Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

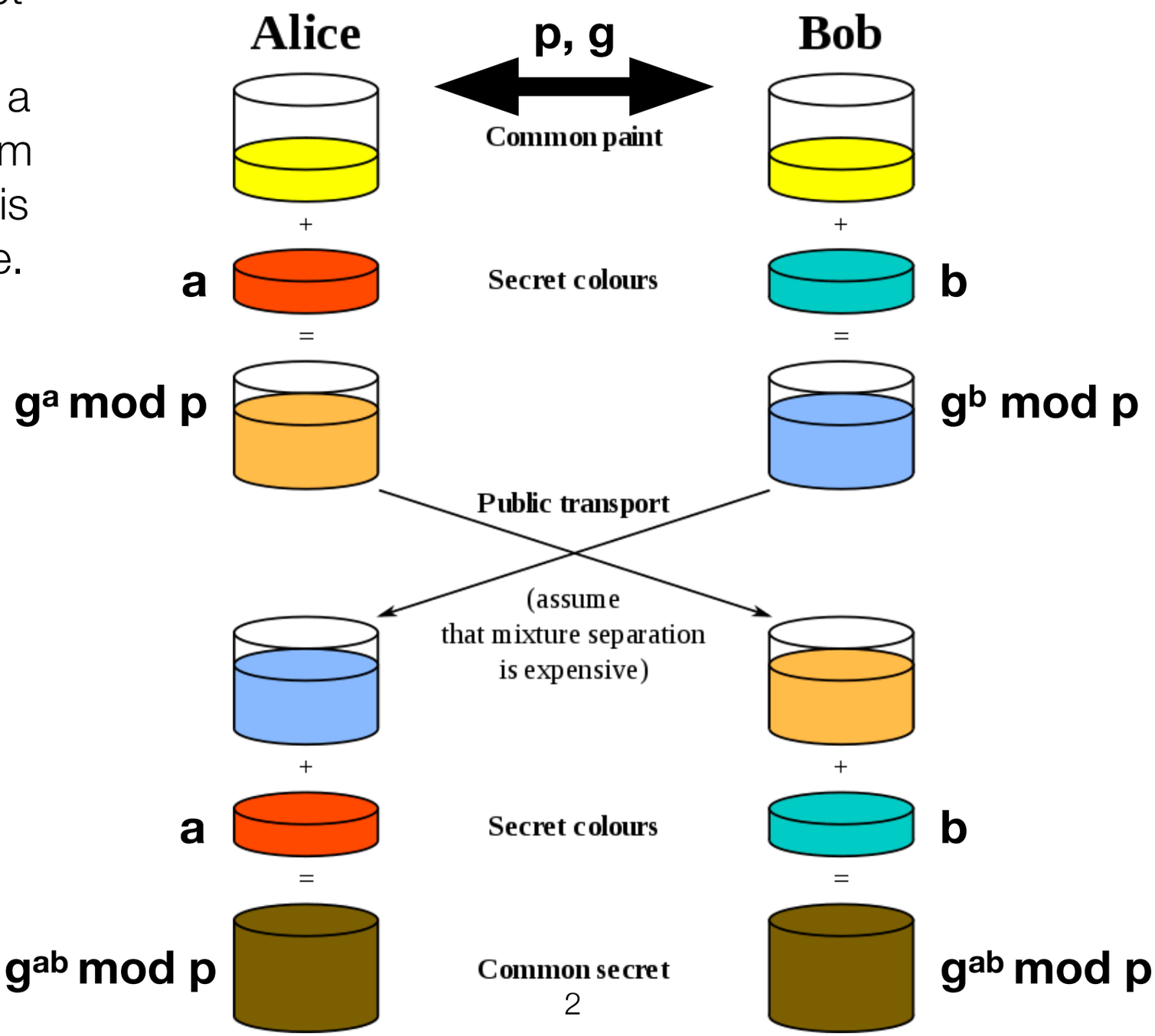
David Adrian et al.
weakdh.org

Presented at the 22nd ACM - CCS ’15, October 2015
Best Paper Award Winner

Presented today by: Reethika Ramesh

What is Diffie-Hellman Key Exchange?

Relies on the fact that using large primes makes it a discrete logarithm problem, which is hard to compute.



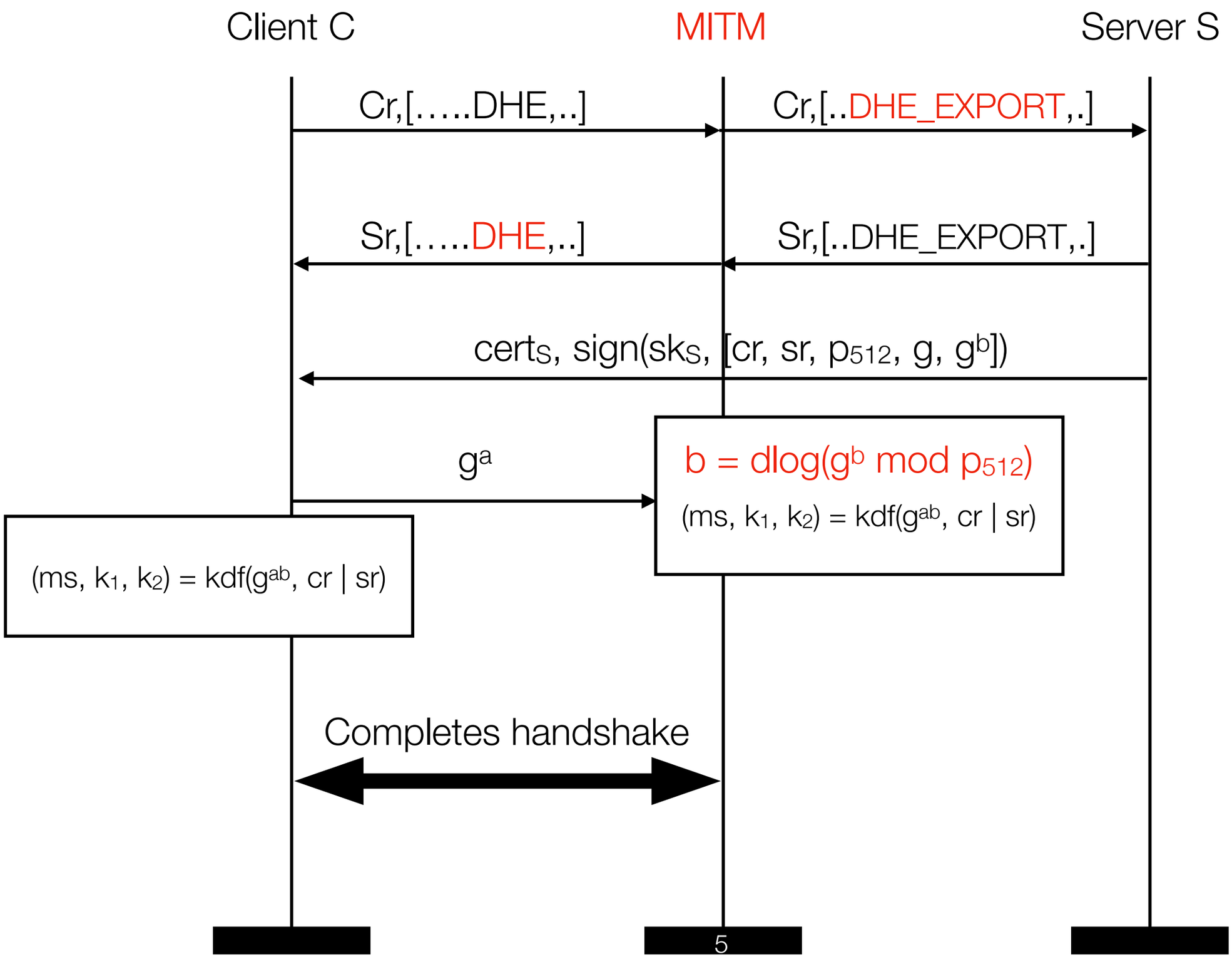
Motivation

- Diffie-Hellman Key Exchange - less secure than once believed.
- Gap between cryptographers and system designers leads to disastrous security issues.

Vulnerability

- DHE relies on the choosing of the prime p and g.
- Large. But how large? 512-bit, 768-bit or 1024-bit or higher?
- 1990-s era U.S. export restrictions applied to crypto - backward compatibility

General idea of Attack



Active downgrade to DHE_EXPORT

Attack

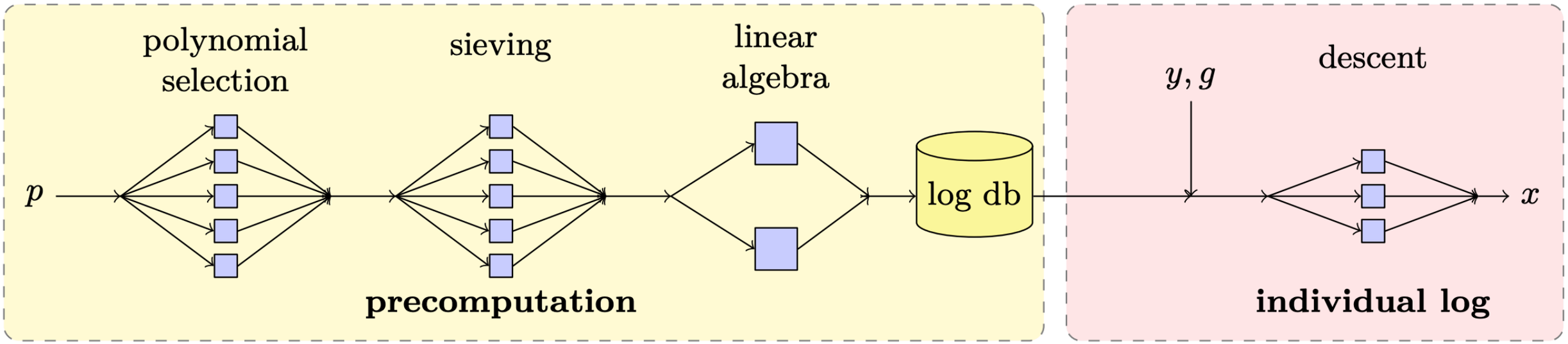
- 92% of vulnerable severs use the same two 512-bit DH groups

- Over 7% of connections to Top 1M sites compromised

- Other weak and misconfigured groups vulnerable:

    1. Non-export grade DHE with 512-bit primes
    2. Composite order subgroups
    3. Programming error with DSA primes

Logjam



$L(1/3, 1.923) = \exp(1.923(\log p)^{1/3}(\log\log p)^{2/3})$ $L(1/3, 1.232)$

	Sieving	Linear Algebra	Descent
RSA-512	0.5 core-years	0.33 core-years	
DH-512	2.5 core-years	7.7 core-years	10 core-mins

Crunching the numbers

- Sieve more than necessary
  - Larger database of known logs - makes descent faster
  - Reduce load on linear algebra step - less parallelize-able

	polysel	sieving	linalg	descent
	2000-3000 cores		288 cores	36 cores
DH-512	3 hours	15 hours	120 hours	70 seconds

- After sieving, they obtained a square matrix of ~2M rows

How secure is DHE?

- Disallowing downgrade
- Stronger groups
- 768-bit and 1024-bit primes?
- Can the attack scale to larger groups?

State-Level Adversaries

- Leverage the widespread reuse of common DH parameters
- Special purpose hardware ~ 80x speedup
- $11B machines can precompute for one 1024-bit prime every year

	Sieving core-years	Linear Algebra core-years	Descent core-time
RSA-512	0.5	0.33	
DH-512	2.5	7.7	10 mins
RSA-768	800	100	
DH-768	8,000	28,500	2 days
RSA-1024	1,000,000	120,000	
DH-1024	10,000,000	35,000,000	30 days

National Security Agency

(der Spiegel documents)

- Attacks on IKE possible if attacker has:

    1. Full transcript with DH keys and nonces and cookies
    2. The Pre-Shared Keys used in deriving SKEYID.

- Possible with developing specific-hardware

- Cost to finish the 1024-bit precomputation stage in a year ~$11B.

Current State

(data from censys.io, updated continually)

Data from Censys	DHE Support	DHE_EXPORT Support
IPv4 hosts with browser trusted certificates https/443	28.71% (6 million)	0.27% (57,841)
Alexa Top 1M (plus churn) https/443	49.34% (467,150)	0.52% (4,933)

Current State

For IPv4 hosts with browser trusted certificates that support DHE

443.https.dhe.dh_params.prime.length	Hosts	
2048	3,729,920	62.07%
1024	1,928,614	32.1%
4096	316,318	5.26%
768	18,793	0.31%
2240	5,828	0.1%
3072	5,145	0.09%
1072	1,516	0.03%
1536	1,337	0.02%
512	580	0.01%

Current State

For Alexa Top 1M HTTPS websites that support DHE

443.https.dhe.dh_params.prime.length	Websites	
2048	288,613	75.14%
1024	69,085	17.99%
4096	25,788	6.71%
3072	340	0.09%
768	143	0.04%
2240	51	0.01%
512	32	0.01%

Recommendations

- Transition to ECDHE
- Increasing minimum key strengths
- Avoid FIXED-prime 1024-bit groups
- Don’t deliberately weaken crypto

Effect of Logjam

- Internet Explorer, Firefox and Chrome transition to minimum 1024-bit.

- On server side, Akamai removed all support for DHE_EXPORT.

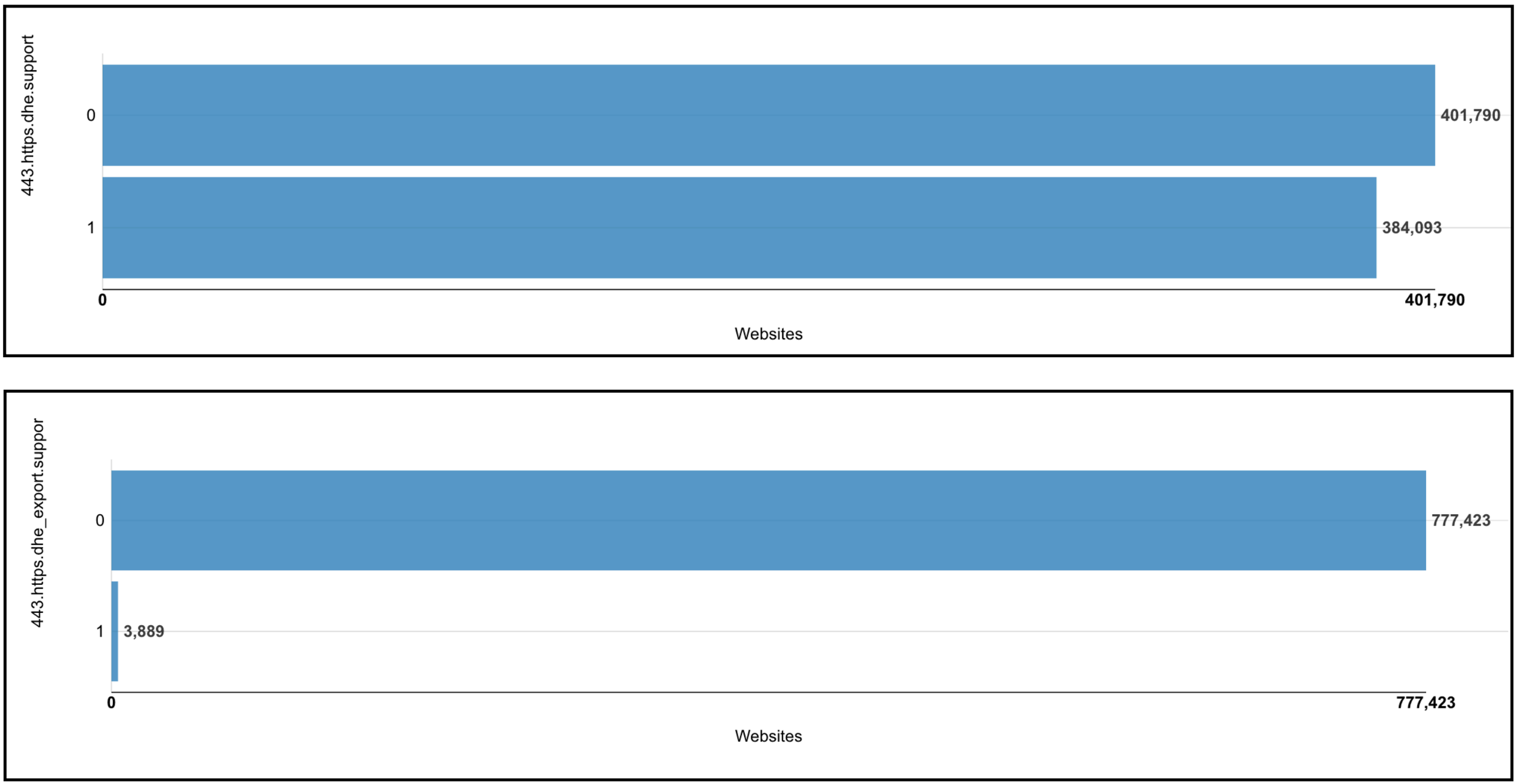
- TLS developers plan to negotiate use of well-known groups 2048-bit primes.

Conclusion

- Reuse of well-known DHE groups leads to the attack reaching further and wider.
- Don’t deliberately weaken crypto (bears repeating)
- Bridge the gap between cryptographers and practical system designers.

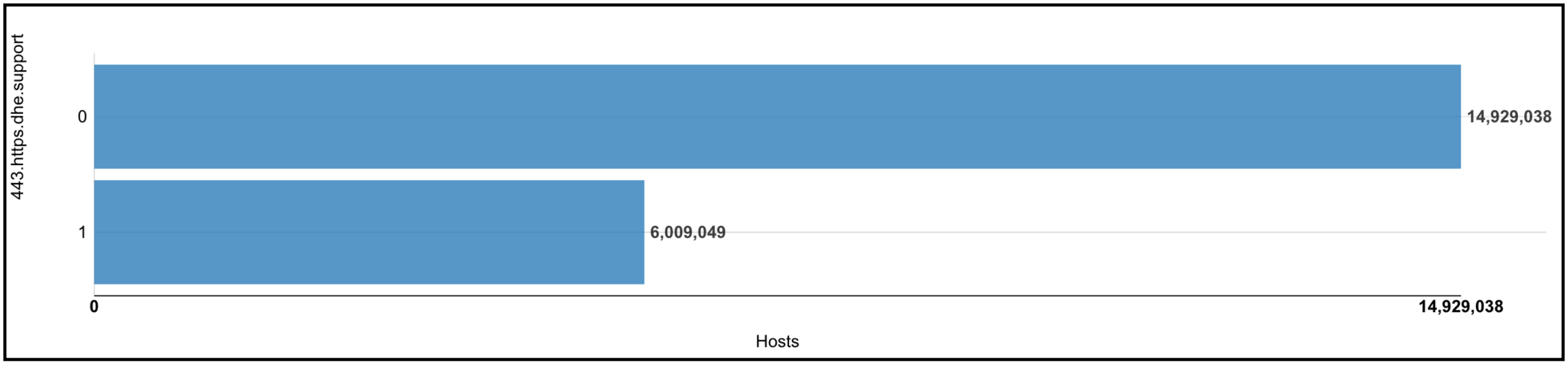
Censys Data

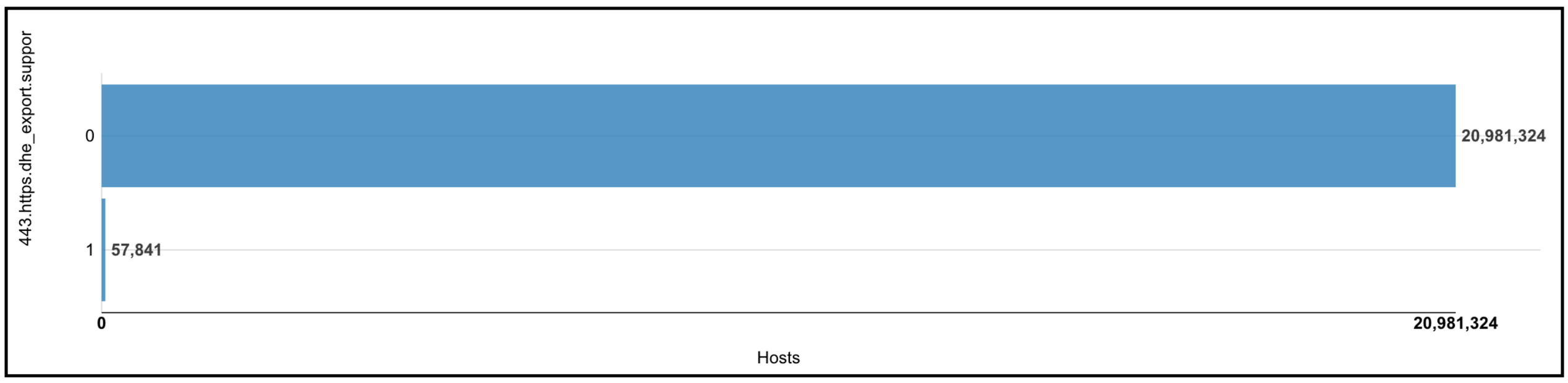
Alexa Top 1M



Censys Data

IPv4 hosts with trusted certificates





Strengths

- Novel flaw discovery
- Extensive research about time and cost complexities
- Other vulnerabilities also exploited
- Expedited disclosure and response

Weakness

- The math behind the precomputation stage was a bit convoluted.

What are the open questions?

- Future work: The engineering behind this attack can be revisited a few years later and with improvements in algorithms and upgraded compute power, can larger groups be broken? Can other algorithms be defeated in a similar way?

Control-Flow Integrity Principles, Implementations, and Applications

M Abadi, M Budiu, U Erlingsson, J Ligatti

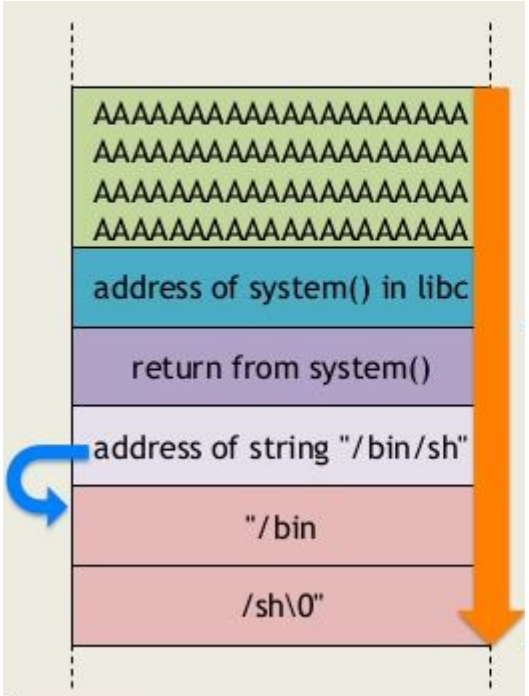
Presented by Hao Lu

Overview

- Attacks built on exploits exploits that
  - *subvert machine-code execution*
  - *arbitrarily control program behavior*
- Control-Flow Integrity (CFI)
  - *Simple, formal & practical*
  - *Foundation for further security policies*

Return-to-libc attack

- Stack buffer overflow
  - *Overrides return address*
  - *Transfers control to code in libc*
- Return-oriented programming



https://www.slideshare.net/saumilshah/dive-into-rop-a-quick-introduction-to-return-oriented-programming

Enforcing CFI

- Software execution must follow a path of a CFG
  - *Dynamic checks*
  - *Machine-code rewriting*
  - *Static inspection*
- An instruction transfers control to a valid destination
  - *Computed control-flow transfers*

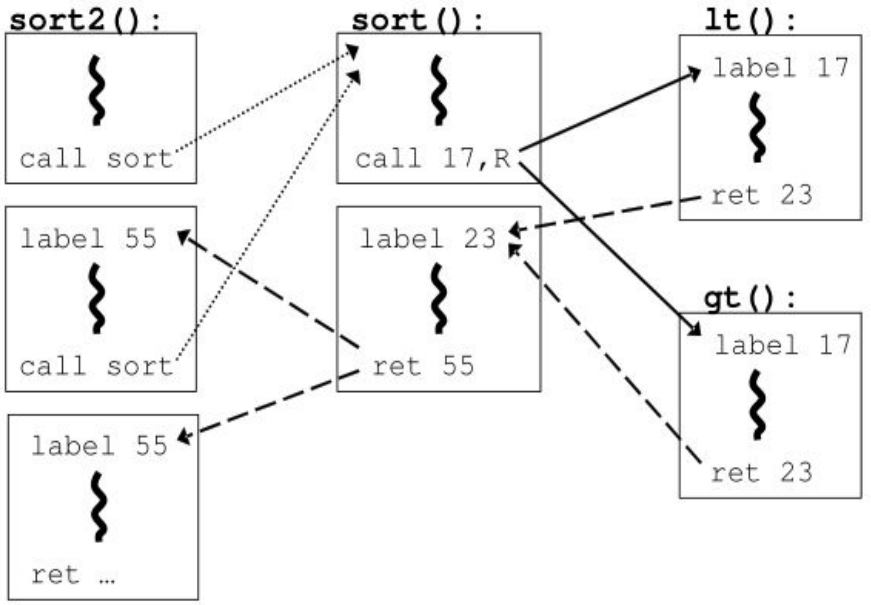
Instrumentation

- Three new machine-code instructions
  - *Label ID*
  - *Call ID, DST*
  - *Ret ID*
- Instruments source code by
  - *Inserting an ID at each destination*
  - *Inserting a dynamic check before each source*

Instrumentation

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Instrumentation for source

Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction
	can be instrumented as (a):	
81 39 78 56 34 12	cmp [ecx], 12345678h	; compare data at destination
75 13	jne error_label	; if not ID value, then fail
8D 49 04	lea ecx, [ecx+4]	; skip ID data at destination
FF E1	jmp ecx	; jump to destination code
	or, alternatively, instrumented as (b):	
B8 77 56 34 12	mov eax, 12345677h	; load ID value minus one
40	inc eax	; increment to get ID value
39 41 04	cmp [ecx+4], eax	; compare to destination opcodes
75 13	jne error_label	; if not ID value, then fail
FF E1	jmp ecx	; jump to destination code

Instrumentation for destination

Bytes (opcodes)	x86 assembly code	Comment
`8B 44 24 04`	`mov eax, [esp+4]`	`; first instruction`
`...`		`; of destination code`

can be instrumented as (a):

`78 56 34 12`	`DD 12345678h`	`; label ID, as data`
`8B 44 24 04`	`mov eax, [esp+4]`	`; destination instruction`
`...`		

or, alternatively, instrumented as (b):

`3E 0F 18 05 78 56 34 12`	`prefetchnta [12345678h]`	`; label ID, as code`
`8B 44 24 04`	`mov eax, [esp+4]`	`; destination instruction`
`...`		

Assumptions

- Unique IDs
  - *Bit patterns must not be present anywhere except in IDs*
- Non-writable code
- Non-executable data

CFI Phases

- Construction of the CFG
- Instrumentation
- Verification

Implementation

- Windows x86, Vulcan
- Instrumentation that does not require
  - *Recompilation*
  - *Source-code access*
- Ret changed to jump
  - *To avoid a time-of-check-to-time-of-use race condition*

Implementation

Bytes (opcodes)	x86 assembly code	Comment
FF 53 08	call [ebx+8]	; call a function pointer

is instrumented using `prefetchnta` destination IDs, to become:

8B 43 08	mov eax, [ebx+8]	; load pointer into register
3E 81 78 04 78 56 34 12	cmp [eax+4], 12345678h	; compare opcodes at destination
75 13	jne error_label	; if not ID value, then fail
FF D0	call eax	; call function pointer
3E 0F 18 05 DD CC BB AA	prefetchnta [AABBCCDDh]	; label ID, used upon the return

Implementation

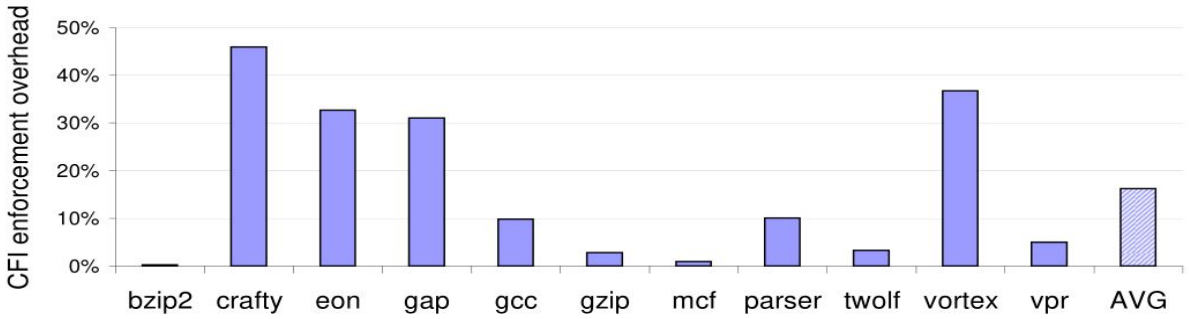
Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes

is instrumented using `prefetchnta` destination IDs, to become:

8B 0C 24	mov ecx, [esp]	; load address into register
83 C4 14	add esp, 14h	; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA	cmp [ecx+4], AABBCCDDh	; compare opcodes at destination
75 13	jne error_label	; if not ID value, then fail
FF E1	jmp ecx	; jump to return address

Measurements

- SPEC CPU benchmarks
- CFG construction and CFI instrumentation
  - *10 seconds*
  - *Binary size increased by 8%*
- Binary execution
  - *Takes 16% longer*
  - *Competitive*



Security-related Experiments

- Return-to-libc
- GDI+ JPEG flaw
  - *Memory corruption overwrites a global variable that holds a C++ object pointer*
  - *Virtual destructor pointing to code of attacker's choice*
- 18 tests for dynamic buffer-overflow prevention

Building on CFI

- Inlined reference monitors
  - *Instruments validity checks & additional state enforcement*
  - *Requires the subject program cannot circumvent the checks*
- Software fault isolation
  - *Emulates traditional memory protection*
  - *Ensures the target memory address lies within a certain range*

SFI with CFI

```
int compute_sum( int a[], int len )
{
    int sum = 0;
    for(int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

```
        ...
        mov   ecx, 0h           ; int i = 0
        mov   esi, [esp+8]      ; a[] base ptr
        and   esi, 20FFFFFFh    ; SFI masking
LOOP:   add   eax, [esi+ecx*4]  ; sum += a[i]
        inc   ecx               ; ++i
        cmp   ecx, edx          ; i < len
        jl    LOOP
```

Building on CFI

- SMAC
  - *Allows different access checks to be inserted at different instructions*
  - *Can create isolated data memory only accessible from specific pieces of code*
  - *Can help eliminate CFI assumptions*
    - NWC: disallowing writes to certain memory addresses
    - NXD: preventing control flow outside code segment
  - *Protected shadow call stack*

Formal Study

- An abstract machine with a simplified instruction set

Theorem 1

Let S_0 *be a state* $(M_c|M_d, R, pc)$ *such that* $pc = 0$ *and* $I(M_c, G)$*, where* G *is a CFG for* M_c*, and let* $S_1, \ldots, S_n$ *be states such that* $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$*. Then, for all* $i \in 0..(n-1)$*, either* $S_i \rightarrow_{\mathrm{a}} S_{i+1}$ *and* $S_{i+1}.pc = S_i.pc$*, or* $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, G, S_i.pc)$.

Conclusion

- Control flow integrity disparity
  - *High-level programming language*
  - *Machine-code level*

Thanks

Other ways to protect against stack-overflow attacks

Other ways to protect against stack-overflow attacks

- Detect that a stack buffer overflow has occurred and thus prevent redirection of the instruction pointer to malicious code
- Prevent the execution of malicious code from the stack without directly detecting the stack buffer overflow
- Randomize the memory space such that finding executable code becomes unreliable.

Other ways to protect against stack-overflow attacks

- Stack canary
- Non-executable stack
- Address space layout randomization

System level call stack