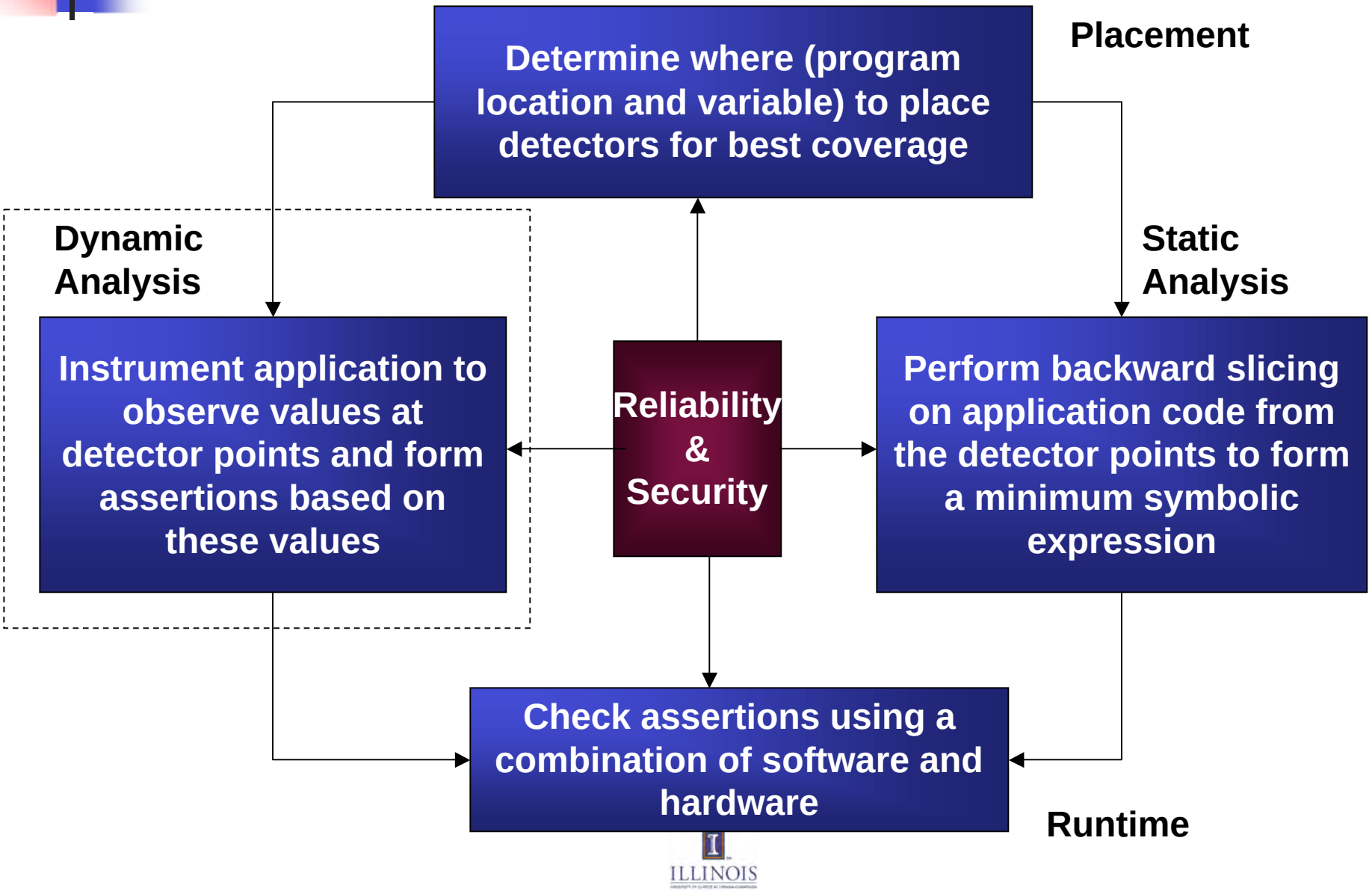
- **OS: Security Micro-Kernel**
- **Applic-transparent OS-level checkpointing**
- **OS health monitoring**

ILLINOIS

Application-aware error detectors

- Provide application-specific error detection at low-cost for high-performance platforms
- Limit error propagation and reduce error detection latency
- Automatically derive fine-grained detectors to
  - Maximize error detection coverage
  - Minimize performance impact
- Implement in software / hardware

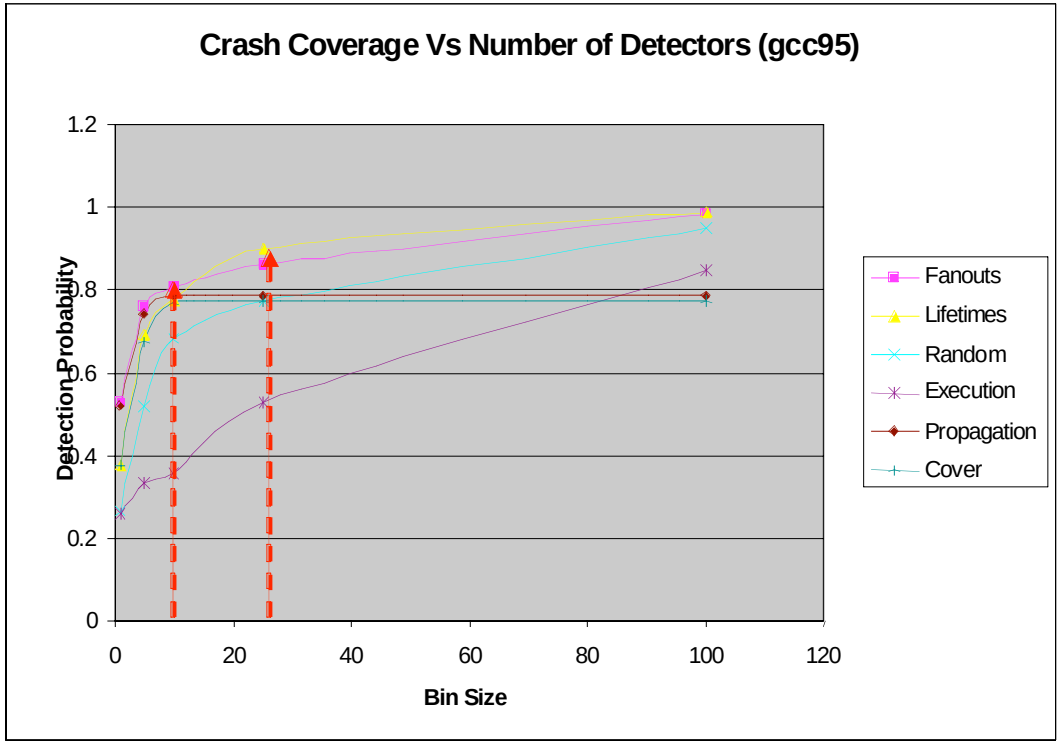
Approach


Determine where (program location and variable) to place detectors for best coverage
Placement
Dynamic Analysis
Static Analysis
Instrument application to observe values at detector points and form assertions based on these values
Reliability & Security
Perform backward slicing on application code from the detector points to form a minimum symbolic expression
Check assertions using a combination of software and hardware
Runtime
ILLINOIS

Where to Place the Detectors?

- Choose variable to check and location to place the detector
- Starting Point: construct Dynamic Dependence Graph of the program
- Compute metrics to choose candidate points for detector placement
  - e.g., fanout, lifetime
- Evaluate detectors placed according to different metrics
  - Fault-injections into data

Coverage for Multiple Detectors



gcc95 benchmark

- Coverage for crashes:
  - 80% with 10 detectors,
    97 % with 100 detectors
- Coverage for fail-silence violations (silent-data corruptions)
  - 60% with 10 detectors,
    80 % with 100 detectors
- Benign errors detected
  - 4 % with 10 detectors,
    10 % with 100 detectors
- Placing detectors randomly on hot-paths:
  - Need ~100 ideal detectors to achieve 90% coverage

ILLINOIS

Reliability Checks

- **Goal:** Automatically derive runtime error detectors based on application properties and implement them in hardware/software
- **Approach:**
  - Placement of error detectors for maximum coverage and to minimize error propagation
  - Dynamic learning approach to derive detectors for the critical variables/locations
  - Static program slicing techniques to form checking expressions for critical variables/locations
- **Faults addressed**
  - Hardware errors: computation errors, cache/memory errors, instruction fetch/devode errors, some control flow errors
  - Software errors: Uninitialized values, memory corruption errors, timing errors that impact values, Some semantic errors in program
- Implement checking expression in **hardware** as part of RSE

Example 1: C Code (matrix mult.)

```
void rInnerproduct(float *result, float a[rowsize+1][rowsize+1], float b[rowsize+1][rowsize+1], int row, int column) {
        /* computes the inner product of A[row,*] and B[*,column] */
        int i;
        *result = 0.0;
        for (i = 1; i<=rowsize; i++)
                *result = *result+a[row][i]*b[i][column];
}

void Mm (int run)    {
    int i, j;
    Initrand();
    rInitmatrix (rma);
    rInitmatrix (rmb);
    for ( i = 1; i <= rowsize; i++ )
            for ( j = 1; j <= rowsize; j++ )
                    rInnerproduct(&rmr[i][j],rma,rmb,i,j);
}
```

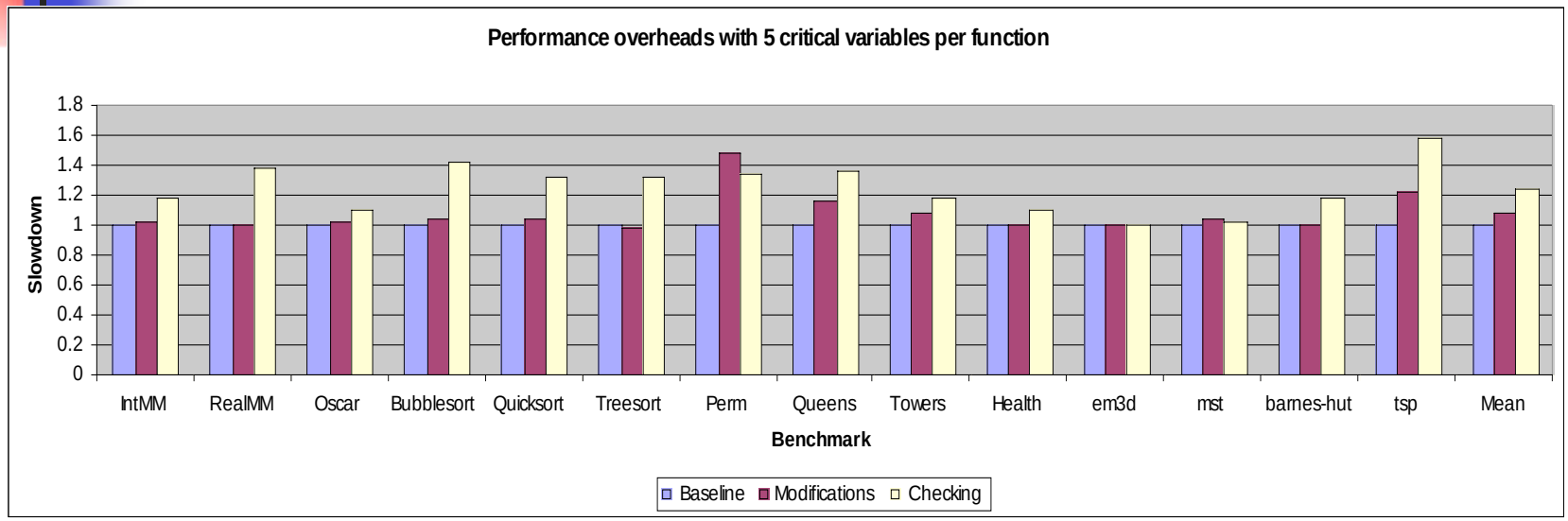
Example 2: Intermediate Code

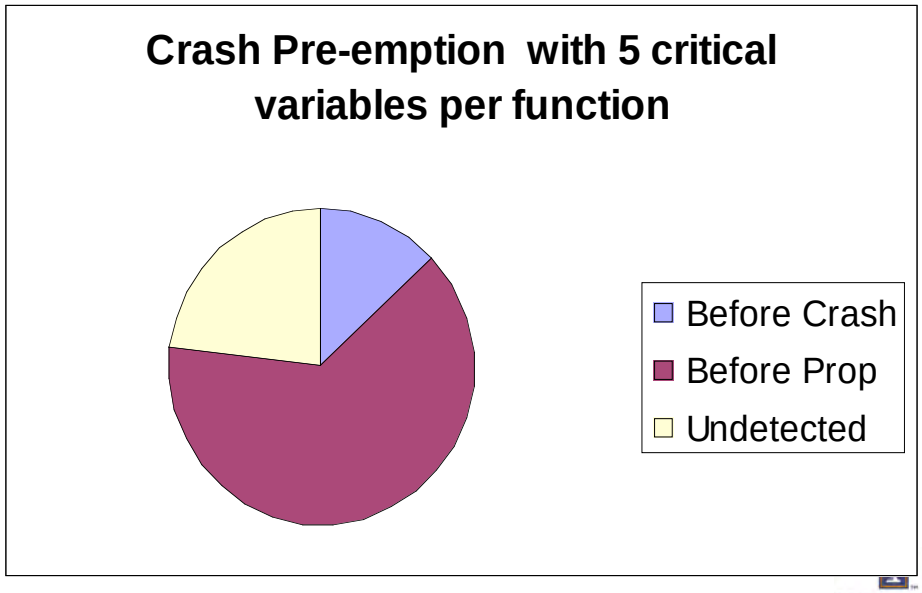
```
void rInnerproduct(double* result, double* a, double* b, int row, int column) {
loopentry:
        ....
        br  tmp.2, label no_exit, label loopexit
no_exit:
        ...
        tmp.7 = load a_addr
        tmp.8 = load row_addr
        tmp.9 = getelementptr tmp.7, tmp.8
        tmp.10 = load int* %i
        tmp.11 = getelementptr tmp.9, 0, tmp.10
        tmp.12 = load tmp.11
        tmp.13 = load b_addr
        tmp.14 = load i
        tmp.15 = getelementptr tmp.13, tmp.14
        tmp.16 = load column_addr
        tmp.17 = getelementptr tmp.15, 0, tmp.16
        tmp.18 = load tmp.17
        tmp.19 = mul tmp.12, tmp.18
        tmp.20 = add tmp.6, tmp.19
        store tmp.20, [ tmp.4 ]
        ....
        br label loopentry
```

Example 3: Transformed Code

```
tmp.20 = add tmp.6, tmp.19

switch pathValue {
            case 2: label path2-8
           case 3: label path3-8
            case 4, label path4-8
}

path2-8:
         new.2.tmp.19 = mul tmp.12, tmp.18
         new.2.tmp.20 = add 0.000000e+00, new.2.tmp.19
        br label Check-8

path3-8:
         new.3.tmp.19 = mul tmp.12, tmp.18
        new.3.tmp.20 = add tmp.20.copy, new.3.tmp.19
        br label Check-8

path4-8:
         new.4.tmp.19 = mul tmp.12, tmp.18
        new.4.tmp.20 = add  tmp.12.copy, new.4.tmp.19
        br label Check-8
```

Results: Crash Pre-emption





Average Performance Overhead

- Checking Overhead = 25%
- Modification Overhead = 8%
- **Total Overhead = 33 %**

Average Coverage (Crashes)

- Before Propagation = 64 %
- Before Crash = 13%
- **Total Coverage = 77 %**

ILLINOIS

Static Analysis for Security

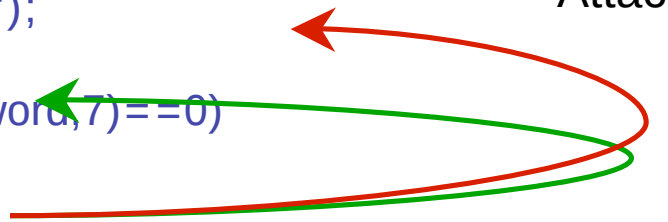
- Motivation: Prevent access to critical Data; Memory corruption attacks
- Goal is to preemptively protect "security-critical data" regardless of vulnerability
  - Can be accomplished by enforcing the source-code semantics on the program binary
- Approach: Encode the entire sequence of dependencies for the critical location, and check that the sequence is not violated during runtime.
  - Static Analysis is performed by the IMPACT compiler
  - Runtime Checking is performed as an RSE module

Information-Flow Signatures

- Use detection of *program data-flow violations* as an indicator of malicious tampering with the system
  - prevent an attacker to exploit disconnect between source-level semantics and execution semantics of the program
- *Security critical* variables chosen based on app semantics
- Employ a compile-time static program analysis to
  - extract a backward slice which collates all dependent instructions along each control-path
  - form a signature, which encodes dependences as a set (or sequence) of instruction PCs along each control-path
- Compute runtime signatures for each critical variable
  - *trusted bit* associated with each instruction
  - only trusted instructions can update runtime signatures
  - check signatures for instructions with trusted-bit set

Security Checking 1: How do signatures detect attacks ?

```
1  int main()
2  {
3   char password[8] = "asecret";
4   char userpass[8];
5   printf("Enter Password:\n");
6   gets(userpass);
8   if(strncmp(userpass,password,7)==0)
9      printf("Success\n");
10   else
11     printf("Failed\n");
12 }
```



Attacker enters “attack! attack!”

User enters “password”

Critical Variable: char password[8]; Signature: {3}

Security Checking 2: Why do we need to encode the entire dependency tree?

```
1   int authenticate(char* username, char* password)
2   {
3    int authenticated=0;
4    int result;
5    char tmpbuf[512];
6       result = !strncmp("asecret",password,7);
7      snprintf(tmpbuf,sizeof(tmpbuf),"user: %s",user);
8    tmpbuf[sizeof(tmpbuf)-1] = '\0';
9    syslog(LOG_NOTICE,tmpbuf);
10   authenticated |= result;
```

Attacker overwrites authenticated

Smarter attacker overwrites result instead, realizing that it can be used to influence authenticated via the format string

Critical Variable: char authenticated; Signature: {10,6},{3}

Critical Variable: char authenticated; Signature: {10},{3}

ILLINOIS

SSH Authentication Function

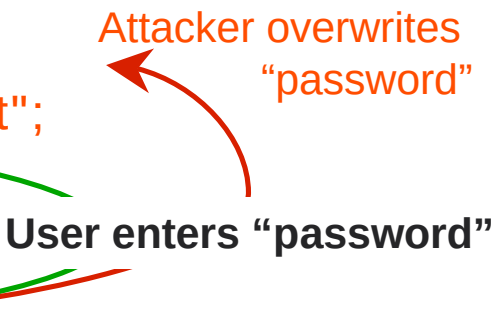
```
int sys_auth_passwd(Authctxt *authctxt, const char *password) {
1:          struct passwd *pw = authctxt->pw;
            char *encrypted_password;
            /* Just use the supplied fake password if authctxt is invalid */
2:          char *pw_password = authctxt->valid ? shadow_pw(pw) :
              pw->pw_passwd; /*Critical Variable Definition*/
            /* Check for users with no password. */
3:          if (strcmp(pw_password, "") == 0 && strcmp(password, "") == 0)
                        return (1);
            /* Encrypt the candidate password using the proper salt. */
4:          encrypted_password = xcrypt(password,
               (pw_password[0] && pw_password[1]) ? pw_password : "xx");
            /* Authentication is accepted if the encrypted passwords match */
5:    return (strcmp(encrypted_password, pw_password) == 0);
}
```

Critical Variable: pw_password

ILLINOIS

Security Checking: How Do Signatures Detect Attacks ?

```
1  int main()
2  {
3   char password[8] = "a_secret";
4   char userpass[8];
5   printf("Enter Password:\n");
6   gets(userpass);
8   if(strncmp(userpass,password,7)==0)
9      printf("Success\n");
10   else
11     printf("Failed\n");
12 }
```



Critical Variable: *char password[8];*
Signature: *{3}*

Security Checking: Why Do we Need to Encode Entire Dependency Tree?

```
1   int authenticate(char* username, char* password)
2   {
3    int authenticated=0;
4    int result;
5    char tmpbuf[512];
6      result = strncmp("asecret",password,7);
7     snprintf(tmpbuf,sizeof(tmpbuf),"user: %s",user);
8    tmpbuf[sizeof(tmpbuf)-1] = '\0';
9    syslog(LOG_NOTICE,tmpbuf);
10   authenticated |= result;
```

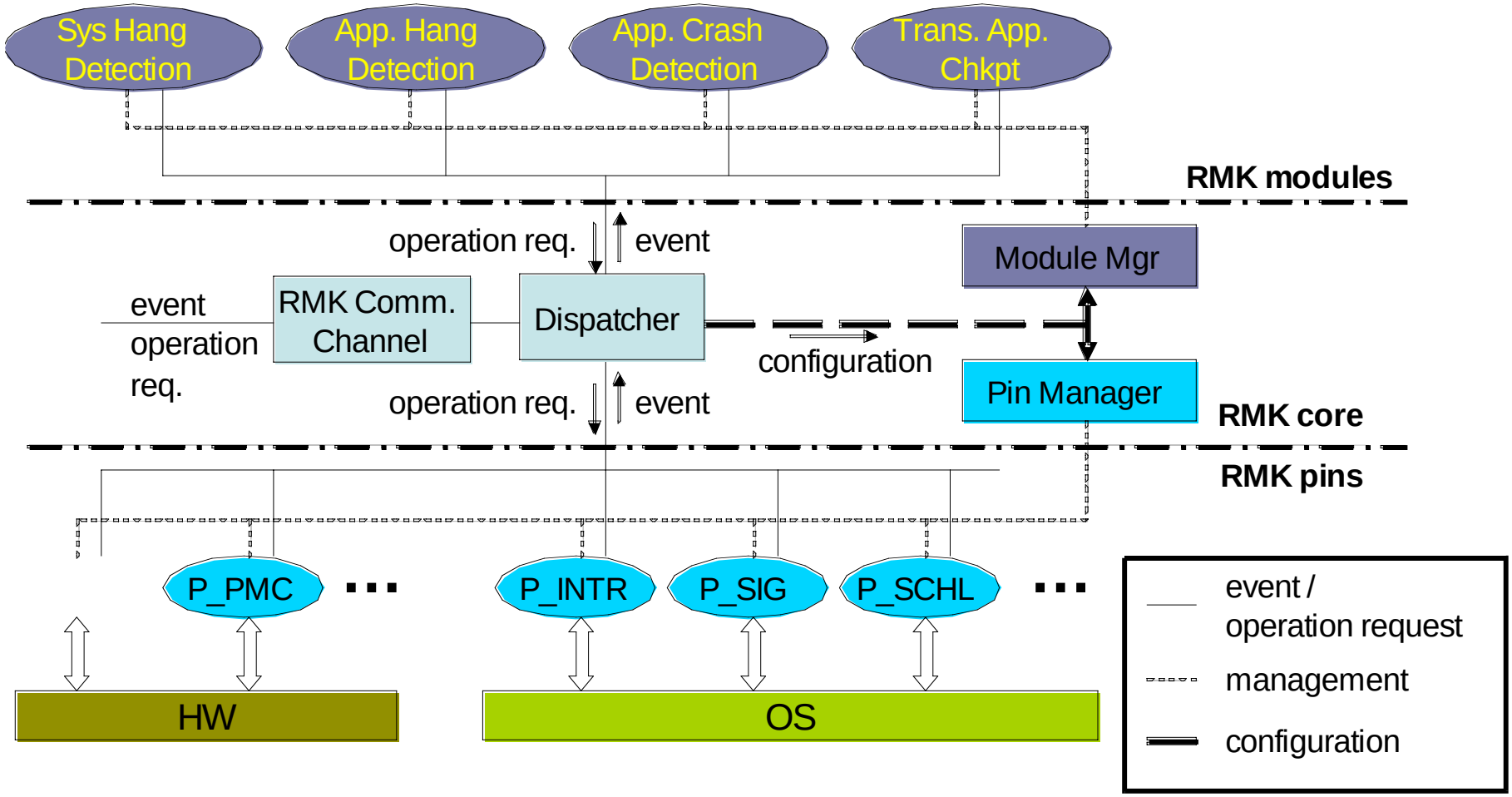
Attacker overwrites *authenticated* via the format string attack
Signature: {10},{3}

However, smarter attacker can overwrite *result* instead, realizing that it eventually writes *authenticated*
New Signature: {10,6},{3}

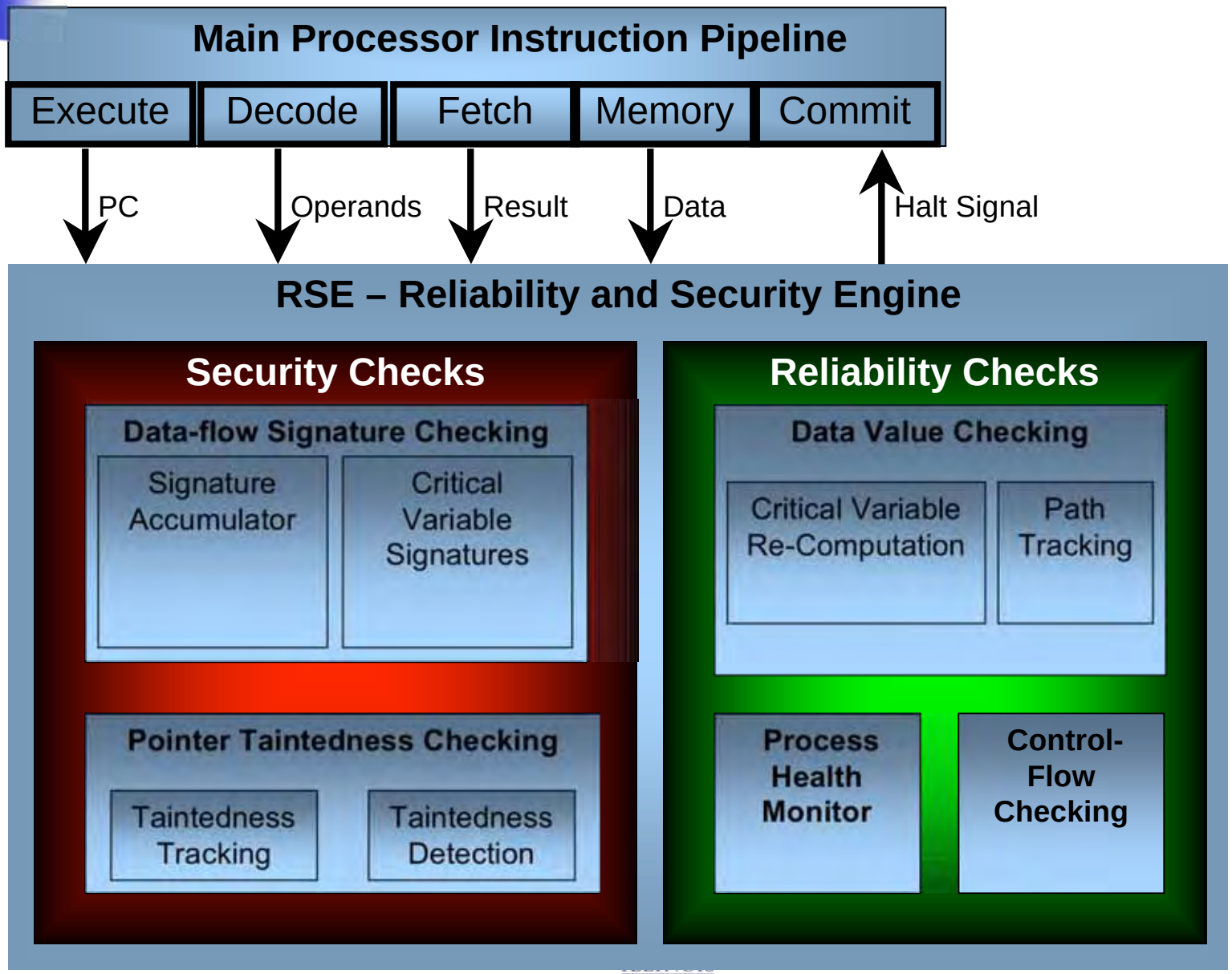
Critical Variable: *char authenticated;*

ILLINOIS

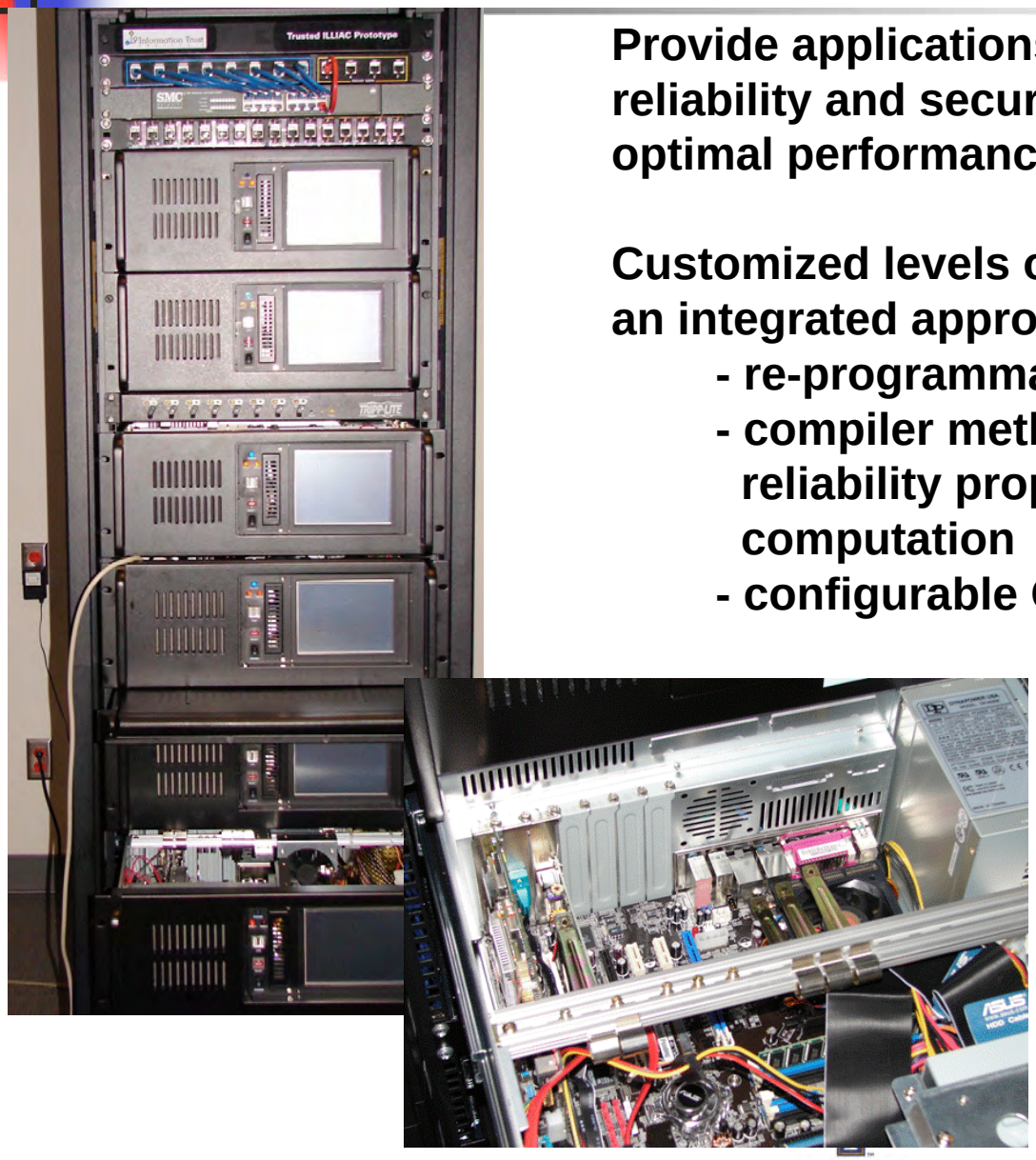
Trusted Microkernel



Hardware Prototype: Reliability and Security Engine



Trusted ILLIAC: The First Hardware



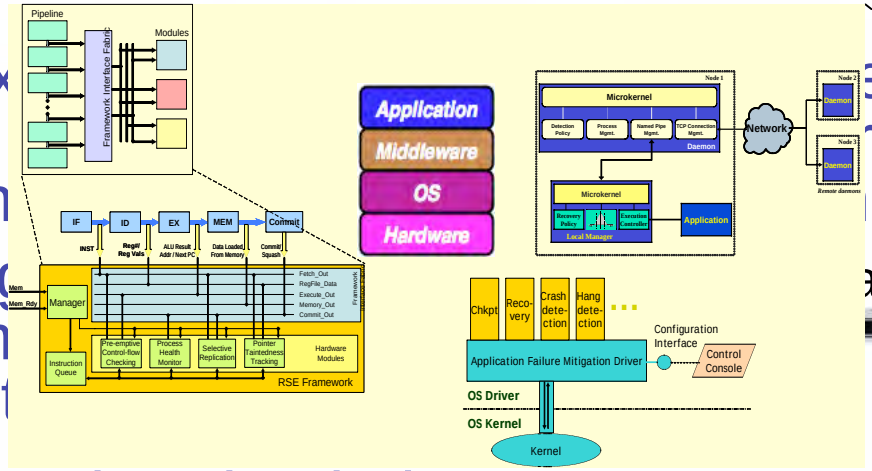
Provide applications-specific level of reliability and security, while delivering optimal performance

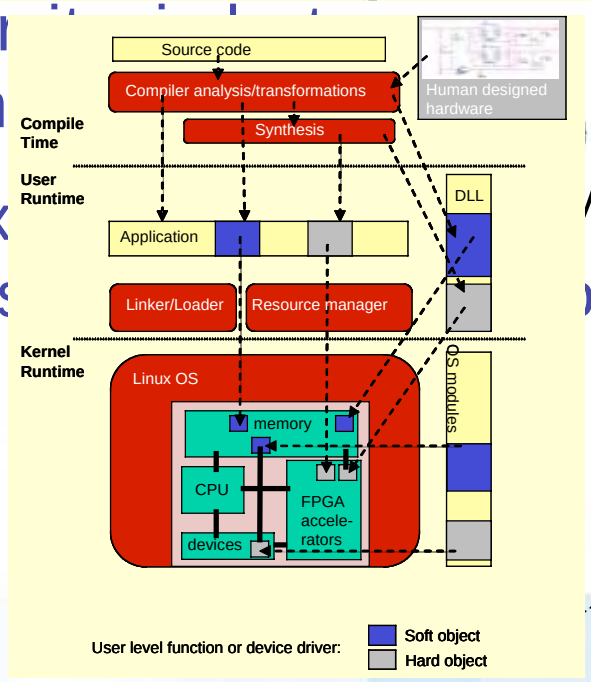
Customized levels of trust enforced via an integrated approach involving:

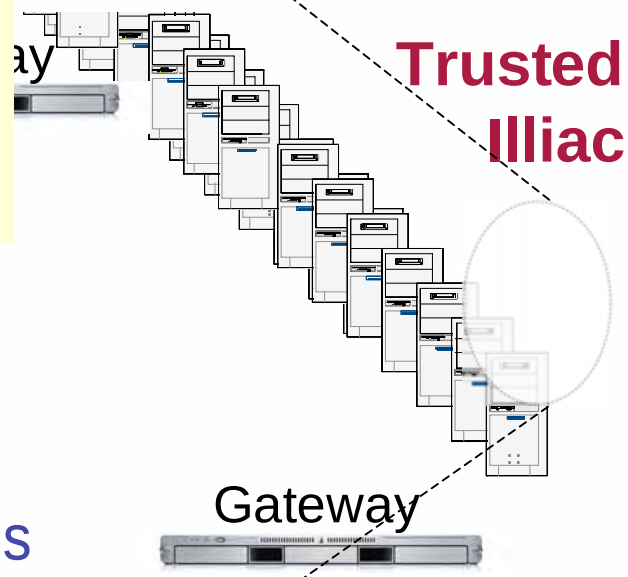
- **re-programmable hardware,**
- **compiler methods to: (i) extract security and reliability properties and (ii) accelerate computation**
- **configurable OS**

Trusted *ILLIAC*: The Broader Context

- New ex… eliable and secure … rt compilers combin… ardware
- Pushing … custom … comput…
- Enable univer… and governm…
- Train the nex… of system des… onals







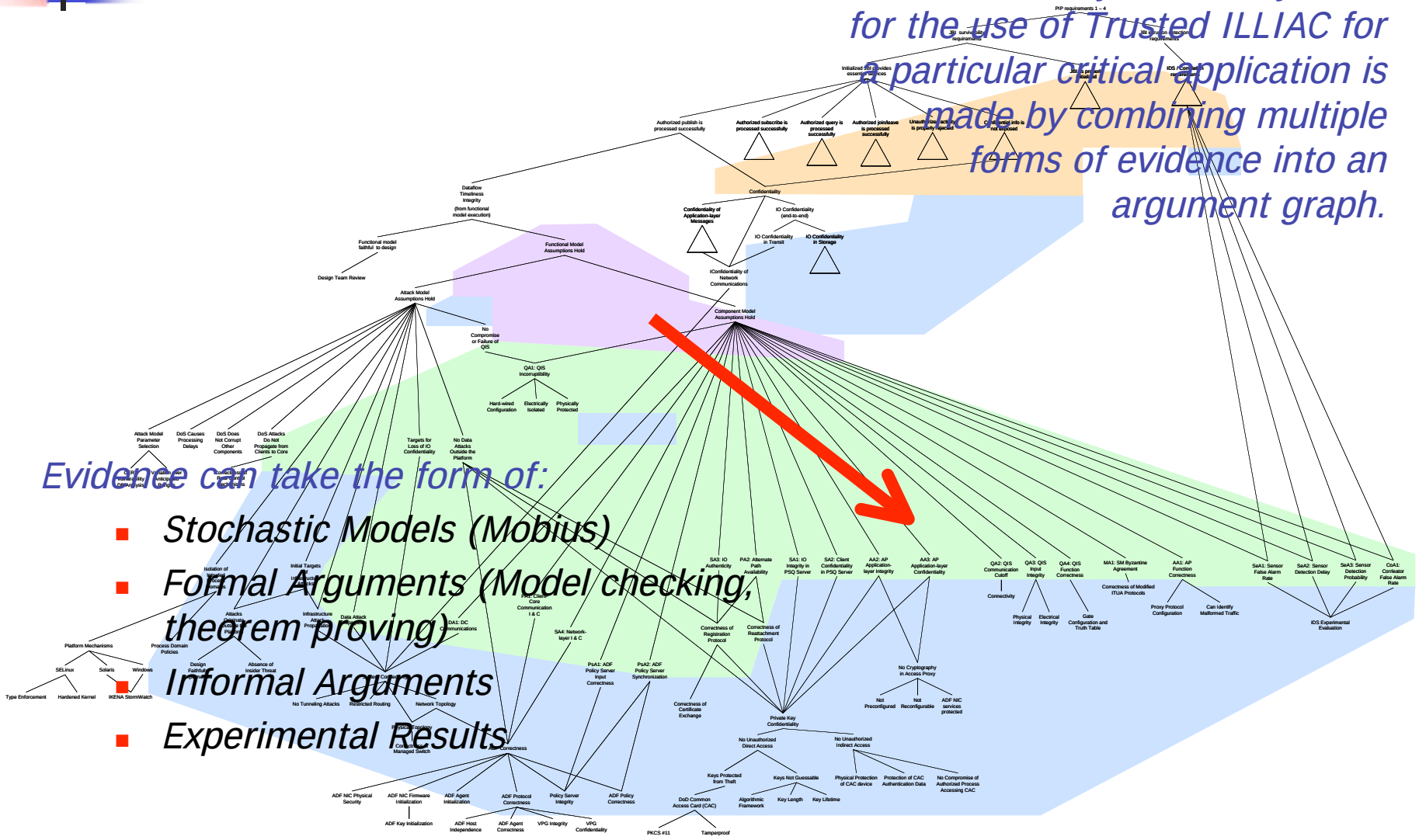
Example Trusted ILLIAC Node

Broad Research Support

- National Science Foundation
  - Programmable Hardware and infrastructure support
- Intel: Hardware/processor-level detection and recovery techniques
  - Reliability and Security Engine (RSE), a processor-level framework to deploy low-overhead application-aware error detection and recovery mechanisms
- IBM: benchmarking and enhancing reliability of operating systems
  - Develop methods for assessment of operating system robustness
  - Targets IBM AIX OS, Linux, Sun Solaris
- Motorola: security and reliability for wireless platforms
  - A testbed to explore seamless reliability issues and provide low-cost detection and recovery for wireless devices (e.g., cell phones) and networks..
- SUN
  - RAS (reliability, availability and serviceability) architecture of next generation dataservers
  - Processor-level error detection and recovery support
- HP
  - Reliable and secure enterprise computing
  - Deployment and automated generation of application-aware detection and recovery techniques

Building a Security or Reliability Case

An overall Reliability & Security Case for the use of Trusted ILLIAC for particular critical application is made by combining multiple forms of evidence into an argument graph.



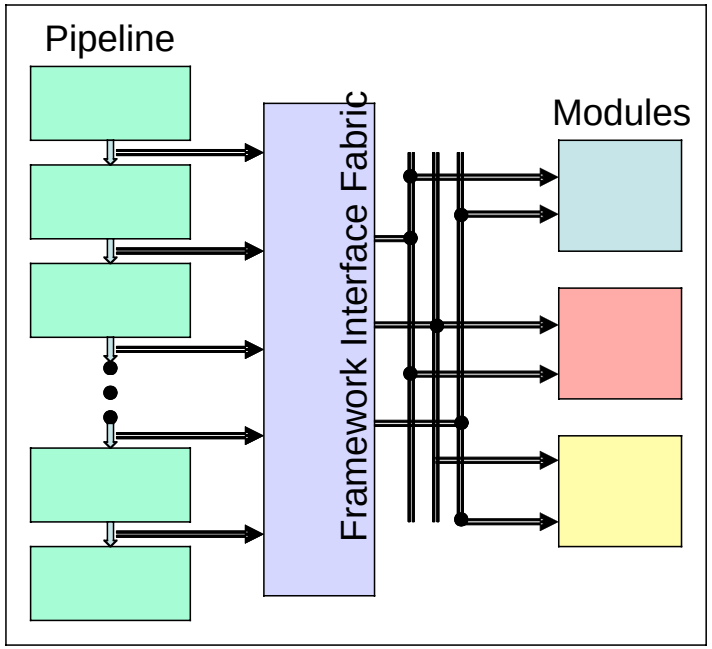
Evidence can take the form of:

- *Stochastic Models (Mobius)*
- *Formal Arguments (Model checking, theorem proving)*
- *Informal Arguments*
- *Experimental Results*

ILLINOIS

Application Aware Checking in Hardware: **Reliability and Security Engine (RSE)**

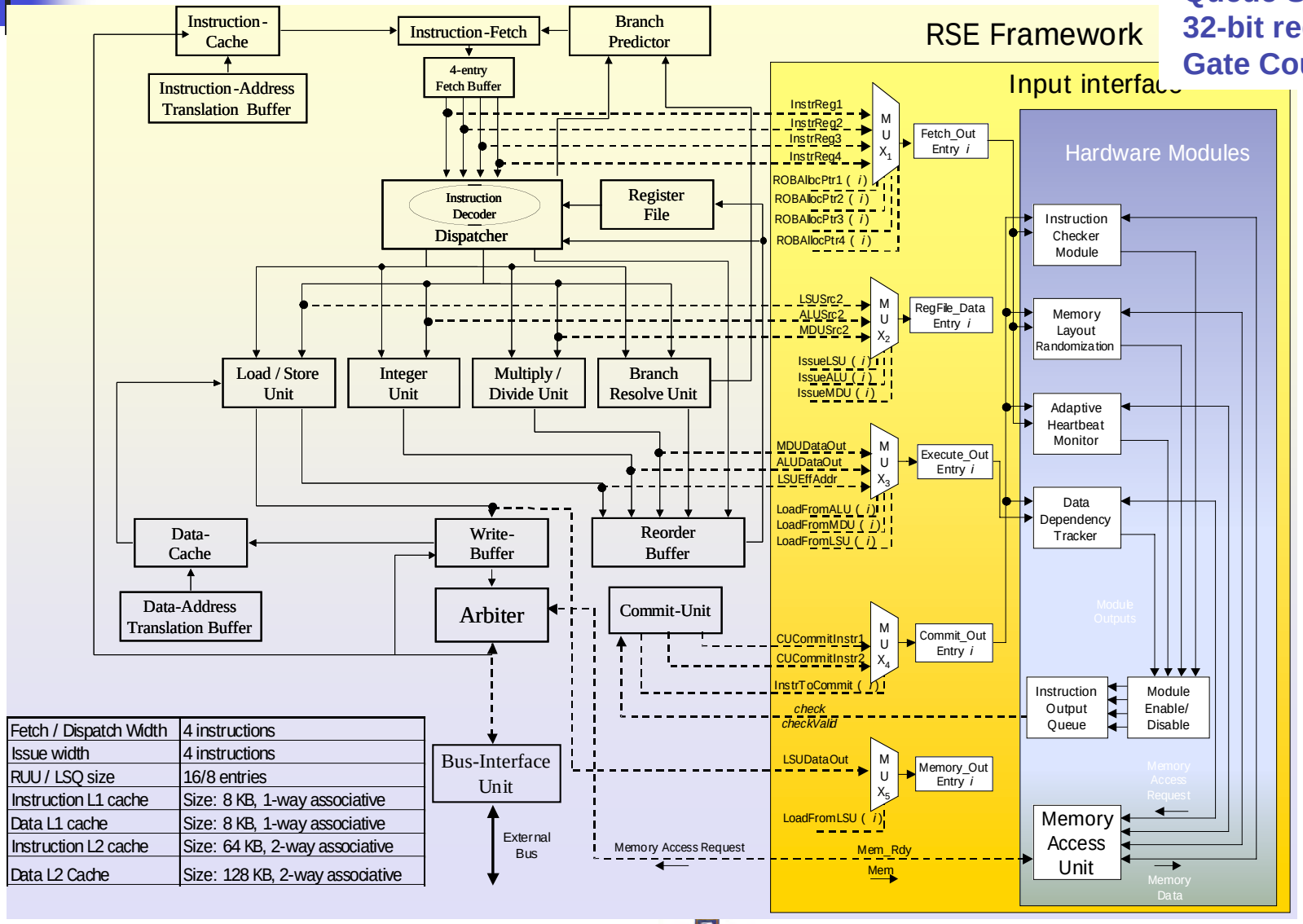
- **Goal:** Provide application-aware checks for reliability and security
- **Approach:** Reconfigurable processor-level hardware framework – Reliability and Security Engine
- **Current features**
  - On-core approach – processor, framework, and modules part of the same core on a single die
  - Framework and modules implemented on an FPGA
  - Framework configured to: (i) embed modules needed by application and (ii) route inputs to modules
- **Available modules**
  - Transparent hang/crash detection for OS and applications
  - Automatic processor-level checkpoint and recovery
  - Malicious attack detection and masking
- **Area and performance overhead of RSE implementation**
  - Area increased by 9.4%
  - Maximum clock period increased by 5.9%



***On-core* approach**

Reliability and Security Engine: Implementation

Input Interface:
Queue Size = 16
32-bit regs = 80;
Gate Count = 12K



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