Why is Security Important?  
(to Architects and Compiler Designers)

- Hardware and system-level solutions are needed to protect software and intellectual property (IP)
- Hardware and low-level software support improves speed and quality of cryptography
- Hardware and system-level software support can most effectively seal up security vulnerabilities
- Hardware and system-level software vulnerabilities enable security attacks
The Security Arms Race

Develop/Debug Applications and Platforms

Attackers Hack Customers

Deploy Countermeasures

Why Do Attackers Attack?

- To gain control of machines, e.g., BotNets
- To gain access to private information, e.g., credit card numbers
- To punish/embarrass individuals and institutions, e.g., Sony
- To educate and advocate, e.g., FireSheep
- To earn reputation in the cracking community, e.g., hackers vs. script kiddies
- Etc…
The Ultimate Goal of the Designer

- Win the bear race…

  Attackers
  Someone more valuable
  You

- Value = f(easy of attack, population, loot therein, goodwill, etc…)

Tutorial Outline

- Security Basics
- Security Exploit Prevention Techniques
- Side-Channel Attacks and Protections
- Hardware for Secure Computing
- Security Vulnerability Analysis
Acknowledgements

- Colleagues: Valeria Bertacco, Seth Pettie
- Students: Joseph Greathouse, Eric Larson, Andrea Pellegrini
- With contributions from:
  - Edward Chow
  - Crispin Cowan
  - Koji Inoue
  - David Lie
  - Igor Markov
  - Ivo Pooters
  - Hovav Shacham
  - Andrew Tanenbaum
  - Kris Tiri
  - Steve Trimberger
  - Wikipedia

Security Basics

- Cryptography
  - Symmetric key cryptography
  - Asymmetric key cryptography
  - Secure sockets layer (SSL) overview
  - Streaming ciphers
  - Cryptographic Hashes
- Security Attacks
  - Buffer overflow attacks
  - Heap spray attacks
  - Return-oriented programming attacks
  - Hardware-based security attacks
- Discussion Points
**Symmetric Key Cryptography**

- Sender and receiver share a private key
- Anyone who knows the private key can listen in
- Often called a “private-key cipher”
- Examples: AES, DES, Blowfish

![Diagram of Symmetric Key Cryptography](image)

**Block Cipher**

Encrypting each block separately not secure: identical plaintext blocks --> identical cipher text blocks

![Diagram of Block Cipher](image)

*Image from: Security Basics for Computer Architects, Ruby Lee*
Block Cipher Operation Modes

Images from: Security Basics for Computer Architects, Ruby Lee

ECB vs. CBC Streaming Modes

Original  ECB Encrypted  CBC Encrypted
**Asymmetric Key Cryptography**

- Sender has the receiver’s public key, receiver has the private key
- Anyone can encrypt a message with the public key, only the holder of the private key can decrypt the message
  - Allows sharing of private information with no initial shared secret
- The reverse path also works: everyone can decrypt a message that was encrypted by the holder of the private key
- Often called a “public-key cipher”
- Examples: RSA, Diffie-Hellman

---

**RSA Authentication**

- Client sends a unique message to server
- Server encrypts unique message with private key
- Client decrypts the message with public key and verifies it is the same

*Authentication*: only server could return private-key encrypted unique message
Symmetric vs. Asymmetric Ciphers

- Symmetric Ciphers
  - Fast to compute
  - Require prior shared knowledge to establish private communication

- Asymmetric Ciphers
  - Orders of magnitude slower to compute
  - No shared secrets required to establish private communication

- Individual benefits create a need for both types of cryptography

Secure Sockets Layer (SSL) Overview

Encrypt client symmetric key with server public key

Decrypt with symmetric key to ensure authentication successful

client: authenticate
          private key
          https get
          https recv
          close

server: Decrypt with server private key to get symmetric key, return “finished” message encrypted by symmetric key
**Verifying Integrity: Hash Functions**

- **Goal:** provide a (nearly) unique “fingerprint” of the message
- **Hash function for L-bit hash must demonstrate three properties:**
  1. Fast to compute $y$ from $m$.
  2. One-way: given $y = h(m)$, can’t find $m'$ satisfying $h(m') = y$ without $O(2^L)$ search
  3. Strongly collision-free: For $m_1 \neq m_2$, we find $h(m_1)=h(m_2)$ with probability $1/2^L$

- Widely useful tool, e.g., Has this web page changed?
- Examples: MD5 (cryptographically broken), SHA-1, SHA-2

---

**Hash Application: Password Storage**

- Never store passwords as plain text
  - If your machine is compromised, so too are all the user passwords
  - E.g., Gawker.com attack in 2010
- Why protect passwords on a compromised machine?
- Instead, store a cryptographic hash of the password
  - Even with a compromised password file, the passwords are still unknown
  - Use “salt” to eliminate the use of “rainbow tables”
Security Vulnerabilities are Everywhere

- Most often born out of software bugs
- NIST estimates that S/W bugs cost U.S. $60B/year
- Many of these errors create security vulnerabilities

![CERT-Cataloged Vulnerabilities](image)

Buffer Overflow Attack

- Buffer overflows constitute a large class of security vulnerabilities
- Goal: inject code into an unsuspecting machine, and redirect control

```c
void foo()
{
    int local_variables;
    int buffer[256];
    ...
    buffer = read_input();
    ...
    return;
}
```

![Buffer Overflow Attack Diagram](image)
**Escalate: No code allowed on stack**

- Use a *heap-spray attack*

  ![Heap Spray Diagram](image)

  - Inject executable data into heap, then perform random stack smash
    - Example, generate many strings in Javascript that are also real code
  - Generous heap sprays will likely be found by stack smash attack

**Escalate: No new code allowed at all**

- Use *return-oriented programming* to attack...

  ![Return-Oriented Programming Diagram](image)

  - Return-oriented programming introduces no new instructions, just carefully craft injected stack returns to link existed function tails
  - New program is formed from sequence of selected function tails composed from existing code
New Threats: Hardware Based Attacks

- 2008: Kris Kapersky announced the discovery of an OS-independent remote code execution exploit based on an Intel CPU bug (not disclosed)
- 2008: UIUC researcher Sam King demonstrate that 1400 additional gates added to a Leon SPARC processor creates an effective Linux backdoor
- 2008: Princeton researcher Ed Felten demonstrates that disk encryption keys can be extraction after system shutdown from frozen DRAM chips
- 2010: Christopher Tarnovsky announced a successful hardware exploit of an Infineon TPM chip

Security Basics: Discussion Points

- Does the security arms race ever end?
- How do I know that I have the server's true public key?
- Can hardware-based security exploits be fixed?
- Do all security protocols and algorithms have a fixed shelf life?
Security Basics: Bibliography

- Applied Cryptography, Bruce Schneier, Wiley, 1996
- CMU's Computer Emergency Response Team, www.cert.org
- Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn, Nozzle: A Defense Against Heap-spraying Code Injection Attacks, USENIX, 2009
- Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage, When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, CCS 2008

Security Exploit Prevention Techniques

- No-Execute (NX) Stacks
- Address Space Layout Randomization (ASLR)
- Stack Canaries
- Encrypted Pointers
- Hardware-Based Buffer Overflow Protection
- Safe Languages
- Discussion Points
No-Execute (NX) Stacks

- Eliminate stack code injection by preventing code execution on stack
- Can be a problem for some safe programs, e.g., JITs
- NX bit in newer x86 PTEs indicates no-execute permission for pages

Address Space Layout Randomization (ASLR)

- At load time, insert random-sized padding before all code, data, stack sections of the program
- Successfully implementing a buffer overflow code injection requires guessing the padding geometry on the first try
- Implemented in recent Windows, Linux and MacOS kernels
**Attacking ASLR**

- ASLR make stack based code injection difficult because the injected return address is different for each execution.

- A successful attack requires a brute-force guess of an address containing injected code **on the first try**.

- ASLR can be compromised with heap-spray attacks.

---

**Escalate: No code allowed on stack**

- Use a *heap-spray attack*

- Inject executable data into heap, then perform random stack smash
  - Example, generate many strings in Javascript that are also real code
  - Generous heap sprays will likely be found by stack smash attack.
Stack Canaries with StackGuard

- Implemented in compiler (GCC), runtime check of stack integrity
- Embed “canaries” in stack frame before the return address, in function prologue, verify their integrity in function epilogue
- Canary is a per-instance random value that attacker must guess on the first try for a successful attack
- About 10% overhead for typical programs
- Can be thwarted with overflow attacks on function pointers

StackGuard Variant - ProPolice

- IBM enhancement of StackGuard, in GCC, deployed in OpenBSD
- Moves pointers in front of arrays, to protect from overflows
Encrypting Pointers with PointGuard

- Encrypt all pointers while in memory
  - Using a per-instance random key, generated when program starts
  - Each pointer is XOR'ed with this key (decrypted) when loaded from memory to registers or when stored back into memory (encrypted)
  - Pointers cannot be overwritten by buffer overflow while in registers

- Protects return addresses and function pointers

- Attackers must guess, on the first try, the random key to implement a successful pointer attack
  - Otherwise, when pointer is overwritten its XOR decrypted value will dereference to a random memory address

- Very difficult to thwart, but pointer encryption/decryption can slow programs by up to 20%

---

Normal Pointer Dereference

1. Fetch pointer value
2. Access data referenced by pointer

**Normal Pointer Dereference**

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

**Corrupted Pointer**

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer
**PointGuard Dereference**

1. Fetch pointer value
2. Access data referenced by pointer

**Sandboxing: Imprison Potential Violators Early**

- Often attackers will infiltrate one program domain to attack another
  - E.g., inter-tab “man-in-the-browser” attacks
- Sandboxes utilize virtual memory system to contain potential damage
  - External interactions require validation
- Generally reliable but still attackable
  - Through missed external interactions
  - Through bugs in the policy manager
  - Through system-level bugs
  - Through external services, e.g., Flash

Chrome NaCL Sandbox Architecture
NaCL Native Execution: The Rules of the Game

- Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
- The binary is statically linked at a start address of zero, with the first byte of text at 64K.
- All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
- The binary is padded up to the nearest page with at least one `hit` instruction (0x14).
- The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
- All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
- All direct control transfers target valid instructions.

<table>
<thead>
<tr>
<th>Constraints for NaCl binaries.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and %eax, 0xffffffff0</code></td>
</tr>
<tr>
<td><code>jmp *%eax</code></td>
</tr>
</tbody>
</table>

Perhaps We Should Go to the Root of the Problem

- Most buffer overflows occur due to memory access errors

- **Spatial** - Buffer overflow
  ```c
  char *c = malloc(100);
  c[101] = 'a';
  ```

- **Temporal** - Dangling reference
  ```c
  char *p1 = malloc(100);
  char *p2 = p1;
  free(p1);
  p2[0] = 'x';
  ```
Safe Languages Prevent Many Attacks

- Runtime checks verify as the program runs that all accesses are in the bounds of intended live storage
  - Examples: Python, Javascript, Java, Ruby, Go
  - Reduces the attack surface available to attackers

- It is also possible to provide runtime checking in non-safe languages, but at some cost

Are Safe Languages Safer?

- Qualys top 5 vulnerabilities for May 2011
  1. Adobe Flash Player Unspecified Code Execution Vulnerability
  2. Oracle Java SE Critical Patch Update
  3. Adobe Reader and Acrobat Security Update
  4. Apple QuickTime Prior to 7.6.9 Multiple Vulnerabilities
  5. Macrovision InstallShield Update Service Multiple Insecure Methods

- Yes, but safe languages are not a panacea
  - Buffer overflows still occur in the interpreter, JIT, runtime, OS, and drivers
  - Doesn't mitigate non-buffer overflow based attacks, such as SQL injection
  - Not easily made available to legacy programs in unsafe languages

- But, if given a choice, why not choose a safer (and likely more productive) language?
Prevention: Discussion Points

- Are hardware based security protection mechanisms worth the silicon to manufacture them?

- Software-based protection mechanisms seem to be more hardened than hardware-based techniques, why is this the case?

Prevention: Bibliography

- H. Shacham et al, On the Effectiveness of Address-Space Randomization, Computer and Communications Security, 2004
- Benjamin A. Kuperman et al, Detection and prevention of stack buffer overflow attacks, CACM, November 2005
- Todd Austin et al, Efficient Detection of All Pointer and Array Access Errors, PLDI 1994
Side Channel Attacks and Protections

- Timing-Based Attacks
- Cache-Based Attacks
- Power Monitoring Attacks
- Fault-Based Attacks
- Discussion Points

Side Channel Attacks

- Even carefully designed systems leak info about internal computation
  - E.g., safes can be cracked by carefully listening to the tumblers
- Clever attackers can utilize leaked information to grain secrets
  - Generally not directly
  - Use statistical methods over time
- These attacks are often considered attacks on the implementation, rather than the algorithm
Cache-Based Side Channel Attack

- Snoop on cache misses to determine code and data accesses
  - Second process can force misses to DRAM
  - Reveals another process' memory accesses
- Algorithms such as AES are designed from the ground up to thwart these attacks

Cache-Based Side Channel Attacks

1. Resource sharing
   Cache accesses observed by spy process evicting cached data
2. Optimization features
   Cache implemented to overcome latency penalty
3. Increased visibility
   Performance counters provide accurate picture
Hardware Design Techniques Facilitate Side Channel Attacks

1. Resource sharing
   - Reduces hardware needed to implement design functionality
   - Results in interaction and competition revealed in timing and power

2. Design optimizations
   - Typical case optimized, thus the corner cases leak information
   - Corner cases run slower and use different hardware leading to distinct timing and power signatures

3. Increased visibility and functionality
   - Provides more information or introduces new interactions
   - Facilitates observation of unique activities/interactions with unique timing and power signatures

Types of Side-Channel Attacks

- Simple attacks
  - Measure the time to perform expected operations
  - Requires precise knowledge of implementation and effect on measurement sample
  - E.g. textbook square-and-multiply RSA algorithm
  - Relatively easy to protect from this attack

- Differential attacks
  - Many observations made holding only one aspect of the input constant
  - Statistical techniques reveal the timing effects of this one input
  - Correlate timings between known and unknown key tests to guess key
**Timing-Based Side-Channel Attacks**

- Daniel Bernstein announces successful timing attacks against AES in April 2005, exploiting cache timing of table lookups
  - Requires cycle-accurate cipher runtime measurement and ability to control plaintext on the attacked machine
  - Requires access to an identical machine with cycle-accurate cipher runtime measurement, and ability to control plaintext and key values
  -Guesses private key by correlating timings of a target machine to those of an identical reference machine with a known key
- Cache conflicts in AES encryption steps slow computation and leak private key information
- High number of samples required, best case as reported by Bernstein is around $2^{27.5}$ tests to recover 128-bit private key

**Side-Channel Power Attacks**

![Diagram of power attack process]

- AES: 128-bit secret key is brute force impossible
- Can be noisy model
- Estimated power is number of changing bits, can be lousy
- AES: 128-bit secret key brute force is impossible
- Compare both and choose key guess with best match

\[ P = S^{-1}(K_G \oplus C) \]
\[ E = H_{mW}(P) \]
**Power Analysis Example**

- Unprotected ASIC AES with 128-bit datapath, key scheduling
- Measurement: $I_{\text{peak}}$ in round 11
- Estimation: HamDistance of 8 internal bits
- Comparison: correlation
- Key bits easily found despite algorithmic noise
- 128-bit key under 3 min.

**Fault-Based Attack of RSA**

**Correct behavior:**
- Server challenge: 
  \[ s = m^d \mod n \]
- Client verifies: 
  \[ m = s^e \mod n \]

**Faulty Server:**
\[ \hat{s} \neq m^d \mod n \]
**Fault-Based Attack of RSA**

- The attacker collects the faulty signatures
- The private key is recovered one window at the time
- The attacker checks its guess against the collected signatures

**Retrieving the Private Key**

- The private key is recovered one window at the time, guessing where and when the fault hits
- The private key is recovered one window at the time, guessing where and when the fault hits

\[ s = (\cdots (m^{d_3})^{16}m^{d_2})^2 \pm 2f^2m^{d_1})^{16}m^{d_0} \]

- Extend the window if no signature confirms value of guess
Fault Injection Mechanisms

How to make hardware fail:

- **Lower voltage** causes signals to slow down, thus missing the deadline imposed by the system clock
- **High temperatures** increase signal propagation delays
- **Over-clocking** shortens the allowed time for traversing the logic cloud
- **Natural particles** cause internal signals to change value, causing errors

All these sources of errors can be controlled to tune the fault injection rate and target some units in the design.

Physical Demonstration of Attack
**Row Hammer Attack**

- Attack flips bits in victim DRAM row, without permission to access
  - Result of wordline crosstalk
  - Creates small pulses on adjacent wordlines, increases bitcell leakage
  - Hammer enough times (~400k) in one refresh cycle (~64ms) and bits will flip in victim row
- Typical protection requires doubling the refresh rate
- Why doesn’t this happen all the time?
**Hardware Trojans**

- Hardware-based back doors inserted into the design by a rogue engineer
- Typically coupled with a trigger circuit that recognizes a code or data sequence
  - Implement with hash function
- Difficult to detect
  - Given range of approaches
  - Many points of entry in the design process

**Processor Trojan Example**

- Processor updates privilege bit in EX stage
- If code sequence precedes update (recognized by trigger hash)
  - Privilege update is always "1" (enter privileged mode)
- Attack: 1) execute trigger code sequence, 2) own machine (as you now have privilege mode access)
Another Example of a Hardware Trojan

- WiFi-enabled inline USB key logger
- Install and it will send key presses to remote site

Side Channels: Discussion Points

- Is it possible to close a side channel completely?

- How much concern should we put on attacks that have unrealistic/favorable pre-requisites, e.g., Bernstein’s requirement to control key and plaintext plus cycle-level timing, Austin’s requirement to control server voltage
Side Channels: Bibliography

- Z. Wang et al, New cache designs for thwarting software cache-based side channel attacks, ISCA 2007
- J. Kong et al, Deconstructing new cache designs for thwarting software cache-based side channel attacks, CSAW 2008
- I.L. Markov, D. Maslov, Uniformly-switching Logic for Cryptographic Hardware, DATE 2005
- Andrea Pellegrini, Valeria Bertacco and Todd Austin, Fault-Based Attack of RSA Authentication, DATE-2010, March 2010

Hardware for Secure Computing

- Secure boot – TPMs
- Generating True Random Numbers
- Crypto-engines – CryptoManiac
- Physical unclonable functions
- Chiplocking Technologies
- Secure Execution
- High-Bandwidth Digital Content Protection
- Discussion Points
Bootstrapping a typical PC

What can go wrong before the kernel runs?

Secure Boot

- Goal of secure boot
  - Ensure only a secure system is booted
  - Operating system that is bootstrapped is based on a untampered foundation

- Why is this useful?
  - Ensure original integrity of system (i.e., no hacking)
  - Protect internal intellectual property (i.e., IP)
  - Examples: iPhone, Xbox 360, SELinux

- Implementation can only be guaranteed if-and-only-if:
  - Base layer is immutable (requires hardware support)
  - The integrity of the base layer is verified
  - Transition to higher layer only occurs after valid verification
**TCG Trusted Platform Module (TPM)**

- Endorsement Key (EK)
- Platform Configuration Register (PCR)
- Non-Volatile Storage
  - Accepts only signed data (EK, AIK, SRK)

**SELinux Trusted Boot Stages**

- BIOS
- Bootloader
- GRUB Stage 1
- GRUB Stage 2
- Operating System
  - JVM
  - SE Linux Kernel
  - MAC Policy
Why Are Random Numbers Important?

- Generally, secret key == random number

- If your random number generator is weak (i.e., guessable), then your secret key are guessable
  - Example: Early Netscape implementation seeded a pseudo-random number generator with \textit{<time of day, process ID, parent process ID>}

- Where can we find true random numbers?
  - Random atomic or subatomic physical phenomenon whose unpredictability can be traced to the laws of quantum mechanics (Best)
  - Human behavior, e.g., moving a mouse (OK)

---

**Intel Random Number Generator**

[Diagram showing the Intel Random Number Generator process]

- Johnson Thermal Noise Source (Resistor) → Noise Amplifier → Voltage Controlled Oscillator → High-Speed Oscillator
- Slow oscillator samples fast one
- Voila! Variable-rate random bit stream
- Von Neumann corrector removes bias
- Control/Status Reg. → FIFO → Digital Corrector

Figure 1: Block diagram of the Intel RNG
**But, If You Need an Exceptional RNG**

- SGI Lavarand – Lava lamp based random number generator
- US Patent #5732138 – hash the image of the lamp
- Provided an online source of random numbers, 1997-2001

**CryptoManiac Crypto Processor**

- **Goal** - fast programmable private-key bulk cryptographic processing
  - Fast: efficient execution of computationally intensive cryptographic workloads
  - Programmable: support for algorithms within existing protocols, support for new algorithms
- **Motivation**
  - Cipher kernels have very domain specific characteristics
- **Solution** - hardware/software co-design
  - Software: crypto-specific ISA
  - Hardware: efficient co-processor implementation
- **Results**
  - More than 2 times faster than a high-end general purpose processor and orders of magnitude less area and power
**CryptoManiac System Architecture [ISCA’01]**

- A highly specialized and efficient crypto-processor design
  - Specialized for performance-sensitive private-key cipher algorithms
  - Chip-multiprocessor design extracting precious inter-session parallelism
  - CP processors implement with 4-wide 32-bit VLIW processors
  - Design employs crypto-specific architecture, ISA, compiler, and circuits

**CryptoManiac Processing Element (CM)**
**Crypto-Specific Instructions**

- **frequent SBOX substitutions**
  - \( X = \text{sbox}(y >> c) \& 0xff \)

- **SBOX instruction**
  - Incorporates byte extract
  - Speeds address generation through alignment restrictions
  - 4-cycle Alpha code sequence becomes a single CryptoManiac instruction

- **SBOX caches provide a high-bandwidth substitution capability (4 SBOX's/cycle)**

---

**Crypto-Specific Instructions**

- **Ciphers often mix logical/arithmetic operation**
  - Excellent diffusion properties plus resistance to attacks

- **ISA supports instruction combining**
  - Logical + ALU op, ALU op + Logical
  - Eliminates dangling XOR’s

- **Reduces kernel loop critical paths by nearly 25%**
  - Small (< 5%) increase in clock cycle time

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add-Xor r4, r1, r2, r3</td>
<td>( r4 &lt;-(r1+r2)\oplus r3 )</td>
</tr>
<tr>
<td>And-Rot r4, r1, r2, r3</td>
<td>( r4 &lt;-(r1&amp;&amp;r2)&lt;&lt;r3 )</td>
</tr>
<tr>
<td>And-Xor r4, r1, r2, r3</td>
<td>( r4 &lt;-(r1&amp;&amp;r2)\oplus r3 )</td>
</tr>
</tbody>
</table>
Crypto-Specific Functional Unit

Scheduling Example: Blowfish

Takes only 4 cycles per iteration to execute!
**Crypto Support in Modern CPUs**

- **IBM Power7 and Power8**: Implement multiple AES block cipher operation modes entirely in hardware (AES-GCM, AES-CTR, AES-CBC, AES-ECB)
- **Intel Westmere (32nm) and newer**: Implement AES block cipher hardware accelerators; software implements operation modes

  - **AESENC**: This instruction performs a single round of encryption.
  - **AESENCLAST**: Instruction for the last round of encryption.
  - **AESDEC**: Instruction for a single round of decryption
  - **AESDECLAST**: Performs last round of decryption.
  - **AESKEYGENASSIST**: Used for generating the round keys used for encryption.
  - **AESIMC**: Is used for converting the encryption round keys to a form usable for decryption using the Equivalent Inverse Cipher.
**Hardware for Per-IC Authentication**

- How can we securely authenticate devices?
  - Keycards, RFIDs, mobile phones
  - Genuine electronics vs. counterfeits
  - Device allowed to display a purchased movie
  - Ensure we are communicating with a specific server

- Each system must have a unique IC
  - Expensive to customize each manufactured IC
  - Physical unclonable functions (PUFs) implement this very cheaply

**Physical Unclonable Functions (PUFs)**

- Extract secrets from a complex physical system

- Because of random process variations, no two Integrated Circuits even with the same layouts are identical
  - Variation is inherent in fabrication process
  - Hard to remove or predict
  - Relative variation increases as the fabrication process advances

- Delay-Based Silicon PUF concept
  - Generate secret keys from unique delay characteristics of each processor chip
### PUF Circuit Using Ring Oscillators

Compare frequencies of two oscillators \( \rightarrow \) The faster oscillator is randomly determined by manufacturing variations.

### Entropy: How Many Bits Do You Get?

- Each of the \( N(N-1)/2 \) distinct pairs will produce a result based on IC-specific process variation.

- Ordering the speed of all of the oscillators produces a \( \log_2(N!) \) bit per-IC unique identifier:
  - Each ordering is equally likely.

- A small number of oscillators can express keys of long length:
  - 35 oscillators produce 133 bits
  - 128 oscillators produce 716 bits
  - 256 oscillators produce 1687 bits
Arbiter-Based Silicon PUF

- Compare two paths with an identical delay in design
  - Random process variation determines which path is faster
  - An arbiter outputs 1-bit digital response
- Multiple bits can be obtained by duplicating circuit or use different challenges
  - Each challenge selects a unique pair of delay paths

XOM Secure Execution

- Programs are encrypted with symmetric key
- XOM processor accesses encrypted program by decrypting IP key with XOM public key
- XOM architecture implements secure and insecure domains, with policies to move data between differing domains
Hardware: Discussion Points

- What are the relative advantages and disadvantages of a crypto engine implemented as an ASIC, for a specific cipher?

- Can PUFs be affected by extreme environmental changes and silicon wearout can compromise PUF integrity?

Hardware: Bibliography

- W. A. Arbaugh et al, A secure and reliable bootstrap architecture, Symposium on Security and Privacy, 1997


**Security Vulnerability Analysis**

- Security Vulnerability Analysis Overview
- Program metadata storage mechanisms
- Heavyweight Analyses for Security Vulnerability Analysis
- Scaling the Performance of Heavyweight Analyses

**How We Find Vulnerabilities Today**

- Develop/Debug Application
- Attackers Hack Customers
- Debug Customer Attacks
- Sorry!
A Better Way - Security Vulnerability Analysis

Deploy

Develop/Debug

Employ

Exposed Vulnerabilities

Take the criminal out of your design cycle!

Bug Bounties: A Clever Approach to Security Vulnerability Analysis

- Humans have proven to be effective at finding security bugs
  - For good or for bad…
- Bug bounties are paid for severe bugs in popular software
  - Google pays $1337 for each severe bug found
  - Mozilla pays $3000, plus a t-shirt!
- Pwn-to-Own contest gives away hardware for newly exposed bugs
- An effective means of finding vulnerabilities and converting blackhats to whitehats
Toward Scalable Vulnerability Analysis

Today we look at three powerful technologies that I helped to develop:

1) **Metadata** that restore programmer intent

2) **Input bounds checking** to expose hidden bugs without an active exploit

3) **Dynamic distributed debug (Testudo)** to scale the performance of vulnerability analysis

Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - Multiple types of tests, runtime protection
  - Extremely high runtime overheads

![Diagram of software dynamic analysis process](image)
Security Vulnerability Analysis Example: Taint Tracking

Key point: Techniques locate bugs without active attack, only need to analyze vulnerable code path

Testudo: Dynamic Distributed Debug [MICRO'08]

- Split analysis across population of users
  - Low HW cost, low runtime overhead, runtime information from the field
    - Analysis only

Users running at virtually full speed

Reported before they are exploited

Our frustration is subsiding

Instrumented Program Potential problems encountered
Vulnerability Analysis: Discussion Points

- What is the trade-off between static vs. dynamic program analysis?
- Is testing all of the paths users execute sufficient to harden a program against security attacks?
- Is it possible to combine static and dynamic program analysis?

Vulnerability Analysis: Bibliography

- Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi, Efficient Detection of All Pointer and Array Access Errors, PLDI 1994
- Joseph Greathouse, Chelsea LeBlanc, Valeria Bertacco and Todd Austin, Highly Scalable Distributed Dataflow Analysis, CGO 2011
Where to Learn More…