17-654: Analysis of Software Systems

Spring 2005

4/21/2005

Topics

- Timing attack
  - Algorithms leak information
  - Nice example of practice trumping theoretical security
  - Hardening algorithms: randomization
- Privilege separation
  - Hardening software: principle of least privilege

Remote Timing Attacks are Practical

with Dan Boneh

Side channel analysis

- Side channel = unintentional leak of information
- Attackers learns secrets by observing *normal* program behavior
  - power
  - noise
  - timing information
- Powerful and realistic approach to breaking crypto

Overview

- Main result: RSA in OpenSSL 0.9.7 is vulnerable to a new timing attack:
  - Attacker can extract RSA private key by measuring web server response time.

- Exploiting OpenSSL's timing vulnerability:
  - One process can extract keys from another.
  - Insecure VM can attack secure VM.
    - Breaks VM isolation.
  - Extract web server key remotely.
    - Our attack works across campus

Why are timing attacks against OpenSSL interesting?

- Many OpenSSL Applications
  - mod_SSL (Apache+mod_SSL has 28% of HTTPS market)
  - stunnel (Secure TCP/IP servers)
  - sNFS (Secure NFS)
  - bind (name service)
  - Many more.
- Timing attacks previously applied to smartcards [K'96]
  - Never applied to complex systems.
  - Most crypto libraries do not defend:
    - libgcrypt, cryptlib, ...
    - Mozilla NSS only one we found to explicitly defend by default.
- OpenSSL uses well-known optimized algorithms

Outline

- **RSA Overview and data dependencies**
- Present timing attack
- Results against OpenSSL 0.9.7
- Defenses

RSA Algorithm

- N is a public modulus. Let N = p*q
  - p,q 512-bit prime numbers
- Let e*d = 1 mod (p-1)(q-1)
  - e is public encryption exponent
  - d is private decryption exponent
- Encryption: m^e mod N = c
- Decryption: c^d mod N = m^{ed} mod N = m mod N
- Secrets: d, p ,q.

RSA & CRT

- RSA decryption: g^d mod N = m
  - d & g are 512 bits

- Chinese remaindering (CRT) uses factors directly. N=pq, and d1 and d2 are pre-computed from d:
  1. $m1 = g^{d1}$ mod q
  2. $m2 = g^{d2}$ mod p
  3. combine m1 and m2 to yield m (mod N)

- CRT gives 4x speedup

- Goal: learn factors (p,q) of N.
  - Kocher's [K'96] attack fails when CRT is used.

RSA Decryption Time Variance

- Causes for decryption time variation:
  - Which multiplication algorithm is used.
    - OpenSSL uses both basic mult. and Karatsuba mult.
  - Number of steps during a modular reduction
    - modular reduction goal: given u, compute u mod q
    - Occasional extra steps in OpenSSL’s reduction alg.

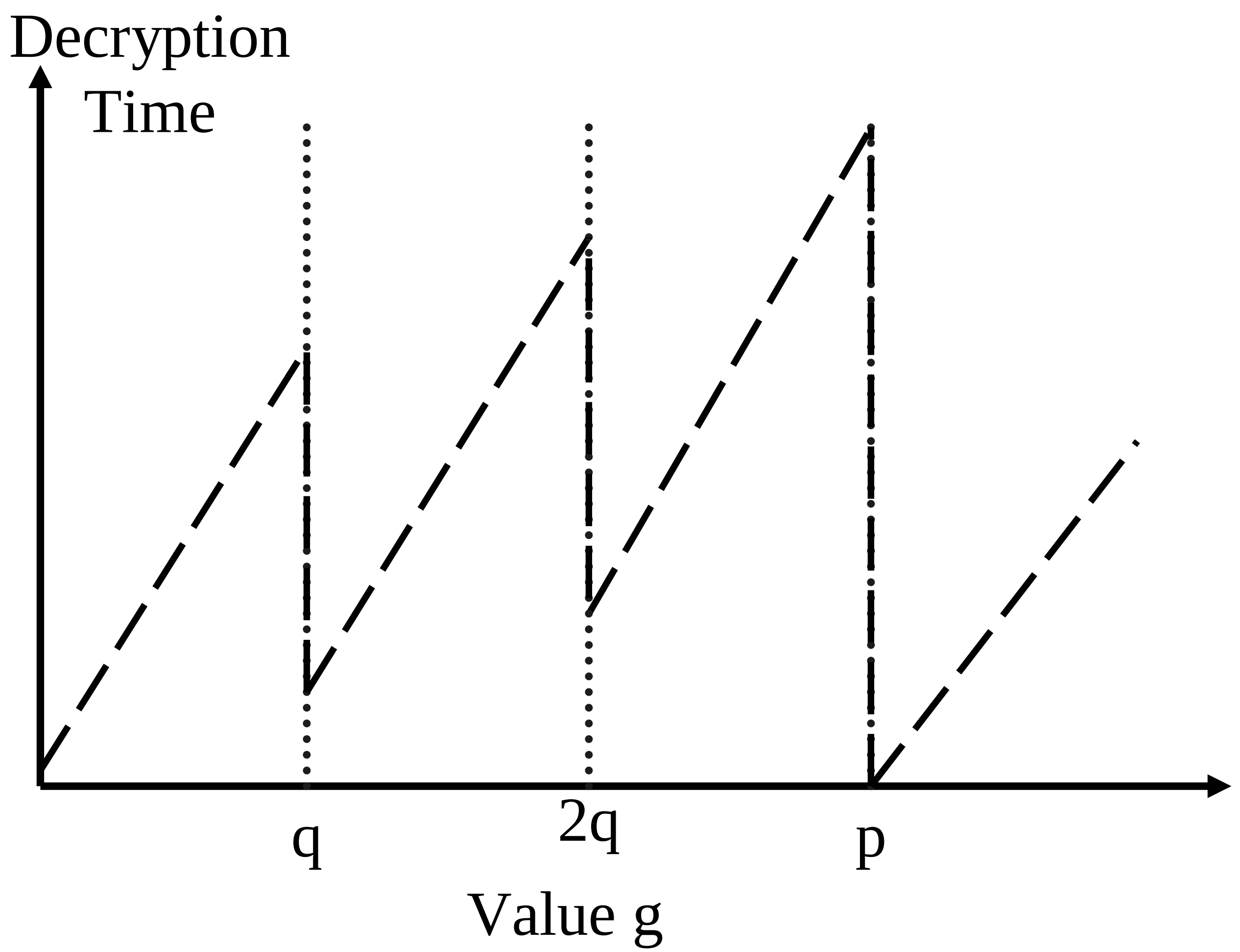
- There are MANY:
  - multiplications by input g
  - modular reductions by factor q (and p)

Reduction Timing Dependency

- Modular reduction: given u, compute u mod q.
  - OpenSSL uses Montgomery reductions [M'85] .
- Time variance in Montgomery reduction:
  - One extra step at end of reduction algorithm with probability

$$\Pr[\text{extra step}] \approx \frac{(g \bmod q)}{2q}$$ [S'00]

$$\Pr[\text{extra step}] \approx \frac{(g \bmod q)}{2q}$$

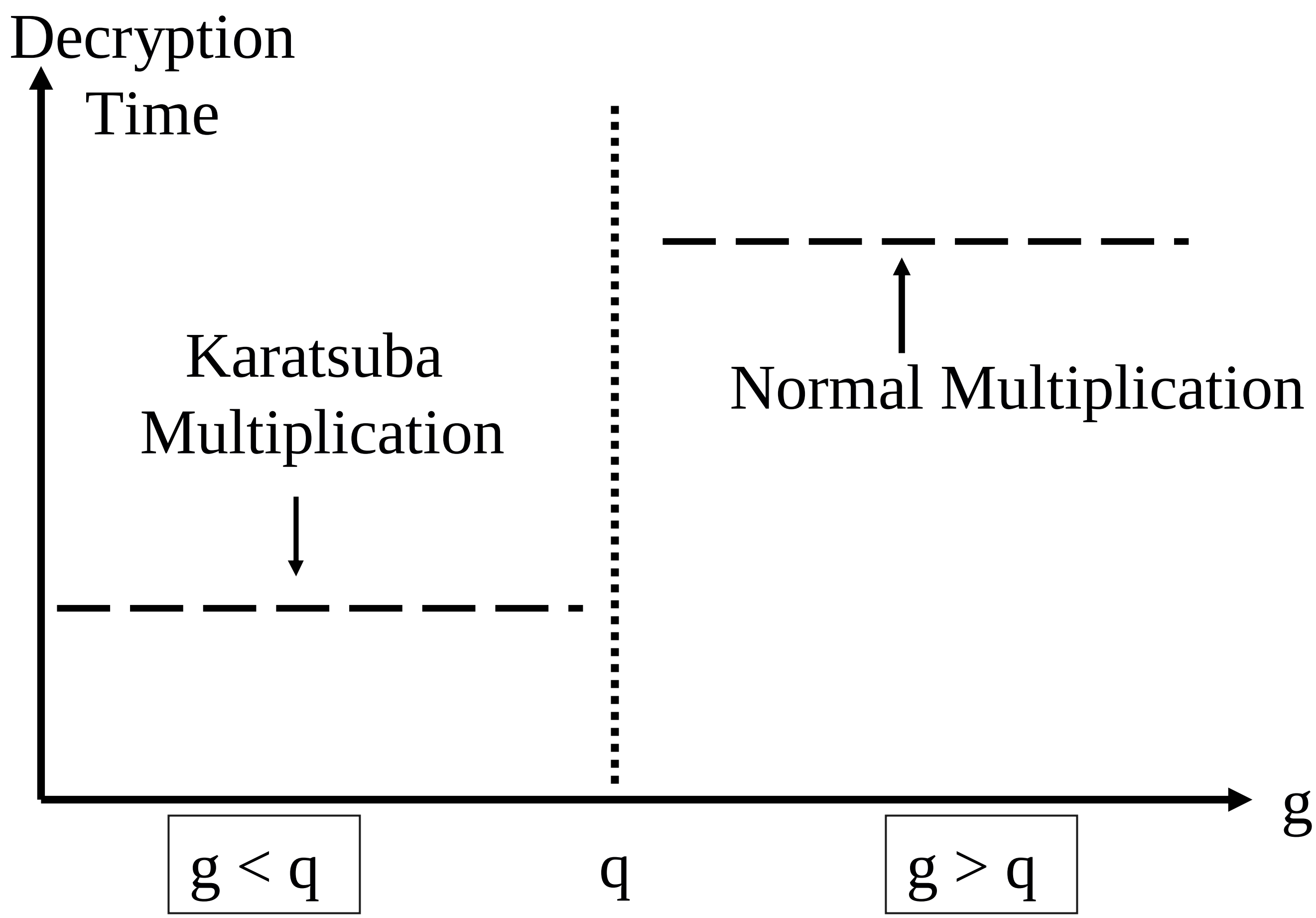


Multiplication Timing Dependency

- Two algorithms in OpenSSL:
  - Karatsuba (fast): Multiplying two numbers of equal length
  - Normal (slow): Multiplying two numbers of different length

- To calc $x \cdot g \bmod q$ OpenSSL does:
  - When x is the same length as (g mod q), use Karatsuba mult.
  - Otherwise, use Normal mult.

Multiplication Summary



Data Dependency Summary

- Decryption value g < q
  - Montgomery effect: longer decryption time
  - Multiplication effect: shorter decryption time

- Decryption value g > q
  - Montgomery effect: shorter decryption time
  - Multiplication effect: longer decryption time

Opposite effects! But one will always dominate

Previous Timing Attacks

- Kocher's attack does not apply to RSA-CRT.

- Schindler's attack does not work directly on OpenSSL for two reasons:
  - OpenSSL uses sliding windows instead of square and multiply
  - OpenSSL uses two mult. algorithms.

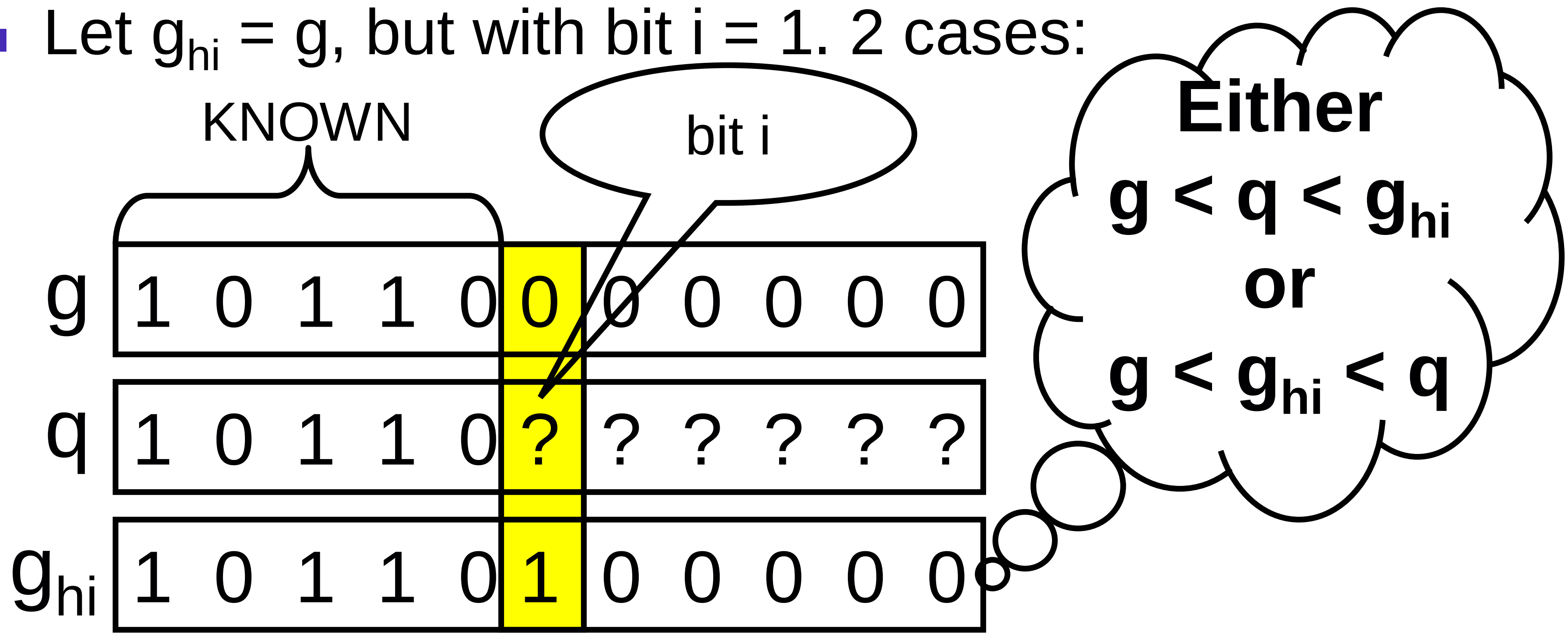
⇒ Both known timing attacks do not work on OpenSSL.

Outline

- RSA Overview and data dependencies during decryption
- **Present timing attack**
- Results against OpenSSL 0.9.7
- Defenses

Timing attack: High Level

- Suppose g = q for the top i-1 bits of q, 0 elsewhere
- Goal: Decide whether bit i = 1 or 0
- Let g_{hi} = g, but with bit i = 1. 2 cases:



Timing Attack: High Level

Goal: Decide $g < q < g_{hi}$ or $g < g_{hi} < q$

1. Sample decryption time for g and g_{hi}:

   t_1 = DecryptTime(g)

   t_2 = DecryptTime(g_{hi})

2. If $|t_1 - t_2|$ is large

   ⇒ g and g_{hi} straddle q

   ⇒ bit i is 0 ($g < q < g_{hi}$)

   else

   ⇒ bit i is 1 ($g < g_{hi} < q$)

large vs. small called ***0 -1 gap***

Timing Attack Details

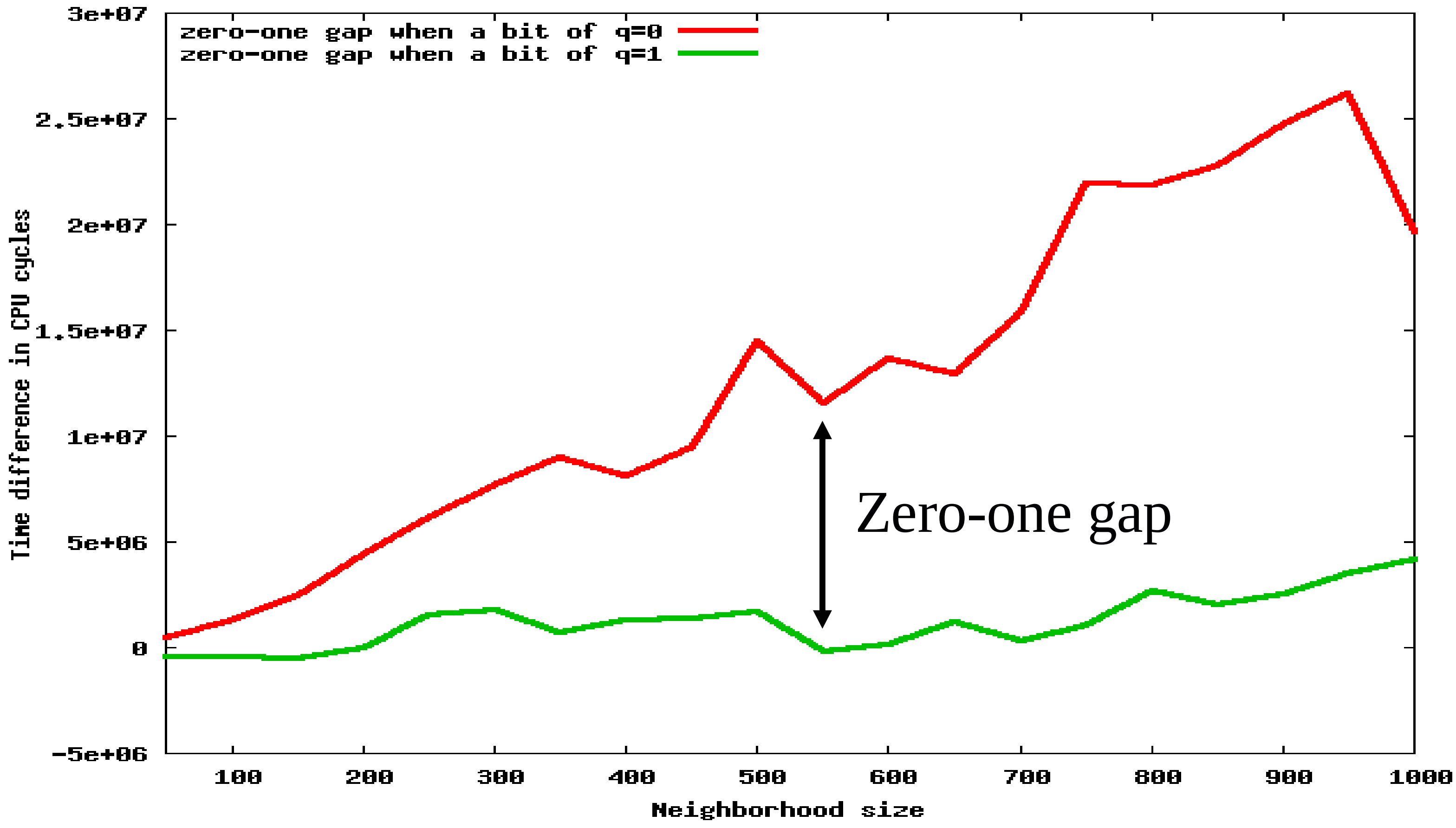
- We know what is “large” and “small” from attack on previous bits.

- Use sampling to filter noise

- Decrypting just g does not work because of sliding windows
  - Decrypt a neighborhood of values near g
  - Will increase diff. between large and small values
    $\Rightarrow$ larger 0-1 gap

- Only need to recover q/2 bits of q [C’97]

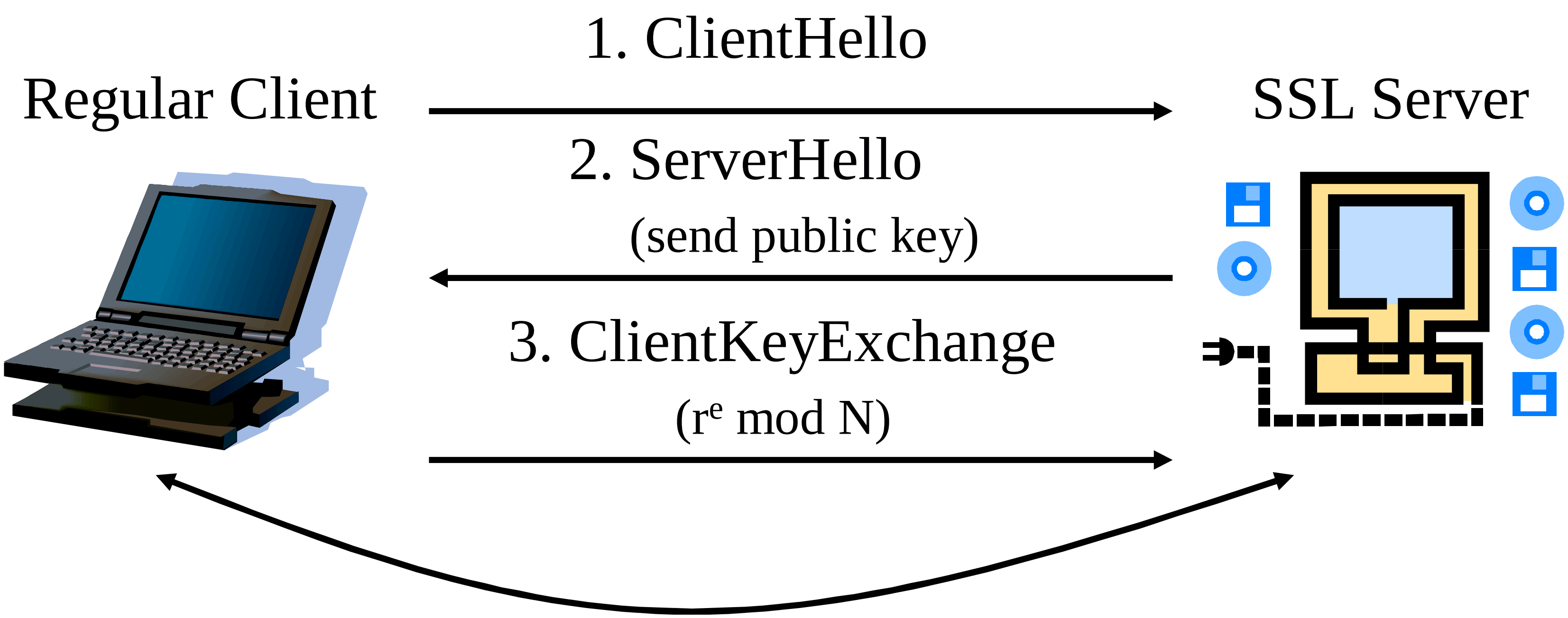
The Zero-One Gap



How does this work with SSL?

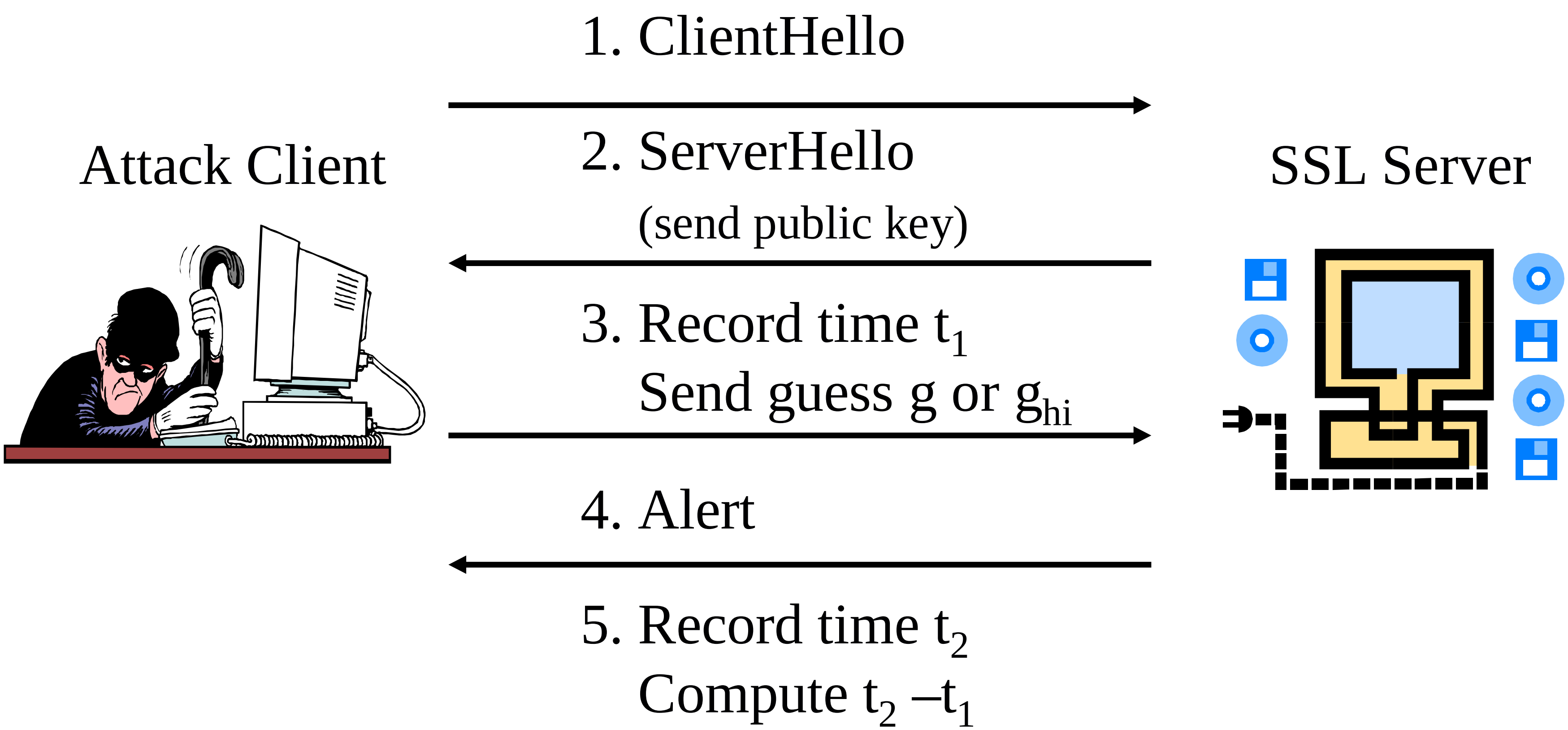
How do we get the server to decrypt our g?

Normal SSL Decryption



Result: Encrypted with computed shared master secret

Attack SSL Decryption



Attack requires accurate clock

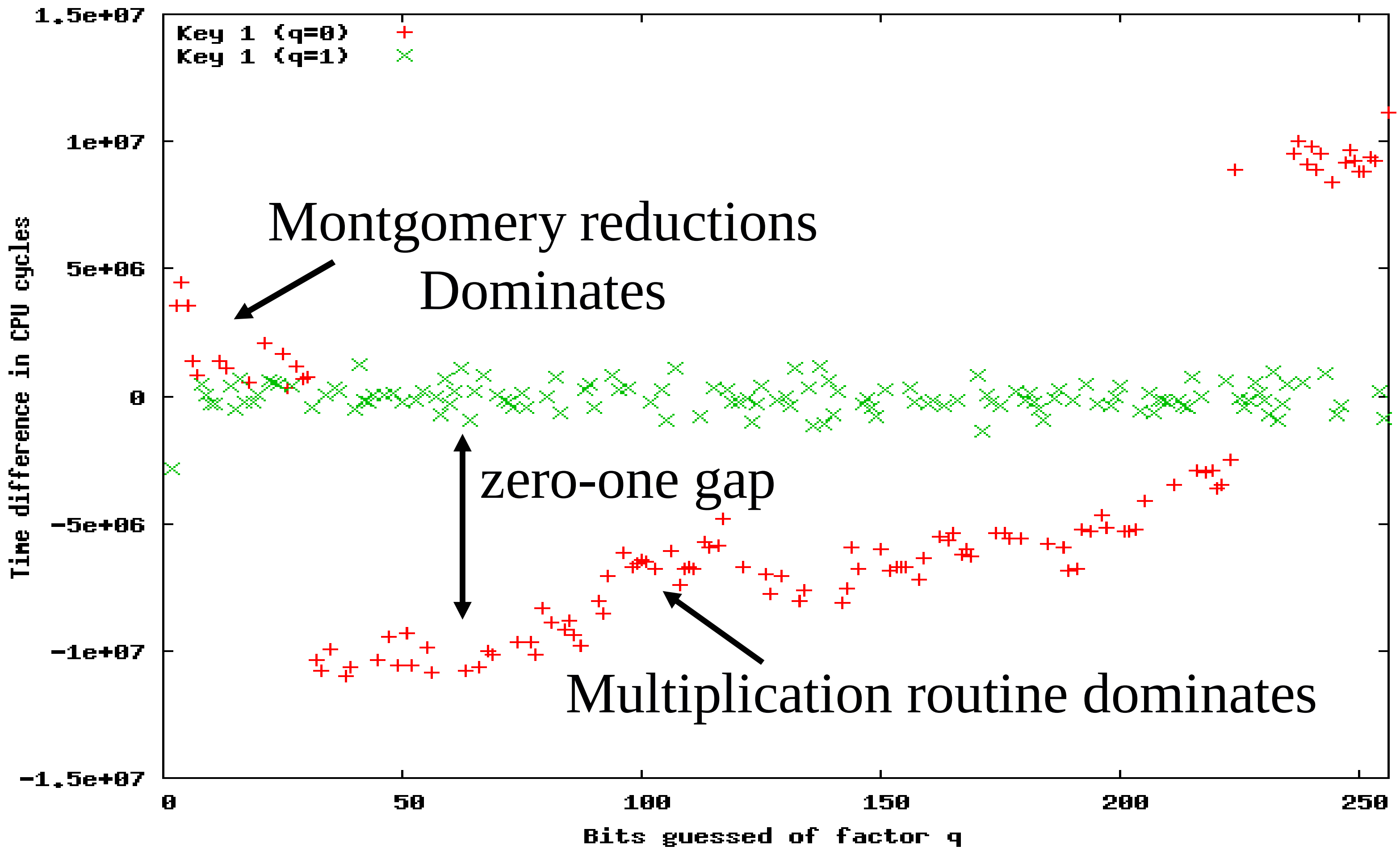
- Attack measures 0.05% time difference between g and g_{hi}
  - << 0.001 seconds on a P4

- We use the CPU cycle counter as fine-resolution clock
  - “rdtsc” instruction on Intel
  - “%tick” register on UltraSparc

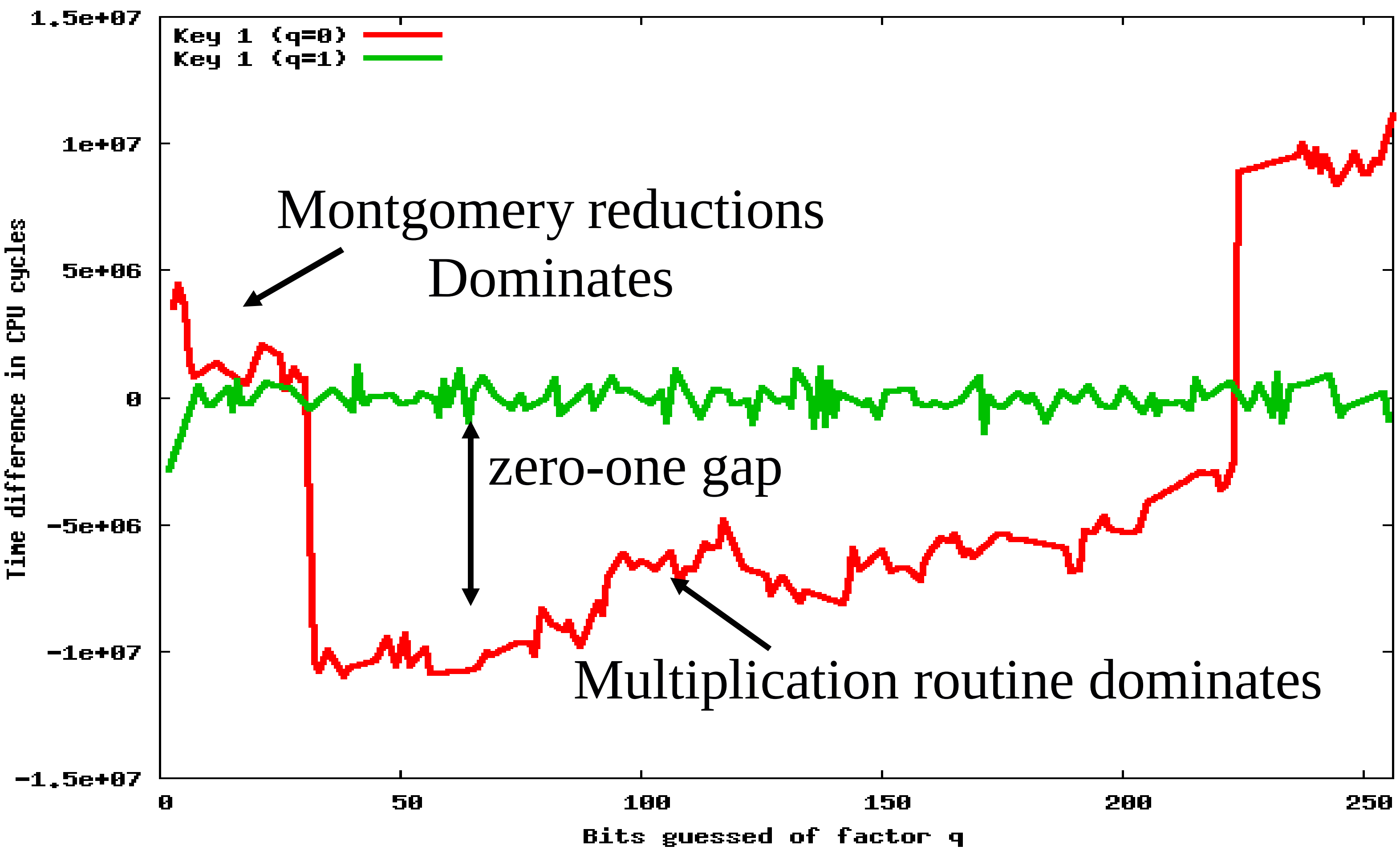
Outline

- RSA Overview and data dependencies during decryption
- Present timing attack
- **Results against OpenSSL 0.9.7**
- Defenses

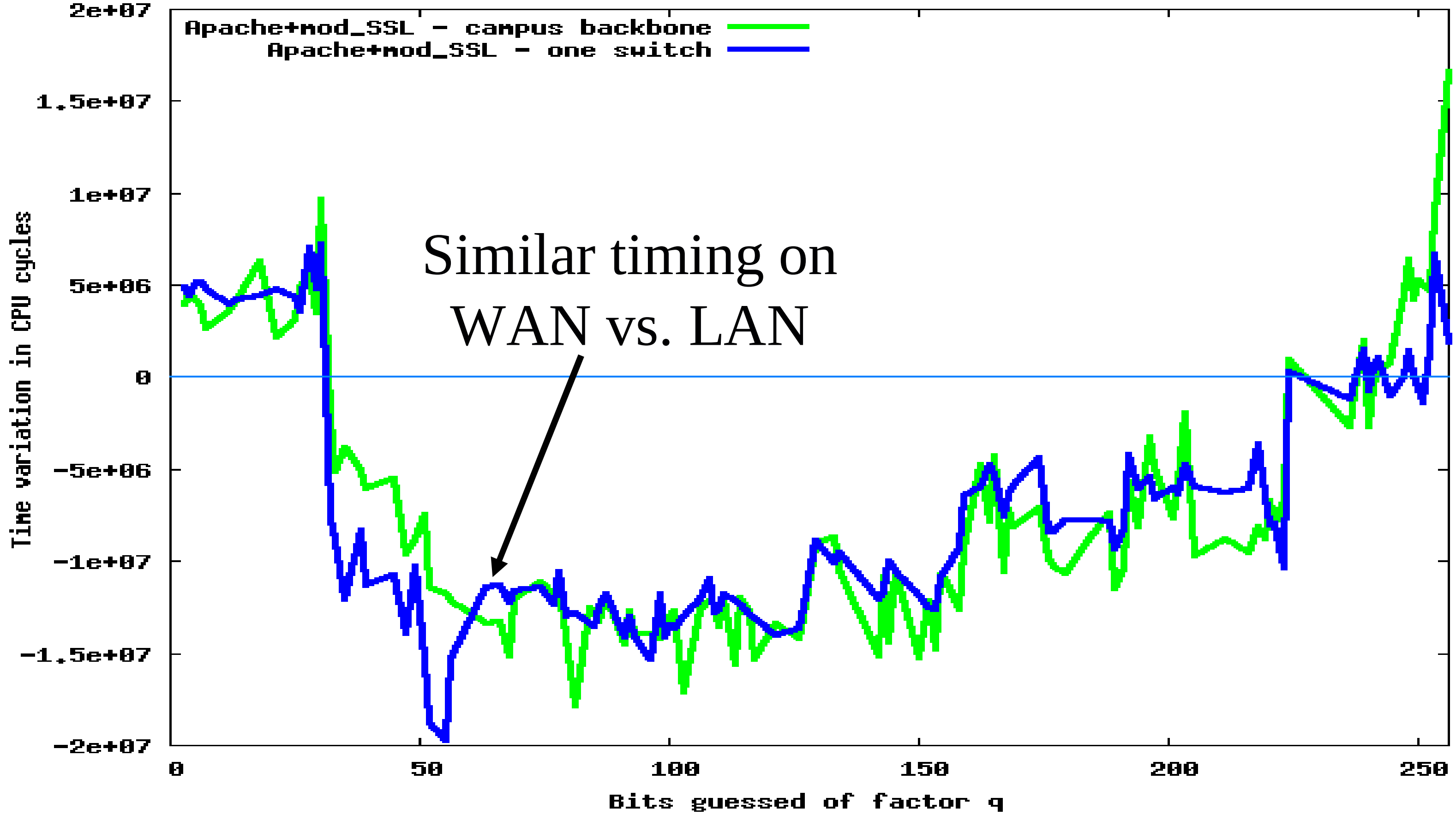
Attack extract RSA private key



Attack extract RSA private key



Attack works on the network



Attack Summary

- Attack successful, even on a WAN

- Attack requires only 350,000 – 1,400,000 decryption queries.

- Attack requires only 2 hours.

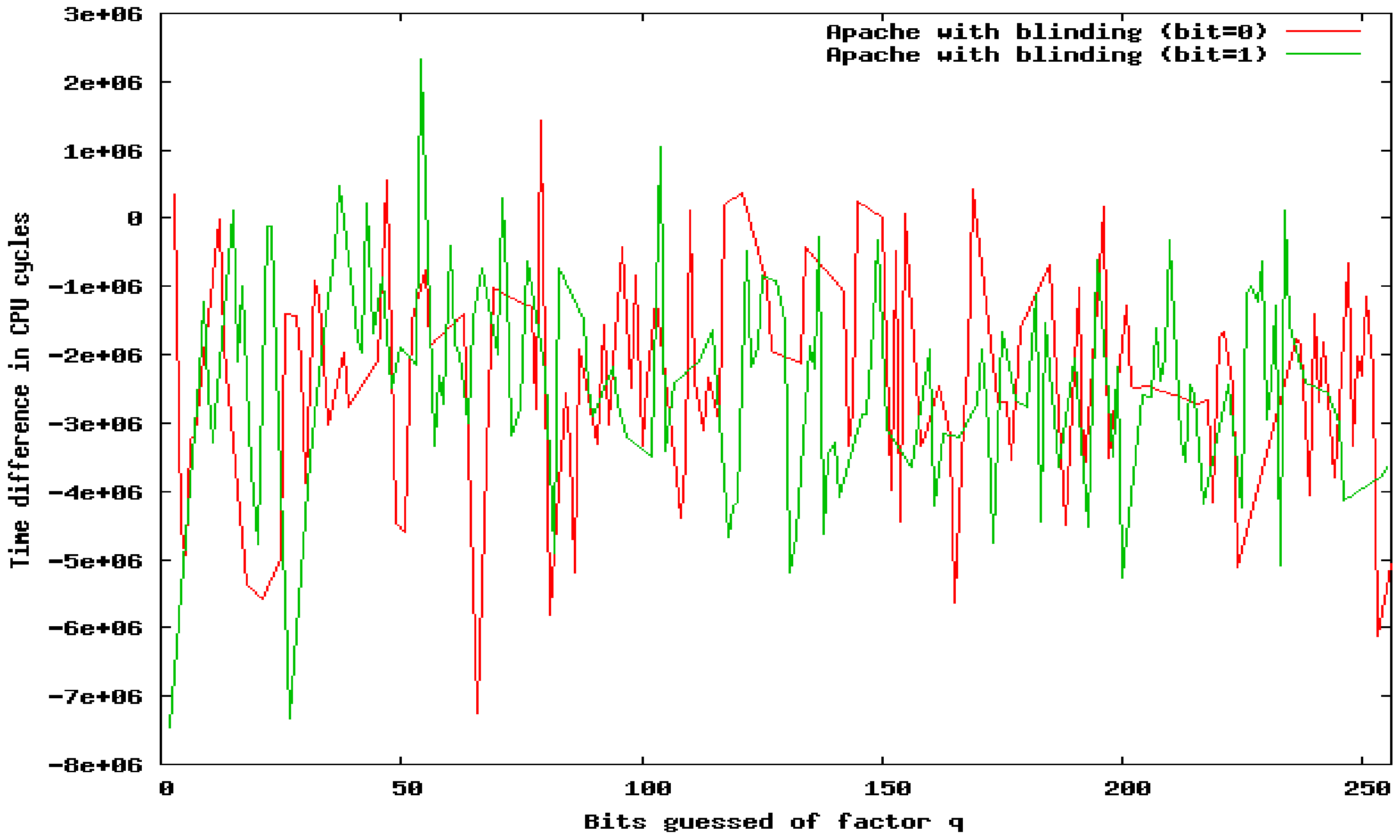
Outline

- RSA Overview and data dependencies during decryption
- Present timing attack
- Results against OpenSSL 0.9.7
- **Defenses**

Recommended Defense: RSA Blinding

- Decrypt random number related to g:
  1. Compute x' = $g*r^e$ mod N, r is random
  2. Decrypt x' = m'
  3. Calculate m = m'/r mod N
- Since r is random, the decryption time should be random
- 2-10% performance penalty

Blinding Works!



Other Defenses

- Require statically all decryptions to take the same time
  - Pros? Cons?
- Dynamically make all decryptions take the same time
  - Only release decryption answers on some interval $\triangle$
  - Pros? Cons?

Conclusion

- Attack works against real OpenSSL-based servers on regular PC’s.
- Well-known optimized algorithms can easily leak secrets
- Randomization of decryption time helps solve problem

Questions?

Privtrans: Automatically Partitioning Programs for Privilege Separation

with Dawn Song

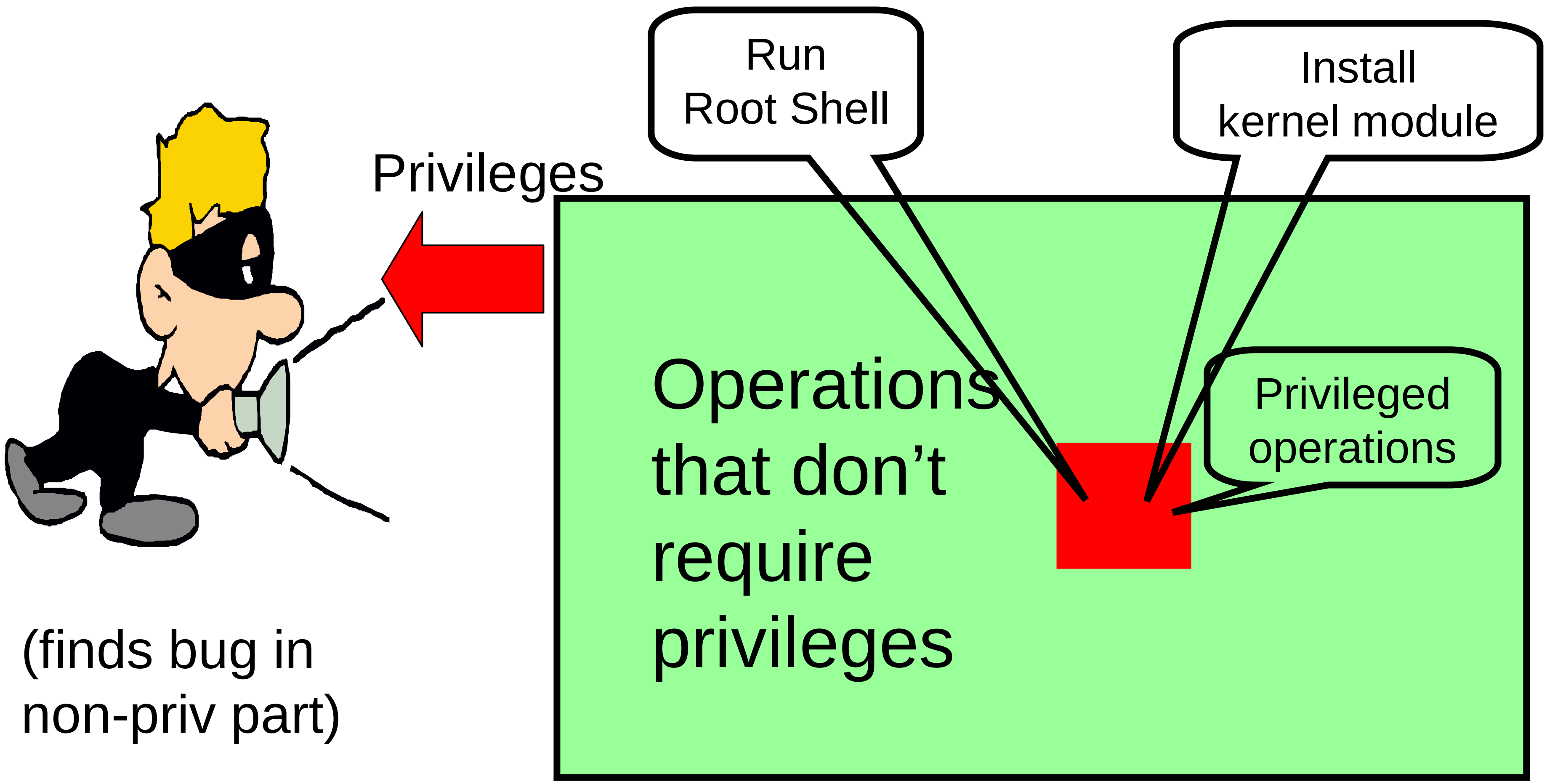
Privileged Programs

- Attackers specifically target privileged programs
  - Large number of privileged programs. Ex: network daemons, setuid(), etc.

- A Privilege may be:
  - OS privilege – Ex: opening /etc/passwd
  - Object privilege – Ex: using crypto keys

- Privileges typically needed for small part of execution

A Security Problem with Privileged C Programs



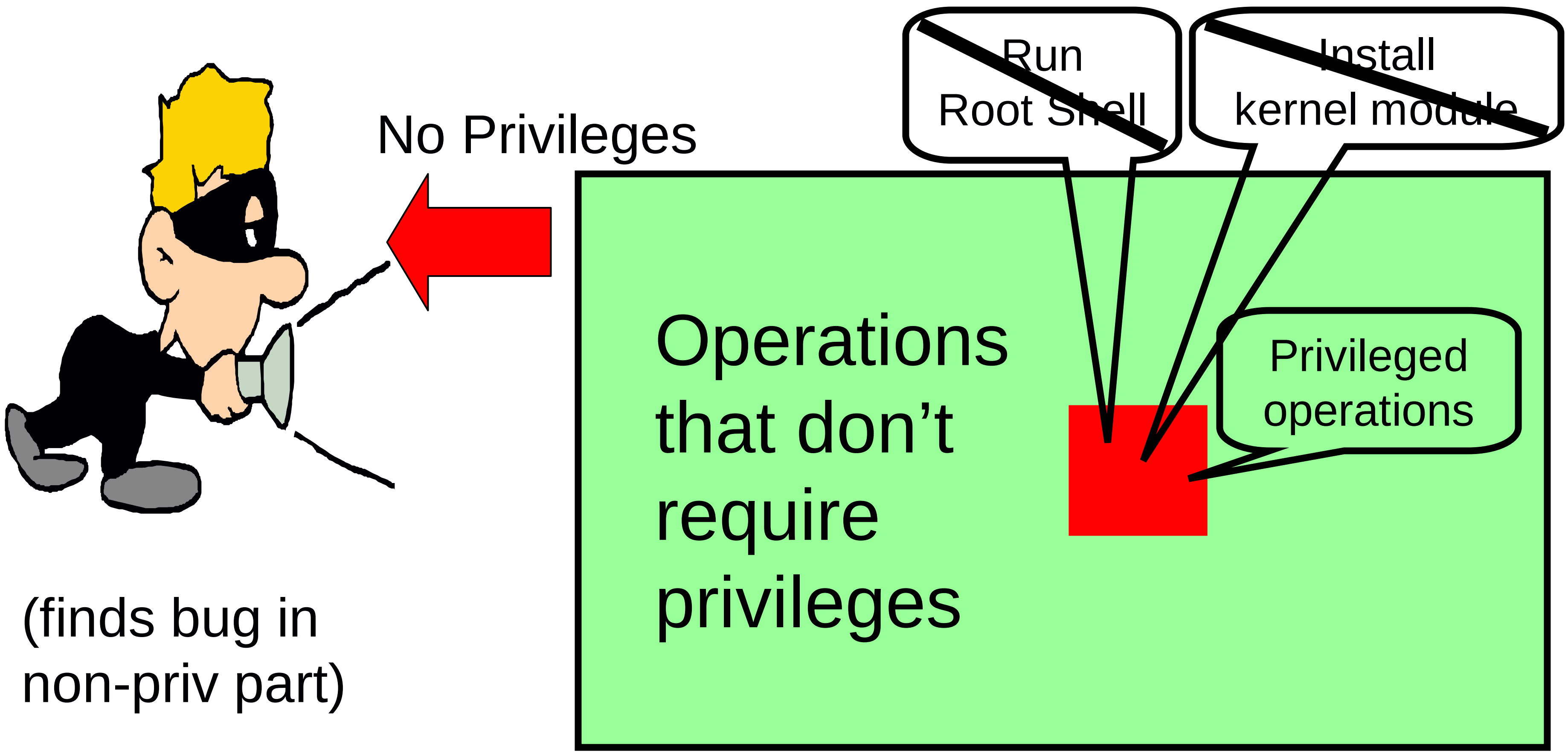
Privilege Separation

- Privilege separation **partitions** program into:
  - Privileged ***Monitor*** (usually small)
  - Unprivileged ***Slave*** (much bigger)

- Enforces principle of least privilege
  - Monitor exports **limited** interface
  - OS provides fault isolation between processes

- Previous work:
  - Privilege separation on OpenSSH [Provos et al 2003]
  - Privman---library assisting privilege separation [Kilpatrick 2003]

Enforcing least privileges
(in a nutshell)



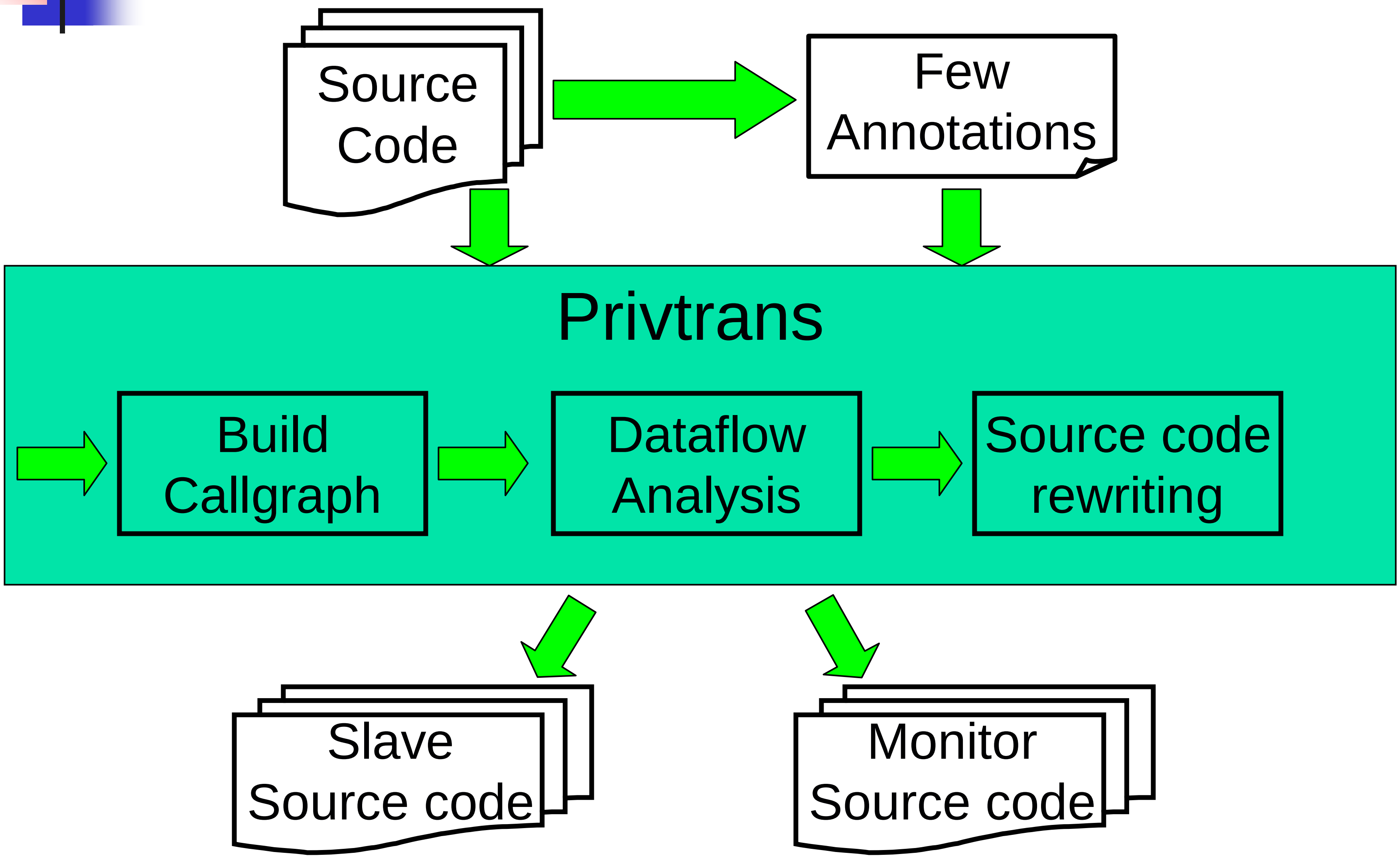
Automatic Privilege Separation

- Previous privilege separation done by hand

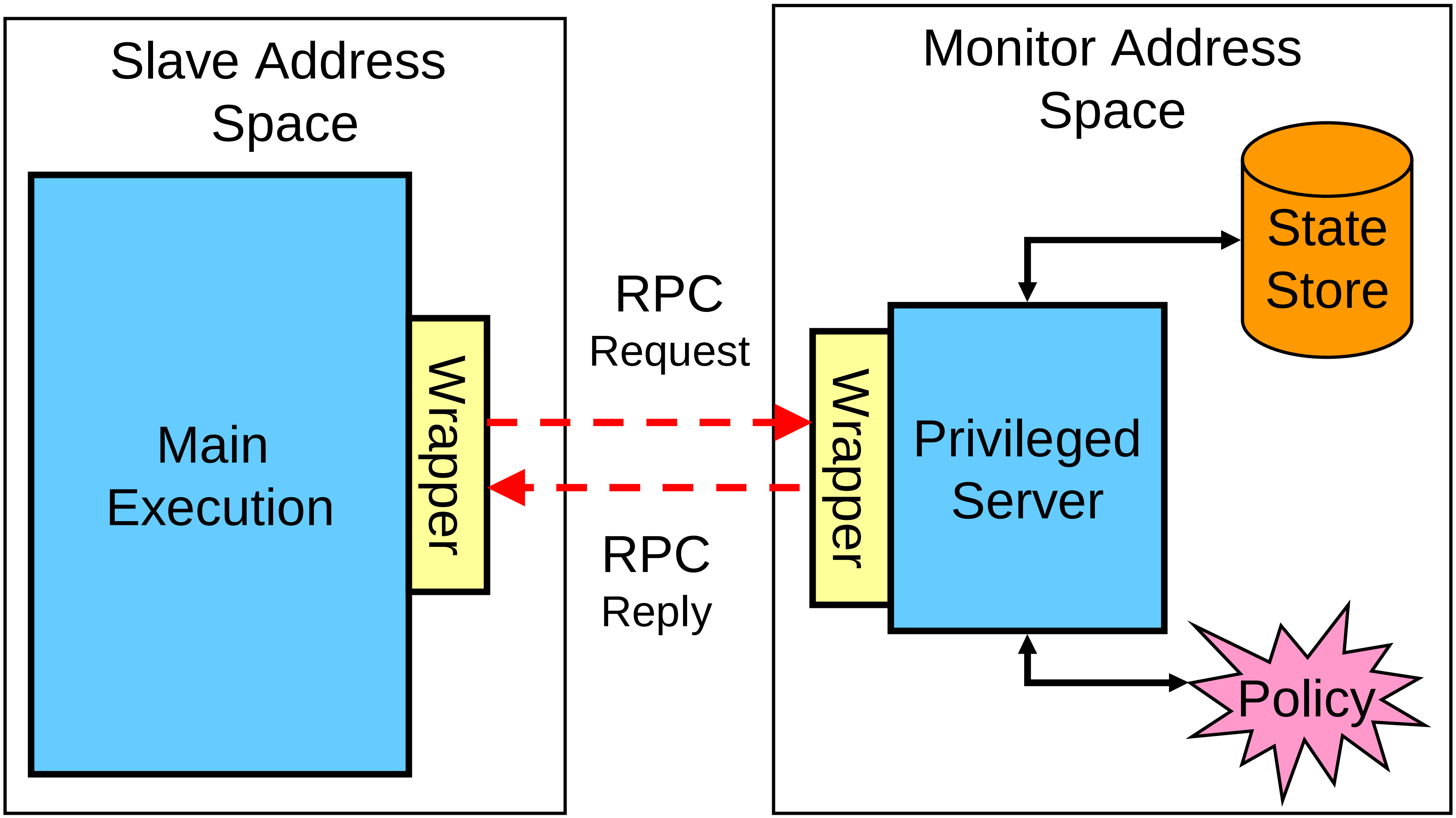
goal:

Automatically integrate privilege separation to existing source code

Privtrans Overview



Privilege Separation at Runtime



Advantages of Our Automatic Privilege Separation

- Quick and easy to use on existing software
  - Can easily re-integrate as source evolves

- Strong model of privilege separation
  - Any data derived from privileged resource is privileged
  - All privileged data protected by monitor
  - More secure than just access control

- Allows fine-grained policies
  - Monitor can allow/disallow any privileged call

- Monitor easier to secure
  - Monitor small → easier to apply other static/dynamic techniques
  - Monitor can be ran on secure host

Talk Outline: Our Techniques & Results

- Techniques in Privtrans:

1. Data type qualifiers
2. Static analysis and propagating qualifiers
3. Qualifier polymorphism and dynamic checks
4. Other components: State Store, Wrappers, Translation
5. Policies

- Experiment results

Program type qualifiers

- Add a type qualifier to every variable and function
  - Privileged – variable or function uses/accesses privileged resource
  - Unprivileged – everything else
- Programmer provides a few initial annotations
  - Variables/functions that are known privileged
  - Annotations are C attributes
    Ex: int __attribute__((priv)) sock;
  - Un-annotated variable/function initially assumed unprivileged

Inferring qualifiers: Static Analysis

- Static analysis infers unknown privileged qualifiers
  - Through assignment
  - Through use in API (i.e., functions declared but not defined)
  - Use as argument or return value to a privileged function

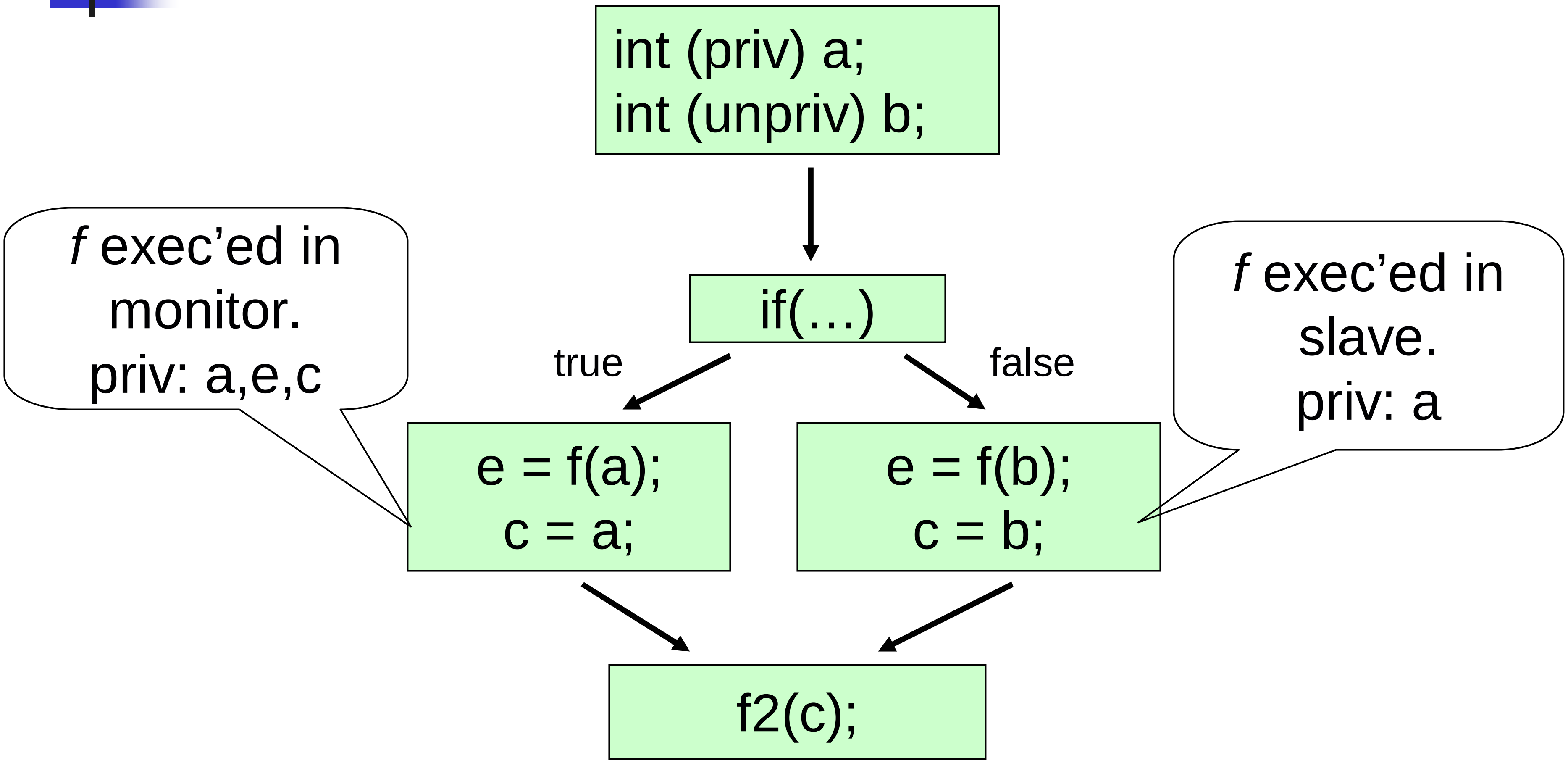
- Result of inference: API calls with privileged arguments
  - Monitor execute these calls
  - Monitor API -- only privileged functions in original source

- Privileged qualifiers found using meet-over-path analysis
  - Conservative
  - Similar to CQual “taint” analysis [foster99,shankar01]

Function Argument Polymorphism

- Function may be polymorphic in argument types
  - Privileged call – called with privileged arguments
  - Unprivileged call – no arguments or return value privileged
- Static analysis is conservative
  - May not be able to decide statically if call privileged or not
  - Must err on conservative side

A small polymorphic example



Dataflow tells us *f2* should be exec'ed in monitor

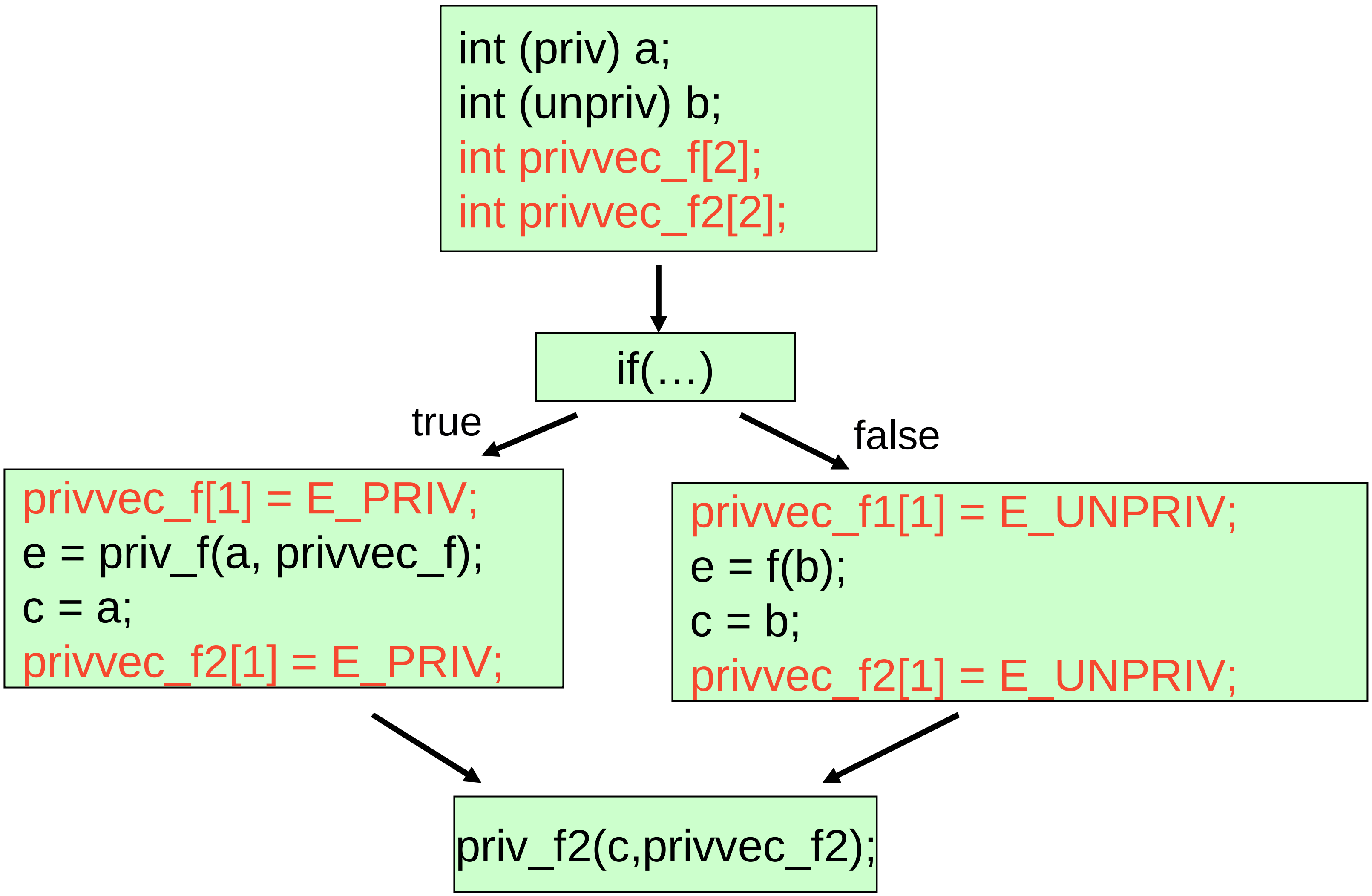
Our solution to polymorphism: Limiting calls to the monitor

- Combine static analysis with runtime information

- Insert code into slave to dynamically track qualifiers
  - Yields check of runtime (dynamic) privileged status
  - Improves accuracy of static analysis
  - Slave wrappers check flags

- Reduced monitor calls = improved performance
  - Monitor must defend against same types of attacks anyway
  - Limit number of calls to monitor

Dynamic Tracking of Privileged Variables



Other components (More information in paper)

- State store: keeps track of monitor values between calls
  - Monitor gives slave opaque index of previous values
  - Slave does not know anything about internal monitor state
  - Monitor can execute on different host than slave
- Wrappers
  - Use RPC as generic transport
  - Slave wrappers check dynamic qualifiers
- Source-to-source translation – Use CIL [necula et al 02]

Fine-grained policies

- Limited monitor interface is default protection
- Fine-grained policies can be added
  - Policies allow/disallow at function call level
  - Monitor can keep full context of call sequences → policies can be precise
- Previous techniques for automatically creating policies
  - Based on FSM/PDA of allowed call sequences
  - Based on call arguments

Experimental results: Changes to code

Program Name	src lines	# user annotations	# calls changed automatically	time to place annotations
chfn	745	1	12	1 hr
chsh	640	1	13	1 hr
ping	2299	1	31	1.5 hrs
thttpd	21925	4	13	2 hrs
OpenSSH	98590	2	42	2 hrs
OpenSSL	211675	2	7	20 min

Experimental Results: API Exported by the monitor

Name	# annotations	API exported by monitor
chfn	1	pam functions
chsh	1	pam functions
ping	1	socket operations
thttpd	4	socket operations
OpenSSH	2	pam operations/crypto key operations
OpenSSL	2	private key operations

Experiences: Potential issues and solutions

- Changing UID of slave
  - complicated but portable in Provos et al
  - Our approach: implement new system call

- Distinguish privileged values in a collection (e.g., array) on slave
  - opaque monitor identifier suffices

- Other issues discussed in paper

Result quality and performance

- Our automatic approach results in similar API to manual separation in OpenSSH
- Performance overhead reasonable
  - Usually ≤ 15% for programs tested, depending on application
  - Overhead amortized over total execution
- Overhead dominated by cross-process call time
  - SFI can reduce or eliminate this cost
- Works on small and large programs

Conclusion

- Type information useful for slicing programs
  - Easy to perform on existing programs
  - Allows for fine-grained policies
  - can re-incorporate privilege separation as source evolves
  - Techniques apply to C program – should also work on Windows
- Privtrans results similar to manual privilege separation
- Improve static analysis precision with dynamic checks
- Techniques work on small and large programs

Questions?

Contact:

David Brumley or Dawn Song
Carnegie Mellon University
{dbrumley,dawn.song}@cs.cmu.edu

Begin backup slides

- Begin backup slides

Potential Issues of Automatic Privilege Separation

- May not work on all programs because:
  - Socket numbering different
  - UID/GID checks different
  - Source code defies static analysis

- Collections are hard to interpret
  - Ex: array of file descriptors
  - Opaque index returned by monitor often enough to distinguish priv from unpriv.

Performance Overhead Numbers

Overhead dominated by cross-domain call

- Similar to Kilpatrick et al.
- No attempt to optimize per-application
- Can be reduced several orders of magnitude by SFI

Call name	Performance penalty factor
socket	8.83
open	7.67
bind	9.76
listen	2.17

Future Work

- Add pointer tracking for better precision
  - Esp. when to free priv. data
- Incorporate automatic policy generation
- Use attribute information to make better system call interposition models

Privileges in a program

A privilege in a program is:

- An OS Privilege:
  - Ex: Reading /etc/passwd
- The ability to access object
  - Ex: Crypto keys

Many different approaches to prevent privilege escalation

- Rewrite application in a safe language – **$$$$$$$$**
- Find and fix all bugs – impractical
- System-call Interposition – too coarse grained
- Runtime checks (stackguard, etc) – usually applied to the whole program

Advantages of dynamic checks

- Improve precision of static analysis

- Do not breach security properties of program.

- Dynamic checks are safe:
  - Attacker tries to make privileged call w/o privileges
    → fails!

  - Attacker tries to make call through monitor
    → Monitor API limits restricts types of calls.
    → Monitor policy should disallow.

Monitor State Store

```
1. int __((priv))__ sock;
2. sock = socket(…);
3. setsockopt(sock,..);
```

- Line 2 – Slave asks monitor to create socket
  - Monitor creates socket.
  - Stores in state store, returns opaque index
- Line 3 – Slave asks monitor to update socket.
  - Slave provides index from line 2.
  - Monitor looks up socket
  - Performs setsockopt().

Automatic Privilege Separation

- Previous privilege separation done by hand

Our goal:

Automatically integrate privilege separation to existing source code