T-110.5211 Cryptosystems

Implementation issues

13.11.2008

Kaufman et al: n/a

Stallings: n/a

(There's very little in either book about these issues)

Outline

- Secure handling of storage media
- Cryptographic libraries
- Side and covert channel attacks

Brief note – random numbers

- Leo Dorrendorf, Zvi Gutterman, Benny Pinkas:
  Cryptanalysis of the Random Number Generator of the Windows Operating System
  - http://eprint.iacr.org/2007/419.pdf
- Windows 2000 cryptographic PRNG is vulnerable (XP/Vista?)
  - CryptGenRandom() - given the internal state of the generator, the previous state can be computed with $O(2^{23})$ effort (up to 128 KB backwards)
  - Implication: if attacker gets PRNG state, he can predict PRNG outputs forwards and backwards (getting state: e.g. buffer overflow => browser SSL attack)
- Contrast this to Fortuna
  - Forward security: attacker cannot predict backwards because AES key is changed after every PRNG request is satisfied
  - Backward security: escape mechanism by reseeding with larger and larger amounts of additional entropy
- If interested, you're welcome to verify this on “hacker course” :)

Secure handling of storage media

Overview

- Cryptography deals with secrets
  - Storing secrets (eg. long-term crypto keys) is dangerous, risk of leaking
  - How to minimize leaks ?
- Two important media (though others exist)
  - Magnetic storage (hard disks, tapes, etc)
  - Memory
- Two important concerns
  - How to **store** secrets while they are needed
  - How to **erase** secrets when they are no longer needed
- Surprisingly difficult to do well
  - See e.g. Peter Gutmann: *Secure Deletion of Data from Magnetic and Solid-State Memory*
- For most systems not critical concerns though
  - Advanced attacker

Secure use of memory

- First, use **memory locking**
  - Prevent OS virtual memory paging (memory to disk)
  - Important if you trust the memory but not the disk
  - **Must** apply to the entire code base dealing with crypto
  - Example: crypto key management daemon, inter-process communication, kernel crypto drivers
  - Difficult in high level programming environments (e.g. Java)
- Second, determine realistic threat model for memory
  - Are attacks against main memory / CPU cache relevant?
  - If memory is trusted, memory locking is enough
- Techniques beyond memory locking for untrusted RAM
  - Memory encryption (against on-line attacks)
  - Memory erasure (against off-line attacks)

Implementing memory locking

- Operating system specific (e.g. Linux: **mlock()**, **mlockall()**)
  - It's often difficult to lock only certain portions
  - Often easiest to lock the entire process, unless it's very large
- Complications
  - Interprocess communication guaranteed (by kernel) not to page?
    - One solution (used by Vista): “SSL-like” (D-H) communication between processes
  - Even if two processes are memory locked, is there a guarantee that data never goes to paged memory in-between ?
  - Hibernation (typically laptops)
  - High level programming environments (Java daemon + entire JVM + ..)
- Even so, memory locking may be a critical function
  - Long term keys – HD encryption, RSA private keys, etc.
  - Example: 3DES HD encryption => try all possible consecutive 24-byte choices in swap file as encryption keys (quick attack)

Memory remanence effect

- Root cause of problems – **memory remanence**
  - Previous states of memory can be determined after power-off
  - Happens with both SRAMs and DRAMs (“ion migration”)
- Remanence is cumulative
  - The longer a bit patterns sits in RAM, the more “burnt in” it becomes, and the longer it takes to erase
  - Repeated overwriting with random data not very effective, erasure needs time relative to burn-in period
- Alternatives
  - 1) Never keep critical data in memory
  - 2) Keep critical data in encrypted form in memory
  - 3) Keep critical data in plaintext form, but avoid burn-in
  - 4) Erase memory properly after usage (problems: theft, power outage)

Memory encryption

- Similar to HD encryption, but applied to main memory
  - Although encrypted data “burns” into memory, it's useless
- Still need some memory to do computation
  - Security critical systems often have a smaller trusted memory which can be relied on for holding plaintext data
  - CPU registers and caches much more trustworthy than main memory
- Memory encryption key needs to be protected
  - Cannot sit in untrusted memory – where should it be stored then?
- Using just registers and caches not practical in PC HW
  - OS and hardware limitations
- => What to do? Is memory encryption feasible on PCs?

Memory encryption
Boojum (Schneier & Ferguson)

- Partial solution (*)
  - Allocate two memory locations, $\mathbf{K_1}$ and $\mathbf{K_2}$, each size of the key **K**
  - Initialize: $\mathbf{K_1 = RND()}$; $\mathbf{K_2 = K \text{ xor } K_1}$ (=> $\mathbf{K = K_1 \text{ xor } K_2}$)
  - Periodically (e.g. every 100ms), update $\mathbf{K_1}$ and $\mathbf{K_2}$ as
    - $\mathbf{T = RND()}$; $\mathbf{K_1 = K_1 \text{ xor } T}$; $\mathbf{K_2 = K_2 \text{ xor } T}$
  - System shutdown: Wipe $\mathbf{K_1}$ and $\mathbf{K_2}$ as effectively as possible
  - **Avoids memory burn-in** (bits change randomly, no cumulation)
  - (Note that this is basically secret splitting.)
- Thus, to read encrypted memory, ideally
  - Extract key **K** into registers as $\mathbf{K_1 \text{ xor } K_2}$
  - Read desired encrypted block into registers
  - Decrypt data in registers, using key (intermediate data in safe regs/mem)
  - Process data in registers; then discard registers
  - In practice: cache and registers, avoid main memory exposure of key

(*) In fact, this works fairly well for any small pieces of data

Memory encryption
Boojum (Schneier & Ferguson)

- Writing to encrypted memory – similar
  - But partial write may require a read-modify-write operation
- All in all, extremely difficult with trusted registers & cache only
  - Amount of data that can be processed is quite minimal with registers and hardware cache architecture not always suitable (may leak to memory)
  - OS limitations, register shortage, etc makes this infeasible in the end
- Alternative – trust the memory for a short period
  - Keep recently accessed data in plaintext form, but keep other data encrypted
  - Avoid burn-in of plaintext data
    - Through Boojum
    - Through limited exposure time (< 1 sec)
    - Move plaintext data around, erasing after each move
  - On system shutdown, wipe all critical data

Memory encryption

- Any solution is dependent on memory technology
  - Approaches such as these are heuristic
    - A hostile memory chip can (easily) overcome them
  - If memory vendor changes memory technology, vulnerability to persistence effects might change
    - This may happen after deployment
    - May even be dependent on RAM production batch
  - Can be solved using specialized secure memory components
    - But using cheap hardware is very attractive in practice
    - **Modern PC hardware: Trusted Platform Module (TPM): https://www.trustedcomputinggroup.org/**
- Memory encryption may seem an extreme measure
  - But if we're using strong cryptography, we should be quite paranoid
    - E.g. loss of a corporate root CA private key may be a problem
  - Remember – a cryptographic system should be balanced
  - Ultimately relevance of each threat depends on the threat model and assets

Implementing memory erasure

- Erasure takes time relative to burn-in period
  - If a certain bit patterns sits for days, it cannot be erased quickly
  - This is the case especially with DRAMs, where “ion migration” effect happens when fixed data is stored
  - High temperature makes the effect quicker, but not under SW control
- Erasure is dependent on memory technology
  - Even different RAM batches have different characteristics
  - Thus outside your control, and difficult to do properly
- Best approach is to avoid the need to erase
  - Memory encryption
  - Boojum and/or moving data around in memory
  - Securely handled memory could be provided by OS, but most OSs don't provide it

Magnetic storage

- Easiest approach to storage is to just encrypt it all
  - Filesystems, virtual memory swap files, etc.
  - More trusted RAM can be used for encryption and decryption processing
- If encryption not feasible, proper erasure is the key
  - Magnetic media is permanent storage so data will stick after power-off
- Proper erasure is not easy
  - Overwriting with various bit patterns may not be fully reliable
    - E.g. modern hard drives and block replacement; error correction
    - Several layers of data used to be recoverable, but it is unclear what the situation is today
  - Strong magnetic fields not fully sufficient
    - Required magnetic field for modern storage may damage the media, making it useless (equivalent to destruction)
  - Most reliable method => physical destruction

Cryptographic libraries

Typical crypto library services

- Very low level services
  - Memory management & other platform dependencies
    - Locking, zeroing, hardware memory interfaces
  - Random number generation
    - “True randomness”, entropy gathering
    - Cryptographic grade PRNG
  - Big number computation
    - Maintaining security of big numbers
- Low level services
  - Symmetric primitives
    - Hashing, encryption (+ padding modes), MAC
  - Asymmetric primitives
    - Public key encryption and signatures (+ padding modes)
    - Diffie-Hellman

Typical crypto library services

- Middle level services
  - File format import and export (e.g. RSA keys)
  - PKI support – certificate parsing, etc.
- High level services
  - SSL/TLS encrypted sockets
  - Other protocol specific services

OpenSSL

- Background
  - Based on earlier SSLeay crypto library (Eric A. Young)
  - Full feature open-source library for SSL/TLS and generic cryptography
  - http://www.openssl.org/
- Implemented in C and assembler
  - No type safety, no automatic resource management
  - => Fragile and prone to leaking, but good performance
    (for evidence of fragility, read through the change logs :-)
- Several security issues have been found in OpenSSL
  - OpenSSL in active use, so many bugs naturally found
  - Even with its bugs, it's (probably) the best open source available
- In wide use, both open source and commercial software

OpenSSL – libcrypto

- Symmetric ciphers
  - Blowfish, CAST, AES, DES/3DES, IDEA, RC2, RC4, RC5
- Public key cryptography and key agreement
  - DSA, Diffie-Hellman, RSA
- Certificates
  - X.509 (ASN.1)
- Hash functions and message auth. codes (MACs)
  - HMAC, MD2, MD4, MD5, MDC2, RIPEMD-160, SHA-*
- Miscellaneous
  - Cryptographic random numbers, big number arithmetic
  - ASN.1 encoding and decoding, PEM, PKCS #7, PKCS #12
  - Abstractions for I/O and cryptographic primitives

OpenSSL – libssl

- Protocol support
  - SSLv2 and SSLv3
  - TLSv1
- Basic paradigm
  - Establish a session context (SSL_CTX)
  - Open a connection and hand it over to the context
  - Event loop uses SSL_read() and SSL_write() for I/O
- The API is a bit complicated
  - Some help in Viega et al: *Network Security with OpenSSL*

Other crypto libraries

- RSA BSAFE
  - Commercial crypto library, C / C++ / Java, FIPS 140 cert
  - Crypto, Cert, SSL, Secure WS (web services)
- MatrixSSL
  - Small footprint SSL crypto library (about 50 kiB) (GPL + commercial)
- Java SSL libraries
  - ssllib, EspreSSL, ...
- libgcrypt
  - Generic crypto library, based on GnuPG
- Tons of Java crypto providers

Big number arithmetic

- Big number arithmetic makes code more complex
  - Programming languages don't often have native big numbers
  - Big numbers are typically implemented as allocated objects
    - Resources need to be managed more carefully to avoid leaks
  - Individual big number operations may fail
    - Typically due to lack of resources
    - Computation failures don't happen with normal integers
  - Optimizations complicate algorithms a great deal
    - Example: optimized exponentiation (Montgomery), optimized RSA

Bad big numbers

- What if big number arithmetic implementation is bad ?
  - Big number arithmetic implemented in terms of fixed size operations on dynamically sized data (e.g. arrays of 32-bit values)
  - Corner cases easy to get wrong – and hard to test
    - For instance, may occur once in 2^{32} or 2^{64} ...
  - Note that similar problem may occur with memory corruption
- Example of RSA private key leak on next slide (w/ CRT)
  - Chinese Remainder Theorem (CRT) can speed up RSA
  - Keep **p** and **q** after key generation, compute **mod p** and **mod q**, then combine using CRT to get **mod pq**
  - We have chosen small encryption exponent, **e=3**
    - Quite common because it optimizes public key operations
  - (Example from *Practical Cryptography*)

RSA private key leak example

- Alice is signing **m** to produce $\mathbf{m^d = s \pmod n}$
  - $\mathbf{m^d = s}$ is the correct signature we want; Bob receives $\mathbf{(m, m^d)}$
- A bug in big number arithmetic occurs during CRT
  - Produces result $\mathbf{s + kq \pmod n}$
- Bob exploits the flaw
  - Bob knows **m** (plaintext) which is equal to $\mathbf{m^{ed} = s^3}$ (3 = pub key)
  - Bob computes:
    $\mathbf{(s + kq)^3 - s^3 = \underline{s^3} + 3s^2kq + 3sk^2q^2 + k^3q^3 - \underline{s^3} =}$
    $\mathbf{q\,[\,3s^2k + 3sk^2q + k^3q^2\,] = tq}$ (where **t** is unknown)
  - Attacker computes greates common divisor of **tq** and **n=pq**
  - **p** and **t** have no common factors => attacker gets **q => p = n/q**
  - Given **p** and **q** it's trivial to compute **d**, the private key
- Can be avoided by not using CRT (not keeping **p** and **q**)...

“Wooping” for big numbers

- Some solutions
  - Just trust the library => non-critical applications
  - Use two different libraries and compare => expensive
  - “Wooping” (Jurjen Bos, *Practical Privacy*, Ph.D. thesis)
- “Wooping” in short
  - Generate small, cryptographically random prime, **t**, e.g. 64-128 bits
    - **t** is strictly local, and kept secret from **all parties**
  - For every big number **x**, also track **x' = x (mod t)**
    - (**x'** is the “woop” of **x**)
  - Shadow all big number computations with woops
    - **a = b + c** => **a' = b' + c' (mod t)**
    - More complicated rules for modular addition and multiplication
  - In the end, check results; e.g. result **x** => check **x (mod t) = x'**

“Wooping” for big numbers

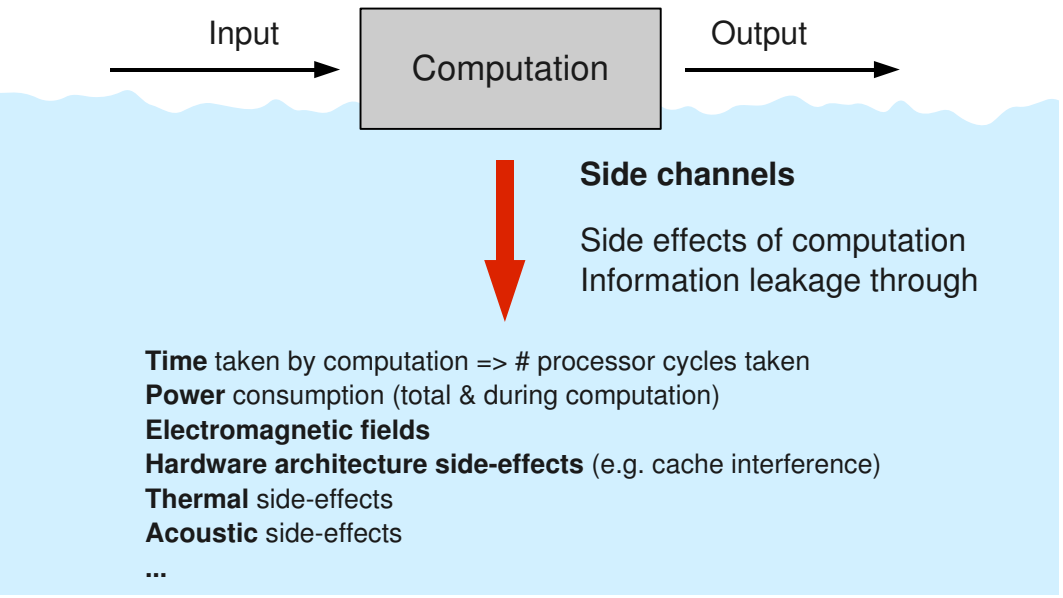
- Why does this help ?
  - If crypto library fails due to a bug, it's unlikely the woop will match
  - Even if the crypto library is malicious, it doesn't know **t**, so it's difficult to fake results (**t** changes on every initialization, e.g. host bootup)
  - Occasional RAM corruption can also be detected
    - Bad memory, bus problems, cosmic rays, ...
- Performance hit ?
  - If computing using big integers (1024+ bits), impact quite small
  - If using crypto hardware, wooping can be done in software (in parallel)
- Is “wooping” used in practice ?
  - Not widely, reason?
  - Added complexity, performance, risk of computation failures too small?

Other cryptographic self-tests

- Test vectors can be used run-time (in addition to testing)
  - Execute a test vector suite for each algorithm during initialization
  - Typically a smaller set than during testing
    - Randomized inputs with lots of iteration can be used during testing
    - Run-time tests must be short and effective
- Motivation
  - Catching compile errors
    - Porting or upgrading a compiler; may compile OK but break
  - Catching load errors
    - Corrupted bits from hard drive / bus
  - Catching linking errors
    - Dynamic library upgraded, binary compatibility lost but linking OK for some reason

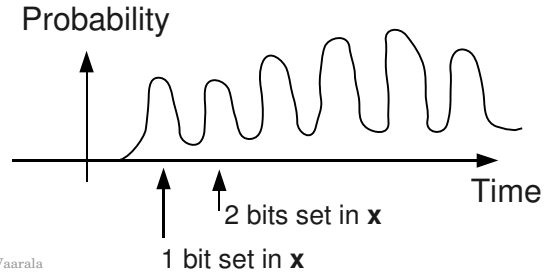
Side channel attacks

Side channels



Idealized timing attack example

- Diffie-Hellman g^x using optimized modular exponentiation
  - Information about value **x** should never be leaked
  - The # of big-number multiplications in simple modular exponentation is correlated with number of **1**-bits in **x (we hope)**
  - Example: $x = 1000101_b$ => n + 3 multiplications
- If you observe a number of Diffie-Hellman operations, g^x timing will have a non-uniform probability distribution



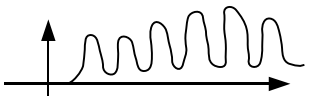
"Side channel" attacks

- What are "side channel" attacks ?
  - Factors other than actual computation (inputs and outputs)
  - Time taken to compute, power drawn during computation, electromagnetic radiation, etc.
- Side channels can leak considerable information
  - Leaking a few hundred bits can be catastrophic if those bits happen to be the encryption key
- Difficult to eliminate
  - Timing etc. often not considered in APIs and other abstractions
  - Example – CPU cache timing leaks information, but is very specific to the processor in question
- Specific techniques needed to avoid side channel attacks
  - Especially relevant to cryptography, of course

Idealized timing attack example

- When you observe one particular D-H, measure g^x time
  - The time will most likely fall within one of the "peaks" of the distribution
  - You can then identify the number if **1**-bits in **x** with some confidence
- How much information is gained ?
  - Assume **x** has **128** bits, and **68** are known to be **1**
    - Original brute force requirement = 2^{128}
    - New brute force requirement = **128! / (68! * (128-68)!) = $2^{123.8}$**
    - Attacker gains **4.2** bits
  - If **37** bits are known to be **1**
    - Original brute force requirement = 2^{128}
    - New brute force requirement = **128! / (37! * (128-37)!) = $2^{107.4}$**
    - Attacker gains **20.6** bits

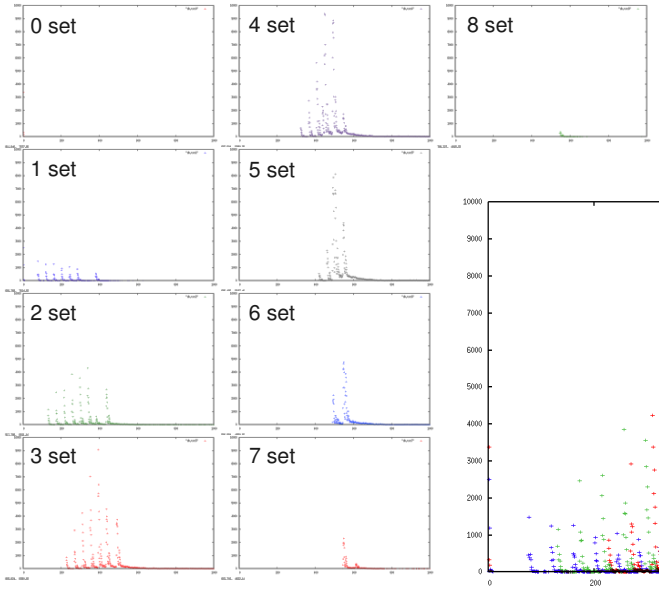
```
Google compute:
(ln (128! / ((68!) * (60!))) / (ln 2) =
(ln (128! / ((37!) * (91!))) / (ln 2) =
```



Actual attacks are more complicated

- Not so easy in practice – distortions in timing include
  - Other data dependencies besides # of bits set
  - Interference from other processes competing for CPU
  - Dynamic compilation (e.g. in Java)
- Let's look at a toy example
  - Diffie-Hellman in a 1024-bit group (IKE group 2, also in TLS)
  - Limit exponent size to 8 bits (with 8th bit set)
    - 256 possible exponent values (256..511)
    - Exponents of the form 1xxxxxxxx binary (highest bit always set)
  - Java implementation, BigInteger
  - One million random exponents, 1 microsecond timing
- => Can we distinguish # of bits based on timing ?
  - Graphs of time taken (microsecs) vs. # of occurrences

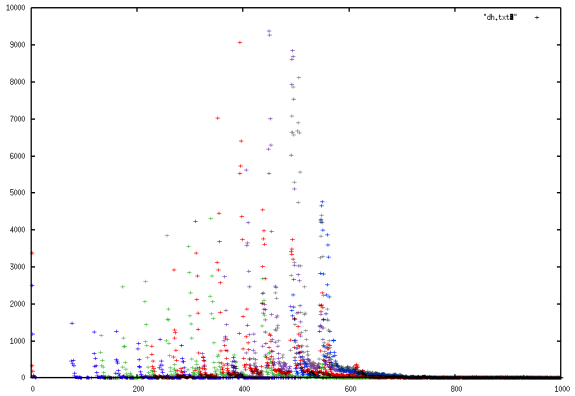
D-H timing, toy example



1024-bit D-H group (IKE group 2)
Exponent limited to 8 bits (0...255)

One million calculations with random exponent, background process noise

Timer resolution: 1 microsecond



Different timing attacks

- Toy example was black box
  - Input, output, total time
  - No information about CPU usage patterns within computation
- More dangerous attack models
  - (Remote) access to same host hardware as victim
  - Fine-grained leaks through hardware and software side-effects, e.g.
    - CPU performance – caches, TLBs, hyperthreading
    - OS process scheduling
  - Can yield bit-by-bit information
    - Example – measure CPU cache hits and misses, and find correlations between Diffie-Hellman computations
    - May allow an attacker to deduce individual bits of **x** in D-H => bad
  - Does **not** require privileges on the host, just access to CPU characteristics data (measurement)

Bernstein's AES attack

- D.J. Bernstein: *Cache-timing attacks on AES*
  - Extract AES key from network server which uses software AES
  - Comparable vulnerabilities in all modern processors
- Attack depends on features of modern CPUs
  - Memory hierarchy – constant time table lookup very difficult
  - **If table lookup indices depend on key data, timing leaks key info**
  - Particular impact on algorithms which employ S-boxes
  - Note that most arithmetic operations are constant time on modern CPUs, but there are exceptions (ARM multiplication)

Bernstein's AES attack

- Simplified description of attack
  - Attacker gets a great number of fixed length plaintext/ciphertext pairs, along with accurate timing information
  - Denote **n[i]** plaintext at offset **i** and **k[i]** key at offset **i**
  - Attacker focuses on a particular offset, e.g. **n[13]** and **k[13]** ...
  - AES uses **(n[13] XOR k[13])** as a table lookup index internally
- Preliminary work
  - Encrypt a great deal of plaintext with identical (= victim) hardware
  - Measure timing for all encryptions, repetitions to eliminate noise
  - Tabulate results with **(n[13] XOR k[13])** as table key
  - Pick value of **(n[13] XOR k[13])** which yields **slowest** encryption
  - For instance, **(n[13] XOR k[13]) = 8** is slowest

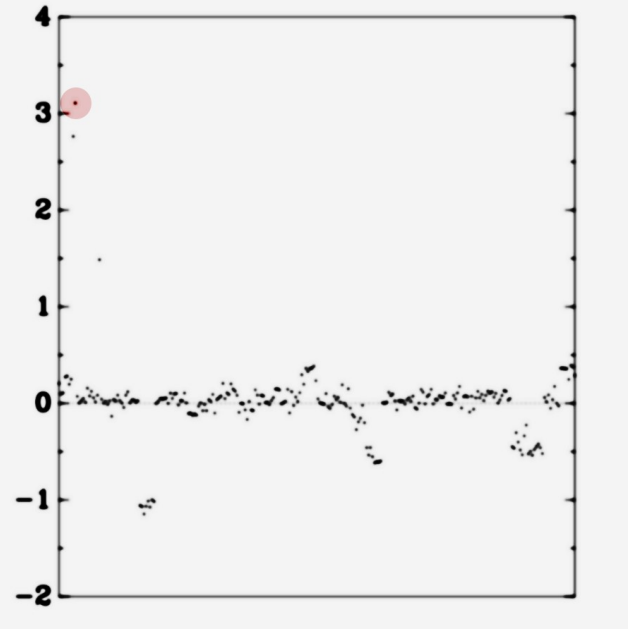
Bernstein's AES attack

- Actual attack
  - Cause victim to encrypt a great deal of plaintext of fixed length
  - Measure timing for all encryptions, repetitions to eliminate noise
  - Tabulate results with **n[13]** as table key, pick slowest, e.g. **147**
    - This eliminates the effect of similar timing effects on other byte positions statistically
  - We now hope that for slowest **n[13]** we have

    k[13] XOR n[13] = 8 (as this was supposed to be slowest)

    => k[13] = 8 XOR n[13] = 8 XOR 147 = 155
  - **=> We extract key byte (8 bits) at offset 13**
  - Repeat for all key offsets

Bernstein's timing example



X-axis

n[13] XOR k[13]
(0...255 values)

Y-axis

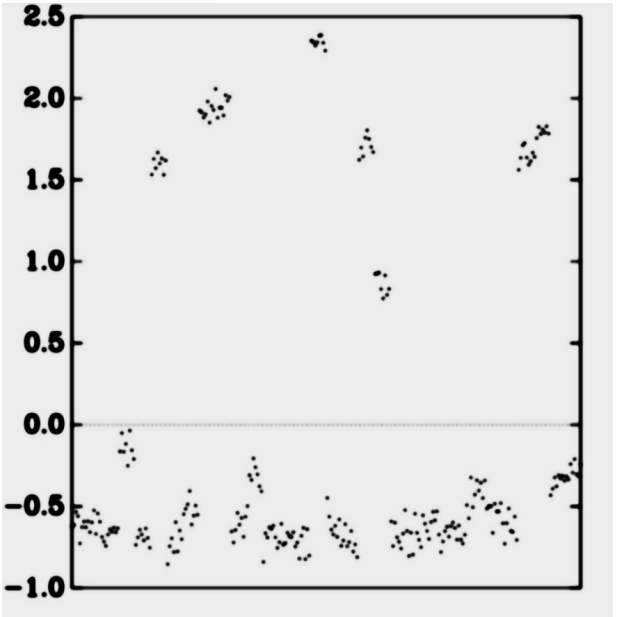
Deviation from average time, in processor cycles (here, Pentium III)

Result

n[13] XOR k[13] = 8 yields longest exec. time – **3.108 cycles** above average

Figure from Bernstein's paper *Cache-timing attacks on AES*, post-processed for slides

Bernstein's timing example



X-axis

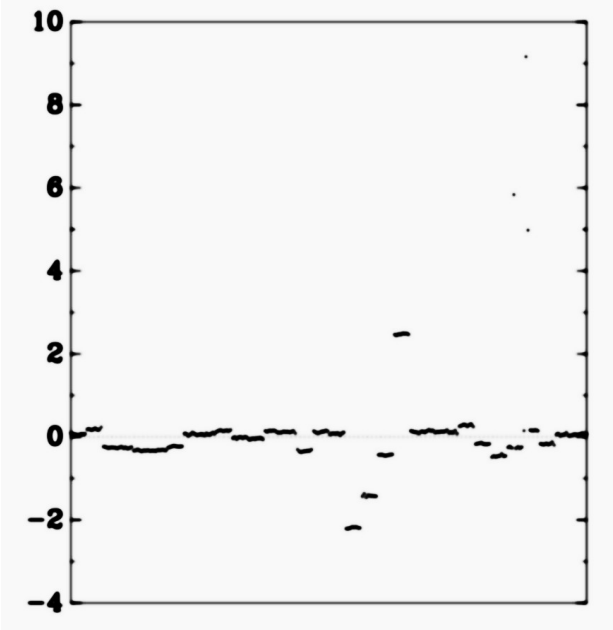
n[5] XOR k[5]
(0...255 values)

Y-axis

Deviation from average time, in processor cycles (here, Pentium III)

Figure from Bernstein's paper *Cache-timing attacks on AES*, post-processed for slides

Bernstein's timing example



X-axis
n[0] XOR k[0]
(0...255 values)

Y-axis
Deviation from average time, in processor cycles (here, Pentium III)

Figure from Bernstein's paper *Cache-timing attacks on AES*, post-processed for slides

A few observations

- Key weakness in software AES
  - Table lookup time is **key dependent**
- Bernstein's attack is not a full real world attack
  - E.g. cycle-accurate timing provided by the server itself
  - But it may still be that similar attacks are possible against real systems
- Timing attacks are very relevant threats
  - Several attacks on asymmetric cryptosystems e.g. smart cards
  - Attacks against symmetric cryptosystems possible, e.g. Bernstein
- Constant-time implementations are secure
  - Very difficult to do in practice
  - Potentially very inefficient
  - May favor hardware over software for critical crypto

Countering timing attacks

- Analyze relevance of timing attacks
  - If unsure, assume timing attacks are relevant (motivated attackers)
  - Cryptosystem is as weak as its weakest link
- Crypto primitive selection
  - Select cryptographic primitives which allow **efficient** constant-time implementation (Bernstein: const time AES is inefficient)
  - This is more difficult than seems at first, see Bernstein's paper
- Crypto libraries
  - Check that used crypto libraries are protected against timing attack
- Other considerations
  - If no constant time primitives, hide timing differences externally
    - Standardize execution time, delay if faster
    - Round execution time to next **N** ms, delay (reduces leak)

Covert channels

- Data transmission through non-obvious channels
  - Bypass security policy (e.g. multi-level security problem)
  - Side channel can be used as a covert channel
  - Attacker leaks keys through a low bandwidth channel and avoids detection
- Example scenario
  - Organization buys crypto equipment from vendor X, and observes that the equipment follows encryption standard
  - Crypto equipment leaks active crypto session keys through e.g. a timing side channel
  - Vendor X is able to wiretap equipment & sell decryption service

Covert channels

- Problems
  - Detection – fatal leakage can be small in crypto (< 1 bit / sec)
  - Prevention – eliminating all side channels is almost impossible
- White box software gives some assurance

Summary

- Secure handling of storage media
- Cryptographic libraries
- Side and covert channels