Integrated Hardware/Software Approaches to Software Security

Bhagi Narahari

Department of Computer Science
The George Washington University

Research Outline

- Area: Software protection
- Focus: How can hardware assist in SW security
  - Without changing processor, add new logic into chip
- Approach: Hardware/Software co-design approach
- Threat Models:
  - physical capture, trojan circuits, access violations and hidden trojans in 3rd party code, automated recovery
  - Embedded systems focus
- Collaborators:
  - GWU: R. Simha, E. Leontie, G. Bloom, O. Chen
  - Northwestern: Prof. Alok Choudhary
  - Iowa State Univ: Prof. Joseph Zambreno
- Sponsors:
  - National Science Foundation (NSF)
  - Air Force Office of Scientific Research (AFOSR)

Our Focus: Software Protection in Embedded Systems

- Embedded systems are everywhere
  - Home appliances, Cell phones, critical infrastructure
  - Avionics, Automobiles,
  - Military – Future Combat Systems, Missile guidance
  - Over 90% of processors are embedded
- Threat?
  - Easily captured!
  - Can be probed in a well equipped laboratory
  - Due to their large number, the attack can be replicated!

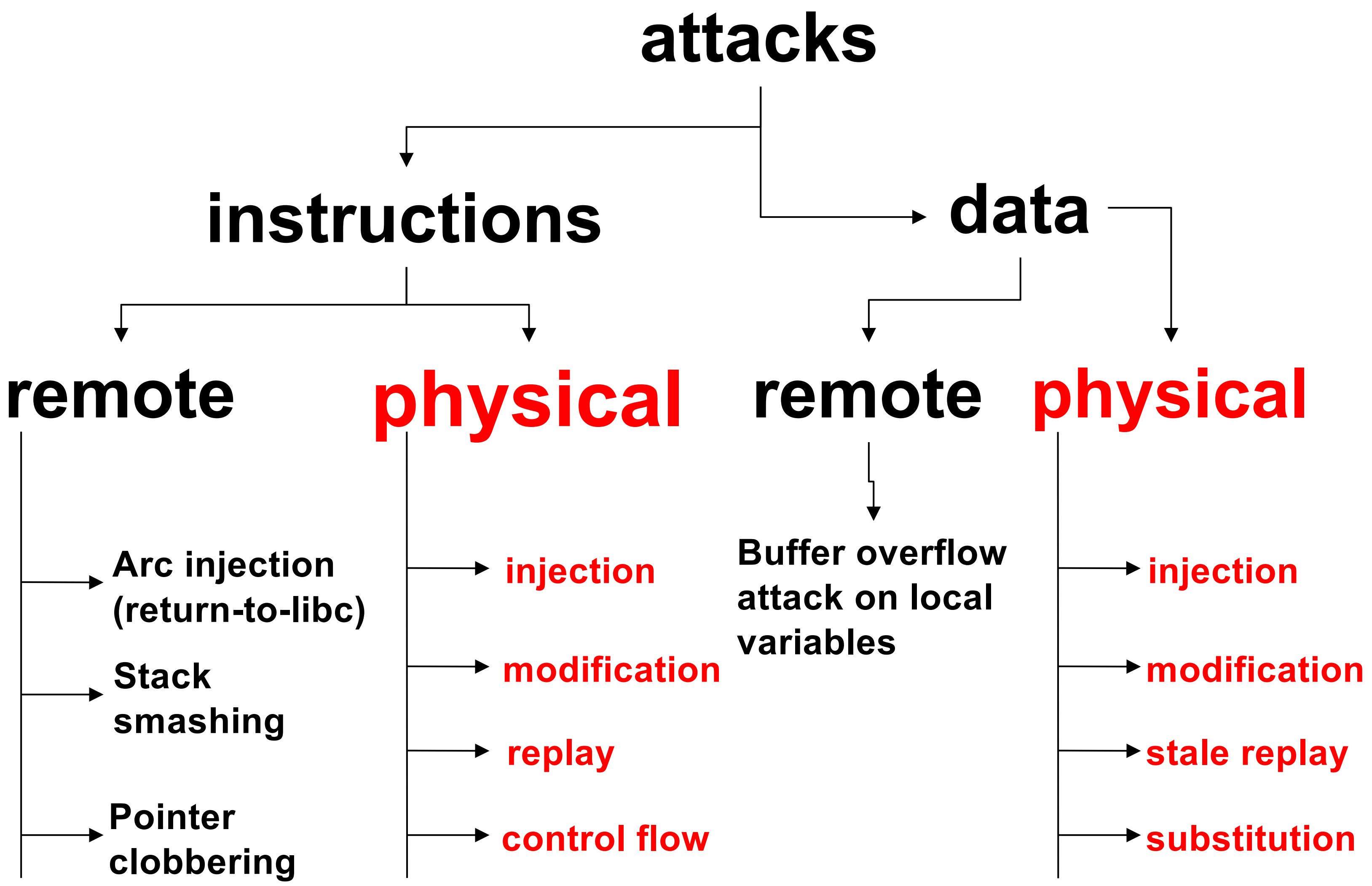
Embedded Systems—Assumptions

- Single (or small set of) application
  - Optimized to meet performance
- Static linking
- No Operating Systems support needed (apart from loaders)
  - Not a necessary condition for us, but we start with this and assume a secure trusted OS
- Usually has stringent size and power constraints
  - Cannot always assume you can use a high performance processor
  - Usually built using COTS technology

Typical Software Integrity Attacks:

- Code tampering attack
  - Executable code tampered with for various purposes
    - Licensing, changing feature set, using code to launch attacks
      - Eg: buffer overflow attack
- Authorization attack
  - Code tampering used to circumvent checking of permission in executing software
- Data tampering attack
  - Modifications of data such as passwords, IP addresses to enable hackers to exert control of SW behavior
- Code understanding
  - Copyright infringement, replay attacks

Attack Models



Software Security/Protection

- Computer security research has produced variety of techniques
  - Crypto, protocols,policy, authentication, intrusion detection,arch support,etc..

- Software protection objectives: provide techniques for authorization, prevent code tampering, make it harder to extract info that can be used to identify system vulnerabilities –code understanding

An abstract view of the software security problem

- Will the program running on the hardware do only what we define/expect and nothing else ?
- Can we model these properties/behaviour ?
- Can we automate the process of checking these properties ?

Our Research Objectives

- How can hardware help in providing software security ?
- Consider the entire language-compiler-OS-hardware tool chain during software and hardware design process
  - How can compiler help in asserting software properties
  - How can hardware validate the assertions
  - What performance penalties are incurred
  - Understand security vs performance tradeoffs
- Focus on ease of adoption
  - No custom hardware design required
  - No changes to existing processor/ISA
    - Example: PowerPC 405 processor with FPGA on Xilinx Virtex II
    - No burden on software developers
    - Techniques are incorporated directly into compiler
  - Backward compatibility with legacy applications
  - Security can be tunable to an individual application

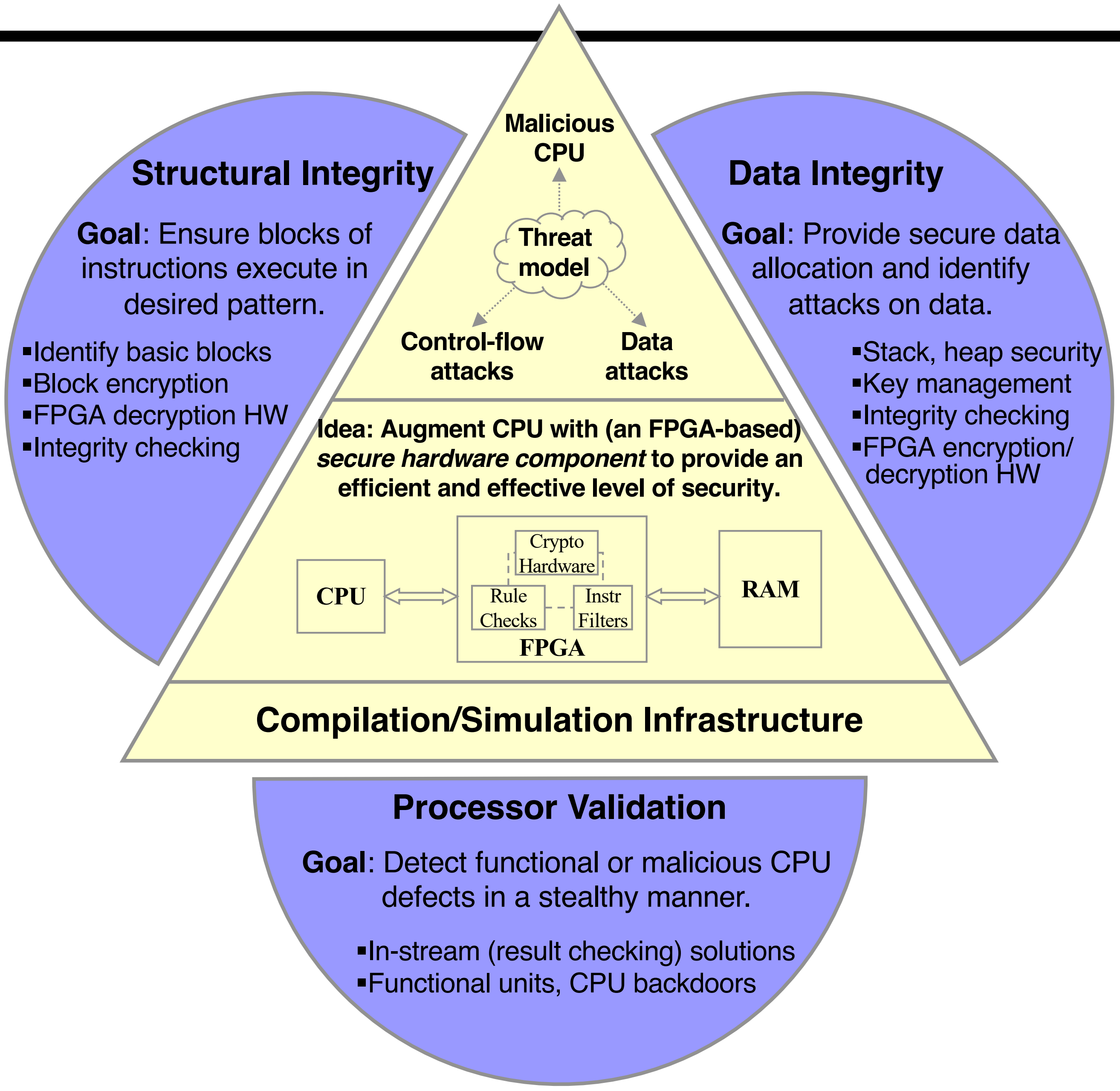
Our HW-SW Security Projects

- SAFEOPS
  - SAFE-OPS: An approach to embedded software security. *ACM Transactions on Embedded Computing Systems (TECS), Vol.4, No.1, 2005.*
- SPEE
  - SPEE: Secure program execution environment with integrity checking. *Journal of High Speed Networks*
- CODESSEAL
  - High Performance Software Protection using Reconfigurable Architectures. *Proceedings of the IEEE.*
  - A compiler hardware approach for software protection for embedded systems. in *Int. Journal of Computers and Electrical Engineering.*
- HW Wrappers/Containers for software components
  - *in SecuCode Workshop, CCS.*
- SHADE: architecture for leakage prevention and detection of trojan circuits
  - in *J. Computers and Security.*
  - *ANCHOR workshop.*

CODESSEAL

- Threat model: Physical capture of devices
- Embedded systems are everywhere
- Threat?
  - Easily captured!
  - Can be probed in a well equipped laboratory
  - Attacker now has access to the hardware
    - Can snoop on bus, can query memory contents
  - Due to their large number, the attack can be replicated!
- Minimal solution: Encrypted Execution and Data (EED) platforms
  - Instructions and data are encrypted

CODESEAL


Structural Integrity
Goal: Ensure blocks of instructions execute in desired pattern.
▪Identify basic blocks
▪Block encryption
▪FPGA decryption HW
▪Integrity checking
Malicious CPU
Threat model
Control-flow attacks
Data attacks
Data Integrity
Goal: Provide secure data allocation and identify attacks on data.
▪Stack, heap security
▪Key management
▪Integrity checking
▪FPGA encryption/ decryption HW
Idea: Augment CPU with (an FPGA-based) secure hardware component to provide an efficient and effective level of security.
CPU
Crypto Hardware
Rule Checks
Instr Filters
FPGA
RAM
Compilation/Simulation Infrastructure
Processor Validation
Goal: Detect functional or malicious CPU defects in a stealthy manner.
▪In-stream (result checking) solutions
▪Functional units, CPU backdoors

Related Work: Typical Software Security Solutions

- S/W solutions
  - Pure S/W approaches
    - Languages, compilers
    - Watermarking, obfuscation, static analyzers (MOPS)
    - System level tools
  - Cannot handle physical capture attacks
- Custom H/W approaches
  - Co-processors – TCPA, IBM coprocessor architecture
  - Crypto processors: Encrypted Execution and Encrypted Data (EED) model
    - Encrypted Execution models (XOM, AEGIS)
  - Memory protection (Mondrian)
  - Require changes to ISA and/or micro-architecture
    - Need buy-in from vendors
    - Our aim is to not modify the ISA or micro-arch
- Trojan circuit detection at design time
  - Not at run-time in cases where trojan circuit has not been detected

HW Solutions

- Common theme: encrypt the executables and data
  - Encrypted Execution and Encrypted data (EED) platform
  - Provide hardware accelerators (dedicated circuits) to provide encryption in hardware
- Shortcomings
  - do not prevent all physical capture attacks
  - require hardware ISA redesign
- Hardware design issue
  - Some solutions need "all new" ISA
    - Difficult to convince vendors, need to recompile applications
  - Some need major redesign of processor datapath
  - need processor manufactures to buy-in

Our premise: EED Attacks

- Does Encryption of Instructions and Data solve everything ?
  - Encrypt instructions and data
  - Everything coming out of the CPU chip is encrypted
  - the 'attack model' ?
    - Physical capture of devices
    - Attacker now has access to the hardware and can snoop on bus, query memory contents
  - How about when the attacker is the chip foundry ?
    - Malicious CPU from an Untrusted Foundry ?
- Can attacker disrupt execution without knowing encryption keys
  - Can they understand what the program does?

CODESSEAL Premise

- EED not sufficient: having everything encrypted does *not* prevent attacks
  - Can still have attacks on data, can understand code functionality, launch replay attacks, inject code
- Project Objective: Practical solutions to protect against attacks on EED platforms.
  - Investigate types of attacks
  - How to effectively combat (prevent & detect) attacks

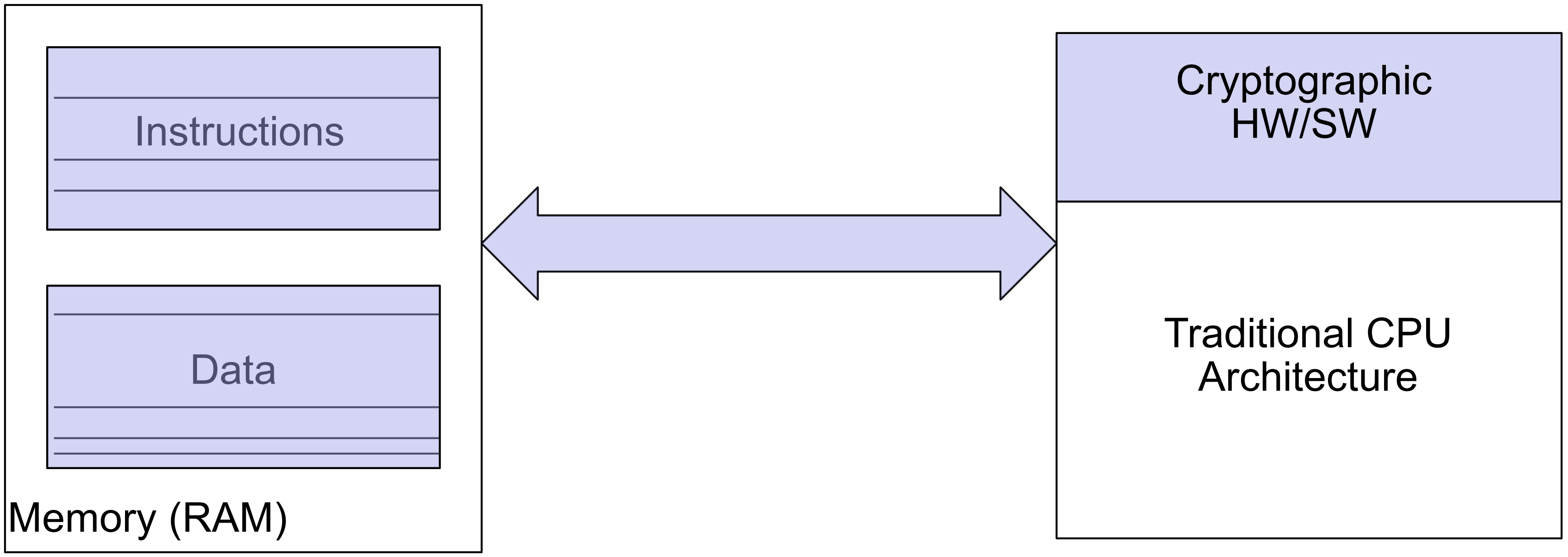
Attacks on EED platforms

Target: An embedded system with standard CPU and memory.

Memory contents encrypted

Decryption takes place on CPU

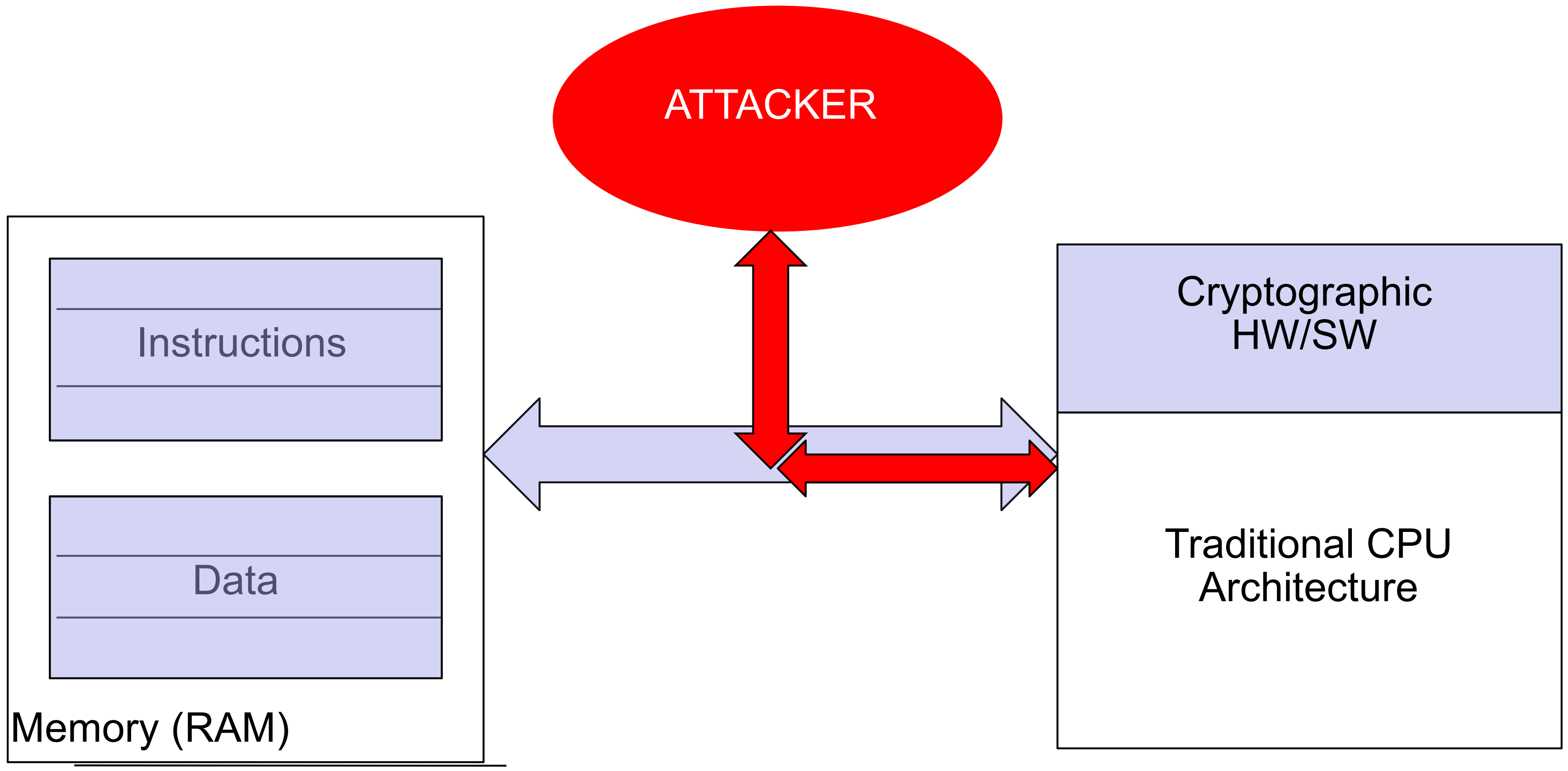
Encrypted Information on Bus



Attack Model

Threat - attacker uses wire probes to gain R/W access to control and data signals on the instruction/data buses

- can also insert HW to snoop on bus and inject during normal execution



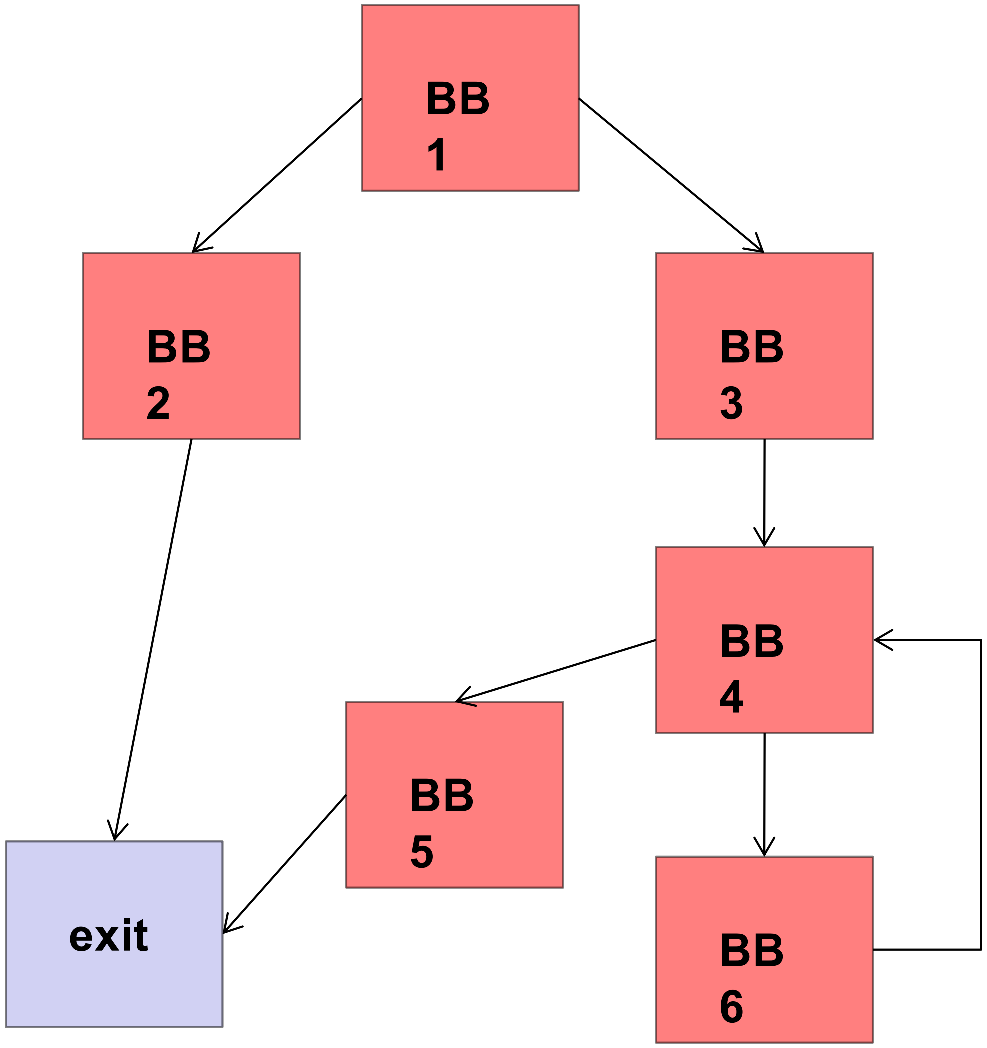
What is a basic block (cache block)? Using compiler techniques in our solution

- Define Program/Data flow in terms of Blocks
  - This is what an attacker can play with
- For data: a cache block
- For instructions: a basic block
- Definition:

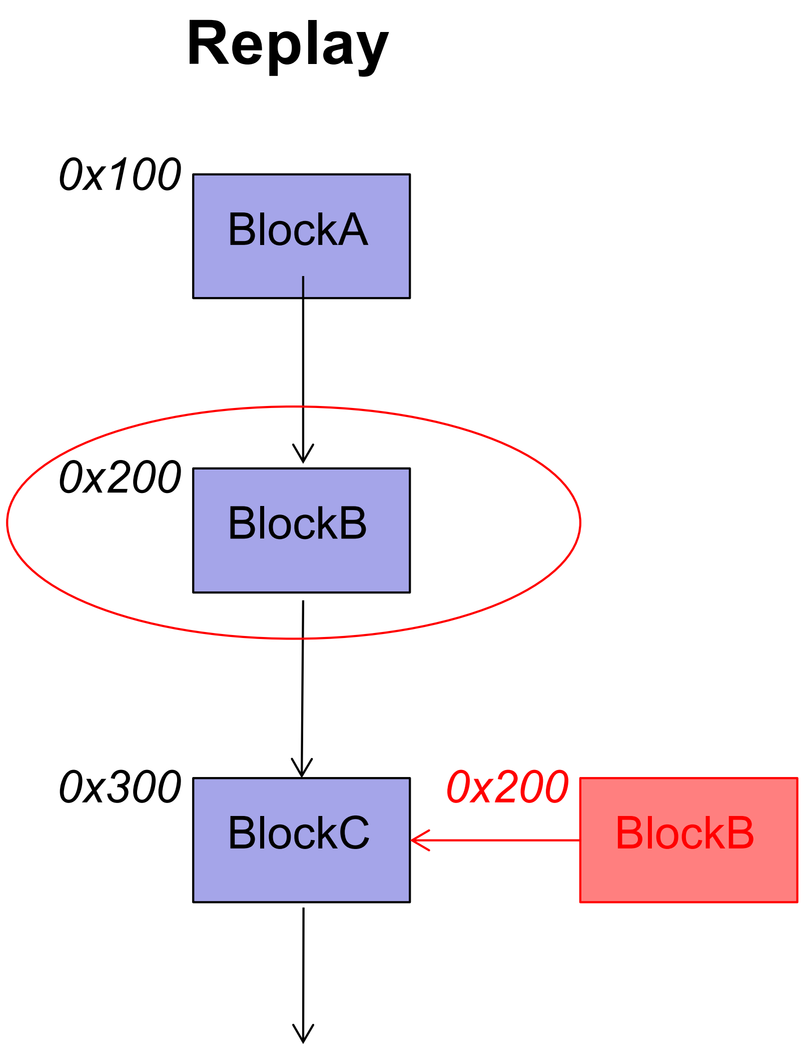
  Basic block is a sequence of consecutive instructions with entry point at the beginning, exit point at the end and containing no branches, except at the end

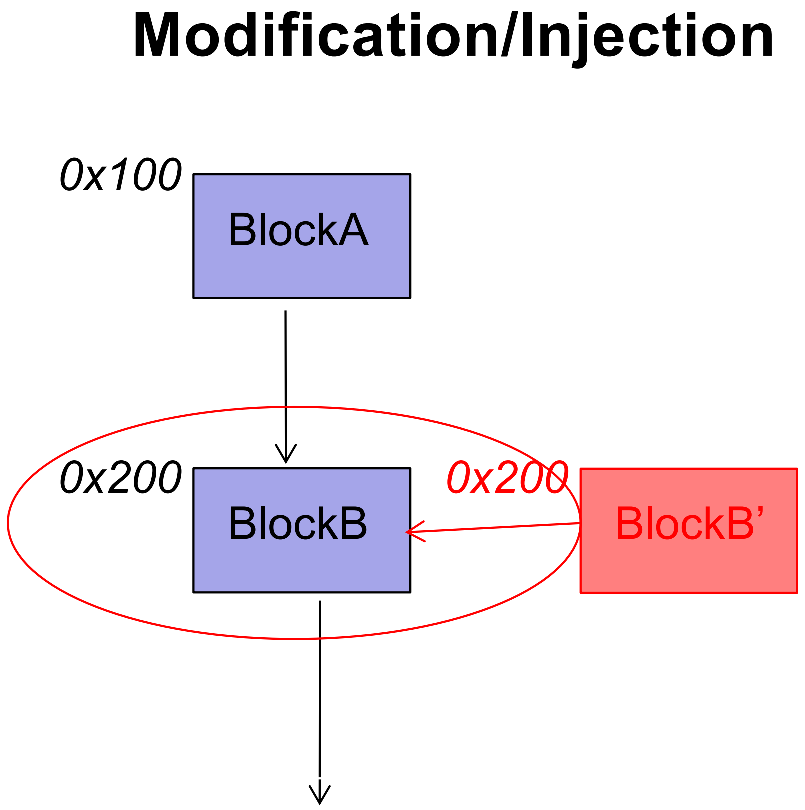
A Basic Block Example

```
          m=y*a+b;
          if m<=1 goto L3        bb1

          i=2                    bb3

L1:       if i<=m goto L2        bb4

                                 bb5
          return x
L2:       x=i+1
          i=i+1                  bb6
          goto L1

L3:       return m               bb2
```



Attacks

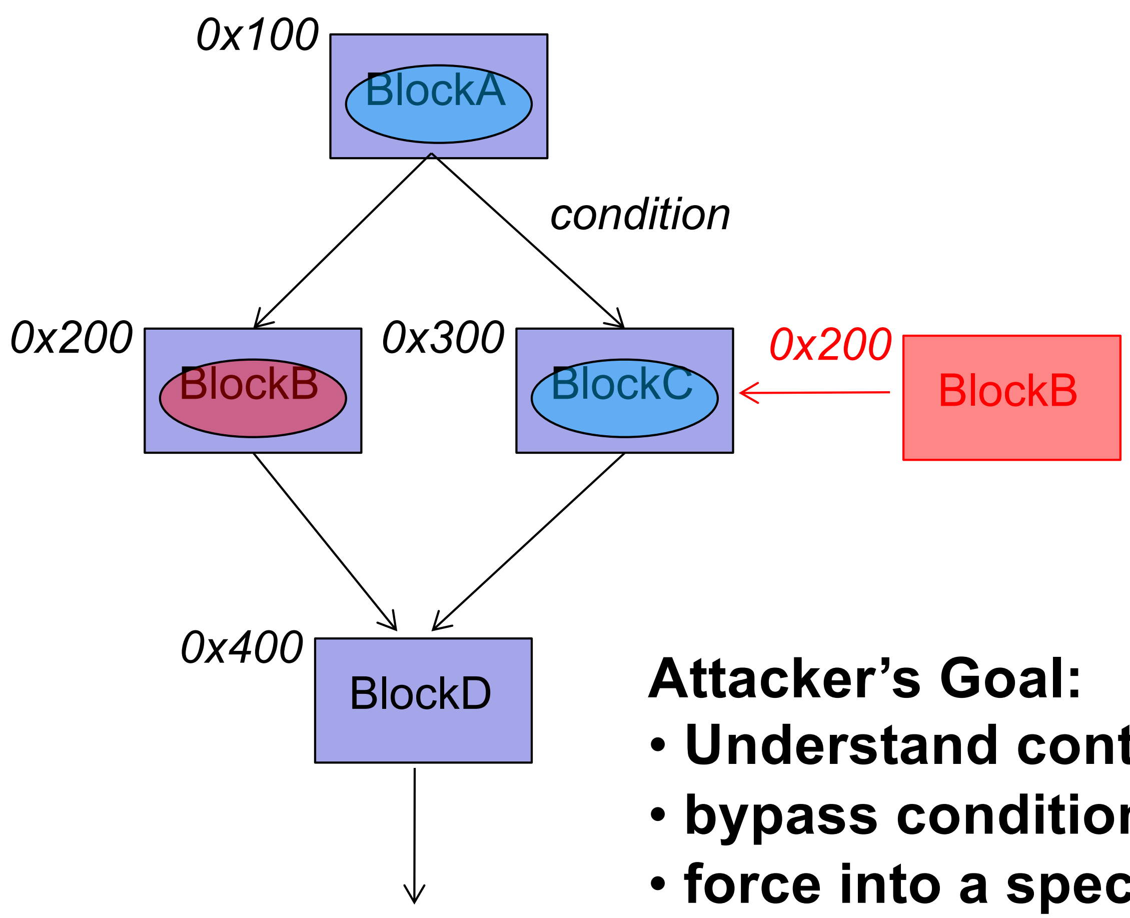




Attacker's Goal: repetition of an observed event or undetected disruption

Attacks: Examples

Control Flow

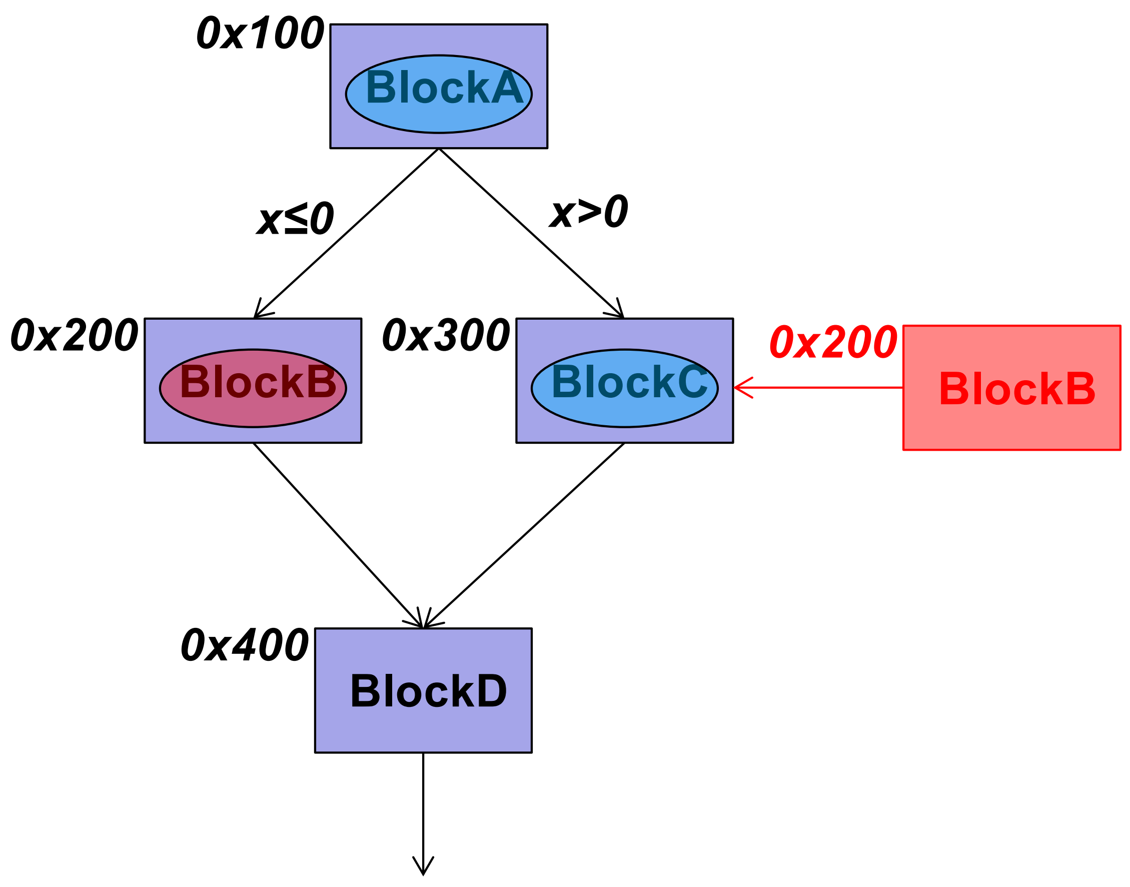


Attacker's Goal:

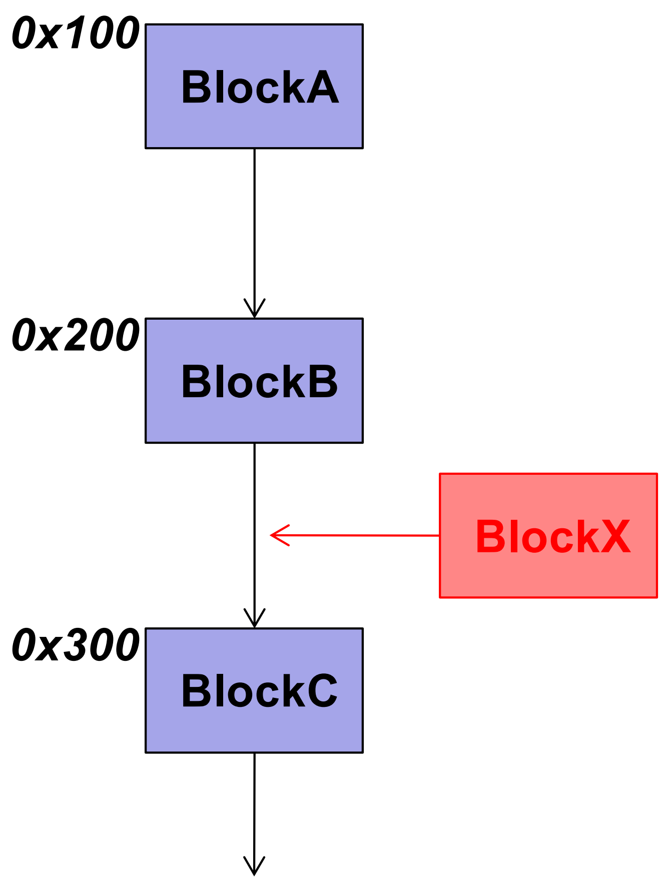
- **Understand control flow**
- **bypass conditions/checks**
- **force into a specific path**

Attack Models (cont)

Control Flow



Injection



Attack Model: Data Attacks

- Privacy of data
  - Attacker can access secrets, passwords
- Attacker can inject data
  - Examine data request addresses
- Data capture and replay attacks
  - Current data replay
  - Stale data replay

Attack Model: Summary

- EED Attacks
- Just having everything encryption does not prevent attacks
- Our objectives: how to prevent/detect such EED attacks
  - Design architecture and compiler techniques
  - Do not change processor ISA
  - The least amount of change to the overall system
  - Backward compatibility

Our Solution: CODESSEAL

- CODESSEAL: COmpiler DEvelopment Suite for SEcure AppLications
  - ❑ Integrated HW-SW approach
  - ❑ Fully encrypted program and data EED
  - ❑ Place on-chip secure hardware component
    - Use reconfigurable (FPGA) fabric to implement HW component
- Consider the interplay between *compiler* and *hardware* during code generation and execution
  - ❑ How can compiler help in asserting SW properties
  - ❑ How can hardware validate the assertions
  - ❑ What performance penalties are incurred
  - ❑ Study security vs performance tradeoffs

CODESSEAL Overview

- Place secure on-chip hardware component – a Guard (using FPGA logic) inside CPU chip
  - All communication to outside world goes through the Guard
- Compiler inserts checks for integrity, control-flow, data timestamp
  - Transparent to programmer
- Guard implements real-time verification
  - Transparent to CPU and Memory
- No change to Processor ISA
- Performance impact study

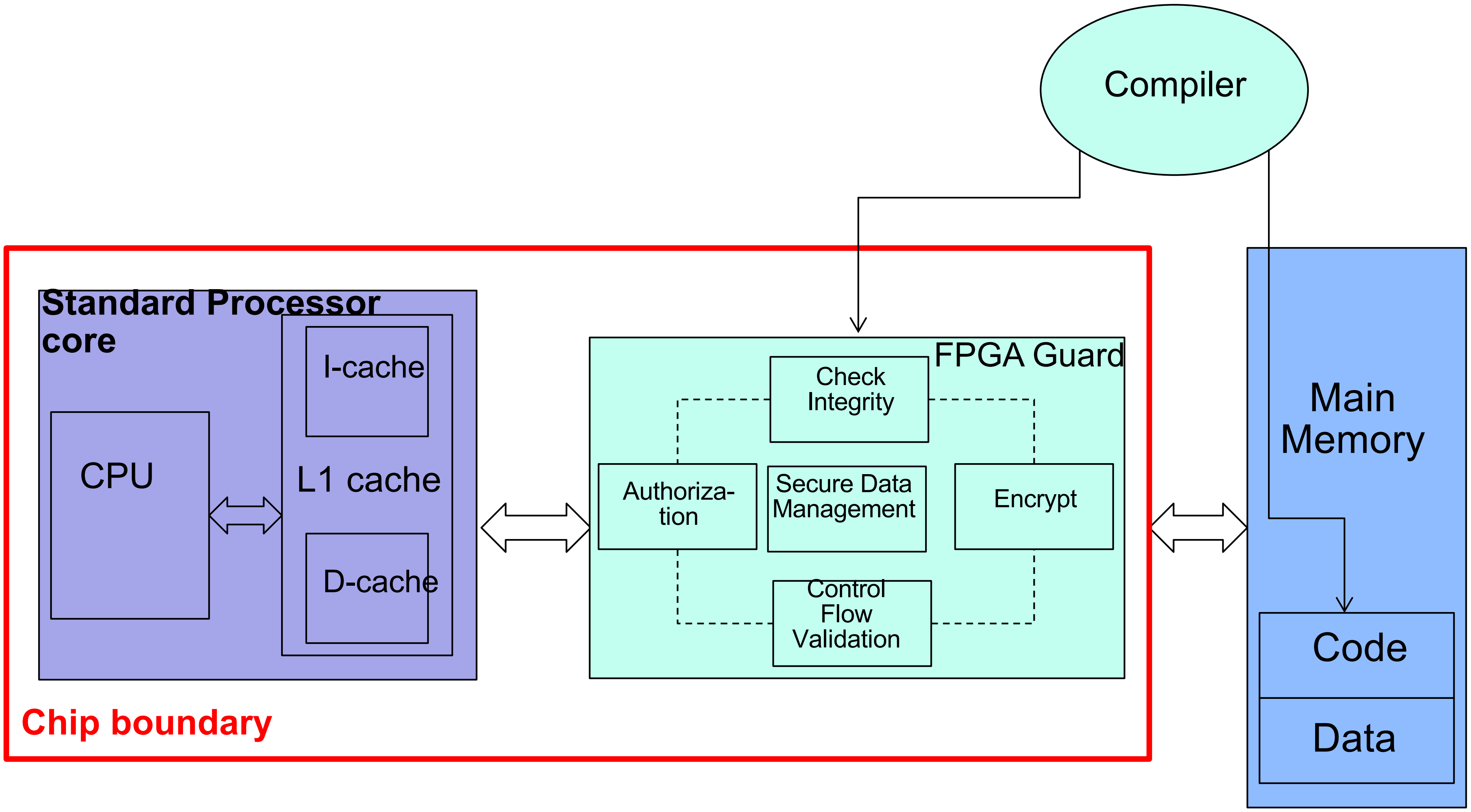
Why Compilers?

- “Gateway” in software creation
- Readily-available program structure
- Optimization on compiler level
- Security can be added at the intermediate code generation stage
- Transparent to programmer

Why FPGA

- Common to have programmable logic on the same chip as processor
  - Example: Xilinx Virtex II Pro from Xilinx Corp.
- No change to the system
- Re-programmability
- Design cycle is the same as processor's
- Can be optimized by the compiler
- Reverse engineering is more difficult than custom hardware

CODESSEAL Concept



30

Security needed for EED attacks

- Recall: to prevent EED attacks under physical capture we need to prevent:
- Code/Data injection and modification attacks
  - Need integrity checking to verify no modification
- Code/Data replay (substitution) attacks
  - Need to check if information has been fetched from requested location
- Control-flow attacks on code and Buffer overflow
  - Check if correct branch outcome is followed
- Stale Data replay
  - Check if data is latest value

Key Aspects

- Secure HW Gateway (FPGA) acts as a *guard* for traffic between CPU and Memory
  - Analogous to an L2 cache controller
- Authorization: Sign/hash and Encrypt each code block – *compiler*
  - Gateway verifies signature
  - Prevents injection of unauthorized code and data
- Control Flow: Embed control flow information in each code block – *compiler*
  - Prevents control flow attacks
  - When fetching blocks from memory, FPGA also checks the control flow information – check for valid parent, valid branch outcome
- Data attacks: Sign, encrypt, "timestamp" each data block – (*done by FPGA controller*)
  - Prevents data injection, provide privacy
- Stack attacks: Provide HW stack in FPGA/HW – *compiler and FPGA*
  - Prevent buffer overflow attacks

Our Security Techniques

- To provide code and data privacy/confidentiality we encrypt the code/data
  - We use AES with CBC for each block
- For integrity checking compute digest
  - Use (1) SHA-1 (secure hash) or (2) CRC
  - Use compiler to embed this digest into the code/data
- Use the FPGA logic to
  - perform the encryption/decryption
  - Check integrity information

The CODESSEAL techniques

- Program blocks defined at what granularity ?
  - Basic block
    - Can be done earlier in the compilation process
    - Architecture independence
  - Cache block (for data)
    - Program graph where each node is a cache block
    - Graph depends on architecture details
    - Need to recompile if we change cache size
- What signature scheme to use,?
- where to store signature

Code and Data Confidentiality

- Need to encrypt all code and data leaving the chip
  - provided by our use of AES encryption
- Instructions
  - encrypted at “code block” granularity – basic block or cache block
  - Prefetching is possible/required for basic block
- Data
  - Encrypted at cache block granularity
  - Decryption performed on every read from memory
  - Encryption performed on every write to memory

Code and Data Integrity

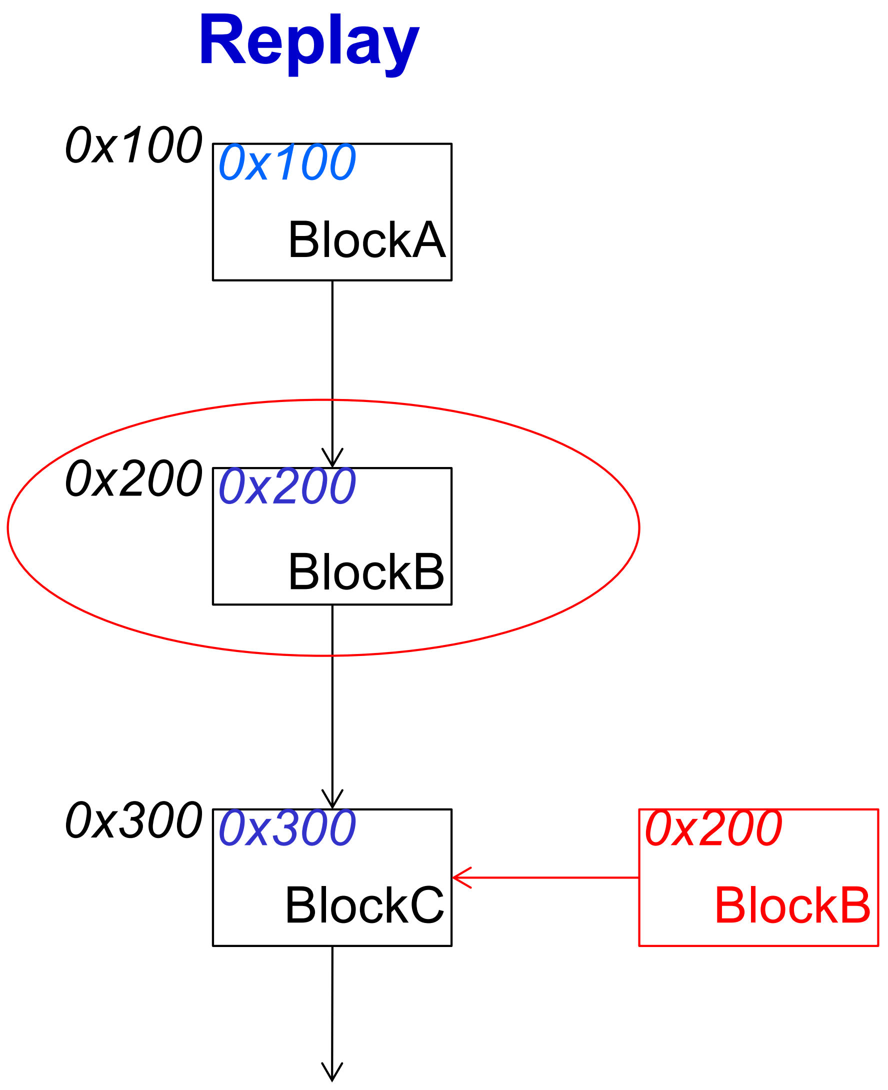
- We embed integrity information into code/data blocks to detect modifications
  - Compute "signature" of the code (data) block at compile time and stored as the "digest"
  - At run-time compute signature and compare with stored signature
- What scheme to use to compute integrity checking information
  - SHA-1 or CRC
- Where to store integrity checking digest
  - Within the block
  - Outside in separate memory (in the Guard?)

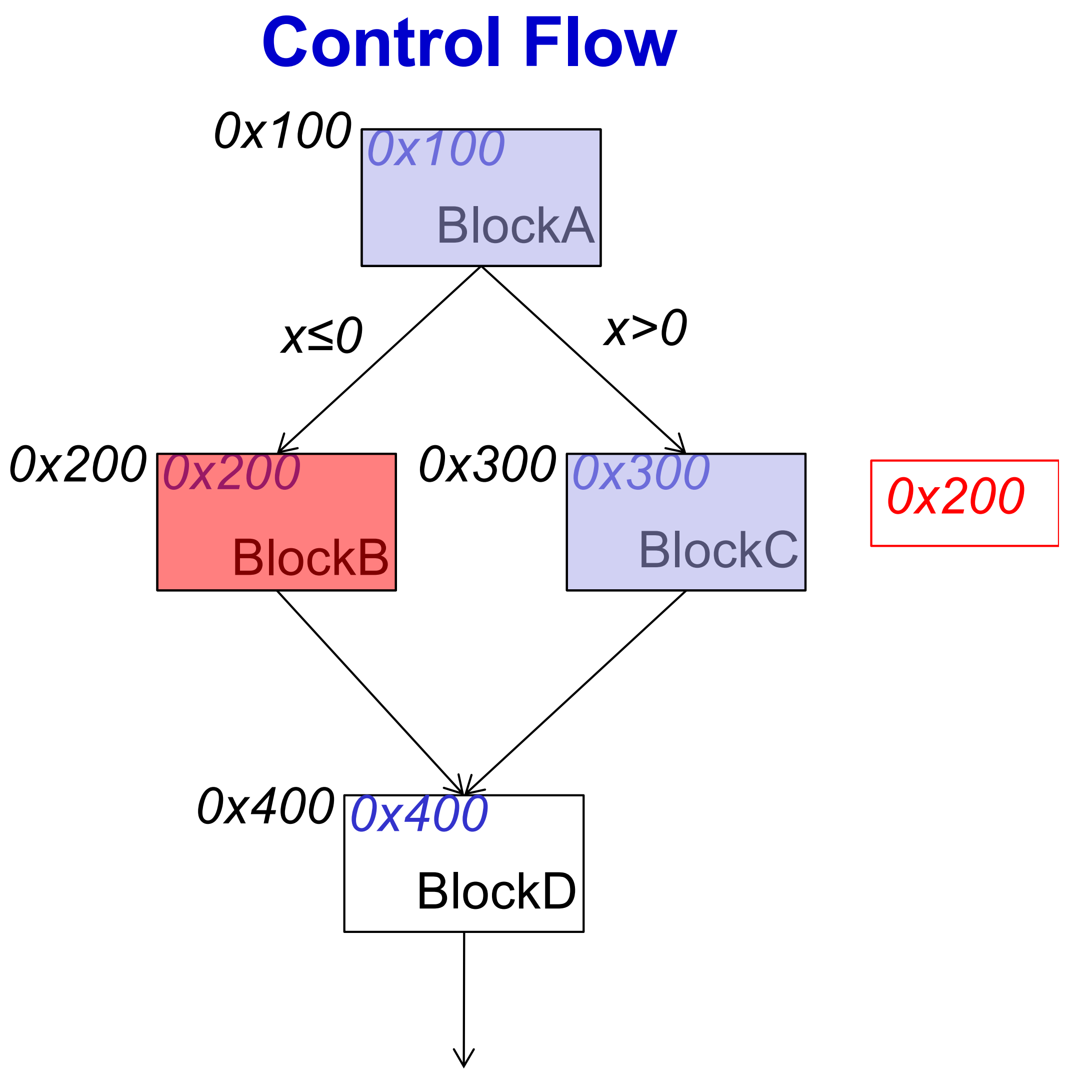
Example: Cache Block (CB) Signatures

- Program consists of a number of cache blocks (CB)
  - Cache block has label – address
- Program has starting address Y
  - Absolute address = starting address + label of CB
- Cache block labels stored with code blocks
  - encrypted
- Processor requests block with address A
- FPGA/Guard captures and stores A
- Memory returns block X
  - Decrypted in FPGA
- Fetched block is valid if and only if label(X)= A+Y
  - Else attack has taken place, halt processor

Example: Replay and Control-flow Protection

- Embed block addresses into signature





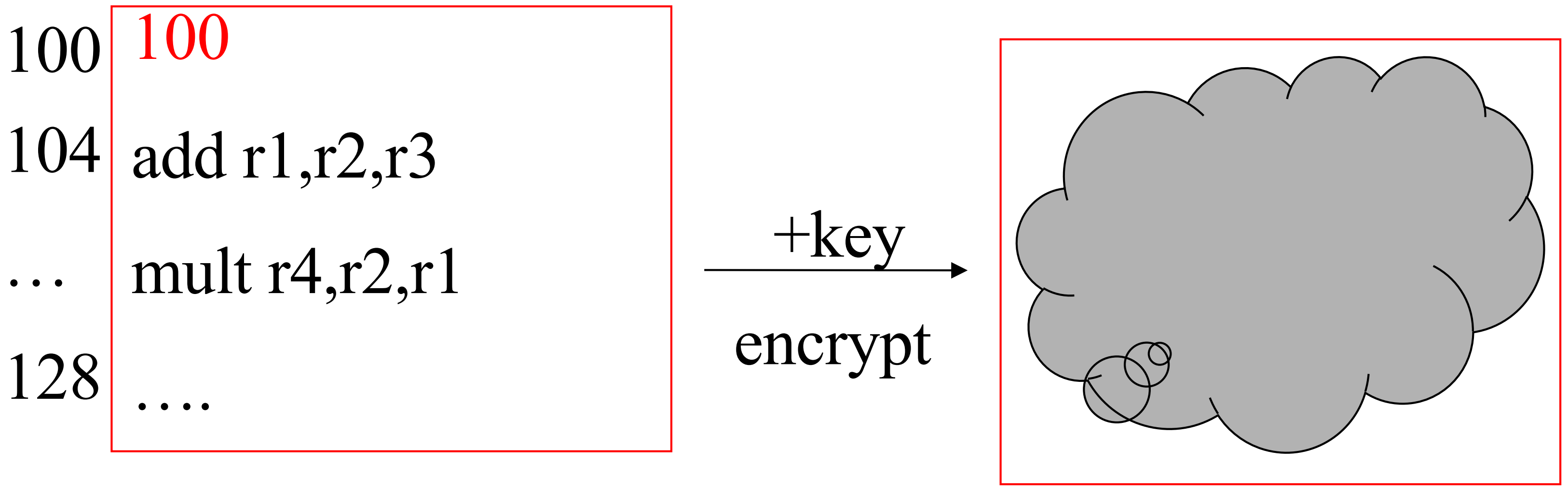
Compiler role

```
100  add r1,r2,r3
104  be 56 (164 )
...  ….
128  mult r4,r2,r1
-----------------------
132  ld r1,#100(R3)
136  add r1,r2,r3
...  …..
160  add r1,r2,r3
-----------------------
164  ld r1,#100(R3)
168  mov r1,r2
...  …..
196  add r1,r2,r3
```

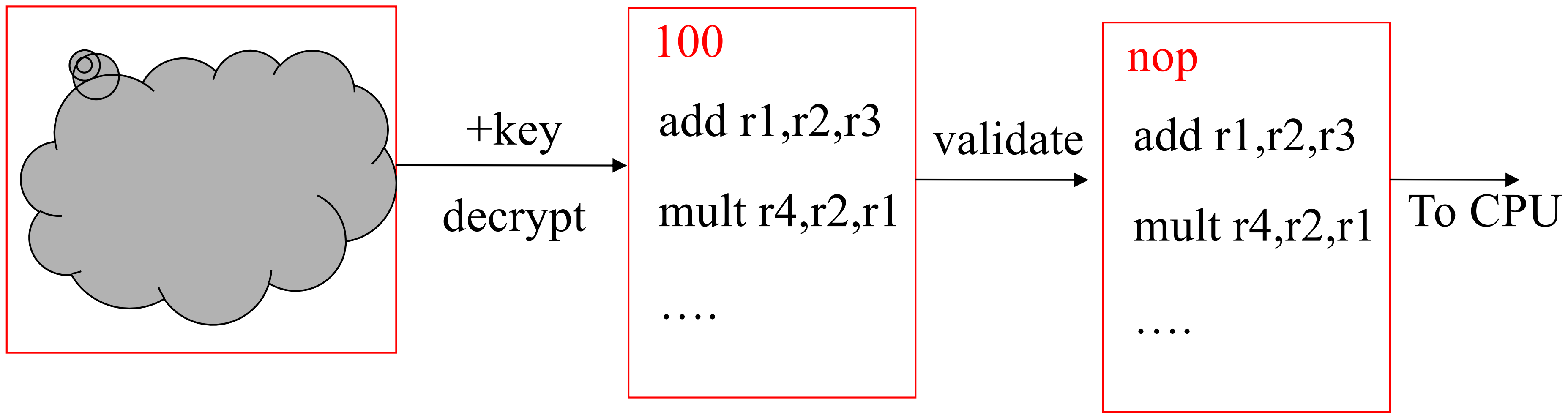
$\longrightarrow$

```
100  100
104  add r1,r2,r3
...  be 60 (172 )
128  ….
-----------------------
132  132
136  mult r4,r2,r1
...  ld r1,#100(R3)
160  ...
-----------------------
164  164
168  add r1,r2,r3
...  ld r1,#100(R3)
196  ...
```

Compiler role (next)



FPGA Guard role



- Decryption increases cache miss penalty
- The NOP inserted by FPGA results in increased computation time

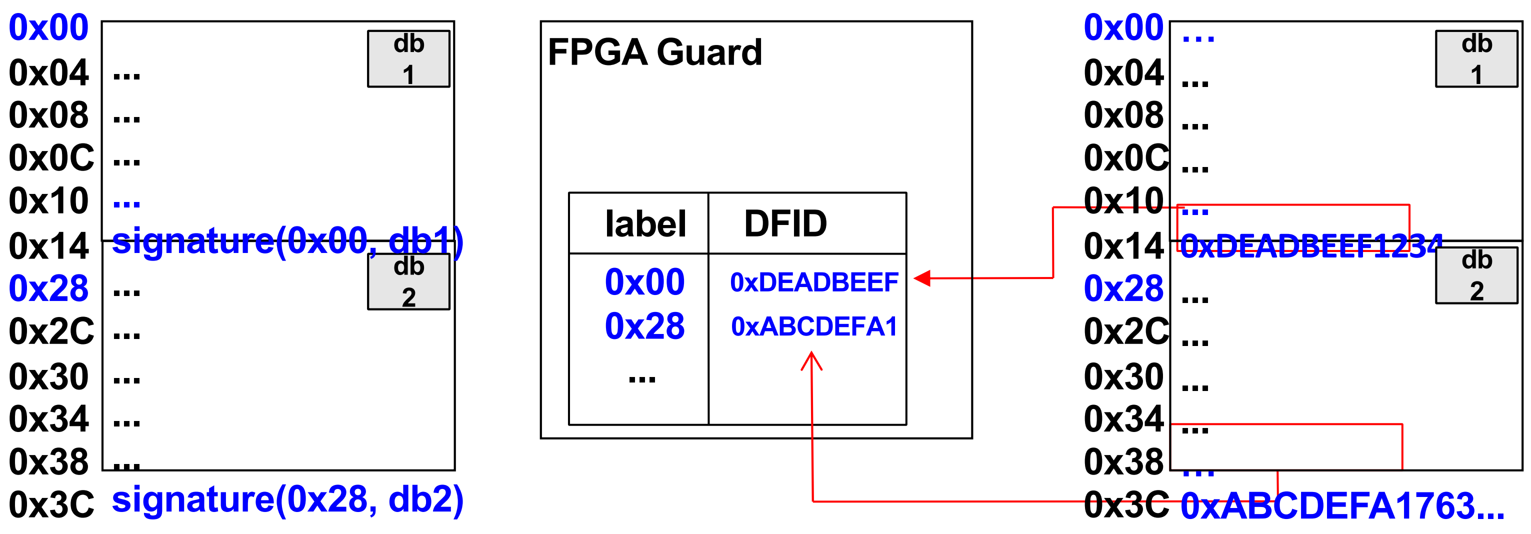
Data Replay Protection

- Similar to Instruction replay
  - Instruction labels ensure control flow and replay protection
- Data block's address is used as a label the same way it is used for instructions
  - Labels are included in the block's digest
- Labels prevent out-of-sequence replay of current data blocks

Stale Data Replay Protection

- DFID – Data Freshness Indicator – prevents stale data replay attacks
- Part of the digest is stored inside the FPGA for verification that the data block is the most current one
- DFID is used instead of a timestamp because one cannot run out of DFIDs

Data Protection



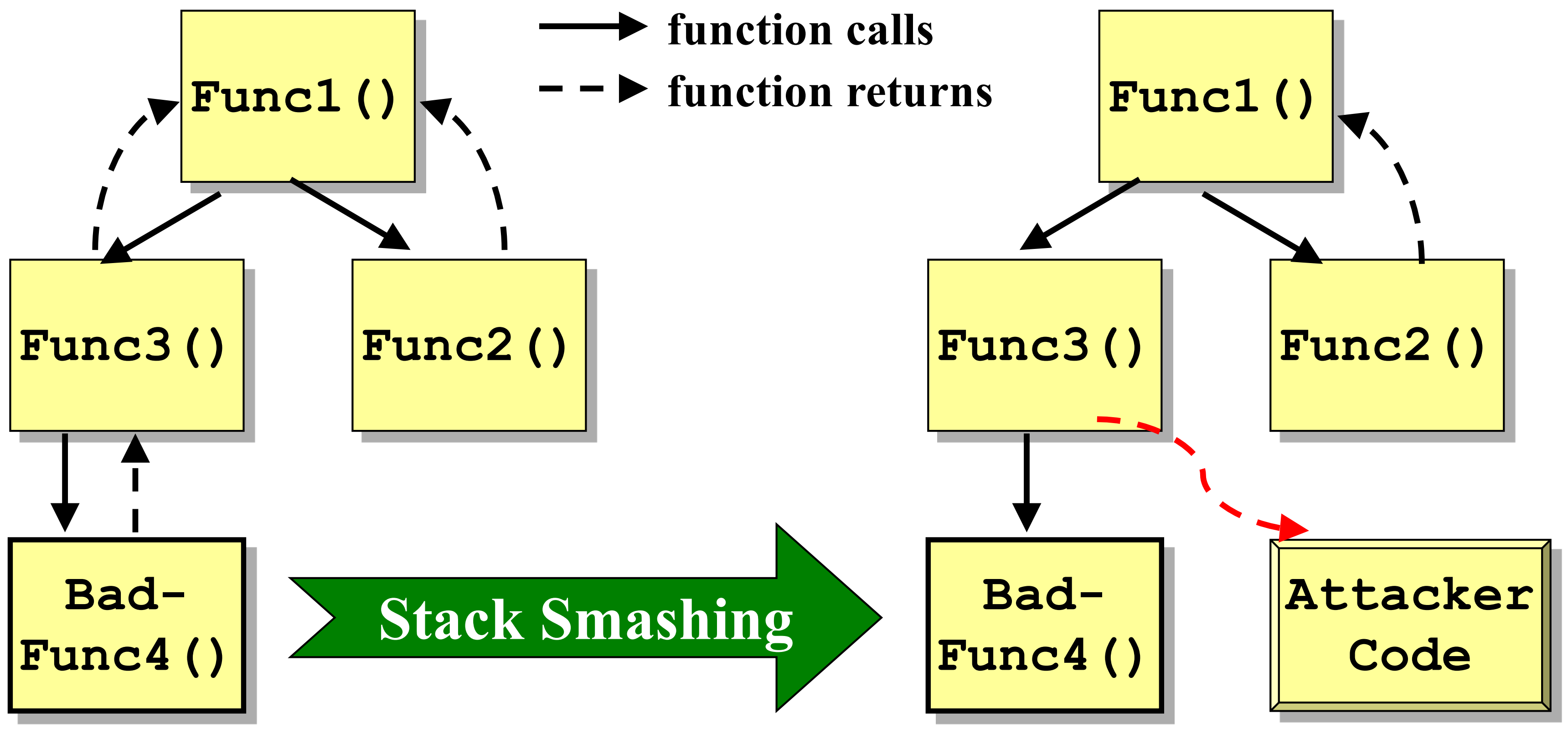
(a)
Signatures are stored sequentially with data blocks in memory

(b)
First 32 bits of signature are used as DFID and stored in FPGA Guard's internal table together with block's address

Buffer Overflow/Function Protection

- Buffer overflow attacks are still the vast majority of attacks – overwrite the return address and route the program to malicious code
- If data is overwritten on run-time stack then we cannot assume that the address requested by the CPU is correct
- If return address is overwritten, then function returns to a “wrong” location

Buffer Overflow Attack



Buffer Overflow Protection: HW Stack

- We provide additional secure hardware function return stack is placed in the FPGA
- On a function call, return address is saved in the hardware stack
  - Push return address onto HW Stack
- On a function return, the address is verified against the one stored in the hardware stack before a function is allowed to return
  - Pop return address from HW Stack
  - Compare with requested address

Overall CODESSEAL Software and Hardware Processes

Compile-time processing

① Identify code blocks (cb/bb)

② Label each block bb_i

③ Sign_compile(bb_i + bb_{label})

④ Embed signature in object file

⑤ Encrypt(AES)

Run-time Validation (loader)

① Set the program context key

② Set program start address a_{start}

Run-time Validation (FPGA Guard)

① Monitor memory reads from cache controller

② Capture requested address a_i=address(bb_i)

③ Fetch and decrypt block bb_i (prefetching if necessary)

④ Fetch stored signature x_i=sig_compile(bb_i + b_{label})

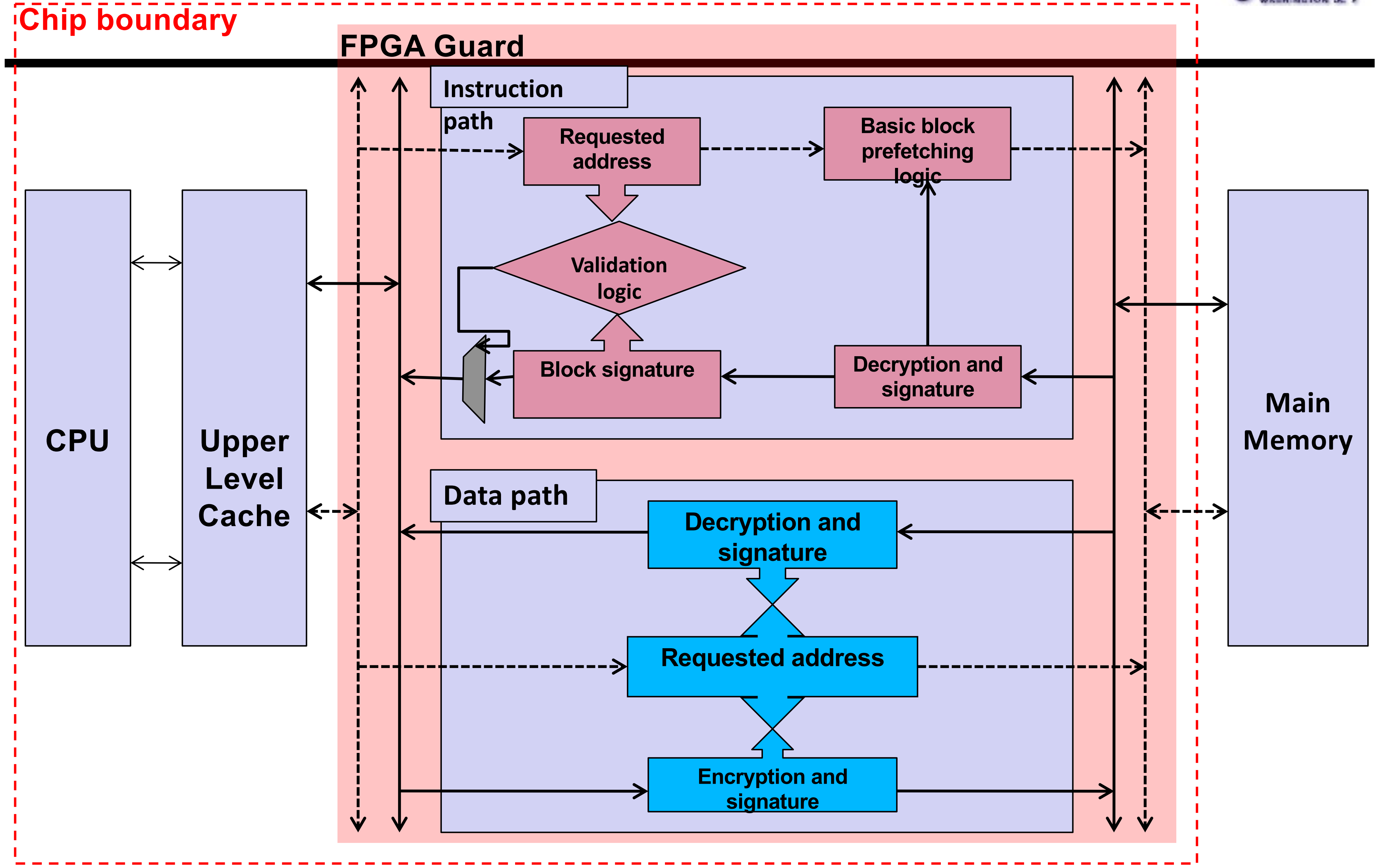
⑤ y_i = sig_runtime(bb_i + (a_i - a_{start}))

⑥ If (y_i == x_i) valid (send to cache), else HALT

Our FPGA Architecture

- Instructions
  - Decryption (AES)
  - Digest calculation/verification (SHA-1/CRC)
  - Label calculation/verification
  - Pre-fetching logic
  - Hardware stack / function protection
- Data
  - Encryption/Decryption (AES)
  - Digest calculation/verification (SHA-1/CRC)
  - Label calculation/verificaiton
  - DFID storage/verification

Architecture: Inside the FPGA Guard



<-----> Address bus <——> Data bus

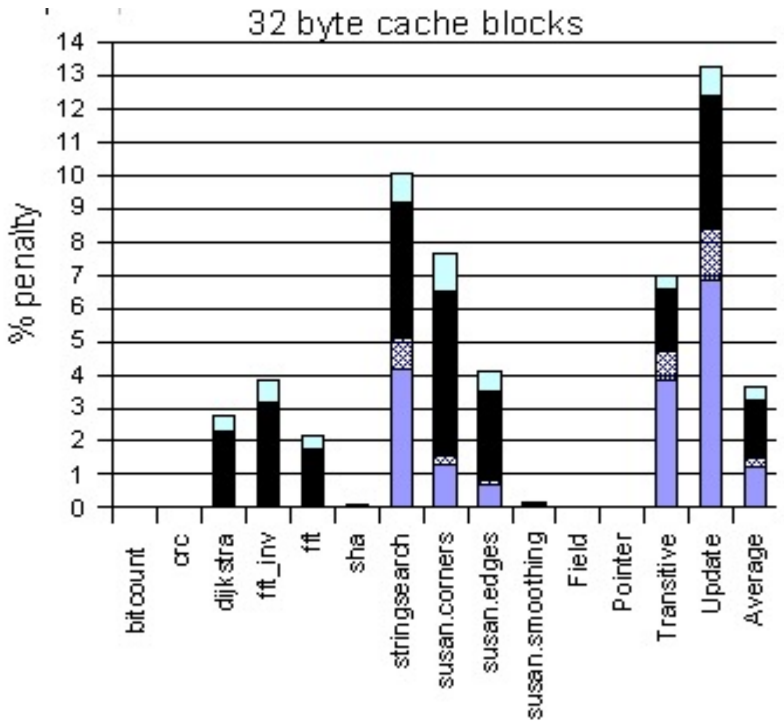
Implementation and Performance

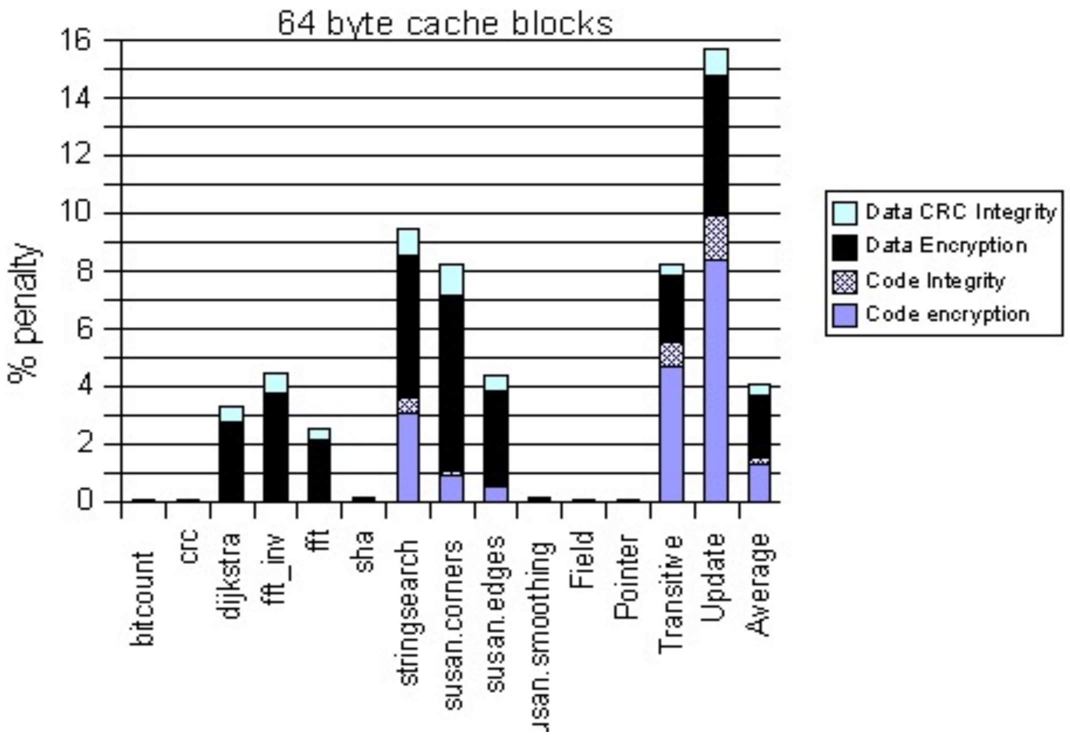
- Software and Simulation Environment
  - modified SimpleScalar cycle level simulator
    - With FPGA functional simulator
  - gcc 3.3 cross-compiler for ARM
    - We provide a module that plugs into gcc backend
    - Our current techniques work at assembly level – can port to other processor ISAs
- Performance depends on security schemes and architecture parameters
  - (1) CRC or (2) SHA-1
    - SHA-1 incurs much larger penalties
  - (a) external storage or (b) internal storage
    - Internal incurs larger penalties
  - Cache block size: 32 byte or 64 byte incurs much larger penalties
    - Smaller cache block size incurs larger penalties
- Hardware Prototype
  - Virtex II Pro FPGA from Xilinx

Simulation Results: Setup

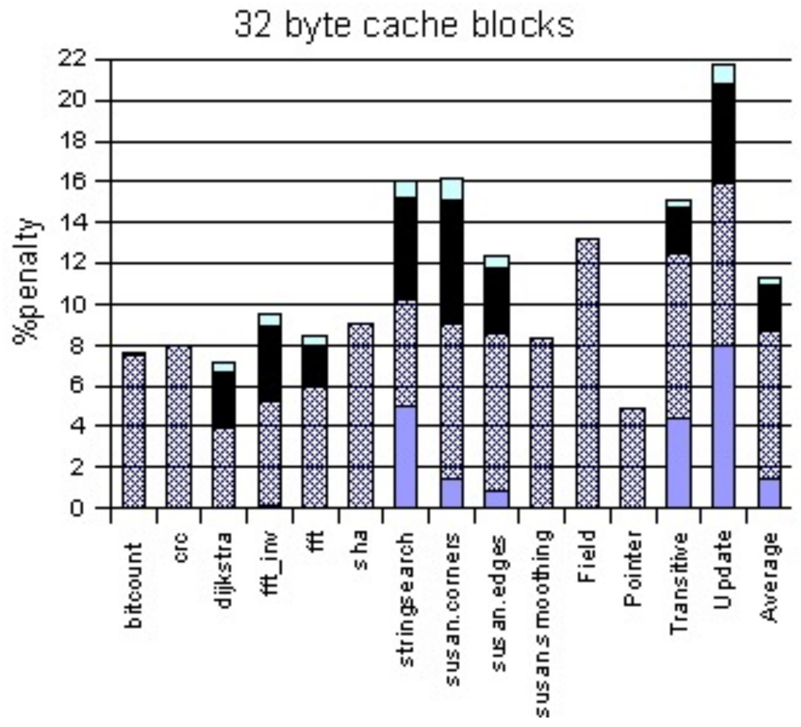
- GCC 3.3, static linking
- Processor:
  - ARM1020E 400 Mhz
  - No L2 cache
  - 32KB cache, 32-byte cache line
  - Cache hit: 1 cycle. Cache miss: 10
- FPGA:
  - Virtex II Pro 200 Mhz
  - Access latency 10 proc. Cycles
  - Upto 3MB on-board memory
- Main Memory:
  - 100 Mhz,
  - Access: 24 + 4
- Hashing Scheme:
  - SHA-1, 20-byte hash
  - Hash calculation: 164 cycles
  - Verification: 2 cycles
- Encryption:
  - AES, 128-bit blocks
  - Encryption latency: 20 cycles/block
- SimpleScalar:
  - FPGA between L1 and Memory
  - In-order execution
- Benchmarks:
  - MiBench: typical benchmark suite for embedded applications
  - DIS: data intensive systems

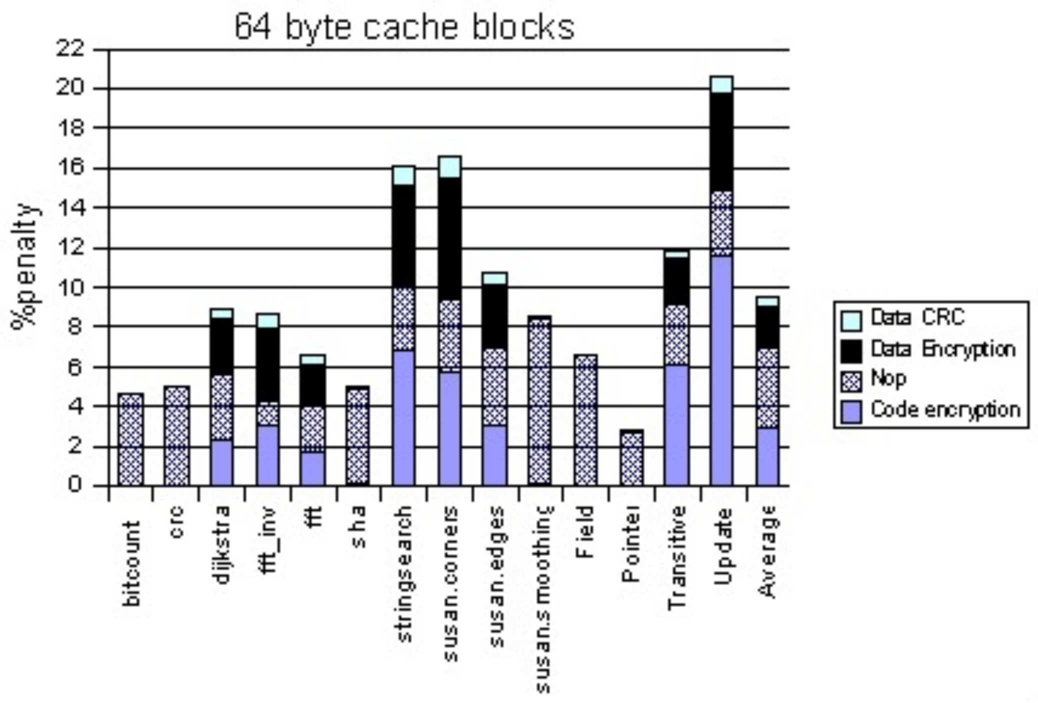
Cache Block Granularity:
(1) CRC for Code and Data, (a) External Storage



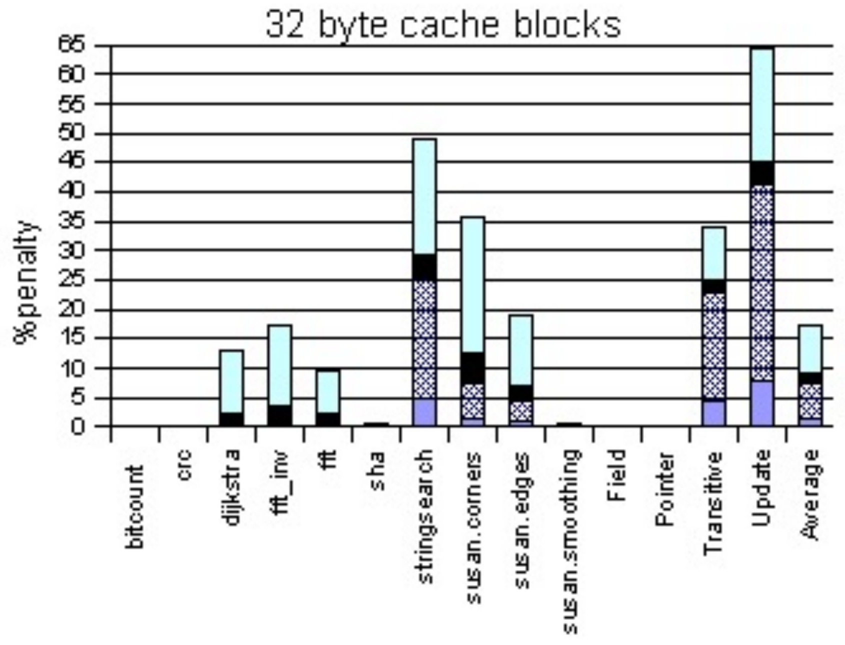


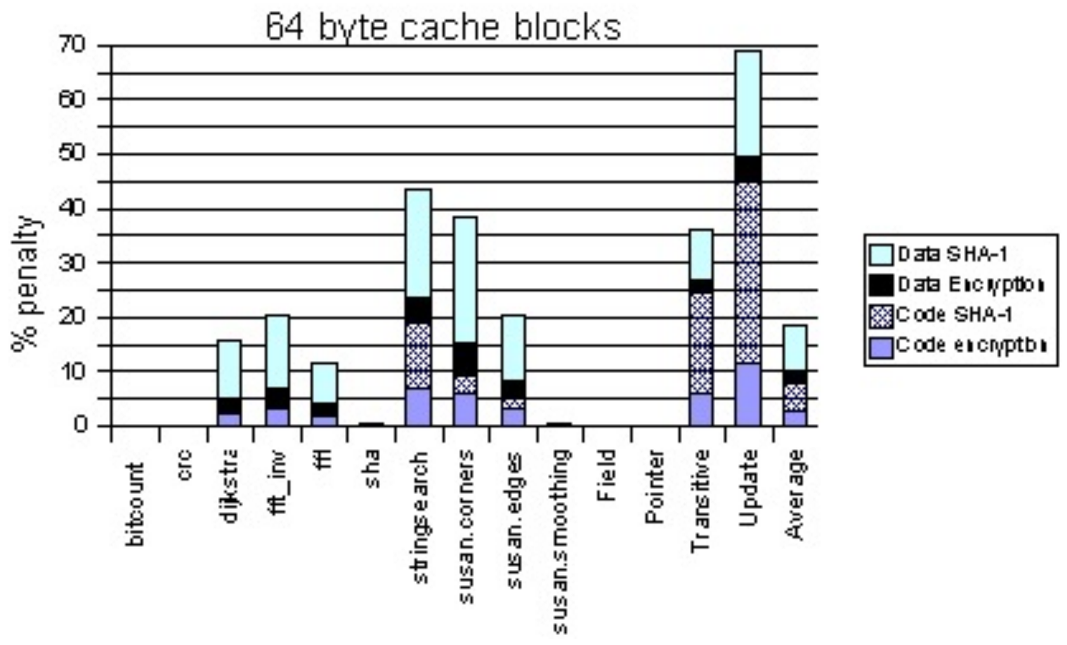
Cache Block Granularity: (1) CRC and (b) Internal Storage





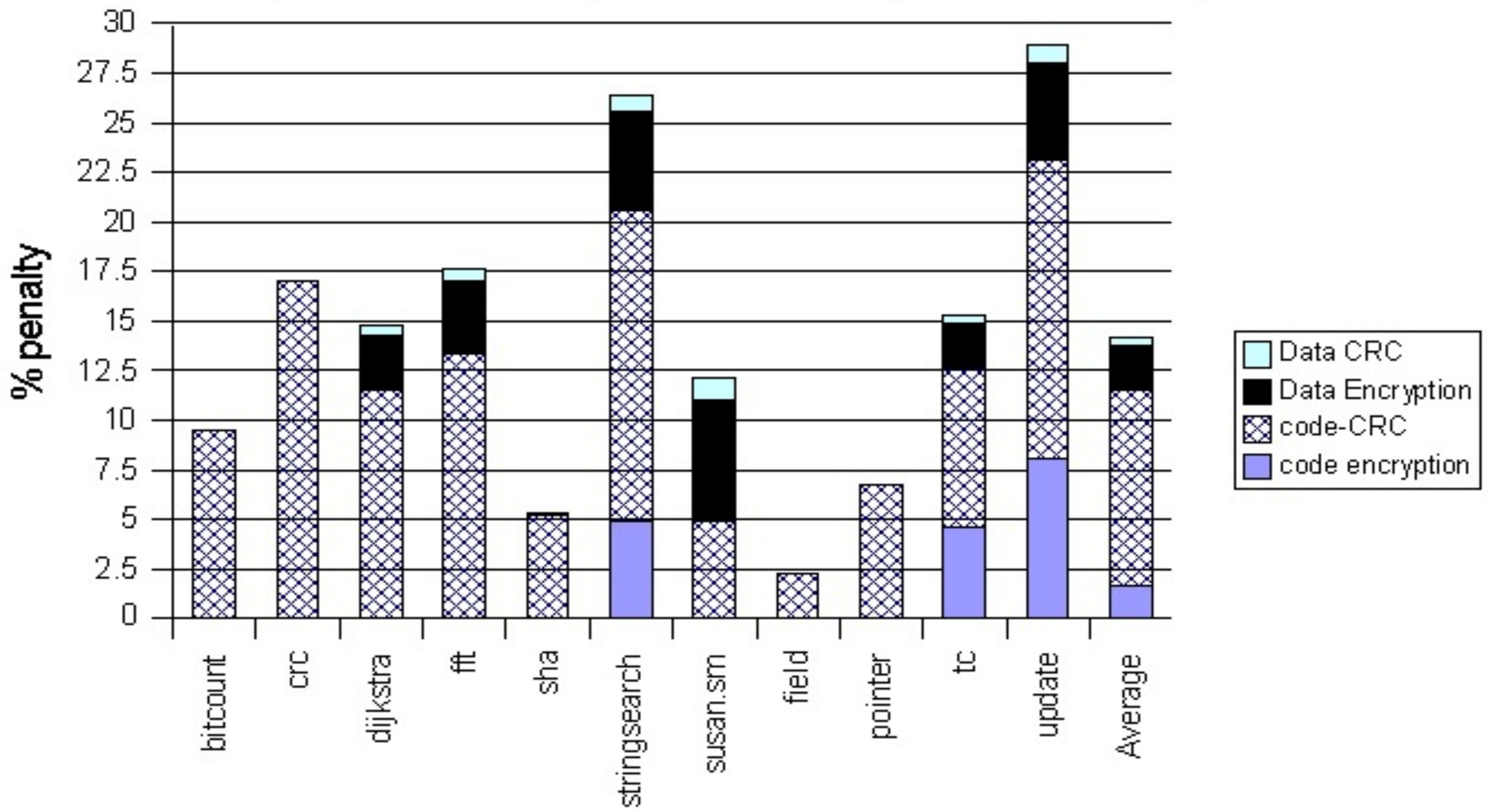
Cache Block Granularity:
(2) SHA1 code and data, (a) External Storage





Basic Block Granularity: (1) CRC and (b) internal storage

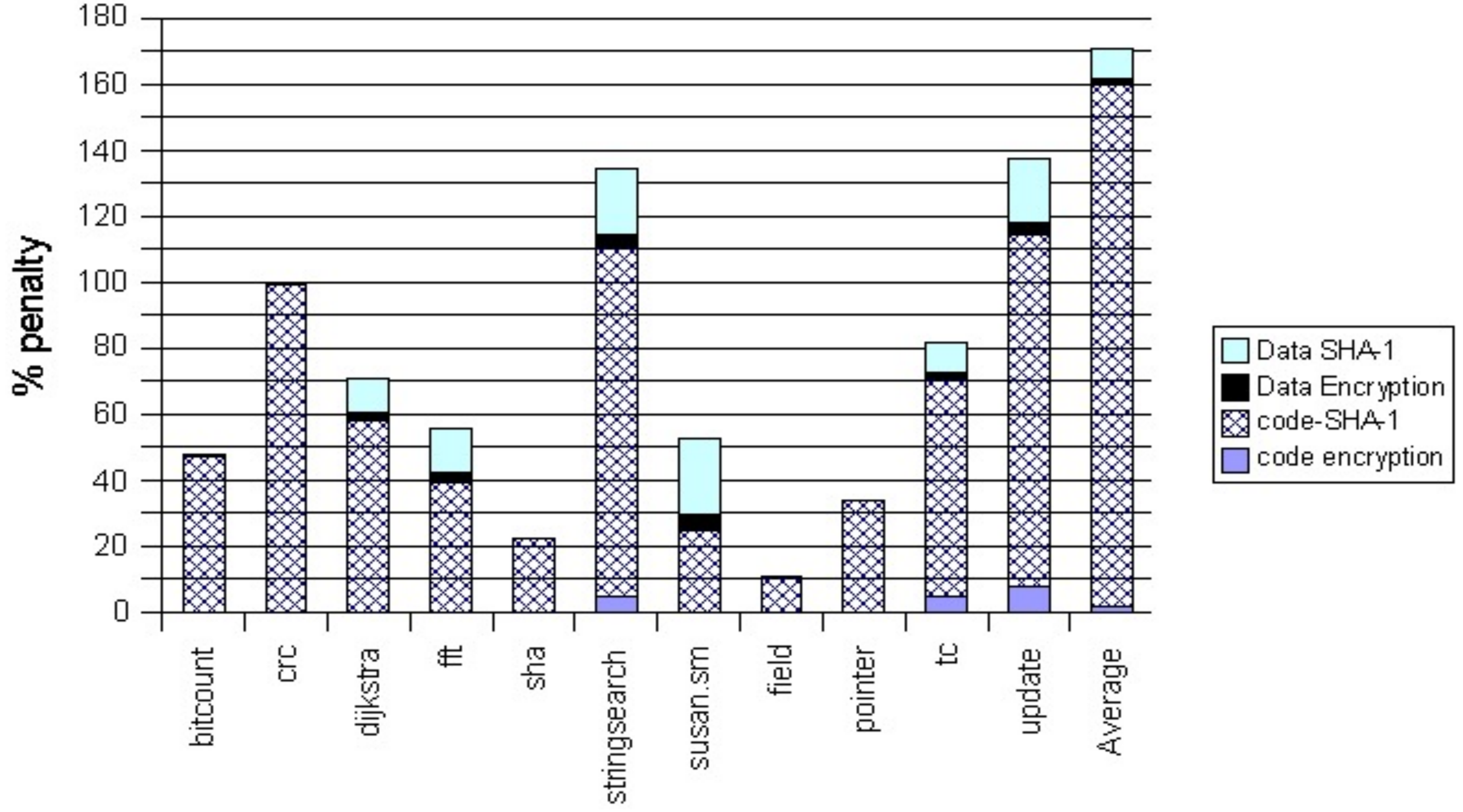
- CRC for basic block integrity check
- Integrity and control-flow information stored inside the basic block
  (replaced with nops at runtime)



32 byte cache blocks

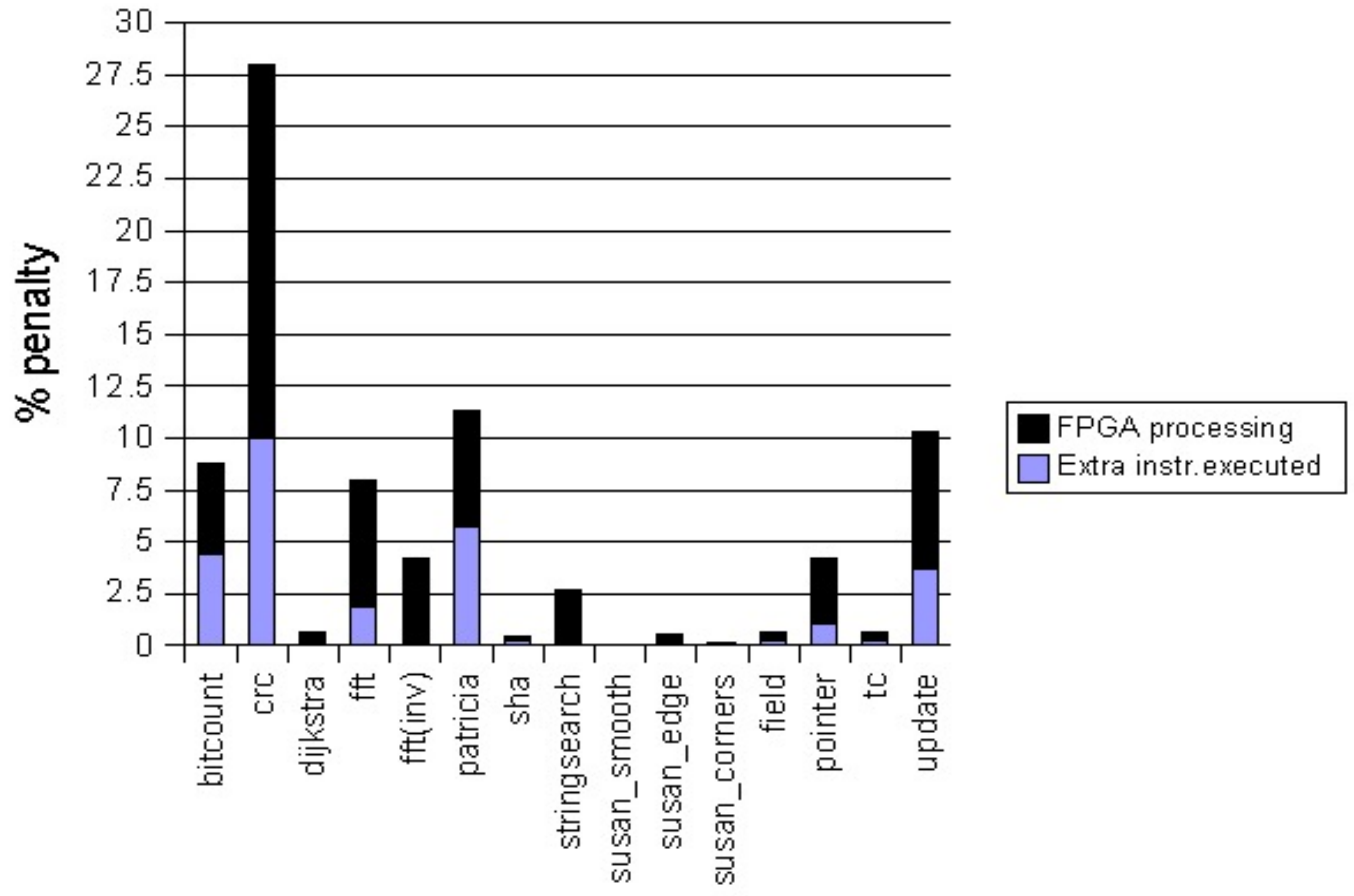
Basic Block Granularity: (2) SHA-1 and (b) internal storage

- SHA-1 for basic block integrity check
- Integrity and control-flow information stored inside the basic block (replaced with nops at runtime)



32 byte cache blocks

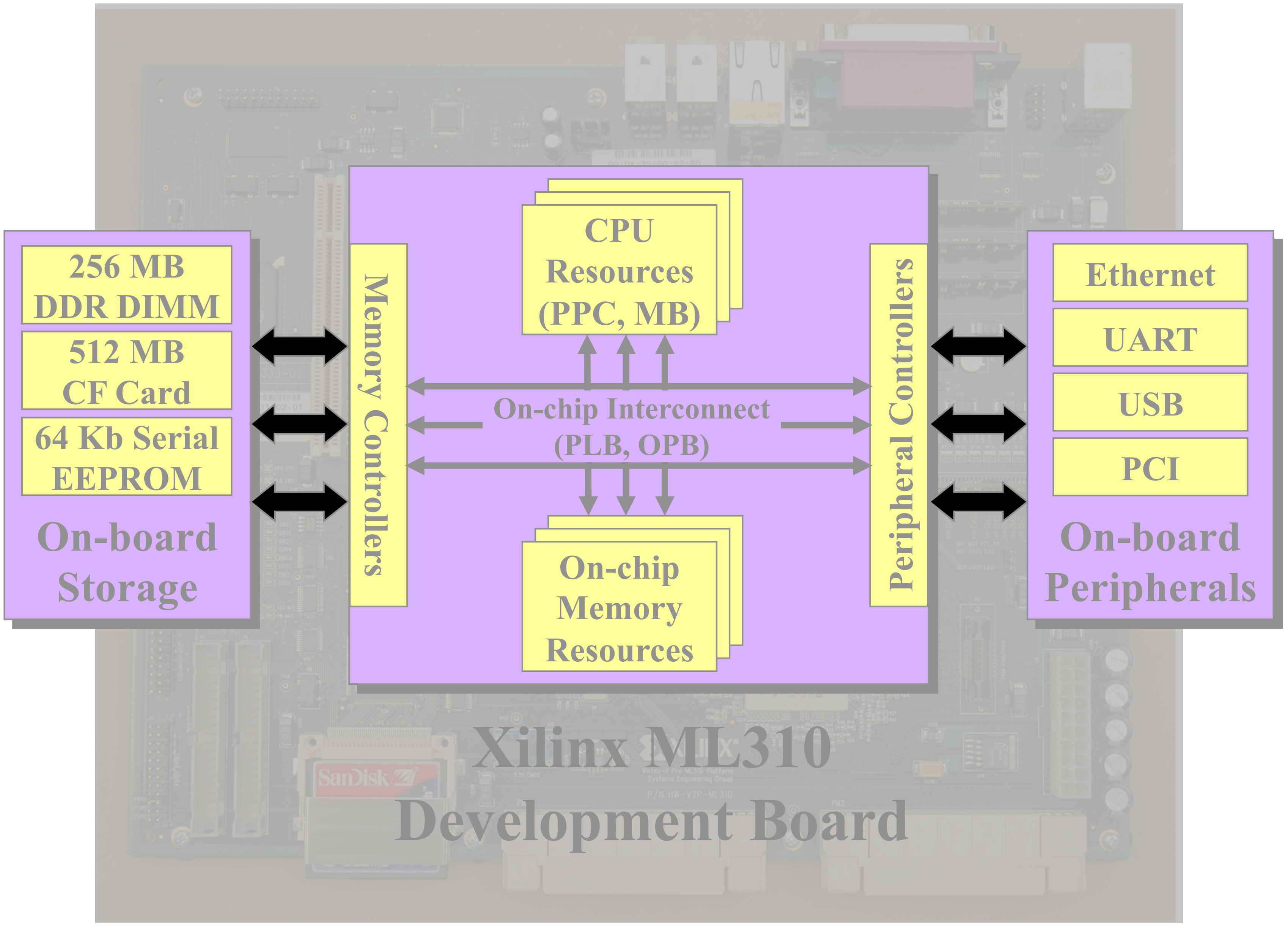
Buffer Overflow Protection Using Hardware Stack in the Guard



Prototype Development

- Hardware Platform
  - Virtex II Pro FPGA from Xilinx
  - compiler
- PowerPC processor + FPGA Guard
- Preliminary results…

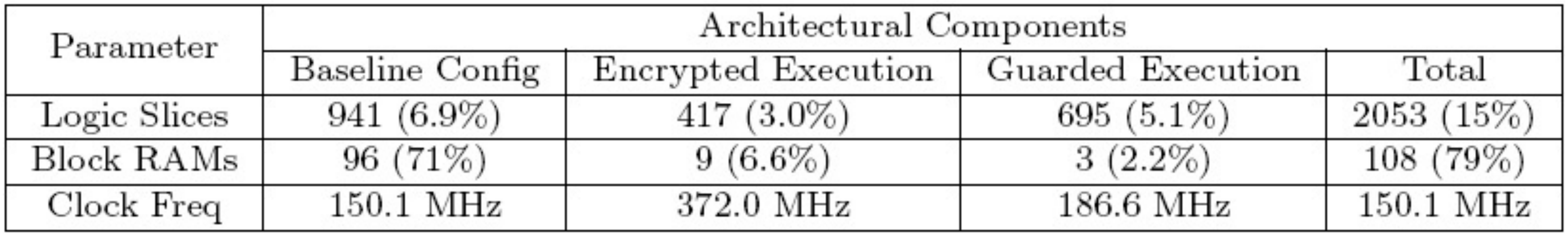
CODESSEAL Prototype Hardware



Synthesis Report

- Prototype implementation of the Guard
- Optimized for ease of implementation and design workflow
- Implementation platform : Xilinx XC2VP30 FPGA
  - PowerPC processor

- Conclusion: enough space even for our unoptimized design
  - Implies an efficient and full implementation of CODESSEAL possible using current COTS technology



Parameter	Architectural Components			
	Baseline Config	Encrypted Execution	Guarded Execution	Total
Logic Slices	941 (6.9%)	417 (3.0%)	695 (5.1%)	2053 (15%)
Block RAMs	96 (71%)	9 (6.6%)	3 (2.2%)	108 (79%)
Clock Freq	150.1 MHz	372.0 MHz	186.6 MHz	150.1 MHz

Key Management..?

- Basic Assumptions:
  - ❑Each Chip (Proc+FPGA) has a unique key
  - ❑Each application will have its own key
- Session key at setup/load time
  - ❑Need ability to swap b/w OS tasks and Application tasks
- What about data?
  - ❑Use same key for all the data generated ??
  - ❑Introduces key management problems that need to be studied

CODESSEAL Summary

- CODESSEAL Examine hardware/software co-design methodologies to secure and protect code
  - ❑ exploits the interplay between compiler and hardware
    - Compiler can play a big role in providing security solutions
  - ❑ Flexibility of FPGAs
  - ❑ Without changing processor ISA and microarchitecture
- Performance
  - ❑ “acceptable” penalties
  - ❑ Average performance for benchmarks as low as 16%

CODESSEAL Summary

- Hardware-software approach for EED attacks
  - comprehensive code and data protections provided
  - Acceptable performance penalty in most cases
- No changes required to processor/ISA, cache or memory organization
  - Past work did not utilize FPGA logic in this manner
  - Off-the-shelf hardware available to implement our system
  - Compiler component ensures no burden on programmer
  - Tradeoff between security and performance
- Acceptable performance penalties
  - As low as 15% penalty for full system protection on average

Collaborators and Acknowledgements

- Collaborators:
  - GWU: R. Simha
  - Doctoral Students: E. Leontie, G. Bloom, O. Chen
  - Northwestern: Prof. Alok Choudhary
  - Iowa State Univ: Prof. Joseph Zambreno
- Sponsors:
  - National Science Foundation (NSF)
  - Air Force Office of Scientific Research (AFOSR)

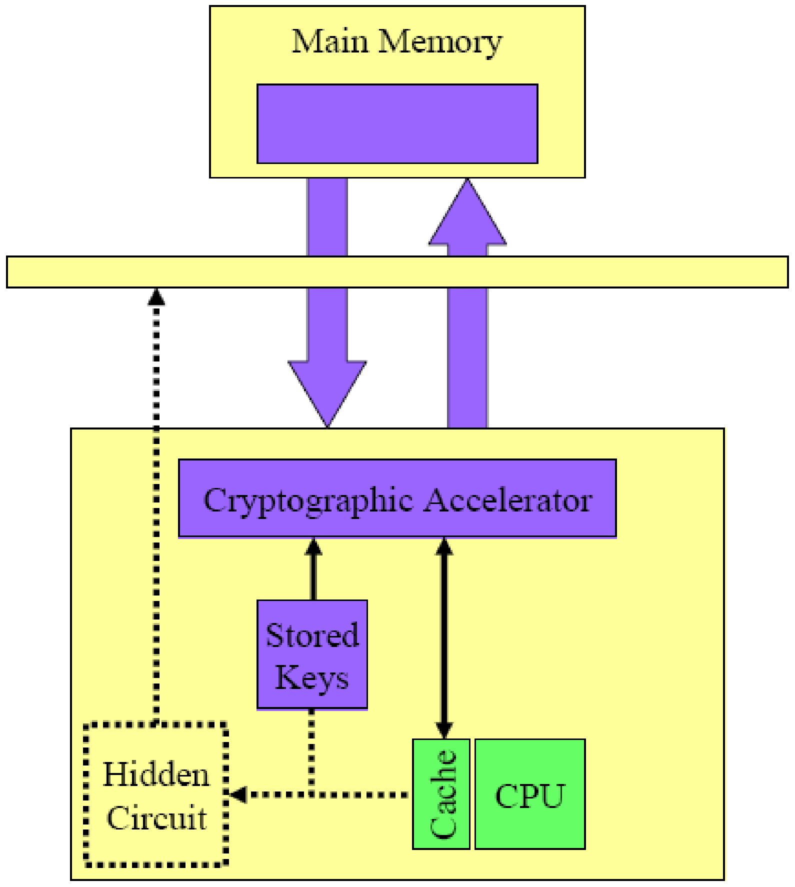
Synergistic research: Brief Overview

- Synergistic efforts on using hardware for SW and Info Assurance
  - Hardware Containers/Wrappers
    - Protecting against backdoors hidden in third party code and enabling recovery from attacks
  - The SHADE Architecture
    - Untrusted chip – when attacker is at the chip foundry and can place hidden/trojan circuit in processor

SHADE: An Architecture for Trusted Execution using Untrusted Components

Untrusted Components: Trojan Circuit

- Example Scenario:
  - Adversary inserts so-called Trojan Circuit into chip
  - Can launch attacks on end products like radios, comm chips even when software has been verified

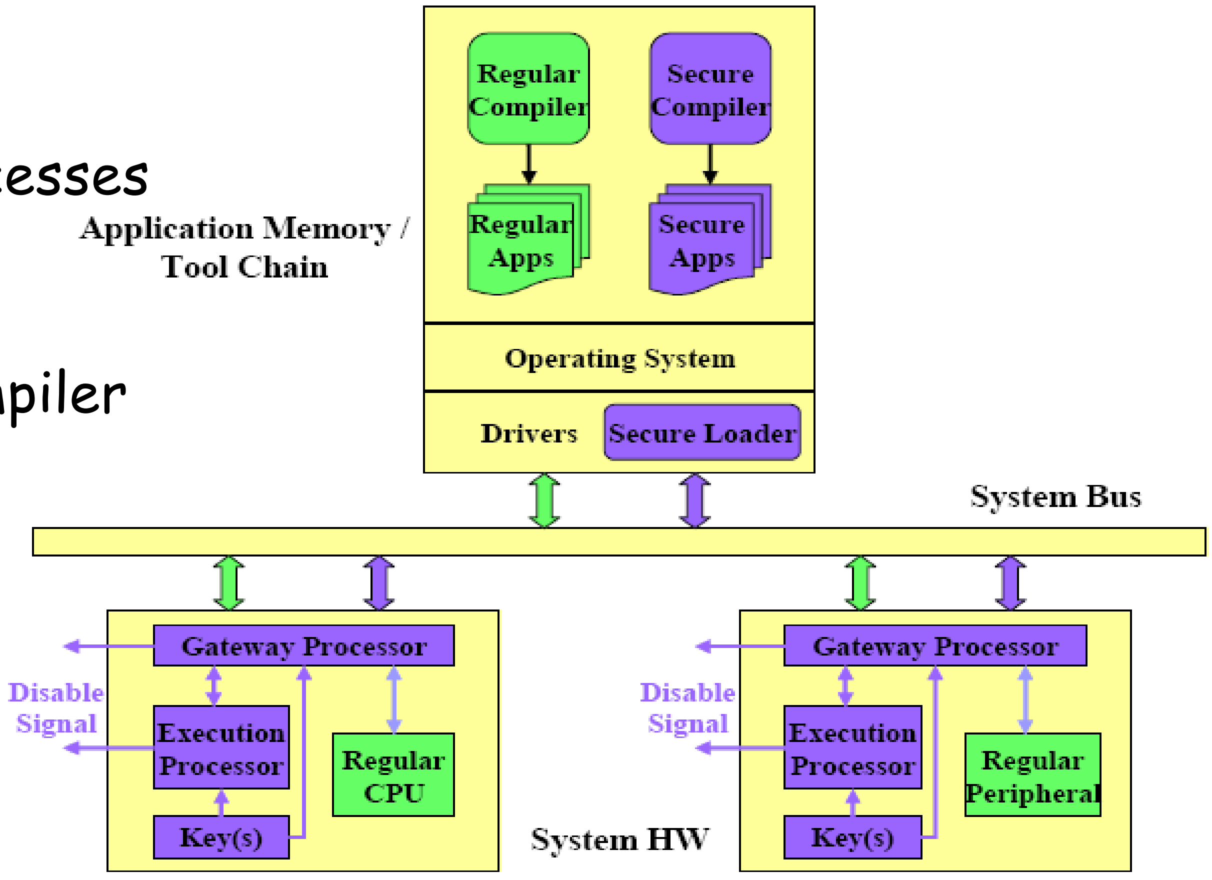


Trojan Circuits: Attack Details

- Leakage attack
  - Seize control of processor and write out decryption keys (for encrypted execution models)
- Denial of service
  - Halt processor at a critical or random time
  - Scan for electromagnetic signals to halt at right cue
- Facilitate reverse engineering
- Hardware firewalls that grant complete external access to the network
  - Packet sent from pre-determined location
  - Key encoded as a series of requests

Our Approach: Secure Heartbeat and Dual Encryption (SHADE)

- Dual processor components
- Gateway
- Dual Encryption
- Backend inserts non-cacheable memory accesses inserted to create a heartbeat
- Architectural and compiler features

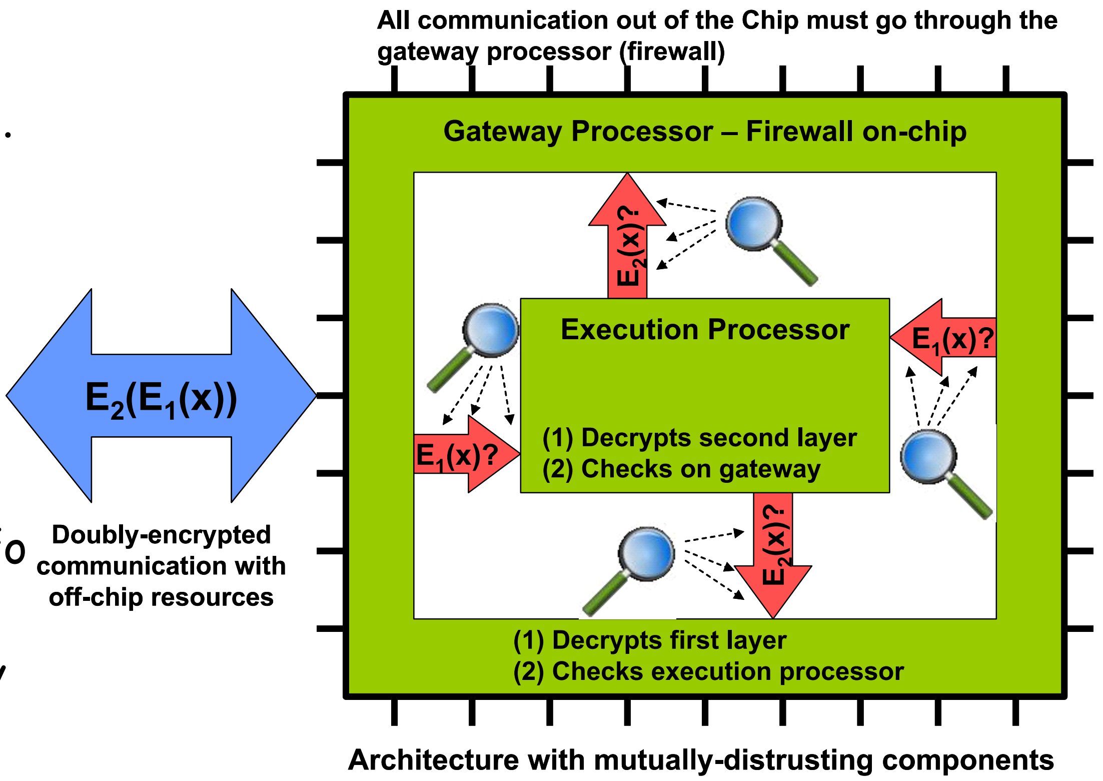


Our Approach

- Dual processor components
  - Gateway and an Execution processor
  - also a standard CPU for non-secure apps – dual use system
- Gateway
  - Perform first level encryption/decryption and send to processor or memory
  - Data values are encrypted first by processor and then again by gateway
- Architectural and compiler features
  - Runs apps in secure and non-secure modes
  - Uses trusted compiler tool-chain
  - Secure apps are doubly encrypted by compiler
- Overhead: varies from 5 – 70%
  - DIS and MIBench, ARM processor.

Single Chip Proposed Approach

- All communication between chip and outside must go through Firewall-On-Chip
- All software is dually-encrypted.
- Firewall = mutually-distrusting components.
- Gateway processor with encryption provides firewall-on-chip capability.
- Hidden circuit in gateway can only leak encrypted (useless) info
- Leak from hidden circuit on processor is trapped by gateway



All communication out of the Chip must go through the gateway processor (firewall)
Gateway Processor – Firewall on-chip
$E_2(x)$?
Execution Processor
$E_1(x)$?
$E_1(x)$?
(1) Decrypts second layer
(2) Checks on gateway
$E_2(x)$?
$E_2(E_1(x))$
Doubly-encrypted communication with off-chip resources
(1) Decrypts first layer
(2) Checks execution processor

Architecture with mutually-distrusting components

Thank you!