Efficient Denial of Service Attacks on Web Application Platforms

Alexander "**alech**" Klink
n.runs AG

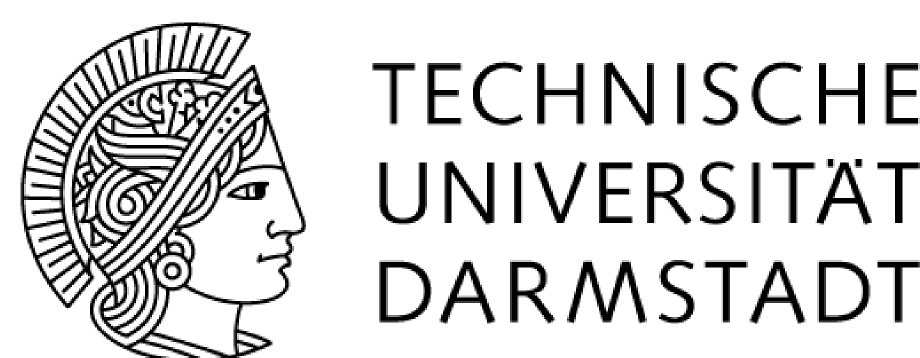
Julian "**zeri**" Wälde
TU Darmstadt

#hashDoS

December 28th, 2011. 28th Chaos Communication Congress. Berlin, Germany.

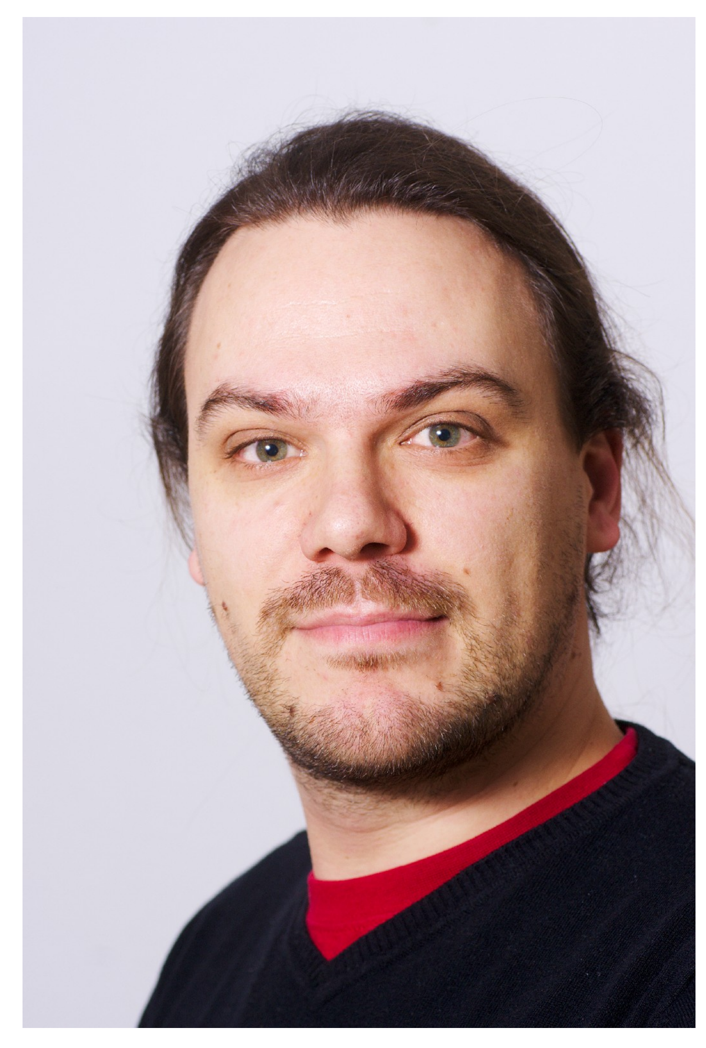
Who are we?

Julian “**zeri**” Wälde



theoretical security

Who are we?



Alexander "**alech**" Klink

applied security

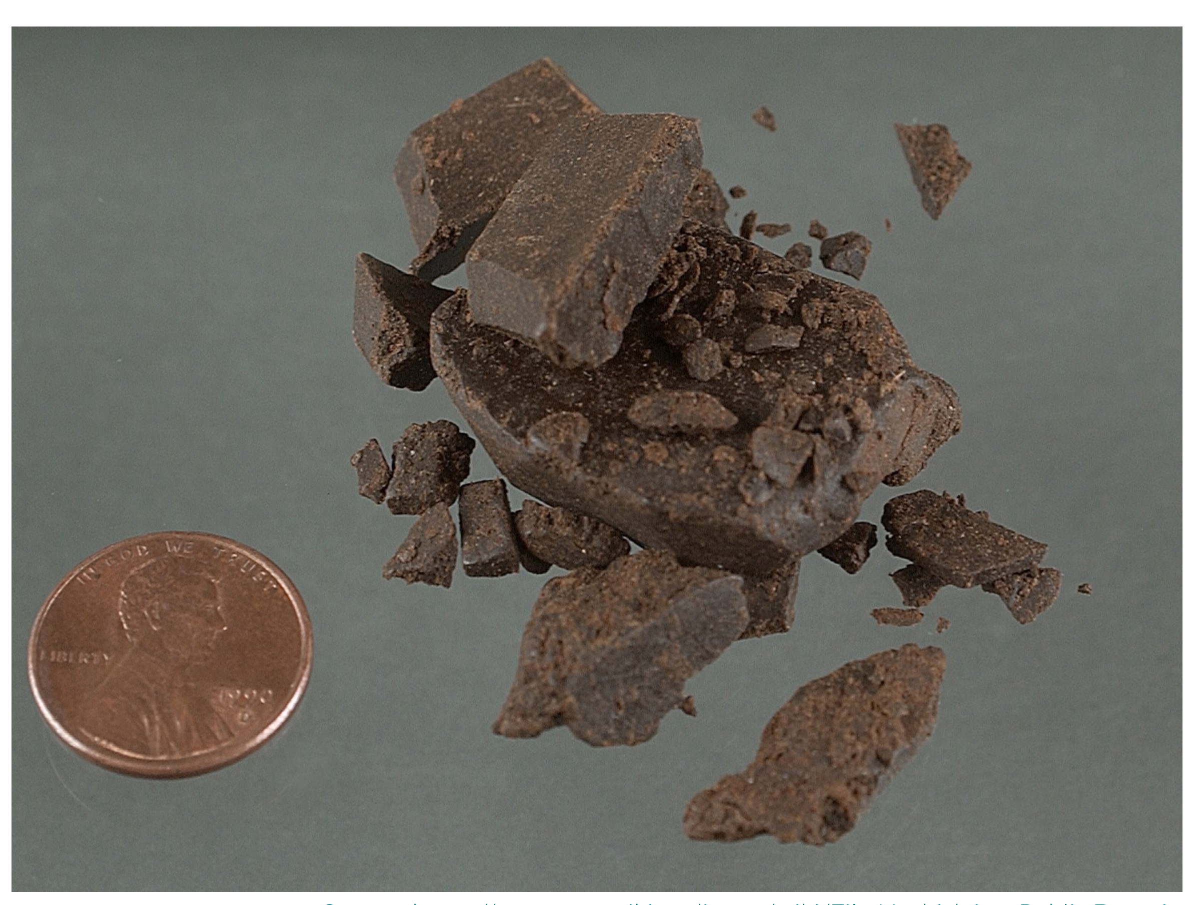
How did we get here?

Trollhöhle (Chaos Darmstadt)

perldoc perlsec, section “Algorithmic Complexity Attacks”

Live demo, part I

Hash table



Source: https://commons.wikimedia.org/wiki/File:Hashish.jpg, Public Domain

Source: https://commons.wikimedia.org/wiki/File:Bernerhof_Large_Salon.jpg, CC-BY Sandstein

Have you seen this code?

```
h = {}                  # empty hash table
h['foo'] = 'bar'        # insert
print h['foo']          # lookup, prints 'bar'
```

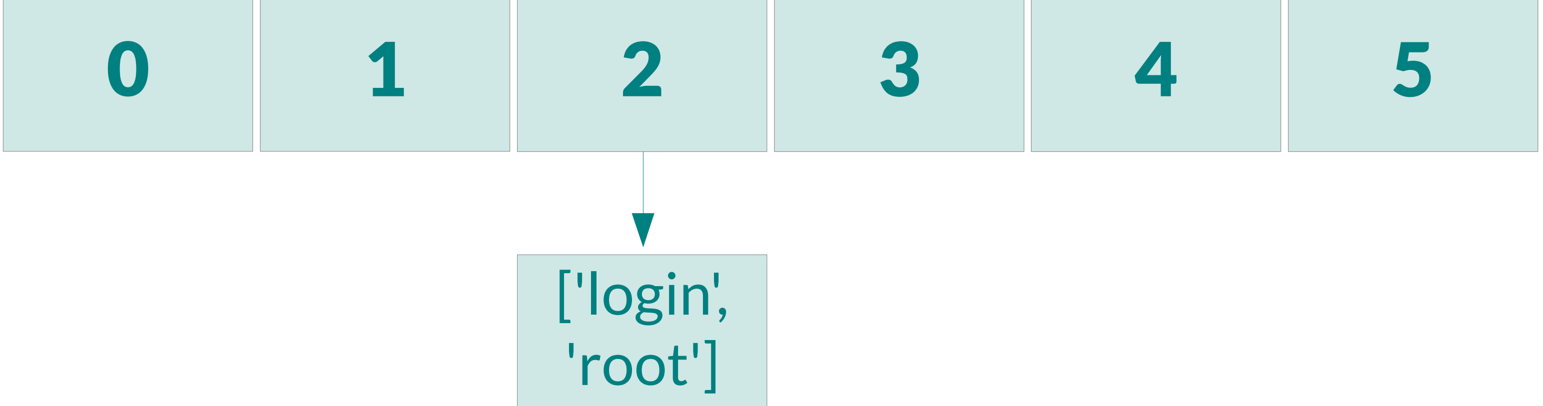
valid Ruby/Python code
(slightly) different syntax elsewhere

Do you know how it works?

How it works (insertion)

h['login'] = 'root'

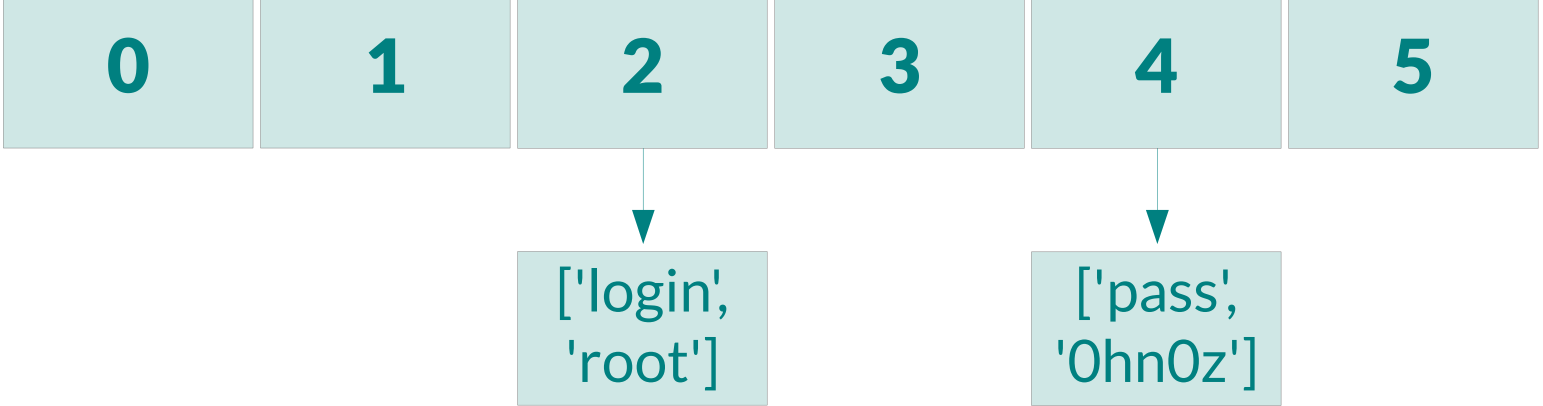
hash('login') = 2



How it works (insertion)

h['pass'] = '0hn0z'

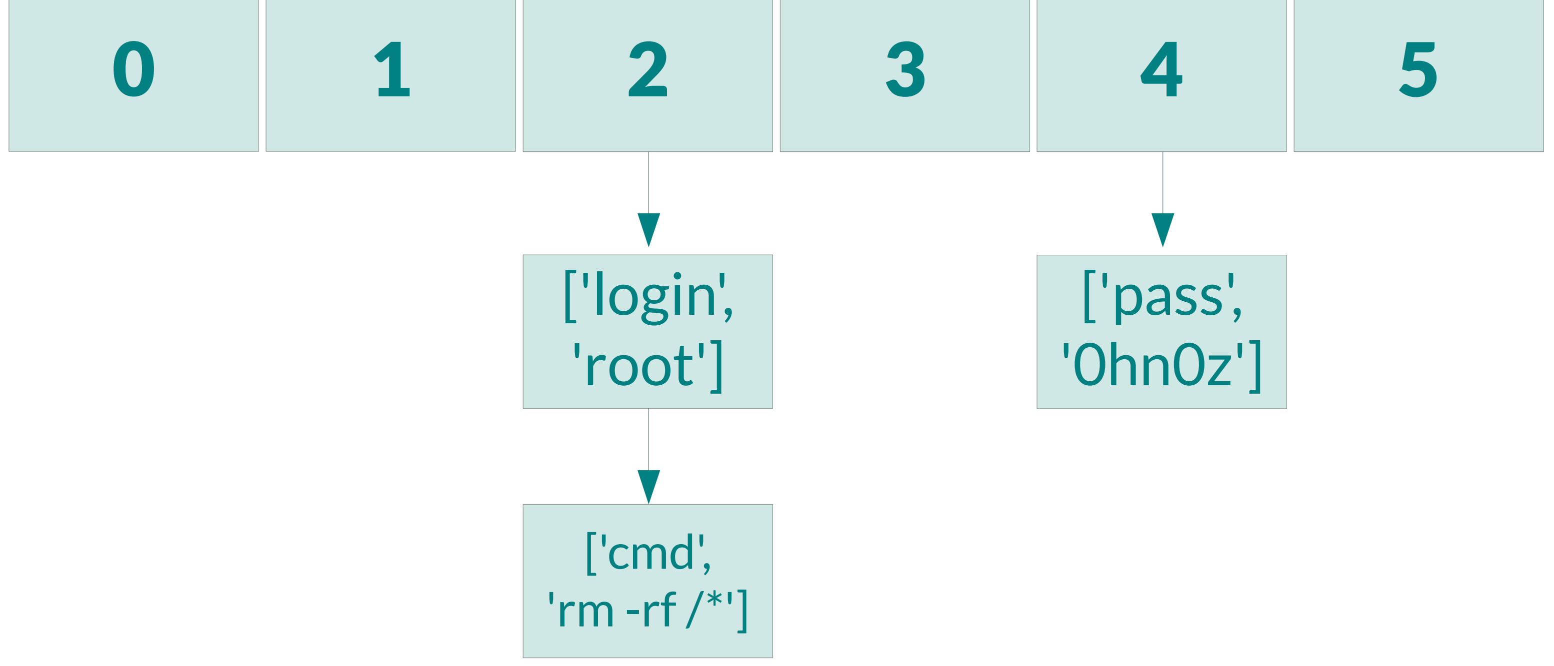
hash('pass') = 4



How it works (insertion)

h['cmd'] = 'rm -rf /*'

hash('cmd') = 2



Complexity: best/average case

One element:

insert → O(1)

lookup → O(1)

(delete) → O(1)

n elements:

insert → O(n)

lookup → O(n)

(delete) → O(n)

aka “pretty damn fast”

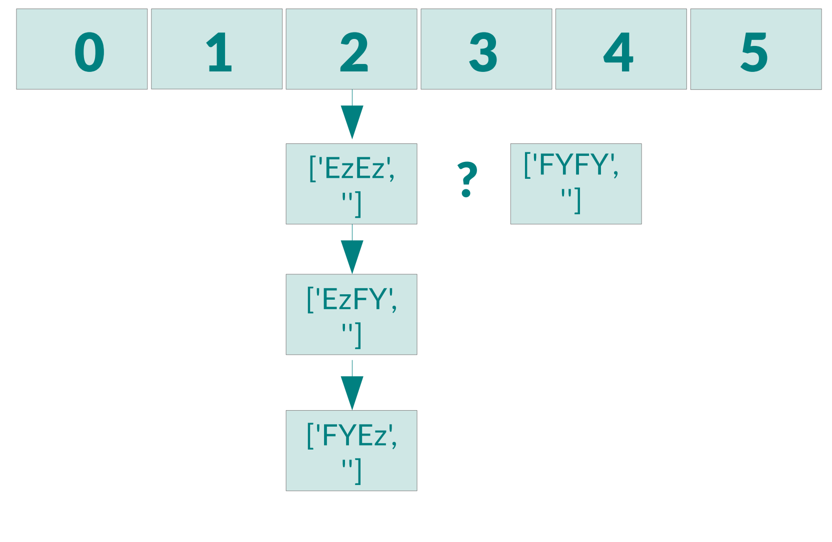
Complexity: worst case

n elements:

insert → $O(n^2)$

lookup → $O(n^2)$

(delete) → $O(n^2)$



aka “a tortoise is fast against it”

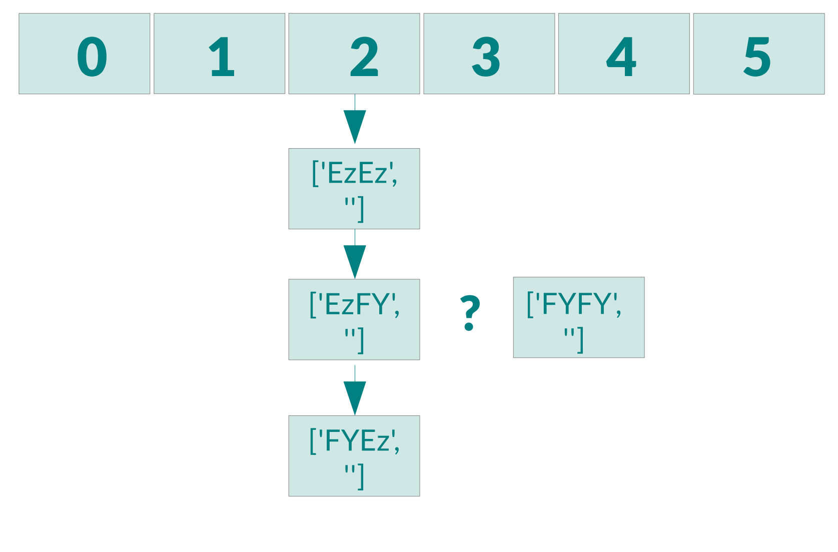
Complexity: worst case

n elements:

insert → O(n^2)

lookup → O(n^2)

(delete) → O(n^2)



aka "a tortoise is fast against it"

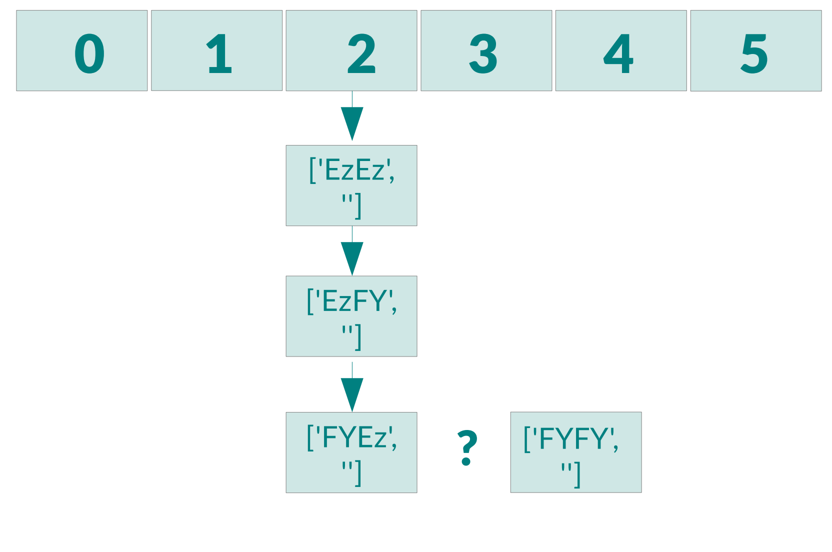
Complexity: worst case

n elements:

insert → O(n²)

lookup → O(n²)

(delete) → O(n²)



aka “a tortoise is fast against it”

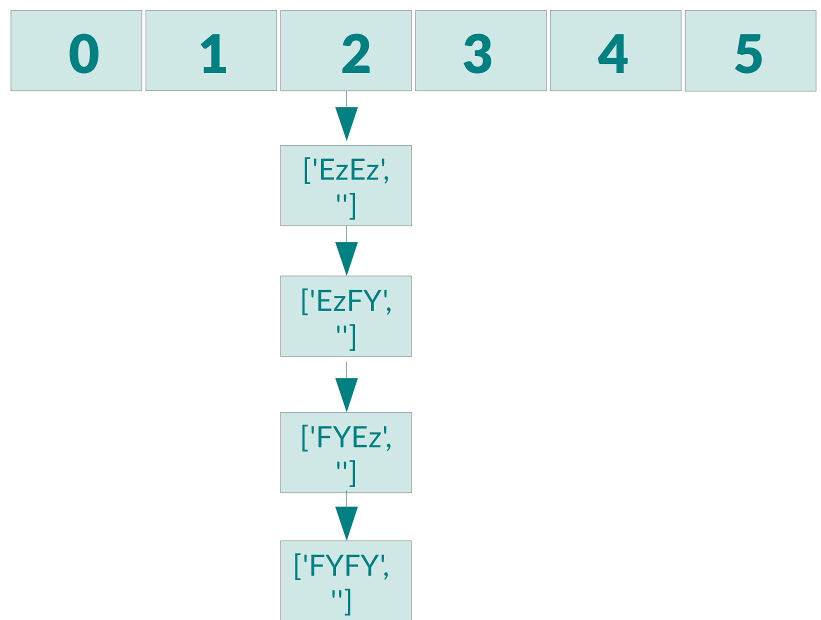
Complexity: worst case

n elements:

insert → $O(n^2)$

lookup → $O(n^2)$

(delete) → $O(n^2)$



aka "a tortoise is fast against it"

The worst case in real life

200.000 multi-collisions à 10 bytes
roughly 2 MB

40.000.000.000 string comparisons
On a 1GHz machine, this is at least 40s

Live demo, part II

Hash functions: definition

- collision resistance?
- one-way?
- fixed output length?

Hash functions: definition

- collision resistance? ✗
- one-way?
- fixed output length?

Hash functions: definition

- collision resistance? ✗
- one-way? ✗
- fixed output length?

Hash functions: definition

- collision resistance? ✗
- one-way? ✗
- fixed output length? ✓

Do you know this guy?

Dan “**djb**” Bernstein (at 27C3)

DJBX33A

times add

```
uint32_t hash(const char *arKey, uint32_t nKeyLength) {
    uint32_t hash = 5381;

    for (; nKeyLength > 0; nKeyLength -=1) {
        hash = ((hash << 5) + hash) + *arKey++;
    }
    return hash;
}
```

hash × **33**

java.lang.String.hashCode()

```
uint32_t hash(const char *arKey, uint32_t nKeyLength) {
   uint32_t hash = 5381;

   for (; nKeyLength > 0; nKeyLength -=1) {
      hash = ((hash << 5) + hash) + *arKey++;
   }
   return hash;
}
```

hash × **33**

java.lang.String.hashCode()

```
uint32_t hash(const char *arKey, uint32_t nKeyLength) {
  uint32_t hash = 0;

  for (; nKeyLength > 0; nKeyLength -=1) {
    hash = ((hash << 5) - hash) + *arKey++;
  }
  return hash;
}
```

hash × **31**

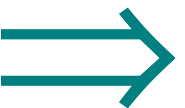
Equivalent substrings

$$h(s) = \sum 31^{n-i} \cdot s_i$$

$$h('Ey') = \mathbf{31^1 \cdot 69 + 31^0 \cdot 121} = 2260$$
$$h('FZ') = \mathbf{31^1 \cdot 70 + 31^0 \cdot 90} = 2260$$

$$h('Eya') = 31 \cdot \mathbf{(31^1 \cdot 69 + 31^0 \cdot 121)} + 31^0 \cdot 97$$
$$= 31 \cdot \mathbf{(31^1 \cdot 70 + 31^0 \cdot 90)} + 31^0 \cdot 97$$
$$= h('FZa')$$

Equivalent substrings



I.	h('EzEz')	(00)
II.	= h('EzFY')	(01)
III.	= h('FYEz')	(10)
IV.	= h('FYFY')	(11)

Equivalent substrings

h('tt') = h('uU') = h('v6')

I.	h('tttt')	(00)
II.	= h('ttuU')	(01)
III.	= h('ttv6')	(02)
IV.	= h('uUtt')	(10)
V.	= h('uUuU')	(11)
VI.	= h('uUv6')	(12)
VII.	= h('v6tt')	(20)
VIII.	= h('v6uU')	(21)
IX.	= h('v6v6')	(22)

Generating 3^n collisions

```
base3_strings = (0..3**n-1).each do |i|
  “%0nd” % i.to_s(3) # “0...0” to “2...2”
end

base3_strings.map do |s|
  s.gsub('0', 'tt')
   .gsub('1', 'uU')
   .gsub('2', 'v6')
end
```

Hash functions: definition

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

typically $n = 32$

Remember this guy?

DJBX33X

times XOR

```
uint32_t hash(const char *arKey, uint32_t nKeyLength) {
    uint32_t hash = 5381;

    for (; nKeyLength > 0; nKeyLength -=1) {
        hash = ((hash << 5) + hash) ^ *arKey++;
    }
    return hash;
}
```

hash × **33**

How To Attack This?

- Equivalent Substrings?
  - No – this function is nonlinear
- Bruteforce?
  - Yes but it takes several minutes per string

Cost of brute-forcing

Hit one specific hash value: 2^{31} attempts

Hit one in two specific hash values: 2^{30} attempts

Hit one in four specific hash values: 2^{29} attempts

...

Hit one in 2^{n} specific hash values: 2^{31-n} attempts

(Let's) Meet In The Middle

```
# Precomputation: filling the lookup table
repeat 2**16 times do
    s := randomsuffix # 3 char string
    h := hashback(s,target)
    precomp[h] := s
end
```

(Let's) Meet In The Middle

```
# Finding preimages
loop do
    s := randomprefix # 7 char string
    h := hashforth(s)
    if h in precomp then
        print s + precomp[h] # 10 char preimage
    end
end
```

(Let's) Meet In The Middle

000cc3f7 : 'RMh'
000cc3f7 : 'Slh'
00a07ae0 : 'Aon'
...
3b847a29 : 'Upl'
3b847a2a : 'vpl'
3b847a2a : 'wQl'
...
99976963 : 'CUu'
99976964 : 'dUu'
99976964 : 'etu'

h(x)

'QCMWaIO'

0

DJBX33X

times XOR

```
uint32_t hash(const char *arKey, uint32_t nKeyLength) {
    uint32_t hash = 5381;

    for (; nKeyLength > 0; nKeyLength -=1) {
        hash = ((hash << 5) + hash) ^ *arKey++;
    }
    return hash;
}
```

hash × **33**

Stand back,
I am going to use math!

XOR

$$A \oplus B \oplus B = A$$

Multiplication

$$33 \cdot 1041204193 = 1$$

false!

Multiplication

$$33 \cdot 1041204193 \equiv 1 \pmod{2^{32}}$$

true in the ring of integers modulus 2^{32}

aka 32 bit integers

DJBX33X done backwards

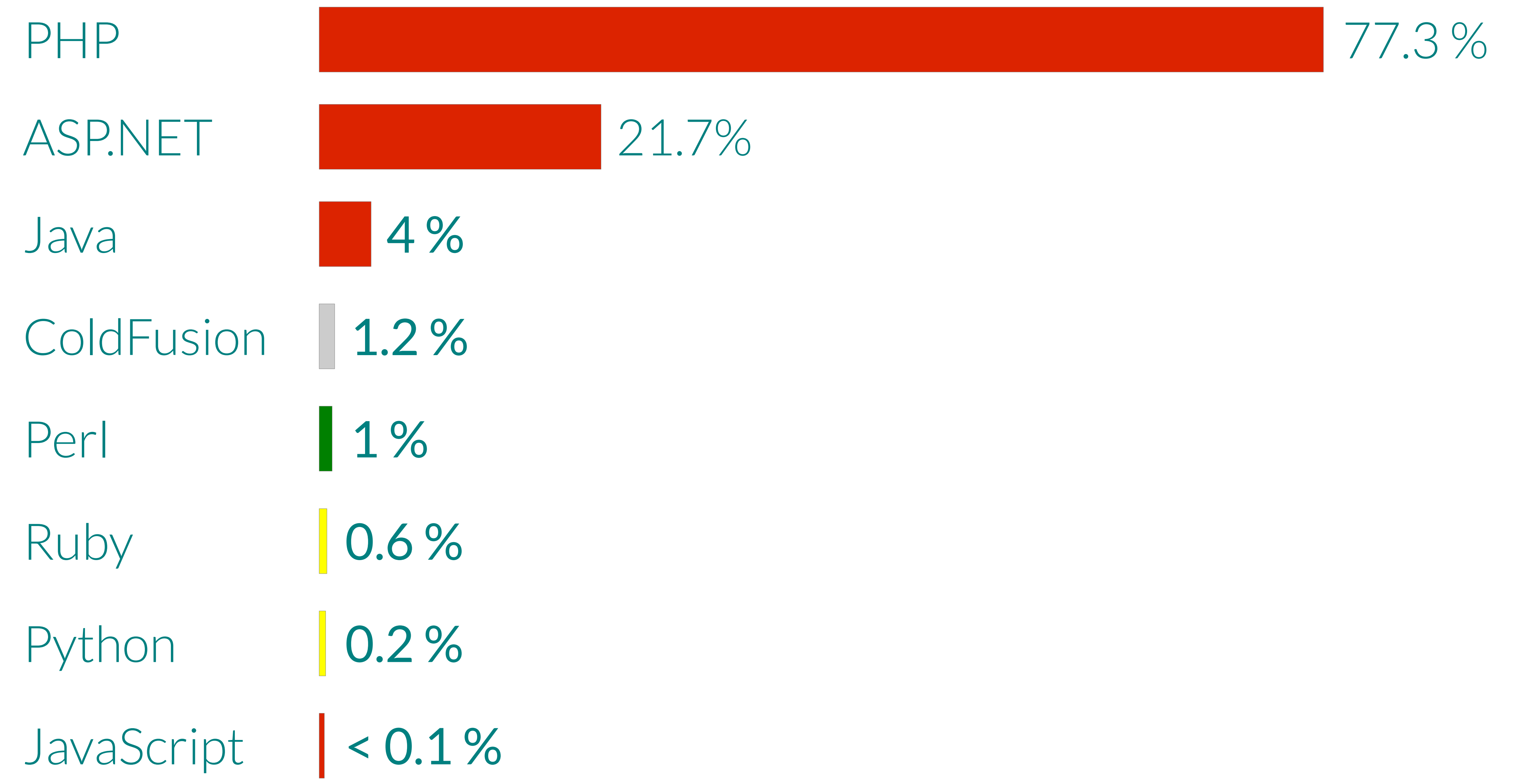
times XOR

```
uint32_t hash(char *suffix, uint32_t length, uint32_t end) {
    uint32_t hash = end;

    for (; length > 0; length -= 1) {
        hash = (hash ^ suffix[length – 1]) * 1041204193 ;
    }
    return hash;
}
```

Attacks

Web application technologies



Source: W3Techs.com, 10 December 2011

POST data in web applications

```
<?php echo $_POST["param"]; ?>
```

```
public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
        out.println(request.getParameter('param'));
}
```

```
Response.Write Request.Form['param']
```

PHP

PHP 5: DJBX33A, 32 bit → equivalent substrings

PHP 4: DJBX33X, 32 and 64 bit → meet in the middle

default post_max_size: 8 MB

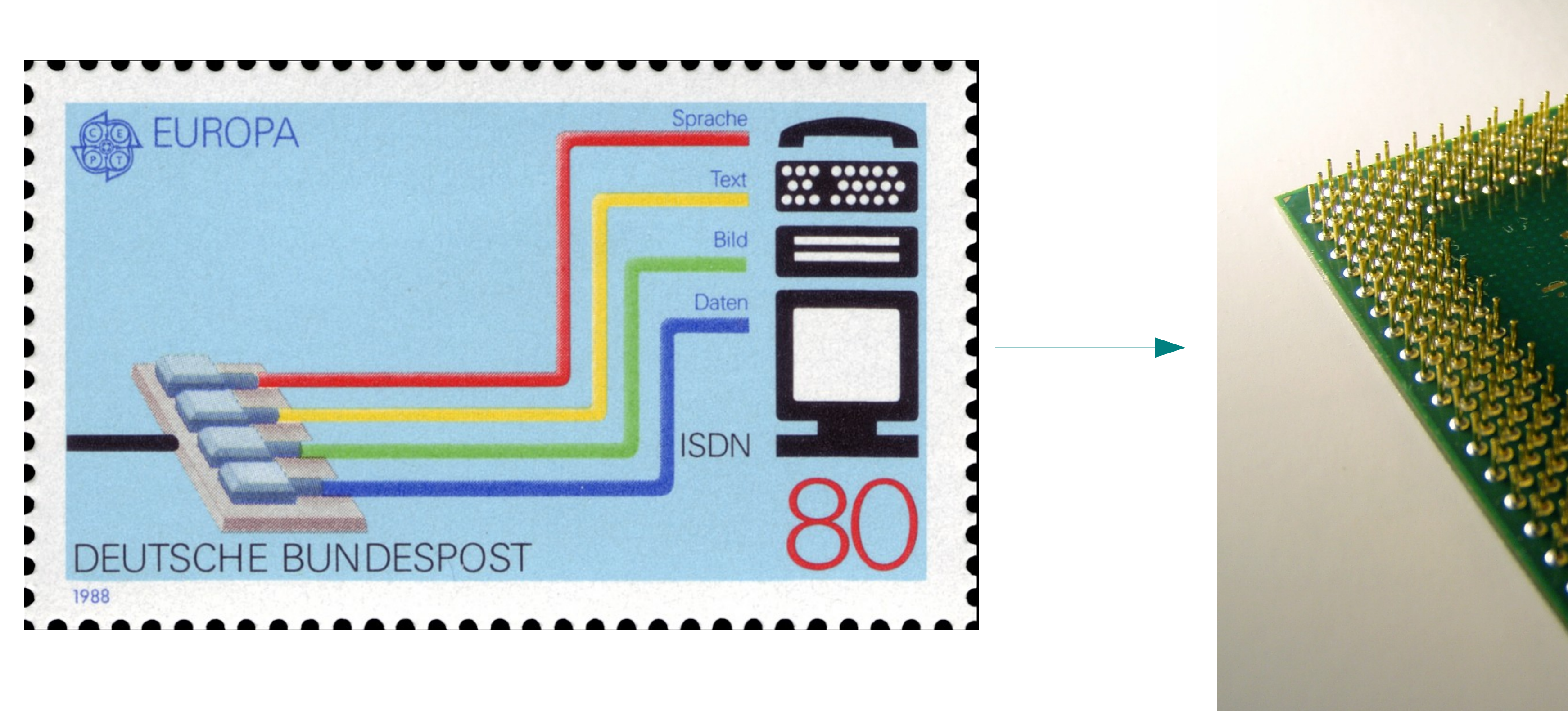
default max_input_time: -1 (unlimited/max_execution_time)

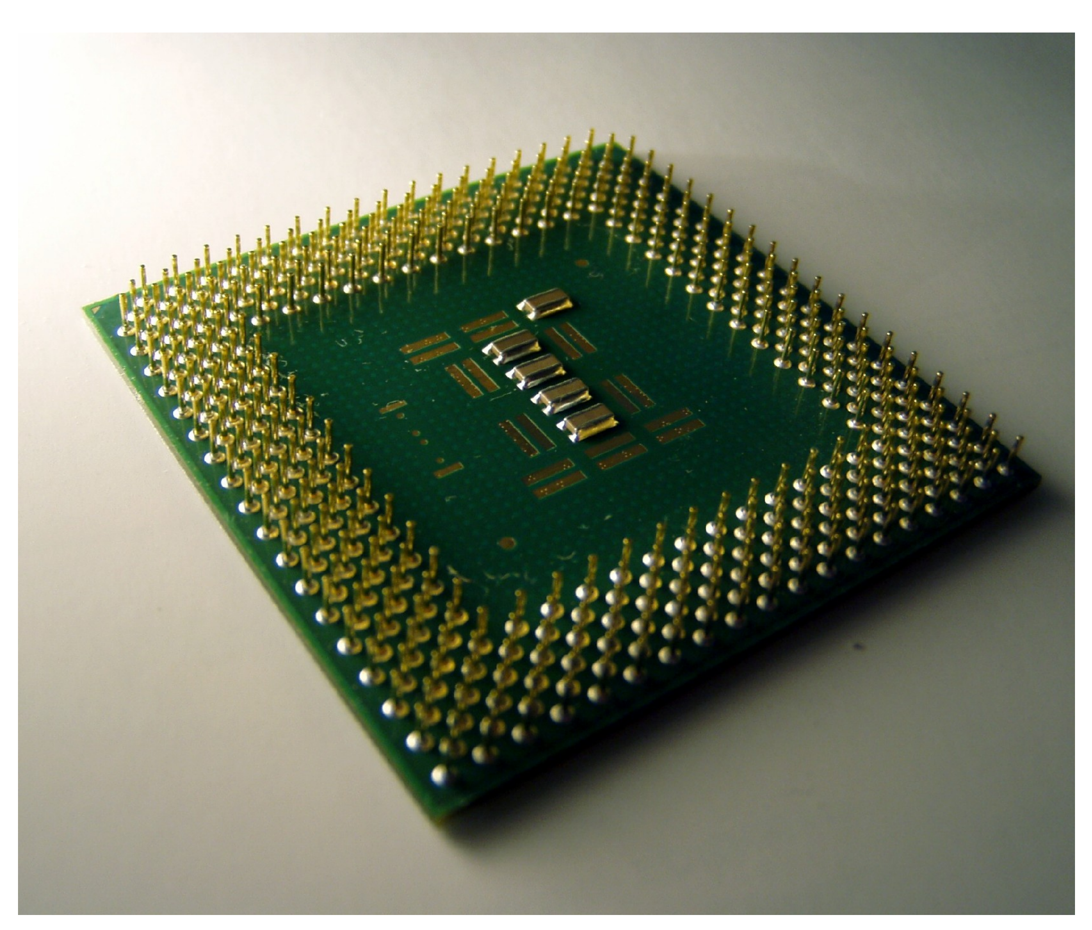
on most distributions: 60 (seconds)

theoretically: 8 MB of POST → 288 minutes of CPU time

realistically: 500k of POST → 1 minute or 300k → 30 secs

PHP: (realistic) efficiency





~70-100kbits/s → keep one i7 core busy

PHP: (realistic) effectiveness

1 Gbit/s → keep ~10.000 i7 cores busy

PHP: disclosure state

disclosed November 1st via oCERT
request for update on November 24th:

"We are looking into it. Changing the core hash function in PHP isn't a trivial change and will take us some time."

– *Rasmus Lerdorf*

PHP: disclosure state

December 15th:
http://svn.php.net/viewvc?view=revision&revision=321040

Log:
Added max_input_vars directive to prevent attacks based on hash collisions

[...]

+- the following new directives were added
+
+ - **max_input_vars** - specifies how many GET/POST/COOKIE input variables may be
+ accepted. default value 1000.
+

ASP.NET

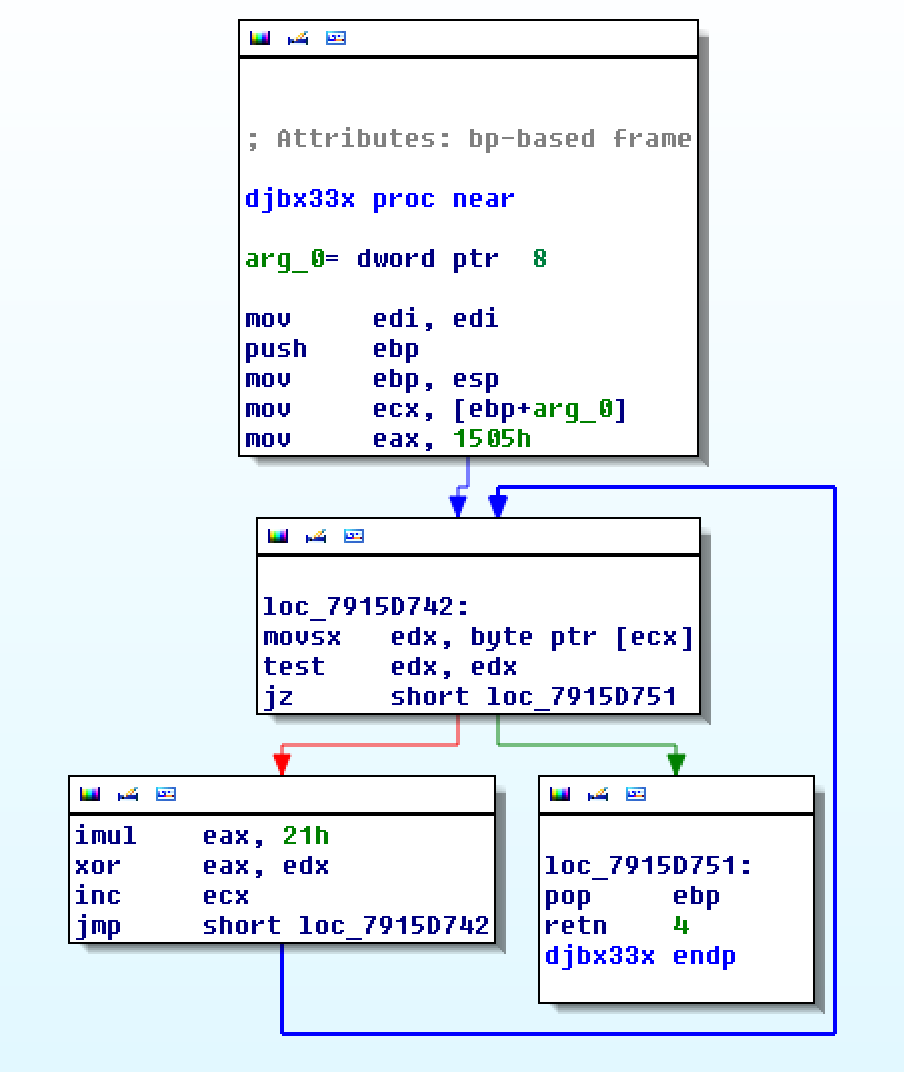
Request.Form is a NameValueCollection object

uses CaseInsensitiveHashCodeProvider.getHashCode()

DJBX33X → meet-in-the-middle

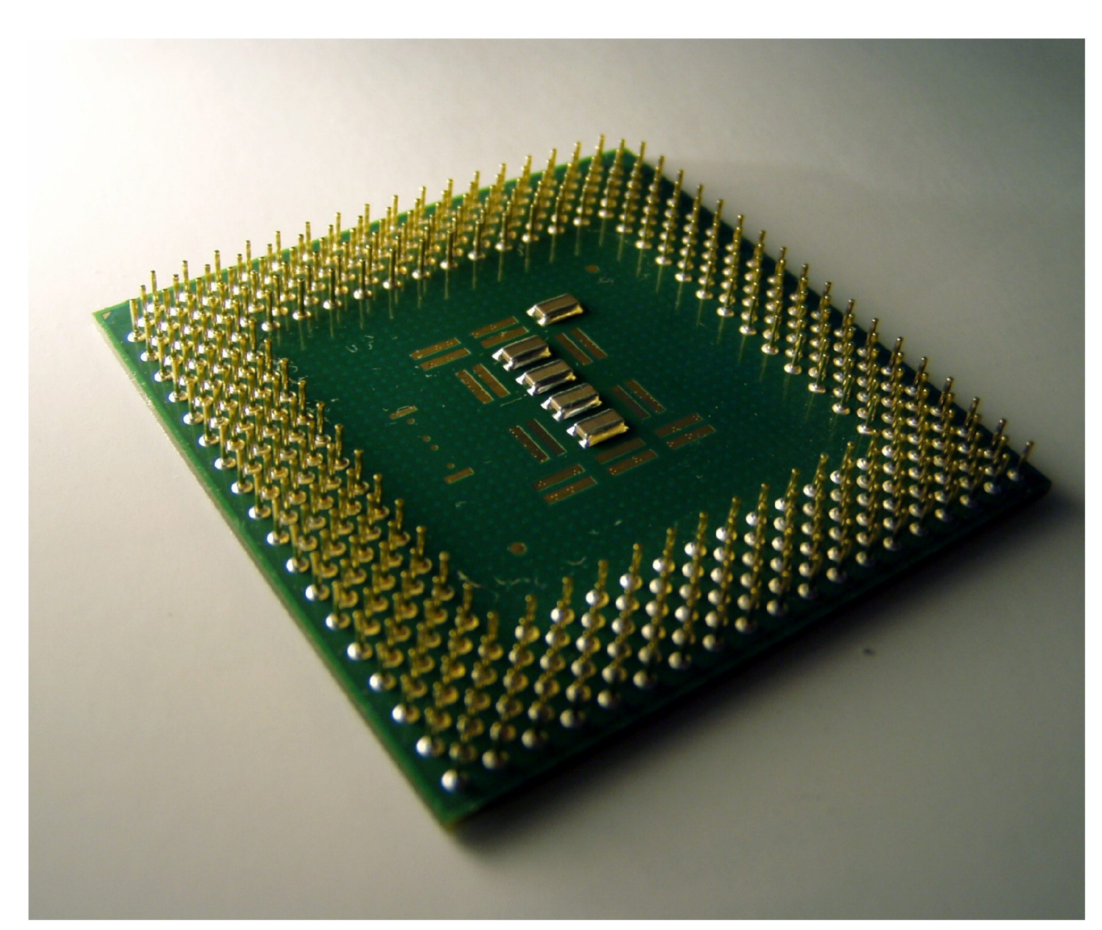
4 MB → 650 minutes of CPU time

IIS limits to 90 seconds typically



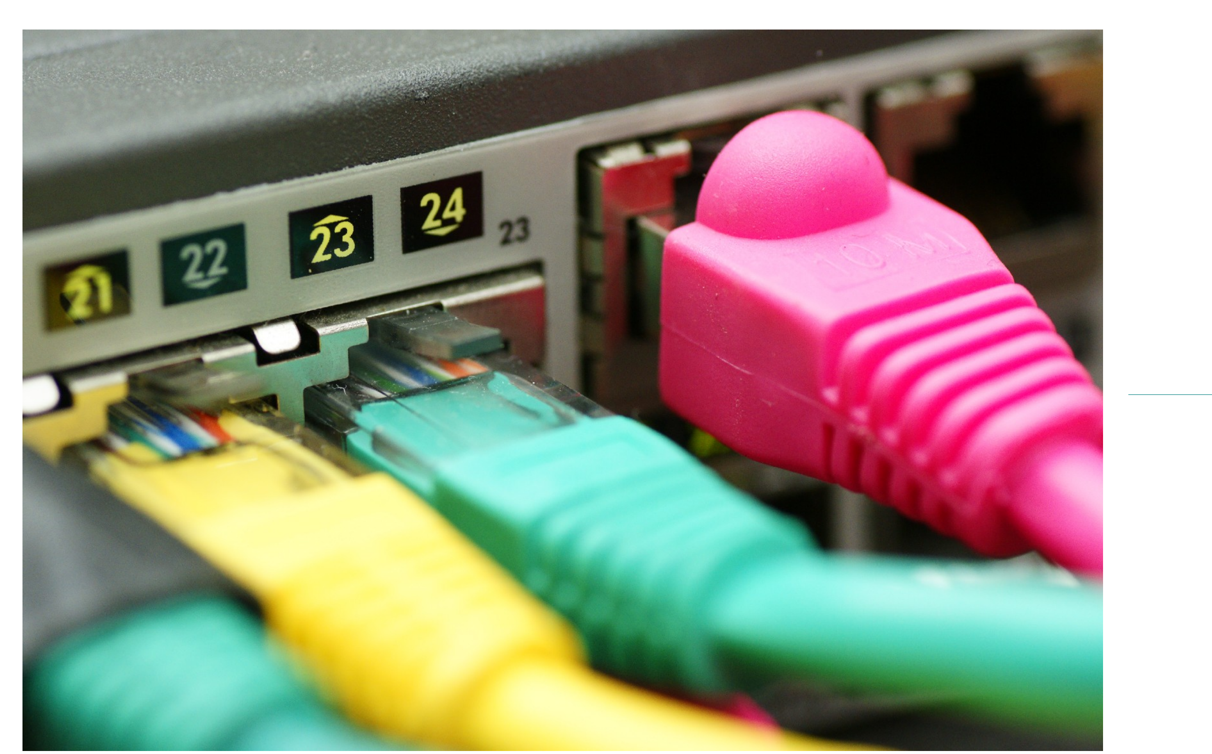
ASP.NET: efficiency

→



~30 kbits/s → keep one Core2 core busy

ASP.NET: effectiveness



1 dot ≈ 3 CPU cores

1 Gbit/s → keep ~30k Core2 cores busy

ASP.NET: disclosure state

disclosed November 29th via CERT
MSRC case number 12038

Working on a workaround patch (limiting number of parameters), randomizing hash function later

Advisory soon at http://technet.microsoft.com/en-us/security/advisory/2659883

Java

String.hashCode(), documented as $h(s) = \sum 31^{n-i} \cdot s_i$

very similar to DJBX33A → equivalent substrings

alternatively, meet in the middle for more collisions

hash result is cached, but only if hash ≠ 0

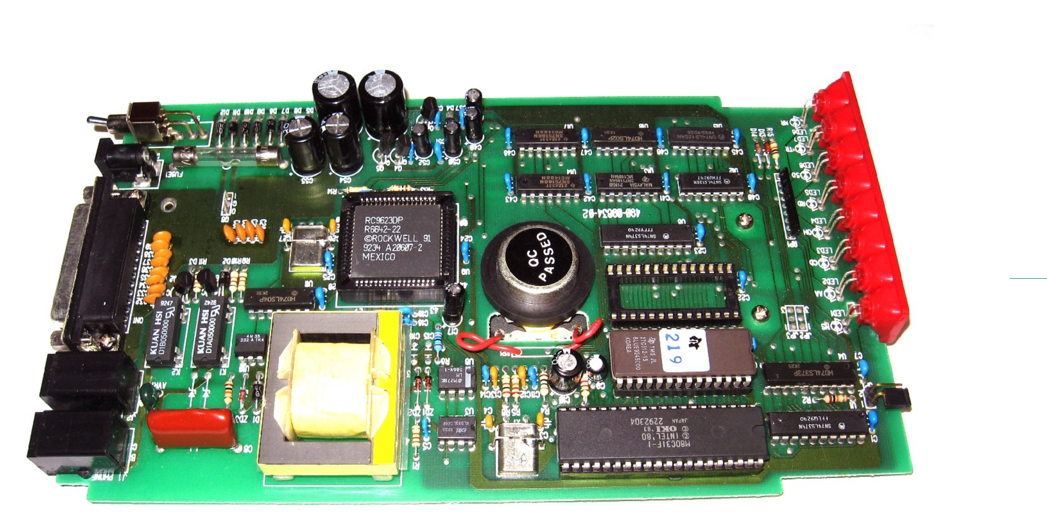
Java – Web Application Servers

- Apache Tomcat
- Apache Geronimo
- Jetty
- Oracle Glassfish
- …

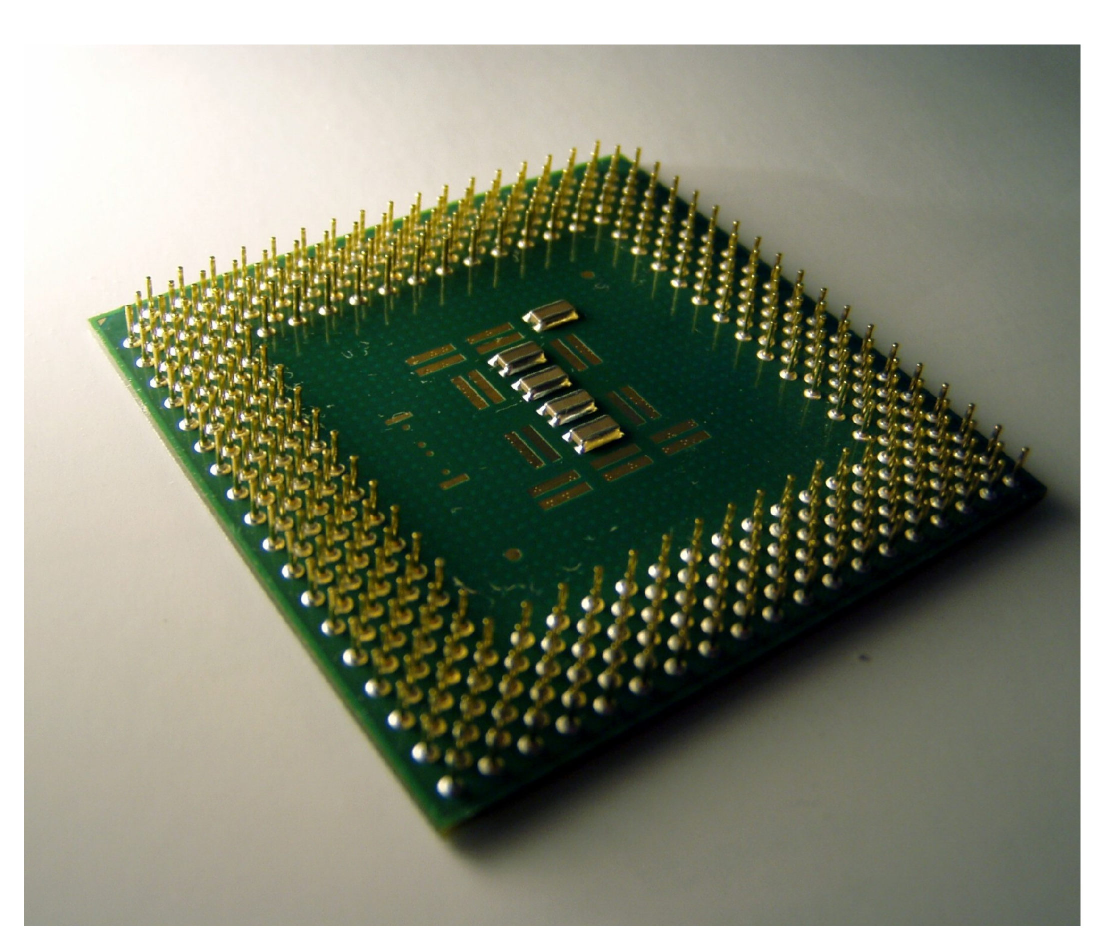
All tested ones use either Hashtable or HashMap to store POST data

Tomcat: 2 MB → 44 minutes of CPU time

Java (Tomcat): efficiency

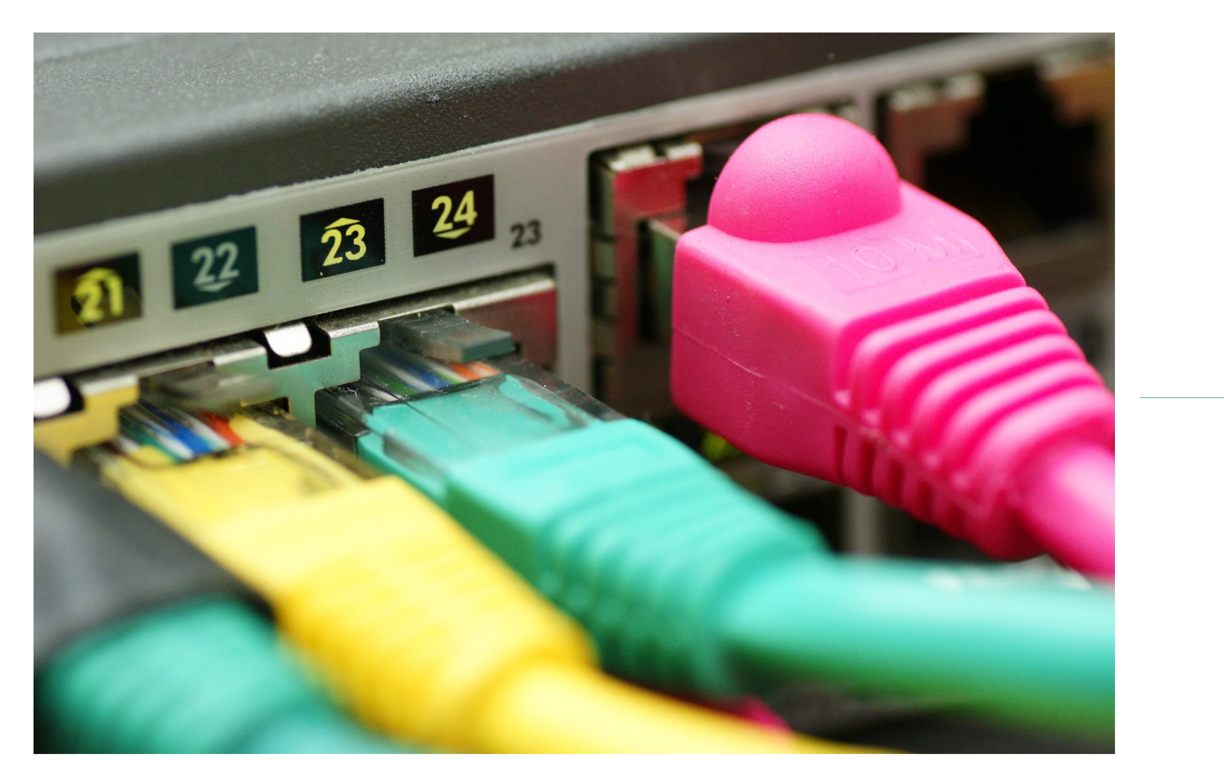


→



~6 kbits/s → keep one i7 core busy

Java (Tomcat): effectiveness



1 dot ≈ 10 CPU cores

1 Gbit/s → keep ~10^5 i7 cores busy

Java: disclosure state

disclosed November 1st via oCERT
Tomcat: workaround in r1189899 (CVE-2011-4084)
Glassfish: will be fixed in a future CPU (S0104869)

"As for Java itself, it does not seem like there is anything that would require a change in Java hashmap implementation."

– *Chandan, Oracle Security Alerts*

Python

hash function very similar to DJBX33X

works on register-size → different for 32 and 64 bits

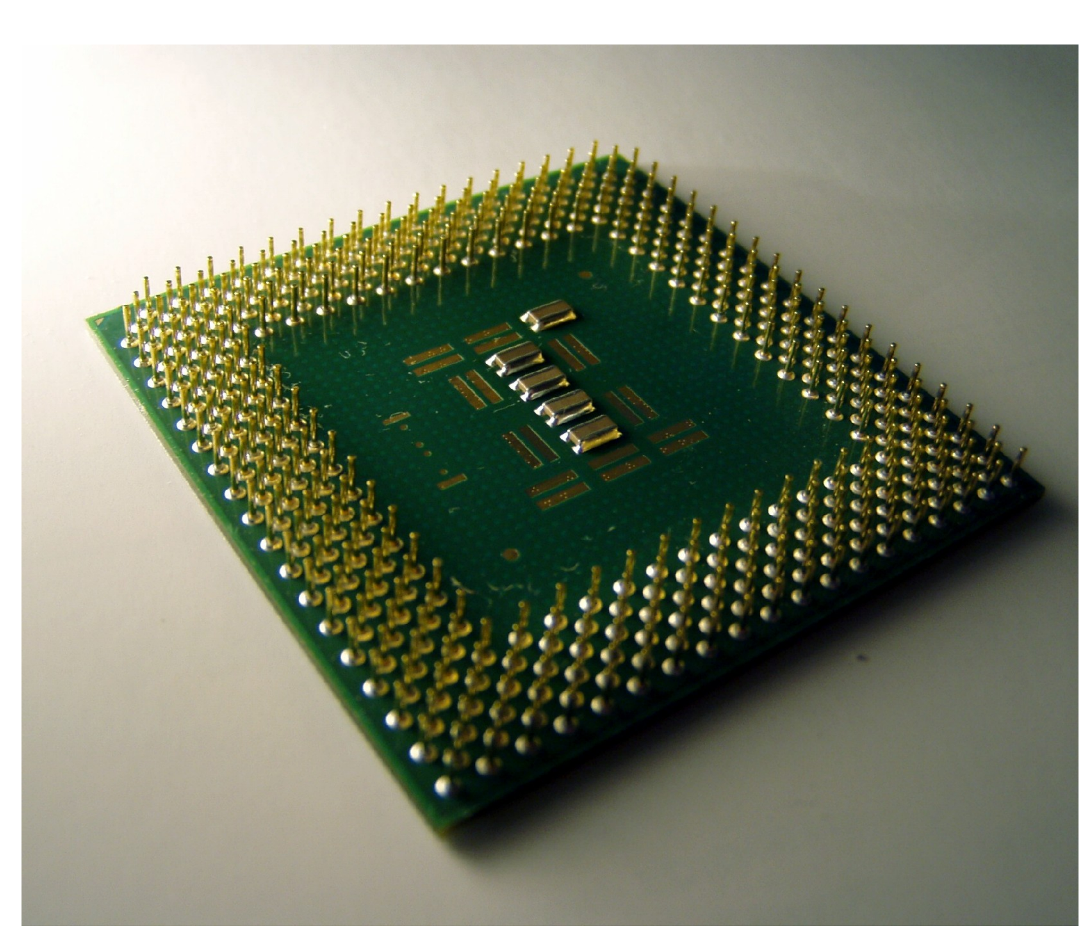
broken using a meet-in-the-middle attack

reasonable-sized attack strings only for 32 bits

Plone has max. POST size of 1 MB

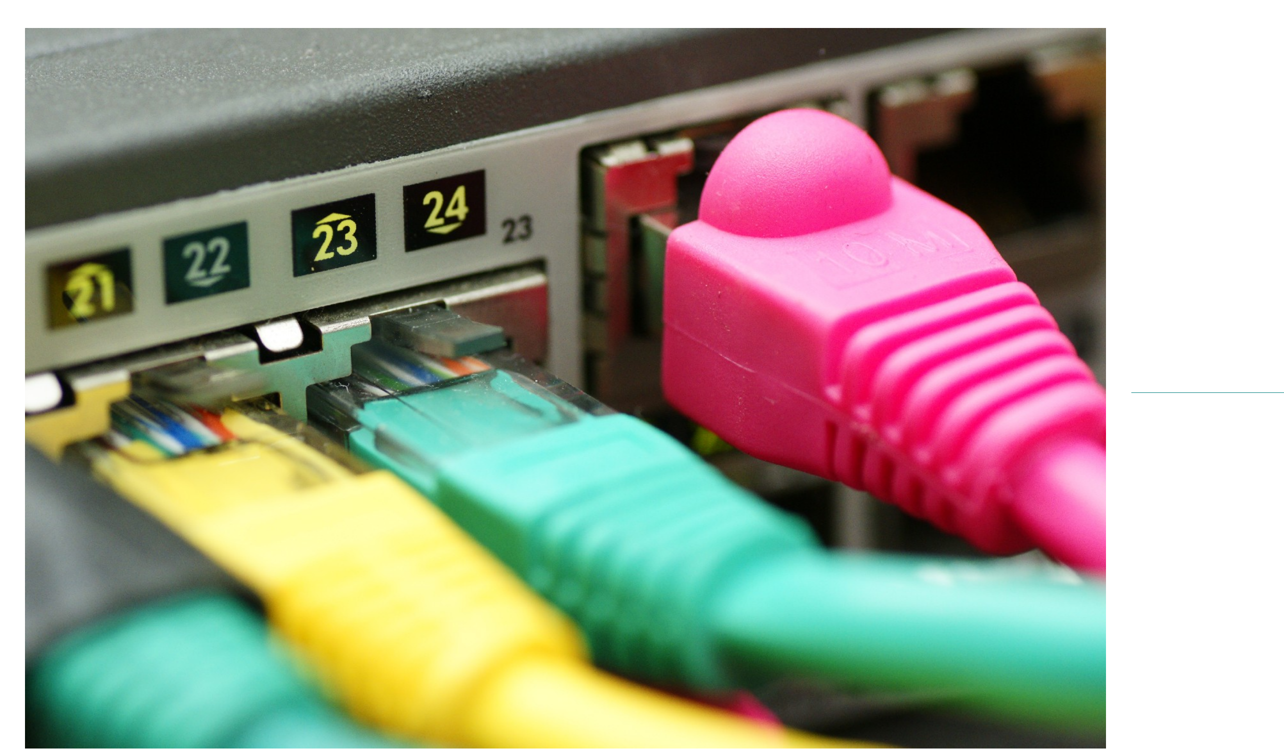
7 minutes of CPU usage for a 1 MB request

Python (Plone): efficiency



~20 kbits/s → keep one Core Duo core busy

Python (Plone) effectiveness



1 dot ≈ 5 CPU cores

1 Gbit/s → keep ~$5 \cdot 10^4$ Core Duo cores busy

Python: disclosure state

disclosed November 1st via oCERT
request for update on November 24th

"Apologies; this message got held in our moderation queue until just now. Because of the USA Thanksgiving holiday, it may be a few days before you get a response to this report."

– Barry Warsaw, Python

Ruby

Already fixed in 2008 in CRuby 1.9

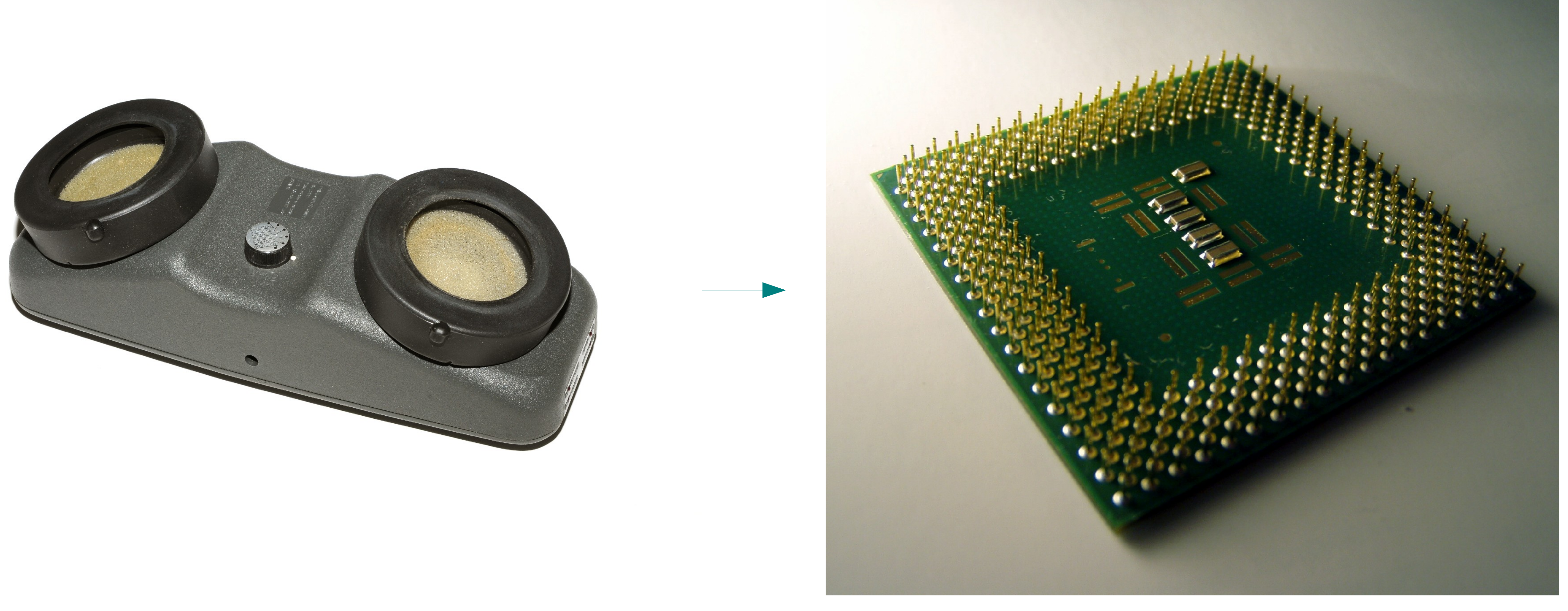
CRuby 1.8: similar to DJBX33A

But: multiplication constant 65599 prevents small equivalent substrings → meet in the middle attack

Different, but vulnerable functions in JRuby and Rubinius (for both 1.8 and 1.9)

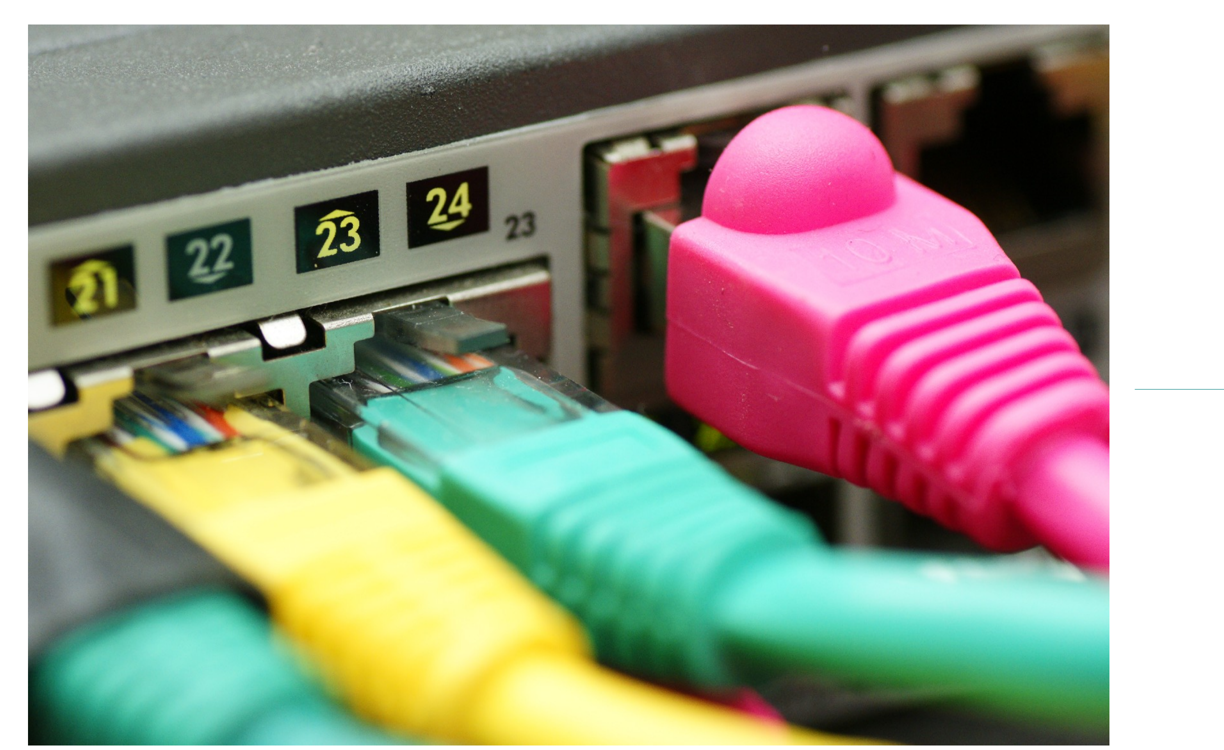
typical max. POST size limit of 2 MB → 6hs of CPU

CRuby 1.8 (Rack): efficiency



~720 bits/s → keep one i7 core busy

CRuby 1.8 (Rack) effectiveness



→

1 dot ≈ 100 CPU cores

1 Gbit/s → keep ~10^6 i7 cores busy

Ruby: disclosure state

disclosed November 1st via oCERT
Ruby Security Team *very* helpful!
New versions of CRuby and JRuby released →
new, randomized hash function, CVE-2011-4815
New version of Rack middleware

v8/node.js

Javascript implementation by Google

```
while (len--) {
    hash += *p++;
    hash += (hash << 10);
    hash ^= (hash >> 6);
}
```

Different than most other stuff, but vulnerable to meet-in-the-middle, too.

node.js: querystring module to parse POST into hashtable

v8: disclosure state

disclosed October 18th via oCERT
Google Security ticket #892388802

Privately contacted Google Security Team member on November 7th → ticket forwarded to Chrome/v8 developers

Web application security

Just a POST request ...

Can be generated on the fly using HTML and JavaScript

next XSS → lots of DDoS participants

Hash tables everywhere

Parsing code

Hash tables in your shell (bash):

```
declare -A hash
hash[foo]=”bar”
echo ${hash[foo]}
```

Live demo, part IV

(we'll skip this and hope you believe us it is still running :-))

How to fix it

Use a randomized hash function!

CRuby 1.9 and Perl already do

```
+ * The "hash seed" feature was added in Perl 5.8.1 to perturb the results
+ * to avoid "algorithmic complexity attacks".
  */
+#if defined(USE_HASH_SEED) || defined(USE_HASH_SEED_EXPLICIT)
+#   define PERL_HASH_SEED     PL_hash_seed
+#else
+#   define PERL_HASH_SEED     0
+#endif
 #define PERL_HASH(hash,str,len) \
     STMT_START       { \
       register const char *s_PeRlHaSh_tmp = str; \
       register const unsigned char *s_PeRlHaSh = (const unsigned char
*)s_PeRlHaSh_tmp; \
       register I32 i_PeRlHaSh = len; \
-      register U32 hash_PeRlHaSh = 0; \
+      register U32 hash_PeRlHaSh = PERL_HASH_SEED; \
       while (i_PeRlHaSh--) { \
         hash_PeRlHaSh += *s_PeRlHaSh++; \
         hash_PeRlHaSh += (hash_PeRlHaSh << 10); \
diff --git a/intrpvar.h b/intrpvar.h
```

Workarounds

- Reduce maximal POST size
  - Typically supported everywhere (but not node.js?)
- Reduce maximal parameters allowed
  - Tomcat, Suhosin: suhosin.{post|request}.max_vars
- CPU limits
  - PHP: reduce max_input_time
  - IIS for ASP.NET: shutdown time limit for processes
  - Typically not available on Java Web Application Servers

Future Work

Linux Kernel

grep -r hashtable linux-3.1.5/

(282 hits)

JSON, YAML, … (AJAX)

What will be put in an hash table?

Other Stuff

- Erlang
- Objective C
- Lua
- GNU ELF binary symbol tables
- Facebook (hiphop-php)

Take Home Messages

Take home: Language Developers

Fix this – soon!

Randomize your hash functions!

Take home: Application developers

Think about whether attacker controlled data ends up in a hash table!

Use different datastructures such as treemaps, etc.

Take home: Penetration testers

Think about whether attacker controlled data ends up in a hash table!

Try to identify used hash functions by hashing the empty string or short strings

Take home: Anonymous

Thank You!

Andrea Barisani of oCERT for lots of coordinating work

CERT for coordinating

Perl for fixing this in 2003

Scott A. Crosby & Dan S. Wallach for the original paper

The Ruby Security Team for taking this seriously and working with us on a fix

Thanks!
Q & A
or later:

hashDoS@alech.de

@hashDoS

@alech

@zeri42