

COMP 4384 Software Security

Module 9: *Algorithmic Complexity and Side-Channel Attacks*

Ahmed Tamrawi



atamrawi



atamrawi.github.io



ahmedtamrawi@gmail.com

Acknowledgment Notice

Part of the slides are based on content from Wikipedia and publicly available articles.

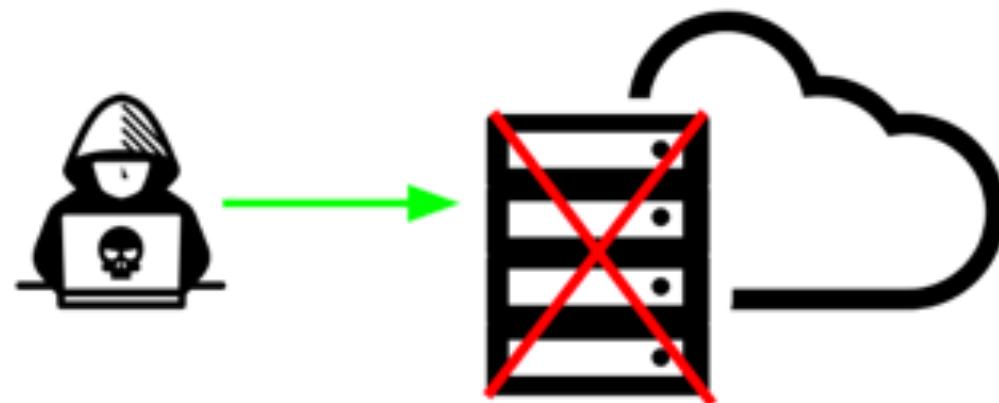
Algorithmic Complexity Vulnerabilities

Algorithmic Complexity Attack (AC)

- An **Algorithmic Complexity** (AC) attack is a *resource exhaustion attack* that takes advantage of *worst-case performance in server-side algorithms*. This type of attack can be used to achieve a denial-of-service.
- Developers select algorithms for performance, for ease of implementation, or because they're the top answer on StackOverflow.
- Most developers test their algorithms for *average-case performance*, checking against the kinds of inputs a typical user would provide.

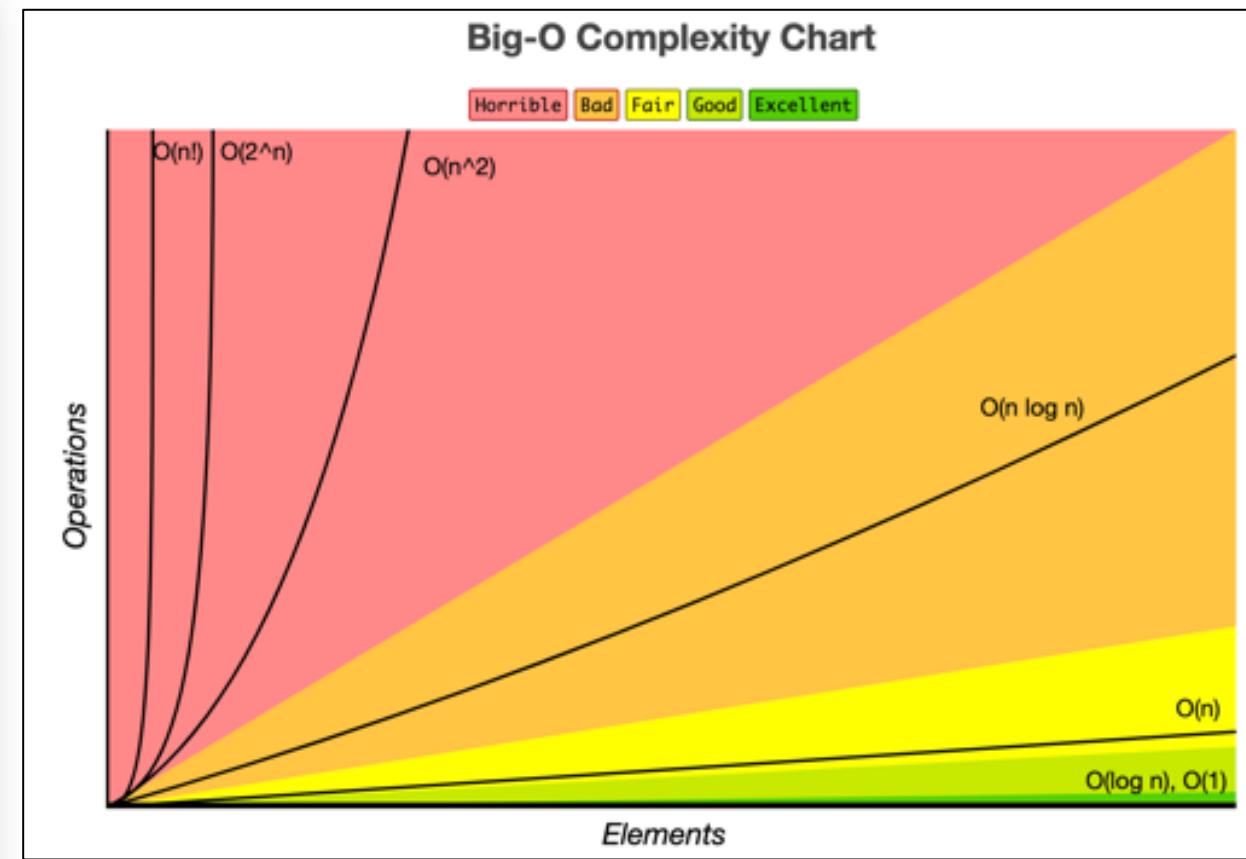
Algorithmic Complexity Attack (AC)

- Algorithmic complexity vulnerabilities arise when the **worst-case performance** for a back-end algorithm results in **resource exhaustion of the server**.
- AC vulnerabilities come in a few flavors.
 - An **AC Time** vulnerability causes denial of service by exhausting CPU.
 - An **AC Space** vulnerabilities exhaust RAM or disk space.



Algorithmic Complexity Attack (AC)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \ log(n))$	$\Theta(n \ log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \ log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \ log(n))$	$O(n \ log(n))$	$O(n)$



Attacker Goal: Find corner case input to trigger worst-case performance!

How do AC Vulnerabilities Differ from Other DoS Attacks?

- In a typical **distributed denial-of-service** (DDOS) attack, the attacker must dedicate *significant resources* to the attack.
 - Attackers will most commonly use a **botnet of thousands or millions of nodes**, each of which initiates a **conversation with the target server**.
 - In this case, there is a *symmetric effort* on part of the attacker vs. the effect on the target.
- In contrast, AC attacks can typically be conducted by a **single user**, with a **relatively small payload**, to cause a disproportionately powerful effect.
- AC vulnerabilities are **much cheaper** than a traditional DDOS attack.

How do AC Vulnerabilities Differ from Other DoS Attacks?

	Effort	Effect
		
AC 		

How do AC Vulnerabilities Differ from Other DoS Attacks?

- Additionally, unlike most DDOS attacks, AC vulnerabilities can be “quieter” than traditional denial of service attacks.
 - Because AC vulnerabilities arise from **intended functionality**, normal indicators of compromise, such as thrown exceptions, unusually high traffic, and excessive logging, may not be present.
- Many AC Time attacks cause **temporary denial of service**, with normal functionality resuming afterwards. This allows AC attacks to escape notice, flying under the standard cybersecurity radar.

AC Attack Example: *Hashtable DoS Attacks*

- In 2011, researchers Alexander ‘alech’ Klink and Julian ‘zeri’ Wälde found vulnerabilities in **several hash table implementations**, including the built-in hash tables in Java, PHP, and Python.

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.

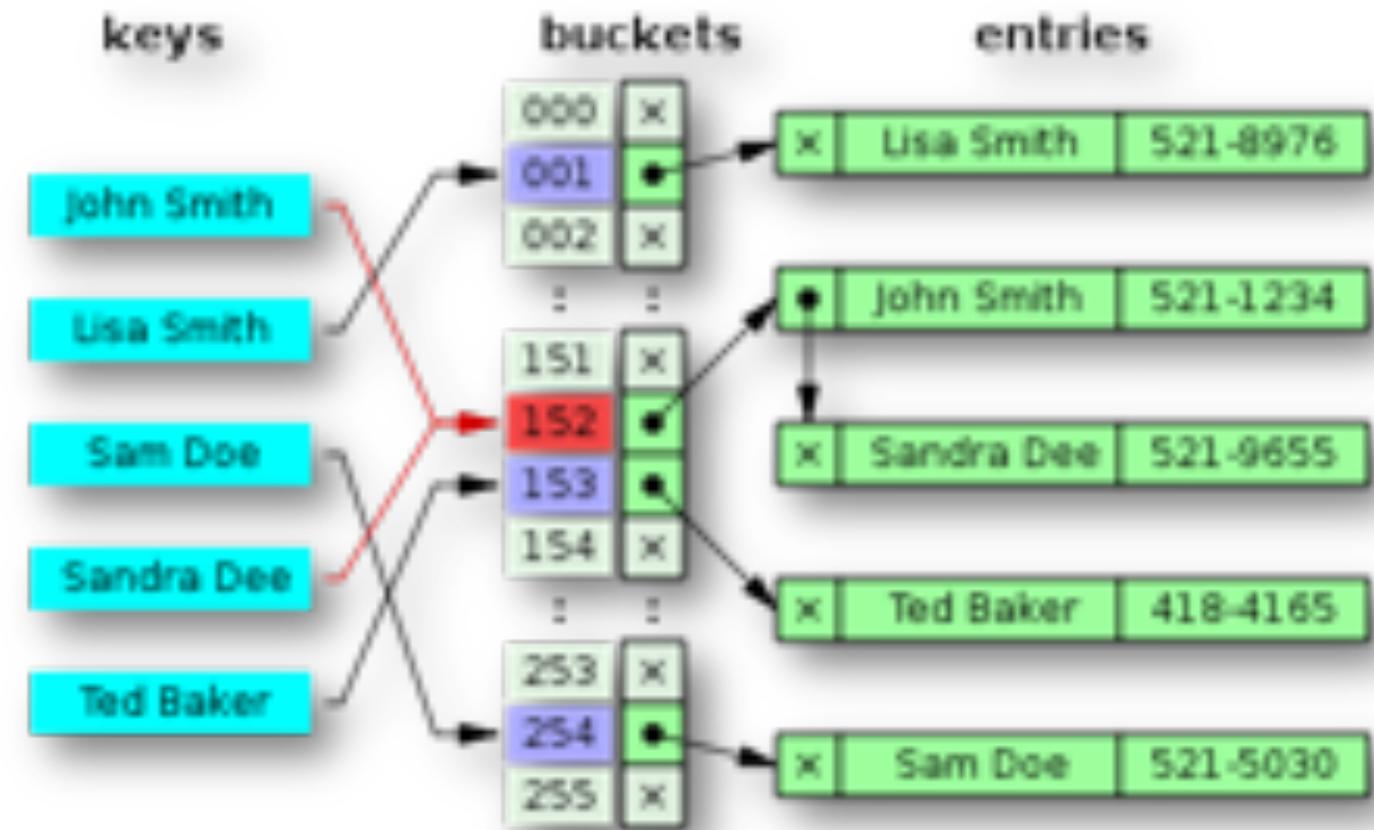
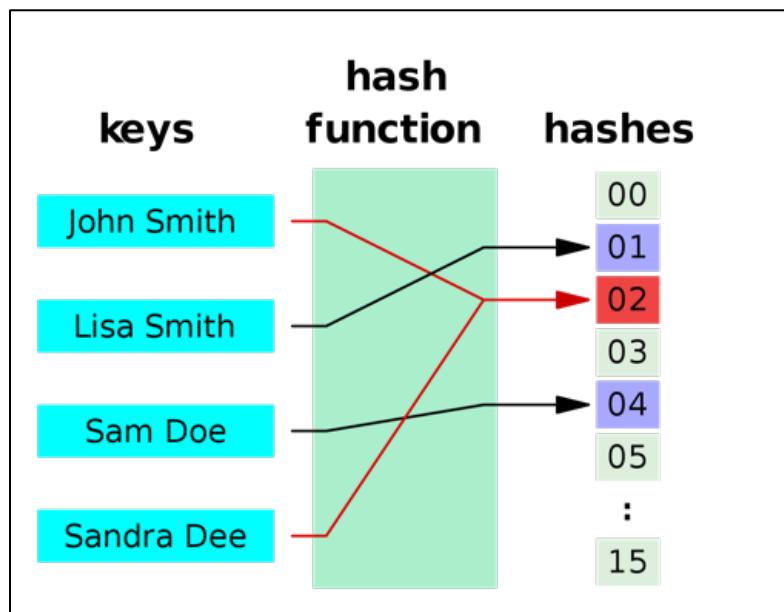
The worst case in real life

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

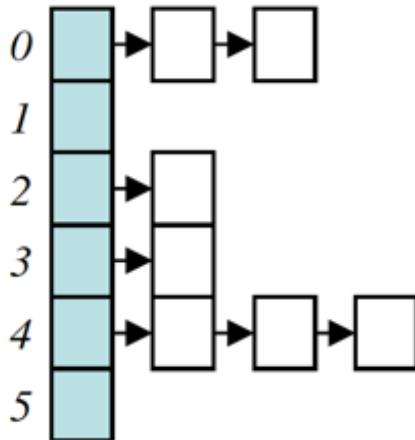
AC Attack Example: *Hashtable DoS Attacks*

Vulnerable hash tables implementations utilized a linked list for **storing hash collisions***.

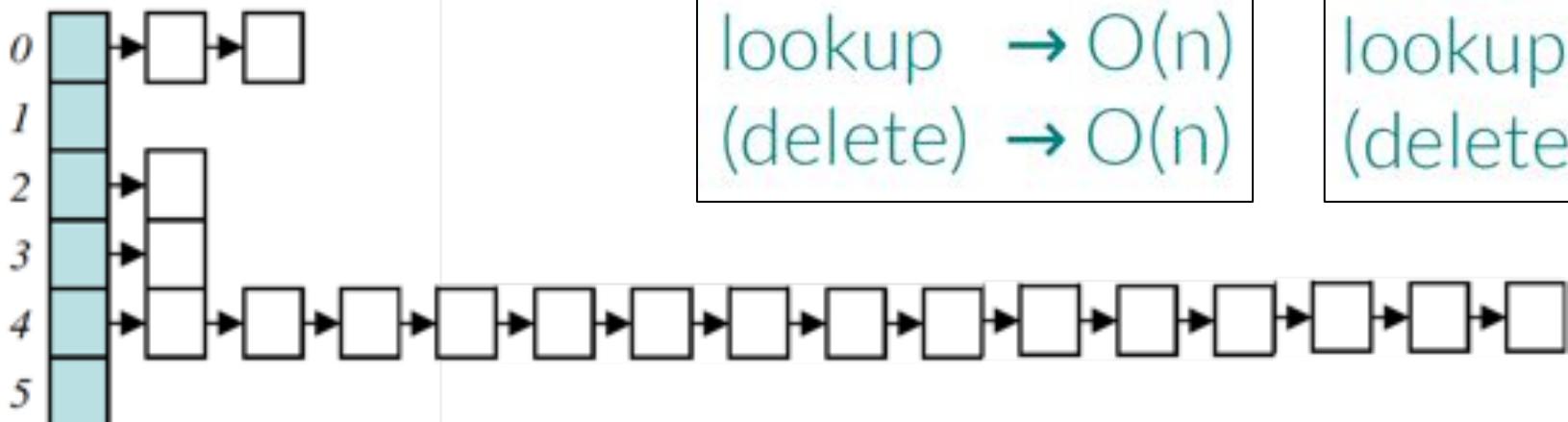


AC Attack Example: *Hashtable DoS Attacks*

Bucket



Bucket



By creating inputs that *collide under the hash functions*, an attacker can insert arbitrarily many hash table keys into **the same linked list**.

Best/Average Case

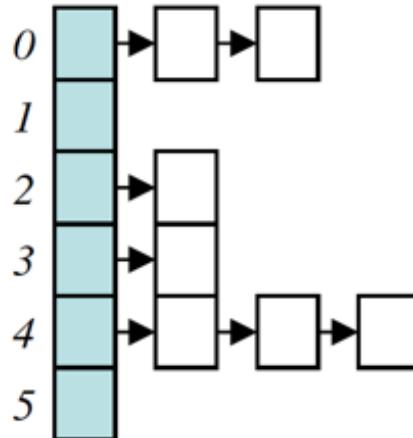
n elements:
insert $\rightarrow O(n)$
lookup $\rightarrow O(n)$
(delete) $\rightarrow O(n)$

Worst Case

n elements:
insert $\rightarrow O(n^2)$
lookup $\rightarrow O(n^2)$
(delete) $\rightarrow O(n^2)$

AC Attack Example: *Hashtable DoS Attacks*

Bucket



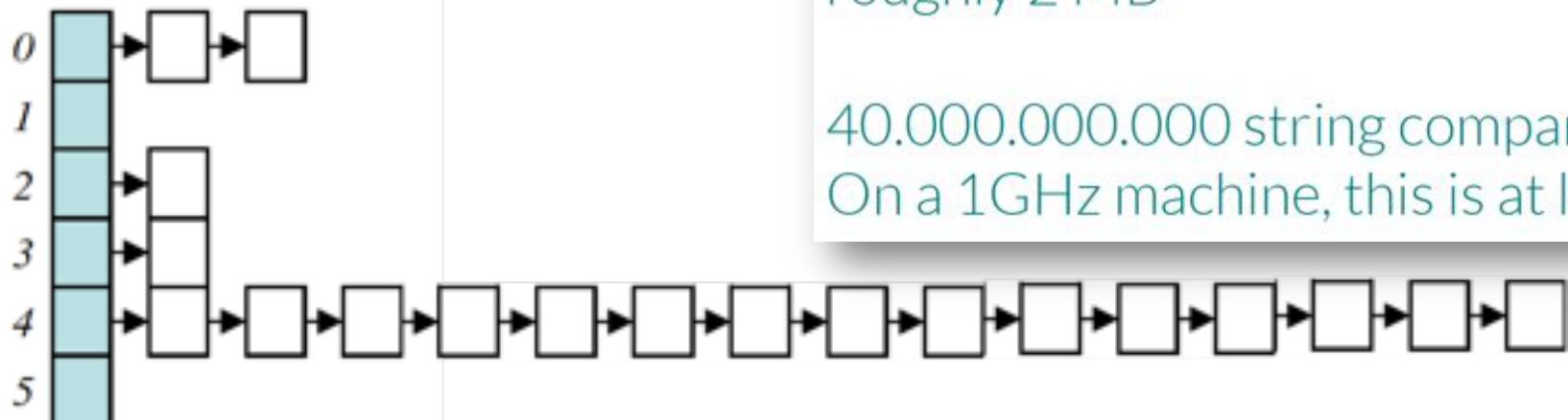
Best/Average Case

n elements:
insert → $O(n)$
lookup → $O(n)$
(delete) → $O(n)$

Worst Case

n elements:
insert → $O(n^2)$
lookup → $O(n^2)$
(delete) → $O(n^2)$

Bucket



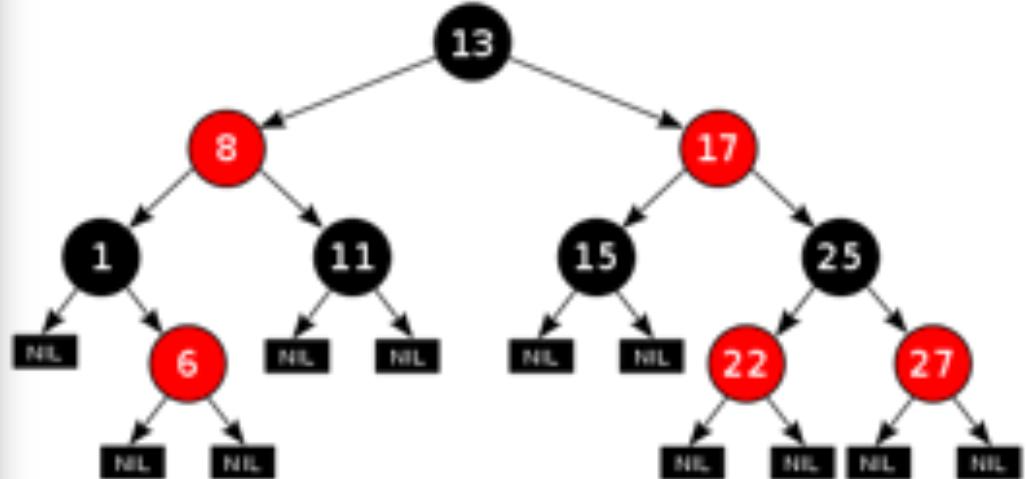
200.000 multi-collisions à 10 bytes
roughly 2 MB

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

Hashtable DoS Attacks Mitigation

- In response to these Hashtable Dos attacks, the developers of the affected languages made fundamental changes to their hash table implementations. Java, for instance, switched from **linked lists** to **balanced red-black trees**.

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log(n))$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
B-Tree	$\Theta(\log(n))$	$\Theta(n)$								
Red-Black Tree	$\Theta(\log(n))$	$\Theta(n)$								
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	
AVL Tree	$\Theta(\log(n))$	$\Theta(n)$								
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	



AC Attack Example: *Hashtable DoS Attacks*



<https://www.youtube.com/watch?v=R2Cq3CLI6H8>

Generating Worst-Case Inputs

- ACsploit is an interactive command-line utility to generate worst-case inputs to commonly used algorithms.
- These worst-case inputs are designed to result in the target program utilizing a large amount of resources (e.g., AC time or AC space).
- Acsploit is publicly available at:
<https://github.com/twosixlabs/acsploit>

```
(acsploit) help
Documented commands (type help <topic>):
=====
help history info options quit reset run set show use

(acsploit) show
Available exploits:
  bombs/compression/gz
  bombs/compression/tar_gz
  bombs/compression/zip
  bombs/fork/fork_bombs
  bombs/git/git_bombs
  bombs/images/png
  bombs/xml/billion_laughs
  geometry/convex_hull
  graphs/min_span_tree/kruskal
  hashes/collisions/adler32
  hashes/collisions/bad
  hashes/collisions/custom_hash
  hashes/collisions/fletcher
  hashes/collisions/java
  hashes/collisions/lrc
  hashes/collisions/sum
  hashes/collisions/xor8
  hashes/collisions/z3_java
  linear_programming/simplex
  sort/bubblesort
  sort/bucketsort
  sort/insertionsort
  sort/mergesort
  sort/quicksort
  strings/matching/boyer_moore
  strings/matching/knuth_moore_pratt
  strings/matching/rabin_karp
  tree/avl_tree
  tree/b_tree
  tree/binary_search_tree
  tree/heap
  tree/huffman
  tree/red_black_tree

(acsploit) use hashes/collisions/bad
exploit => hashes/collisions/bad
(acsploit : hashes/collisions/bad) options
  output: stdout
  Output options
    separator: newline
    number_format: decimal
  Exploit options
    n_collisions: 10
    length: 10
    hash_table_size: 65535
    target_type: preimage
    target: hello

(acsploit : hashes/collisions/bad) set exploit.n_collisions 3
exploit.n_collisions => 3
(acsploit : hashes/collisions/bad) run
Running hashes/collisions/bad...
#0#D#b#||{|
|<AA1<-|-|
|<AA1>|||=|
Finished running hashes/collisions/bad
(acsploit : hashes/collisions/bad)
```

AC Attack Example: *Decompression Bombs*

- Decompression bombs (aka "zip of death") exploit the ability of **efficient compression algorithms** to compress a large amount of repeated data into a small package. It is often employed to disable antivirus software, in order to create an opening for more traditional viruses.
- Decompression bombs typically causes an **AC Space effect on the memory use of the file parser**: as the bomb is decompressed, it expands to consume all the system's memory.
- Modern parsers offer some protections against decompression bombs, e.g. (optional) safeguards and sandboxes to limit resource consumption during parsing.

Vulnerable Vectors

- Chat clients
- Image hosting
- Web browsers
- Web servers
- Everyday web-services software
- Everyday client software
- Embedded devices (especially vulnerable due to weak hardware)
- Embedded documents
- Gzip'd log uploads

[root@netsec /]#

[comments](#)

other discussions (2)

↳ /r/netsec Q3 2016 Hiring Thread ↳



This is an archived post. You won't be able to vote or comment.

+ 420 bytes file that uncompresses to a 141.4 GB 225,000 × 225,000 pixels (50.625 gigapixels) PNG.
679 Upload as your profile picture to some online service, try to crash their image processing scripts. Set as
your web site's favicon; try to crash browsers that don't check the size.

OsmaSoftware.com]

submitted 10 months ago by [spectre](#)

81 comments share

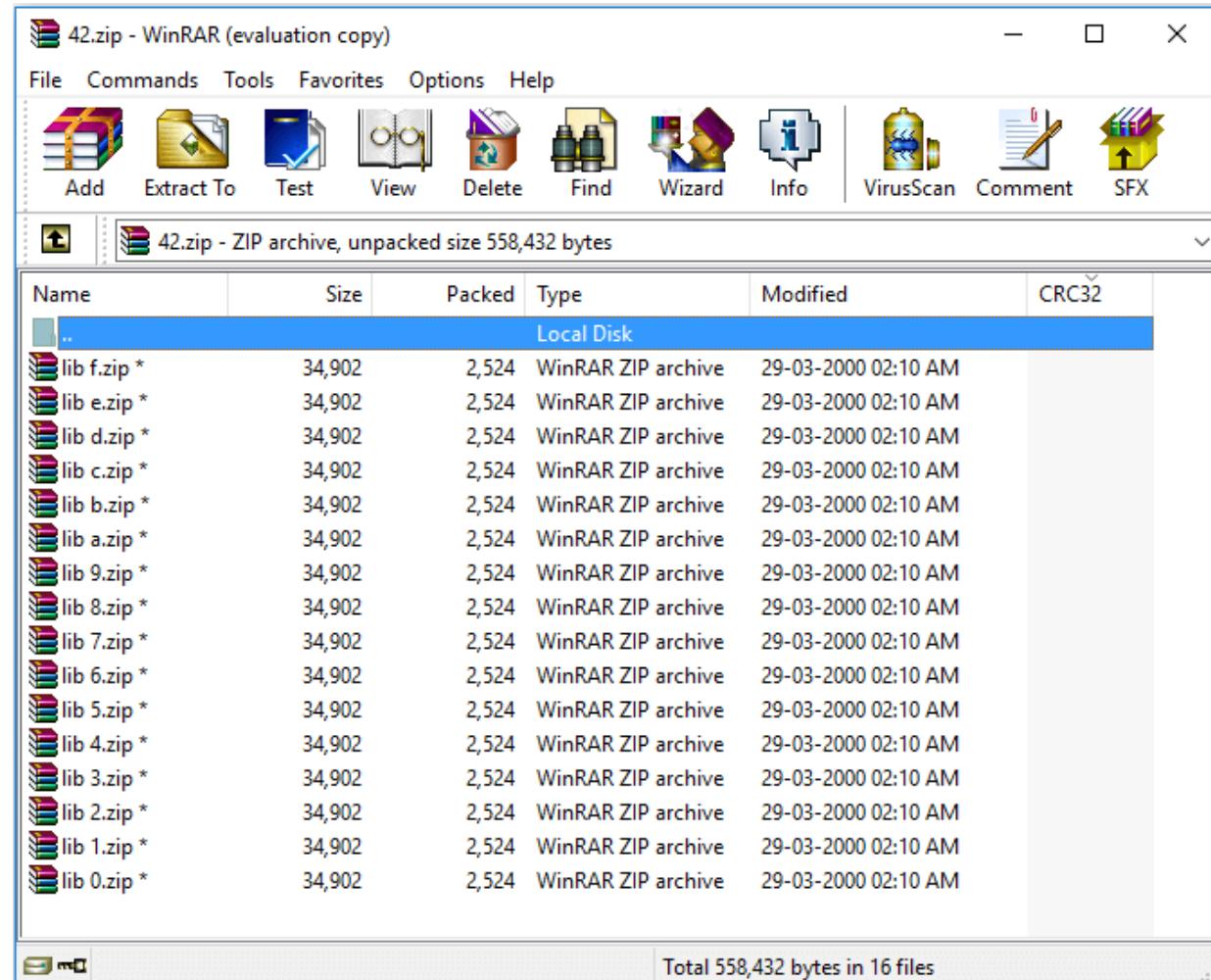
at 81 comments • sorted by: Best ▾

- [BeD1834633](#) 177 points 10 months ago
- Graphical version of a zip bomb...

permissions enabled score give gold

AC Attack Example: *Decompression Bombs*

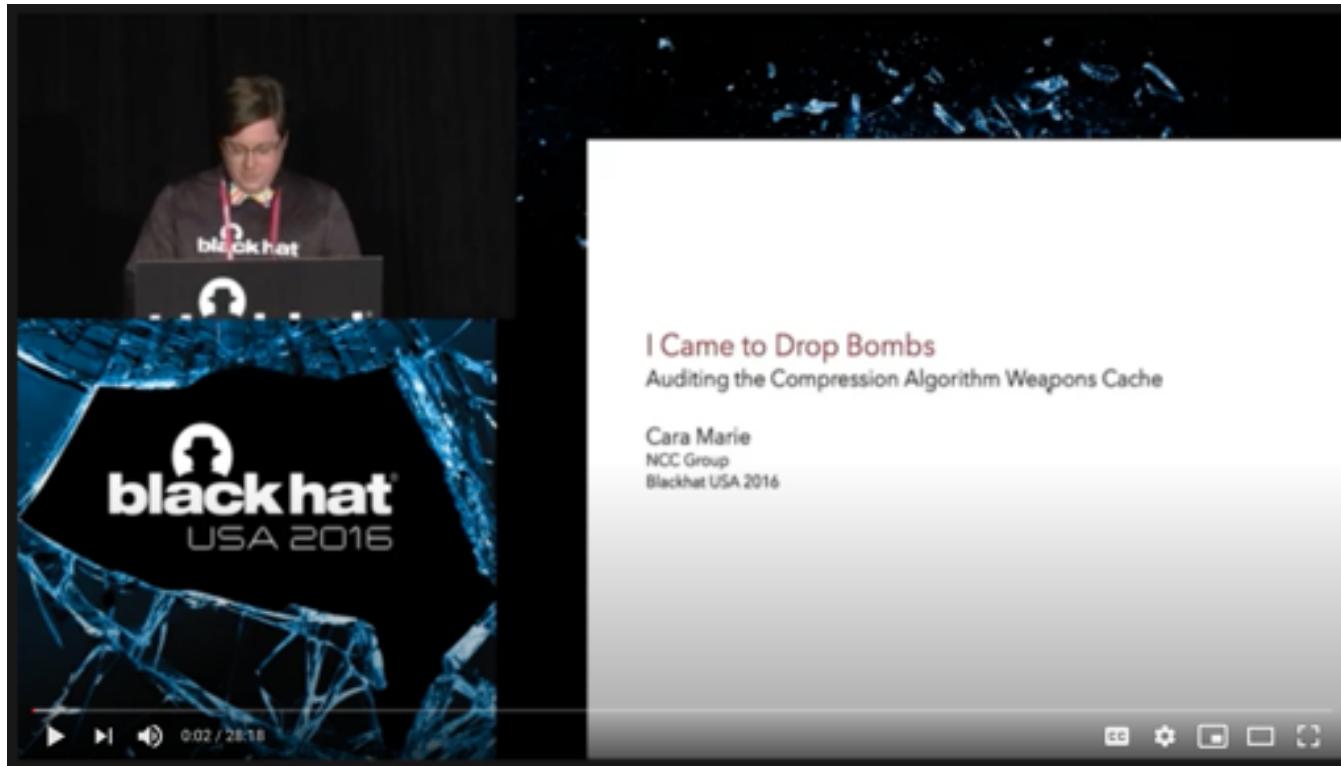
- One example of a zip bomb is the file 42.zip:
 - A zip file consisting of 42 kilobytes of compressed data.
 - It has five layers of nested zip files in sets of 16.
 - Each bottom-layer archive containing a 4.3-gigabyte file for a total of **4.5 petabytes of uncompressed data**.



Mitigations

- Restrict resources — place limits on processes and their children
- Don't rely on size alone — check image dimensions prior to rendering
- Restrict file size output — verify that the output file size won't max out storage
- Limit number of extracted files — calculate the file total to ensure that storage/processing power won't be overloaded
- Perform dynamic testing — always verify mitigations via manual testing to ensure that they are functioning properly
- *Archive bombs are decompression bombs, but not all decompression bombs are archive bombs.*

AC Attack Example: *Decompression Bombs*



<https://www.youtube.com/watch?v=IXkX2ojrKZQ>

More about Decompression Bombs

- One great resource for learning about decompression bombs is the website <https://bomb.codes/>.
- You can also generate decompression bombs using ACsploit.

Menu

The (Decompression) Bomb Site

=====

Home * ## What is a decompression bomb?

Bombs *

Mitigations *

A decompression bomb is a file designed to crash or render useless the program or system reading it, i.e. a denial of service. The following files can be used to test whether an application is vulnerable to this type of attack.

When testing, it's always better to start small and work your way up. Starting with the largest file available can seriously harm an application or system – use these bombs with caution.

> When you see something that is technically sweet, you go ahead and do it and you argue
> about what to do about it only after you have had your technical success. That is the
> way it was with the atomic bomb.

Format	<u>10GB</u>	<u>30GB</u>	<u>50GB</u>	<u>100GB</u>	<u>200GB</u>	<u>300GB</u>
<u>7z</u>	-	-	-	-	-	-
<u>bzip2</u>	-	-	-	-	-	-
<u>Gzip</u>	-	-	-	-	-	-
<u>LZ4</u>	-	-	-	-	-	-
<u>LZFSE</u>	-	-	-	-	-	-
<u>RAR</u>	-	-	-	-	-	-
<u>xar</u>	-	-	-	-	-	-
<u>XZ</u>	-	-	-	-	-	-
<u>ZIP</u>	-	-	-	-	-	-
<u>zstd</u>	-	-	-	-	-	-

AC Attack Example: *REDoS*

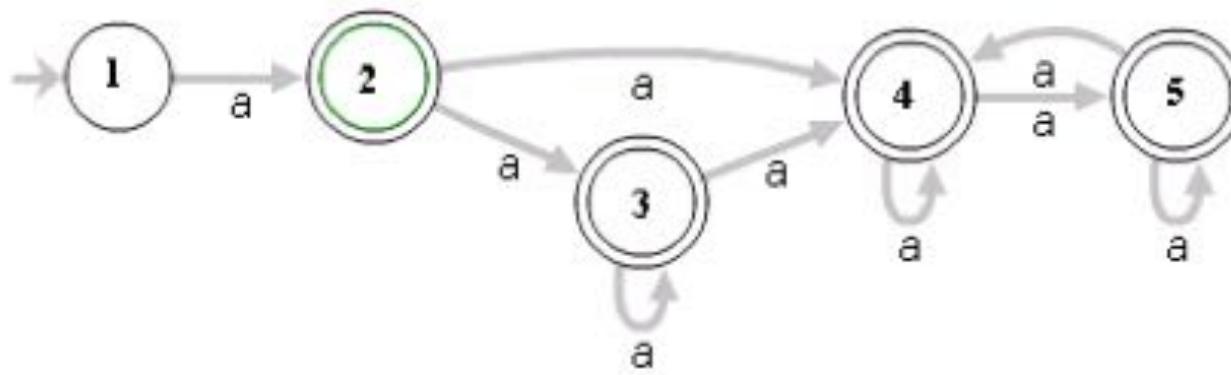
- REDoS, or **Regular Expression Denial of Service**, refers to a class of vulnerabilities in *regular expression parsing engines* that causes a denial-of-service attack.
- The REDoS attack exploits the fact that most **Regular Expression implementations** may reach **extreme situations** that cause them to work very slowly (exponentially related to input size).
- An attacker can then cause a program using a Regular Expression to enter these extreme situations and then hang for a very long time.

AC Attack Example: *REDoS*

- The Regular Expression naïve algorithm builds a **Nondeterministic Finite Automaton** (NFA), which is a finite state machine where for each pair of state and input symbol there may be several possible next states. Then the engine starts to make transition until the end of the input.
- Since there may be several possible next states, a deterministic algorithm is used. This algorithm tries one by one all the possible paths (if needed) until a match is found (or all paths are tried and fail).

AC Attack Example: *REDoS*

- For example, the Regex $^((a+)^*)^*$ is represented by the following NFA:



- For the input `aaaaX` there are 16 possible paths.
- For the input `aaaaaaaaaaaaaaaaaaX` there are 65536 possible paths, and the number is double for each additional `a`.
 - This is an extreme case where the naïve algorithm is problematic, because it must pass on many many paths, and then fail.

AC Attack Example: *REDoS*

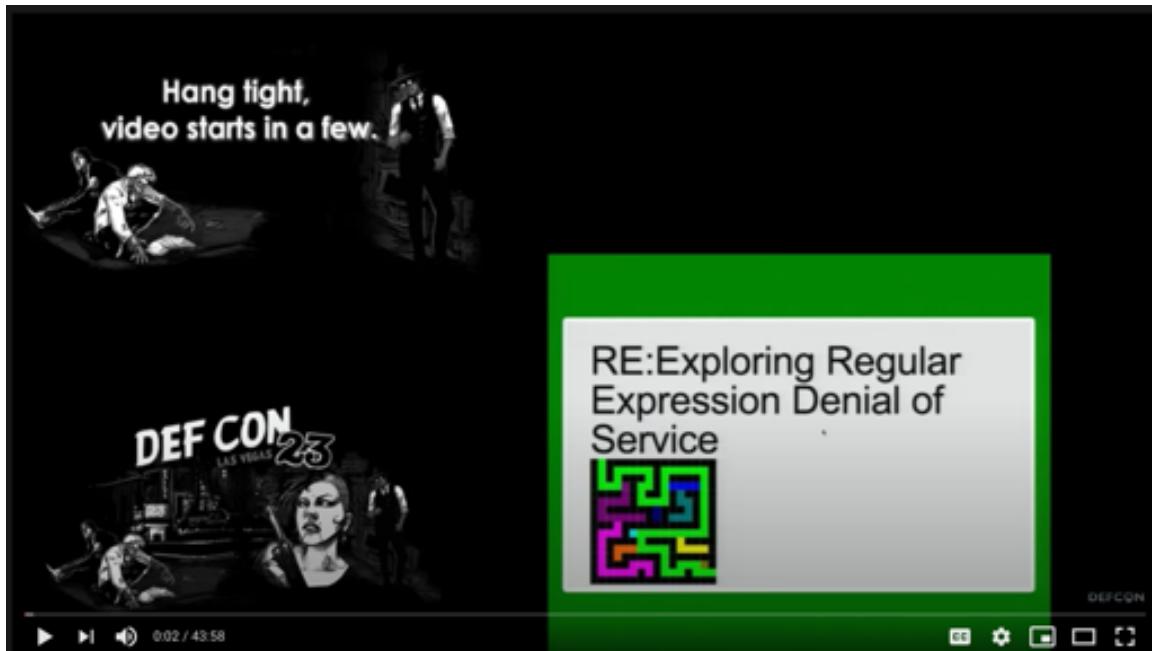
The screenshot shows a screenshot of the regex101.com website. The interface includes a sidebar with icons for saving, sharing, and testing regular expressions across various platforms like PCRE (PHP), ECMAScript (JavaScript), Python, and Golang. The main area displays a regular expression pattern `/^(a{100}+)/` and a test string consisting of 100 'a' characters followed by an 'X'. The status bar at the bottom right indicates "no match, 196608 steps (~155ms)".

<https://regex101.com/>

Evil Regex

- They are the regular expressions that make an application vulnerable to ReDoS attacks, they occur whenever these factors occur:
 - The regular expression applies repetition (“+”, “*”) to a complex subexpression
 - for the repeated subexpression, there exists a match which is also a suffix of another valid match.
- Examples of malicious regexes include the following:
 - $(a^+)^+$
 - $([a-zA-Z]^+)^*$
 - $(a|aa)^+$
 - $(a|a?)^+$
 - $(.^*a)\{x\}$ for $x > 10$

More about ReDoS



<https://www.youtube.com/watch?v=Hbih2lG2v0s>

The Regular Expression Denial of Service (ReDoS) cheat-sheet



James Davis [Follow](#)

May 23 · 11 min read



Introduction

This post is intended as a “technical two-pager” to summarize a security vulnerability called Regex-based Denial of Service (AKA *Regex DoS*, *ReDoS*). There are a variety of write-ups about ReDoS, but I’m not aware of a good one-stop-shop with a higher-level treatment of all aspects of the subject. I have included links at the end to more detailed treatments.

I have used headings liberally to help you navigate to your issue.

<https://levelup.gitconnected.com/the-regular-expression-denial-of-service-redos-cheat-sheet-a78d0ed7d865>

More about REDoS

- REDoS has been a well-known issue for many years at this point, so it may surprise you to hear that some high-profile applications still fall victim to this class of vulnerability.
 - In 2016, **StackExchange** experienced a half hour outage due to a bad regex. Remarkably, the StackExchange team was able to resolve the issue without consulting StackOverflow. You can read their post-mortem
 - Article: <https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016>
 - More recently, on July 2, 2019, **Cloudflare** experienced a blackout due to a poorly implemented regex. You can read their post-mortem
 - Article: <https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/>

Eliminating ReDoS Vulnerabilities

- Avoid using regex
 - The most foolproof way of avoiding ReDoS attacks is to avoid using regex. Instead, find alternative methods that can achieve the same results.
- Use safe regex engines
 - Instead of using built-in, unsafe regex engines (like the Node.js regex engine), you can opt to use safe alternatives instead.
 - For example, re2 (<https://www.npmjs.com/package/re2>) is a safe alternative that you can use without fear of ReDoS.
- Detect and sanitize evil regexes
 - You can also prevent ReDoS by detecting evil regex in your code or in user input, then sanitizing them.
 - For this, you can utilize evil regex detection libraries like safe-regex:
<https://www.npmjs.com/package/safe-regex>

OpenJDK's Collection.sort() is broken!

Execution error in Java's Timsort						
Details						
Type:	<input checked="" type="checkbox"/> Bug	Status:				
Priority:	<input checked="" type="checkbox"/> P3	Resolution:				
Affects Version/s:	None	Fix Version/s:				
Component/s:	core-libs					
Labels:	None					
Subcomponent:	java.util:Collections					
Introduced In Version:	6					
Resolved In Build:	b20					
Backports						
Issue	Fix Version	Assignee	Priority	Status	Resolution	Resolved In Build
JDK-8206770	12	Doug Lea	P3	Resolved	Fixed	team
JDK-8206547	11.0.1	Doug Lea	P3	Resolved	Fixed	b01

▼ Description

Carine Pivoteau wrote:

While working on a proper complexity analysis of the algorithm, we realised that there was an error in the last paper reporting such a bug (<http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf>). This implies that the correction implemented in the Java source code (changing TimSort stack size) is wrong and that it is still possible to make it break. This is explained in full details in our analysis: <https://arxiv.org/pdf/1805.08612.pdf>.

We understand that coming upon data that actually causes this error is very unlikely, but we thought you'd still like to know and do something about it. As the authors of the previous article advocated for, we strongly believe that you should consider modifying the algorithm as explained in their article (and as was done in Python) rather than trying to fix the stack size.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at java.util.TimSort.pushRun(TimSort.java:386)
    at java.util.TimSort.sort(TimSort.java:213)
    at java.util.Arrays.sort((Arrays.java:659)
    at TestTimSort.main(TestTimSort.java:18)
```

```

final static int[] runLengths = new int[] { 76405736, 74830360, 1181532, 787688, 1575376, 2363064, 3938440, 6301504,
1181532, 393844, 15753760, 1575376, 787688, 393844, 1969220, 3150752, 1181532, 787688, 5513816, 3938440,
1181532, 787688, 1575376, 18116824, 787688, 1575376, 2363064, 3938440, 787688, 26781392, 1181532,
787688, 1575376, 2363064, 393844, 4332284, 1181532, 787688, 1575376, 12209164, 1181532, 787688, 1575376,
2363064, 787688, 393844, 4726128, 1575376, 787688, 1969220, 76405758, 53168940, 1181532, 787688, 1575376,
2363064, 3938440, 1575376, 787688, 393844, 10633788, 1181532, 787688, 1575376, 2363064, 4332284, 1181532,
787688, 1575376, 12996852, 1181532, 787688, 1575376, 2363064, 393844, 17329136, 1575376, 787688, 393844,
1969220, 3150752, 1181532, 393844, 7483036, 1575376, 787688, 1969220, 2756908, 1181532, 787600, 76405780,
38202802, 114608494, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264, 22528, 45056, 90112, 180224,
360448, 720896, 1441792, 2883584, 5767168, 11387222, 22495132, 319836, 213224, 426448, 639672, 1066120,
1705792, 426448, 213224, 106612, 4584316, 426448, 213224, 106612, 533060, 106612, 852896, 426448, 213224,
1599180, 1172732, 319836, 213224, 426448, 5223988, 319836, 213224, 426448, 639672, 1066120, 319836, 213224,
7782676, 426448, 213224, 533060, 746284, 213224, 1705792, 319836, 213224, 426448, 639672, 2238852, 426448,
213224, 106612, 2345464, 426448, 213224, 106612, 533060, 106612, 852896, 426448, 213224, 22921602,
15245516, 319836, 213224, 426448, 639672, 1172732, 319836, 213224, 426448, 3304972, 319836, 213224, 426448,
639672, 213224, 1279344, 426448, 213224, 533060, 3838032, 319836, 213224, 426448, 639672, 213224, 106612,
5330600, 319836, 213224, 426448, 639672, 1066120, 213224, 2345464, 426448, 213224, 106612, 533060, 106612,
852896, 426448, 213224, 106524, 22921624, 11460724, 34382260, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632,
11264, 22528, 45056, 90112, 180224, 360448, 720896, 1001792, 17383584, 2649020, 67397370, 102630, 68420,
136840, 205260, 342100, 547360, 102630, 68420, 1436820, 102630, 68420, 136840, 205260, 342100, 547360,
102630, 68420, 136840, 205260, 68420, 34210, 1607870, 102630, 68420, 136840, 205260, 342100, 68420, 34210,
2428910, 102630, 68420, 136840, 205260, 34210, 410520, 102630, 68420, 136840, 1094720, 102630, 68420,
136840, 205260, 68420, 34210, 444730, 136840, 68420, 34210, 171050, 34210, 6876232, 46181350, 102630, 68420,
136840, 205260, 34210, 342100, 136840, 68420, 34210, 992090, 102630, 68420, 136840, 205260, 68420, 342100,
205260, 102630, 68420, 1163140, 102630, 68420, 136840, 205260, 68420, 1607870, 102630, 68420, 136840,
```

```
n" java.lang.ArrayIndexOutOfBoundsException: 40
        .pushRun(TimSort.java:386)
    .sort(TimSort.java:213)
    sort((Arrays.java:659)
n(TestTimSort.java:18)

19536, 3256, 143264, 13024, 6512, 3256, 16280, 26048, 9768, 3256, 61864, 13024, 6512, 16280, 22792, 9768,
3168, 618684, 309254, 927850, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264, 22460, 23056, 45056,
72314, 181632, 2838, 1892, 3784, 5676, 9460, 15136, 2838, 946, 37840, 3784, 1892, 946, 4730, 7568, 2838,
1892, 13244, 9460, 2838, 1892, 3784, 43516, 2838, 1892, 3784, 5676, 9460, 1892, 65274, 2838, 1892, 3784,
5676, 946, 10406, 2838, 1892, 3784, 30372, 2838, 1892, 3784, 5676, 946, 12298, 3784, 1892, 946, 4730,
185438, 127710, 2838, 1892, 3784, 5676, 9460, 3784, 1892, 946, 26488, 2838, 1892, 3784, 5676, 946, 10406,
2838, 1892, 3784, 31218, 2838, 1892, 3784, 5676, 946, 42570, 2838, 1892, 3784, 5676, 9460, 17974, 3784,
1892, 4730, 6622, 2838, 1804, 185460, 92642, 278014, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 9064,
11528, 23606, 54340, 858, 572, 1144, 1716, 2860, 4576, 858, 286, 11440, 1144, 572, 286, 1430, 2288, 858,
572, 4004, 2860, 858, 572, 1144, 13156, 858, 572, 1144, 1716, 2860, 572, 19448, 858, 572, 1144, 1716, 286,
3146, 858, 572, 1144, 8866, 858, 572, 1144, 1716, 572, 286, 3432, 1144, 572, 1430, 55506, 38610, 858, 572,
1144, 1716, 2860, 1144, 572, 286, 7722, 858, 572, 1144, 1716, 3146, 858, 572, 1144, 9438, 858, 572, 1144,
1716, 286, 12584, 1144, 572, 286, 1430, 2288, 858, 286, 5434, 1144, 572, 1430, 2002, 858, 484, 55528, 27676,
83116, 66, 44, 88, 176, 352, 704, 1408, 1716, 3872, 8816, 16192, 264, 176, 352, 528, 176, 88, 1144, 352,
176, 88, 440, 88, 3432, 352, 176, 440, 616, 176, 88, 1408, 264, 176, 352, 528, 176, 88, 3960, 264, 176, 352,
528, 880, 88, 5808, 264, 176, 352, 528, 968, 264, 176, 352, 2640, 264, 176, 352, 528, 968, 264, 176, 352, 2816,
440, 16566, 11264, 264, 176, 352, 528, 880, 352, 176, 2376, 264, 176, 352, 528, 968, 264, 176, 352, 2816,
264, 176, 352, 528, 88, 3872, 264, 176, 352, 528, 880, 1584, 528, 264, 88, 616, 176, 16588, 8206, 24706, 66,
44, 88, 176, 352, 704, 1408, 2090, 4708, 66, 44, 88, 132, 220, 352, 66, 44, 88, 990, 66, 44, 88, 132, 220,
418, 88, 44, 110, 154, 66, 44, 1122, 66, 44, 88, 132, 220, 88, 44, 22, 1716, 88, 44, 110, 154, 44, 352, 66,
```

OpenJDK's Collection.sort() is broken!

On the Worst-Case Complexity of TimSort

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivotteau
Université Paris-Est, LIGM (UMR 8049), UPEM, F77454 Marne-la-Vallée, France

Abstract

TIMSORT is an intriguing sorting algorithm designed in 2002 for Python, whose worst-case complexity was announced, but not proved until our recent preprint. In fact, there are two slightly different versions of TIMSORT that are currently implemented in Python and in Java respectively. We propose a pedagogical and insightful proof that the Python version runs in time $\mathcal{O}(n \log n)$. The approach we use in the analysis also applies to the Java version, although not without very involved technical details. As a byproduct of our study, we uncover a bug in the Java implementation that can cause the sorting method to fail during the execution. We also give a proof that Python's TIMSORT running time is in $\mathcal{O}(n + n\mathcal{H})$, where \mathcal{H} is the entropy of the distribution of runs (i.e. maximal monotonic sequences), which is quite a natural parameter here and part of the explanation for the good behavior of TIMSORT on partially sorted inputs. Finally, we evaluate precisely the worst-case running time of Python's TIMSORT, and prove that it is equal to $1.5n\mathcal{H} + \mathcal{O}(n)$.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Sorting algorithms, Merge sorting algorithms, TimSort, Analysis of algorithms

1 Introduction

TIMSORT is a sorting algorithm designed in 2002 by Tim Peters [9], for use in the Python programming language. It was thereafter implemented in other well-known programming languages such as Java. The algorithm includes many implementation optimizations, a few heuristics and some refined tuning, but its high-level principle is rather simple: The sequence S to be sorted is first decomposed greedily into monotonic runs (i.e. nonincreasing or nondecreasing subsequences of S as depicted on Figure 1), which are then merged pairwise according to some specific rules.

$$S = (\underbrace{12, 10, 7, 5}_{\text{first run}}, \underbrace{7, 10, 14, 25, 36}_{\text{second run}}, \underbrace{3, 5, 11, 14, 15, 21, 22}_{\text{third run}}, \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}})$$

Figure 1 A sequence and its run decomposition computed by TimSort: for each run, the first two elements determine if it is increasing or decreasing, then it continues with the maximum number of consecutive elements that preserves the monotonicity.

The idea of starting with a decomposition into runs is not new, and already appears in Knuth's NATURALMERGESORT [6], where increasing runs are sorted using the same mechanism as in MERGESORT. Other merging strategies combined with decomposition into runs appear in the literature, such as the MINIMALSORT of [10] (see also [2] for other considerations on the same topic). All of them have nice properties: they run in $\mathcal{O}(n \log n)$ and even $\mathcal{O}(n + n \log \rho)$, where ρ is the number of runs, which is optimal in the model of sorting by comparisons [7], using the classical counting argument for lower bounds. And yet, among all these merge-based algorithms, TIMSORT was favored in several very popular programming languages, which suggests that it performs quite well in practice.

TimSort running time was implicitly assumed to be $\mathcal{O}(n \log n)$, but our unpublished preprint [1] contains, to our knowledge, the first proof of it. This was more than ten years after TIMSORT started being used instead of QUICKSORT in several major programming languages. The growing popularity of this algorithm invites for a careful theoretical investigation. In the present paper, we make a thorough analysis which provides a better understanding of the inherent qualities of the merging strategy of TIMSORT. Indeed, it reveals that, even without its refined heuristics,¹ this is an effective sorting algorithm, computing and merging runs on the fly, using only local properties to make its decisions.

OpenJDK's java.util.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw^{1,2}, Jurriaan Rot^{3,1}, Frank S. de Boer^{1,3}, Richard Bubel⁴, and Reiner Hähnle⁴

¹ CWI, Amsterdam, The Netherlands

² SDL, Amsterdam, The Netherlands

³ Leiden University, The Netherlands

⁴ Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

1 Introduction

Some of the arguments often invoked against the usage of formal software verification include the following: it is expensive, it is not worthwhile (compared to its cost), it is less effective than bug finding (e.g., by testing, static analysis, or model checking), it does not work for “real” software. In this article we evaluate these arguments in terms of a case study in formal verification.

The goal of this paper is functional verification of sorting algorithms written in Java with mechanical proofs. Because of the complexity of the code under verification, it is essential to break down the problem into subtasks of manageable size. This is achieved with *contract-based deductive verification* [3], where the functionality and the side effects of each method are precisely specified with expressive first-order contracts. In addition, each class is equipped with an invariant that has to be re-established by each method upon termination. These formal specifications are expressed in the Java Modeling Language (JML) [9].

We use the state-of-art Java verification tool KeY [4], a semi-automatic, interactive theorem prover, which covers nearly full sequential Java. KeY typically finds more than 99% of the proof steps automatically (see Sect. 6), while the remaining ones are interactively done by a human expert. This is facilitated by the use in KeY of symbolic execution plus invariant reasoning as its proof paradigm. That results in a close correspondence between proof nodes and symbolic program states which brings the experience of program verification somewhat close to that of debugging.

* Partly funded by the EU project FP7-610582 ENVISAGE and the NWO project 612.063.920 CoRE.

An Example from DARPA STAC Program

```
1 Challenge Program:  
2  
3     railyard-manager.jar  
4  
5 Challenge Question:  
6  
7     Is there an algorithmic complexity vulnerability in space that would  
8         cause the challenge program to store a file with a logical size that  
9         exceeds the resource usage limit given the input budget?  
10  
11 Additional Background for Question:  
12  
13     Attacker is privileged (has railyard manager credentials for server  
14         under attack)  
15  
16 Input Budget:  
17  
18     Maximum sum of the PDU sizes of the application requests sent from the  
19         attacker to the server: 5 MB (measured via sum of the length field in  
20             tcpdump)  
21  
22 Resource Usage Limit:  
23  
24     Available Logical Size: 5 GB (logical size of output file measured  
25             with 'stat')  
26  
27 Probability of Success: 99%  
28
```



Private Repository: https://github.com/EnSoftCorp/STAC-E6-Engagement-Apps/tree/master/engagement6/decompiled_source/STAC6-railyard_src_cfr
Vulnerability Report: <https://docs.google.com/document/d/1-vAyjZbzKBqjVOohrFHFAJ6amXfSWraUZVPPWI33PXQ/>

What Can Be Done About AC Vulnerabilities?

- AC vulnerabilities arise because of **design decisions**, so solutions must address these issues in design. But you can't know what safeguards to put in place if you don't know how you're vulnerable.
- Mitigation techniques include:
 - **Select a new algorithm.** As a result of the hash table collision attacks in 2011, most programming languages changed the data structure, used as bins, for their hash table implementation.
 - **Use input sanitization.** Sometimes the AC vulnerability present in a given algorithm only happens for a specific class of inputs.
 - You can restrict the input space a user can submit by placing explicit limits in your application (e.g., limit the length of input, the use of certain options or characters, etc.).

What Can Be Done About AC Vulnerabilities?

- **Implement hard resource limits.** Occasionally, you need the strength and flexibility of an algorithm that is vulnerable to attack, and the input space is too difficult to restrict with input sanitization.
 - In this case, you can implement hard resource limitations for your application.
 - Many applications will abort decompression when they encounter a decompression bomb by refusing to extract data beyond a certain size.

Side Channel Vulnerabilities

Extracting audio from visual information

Algorithm recovers speech from the vibrations of a potato-chip bag filmed through soundproof glass.

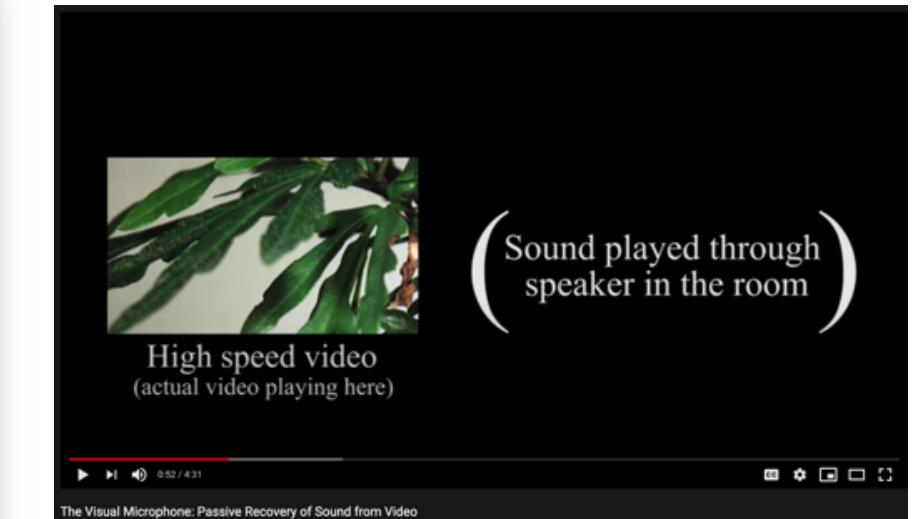
 Watch Video

Larry Hardesty | MIT News Office

August 4, 2014



<https://news.mit.edu/2014/algorithm-recovers-speech-from-vibrations-0804>



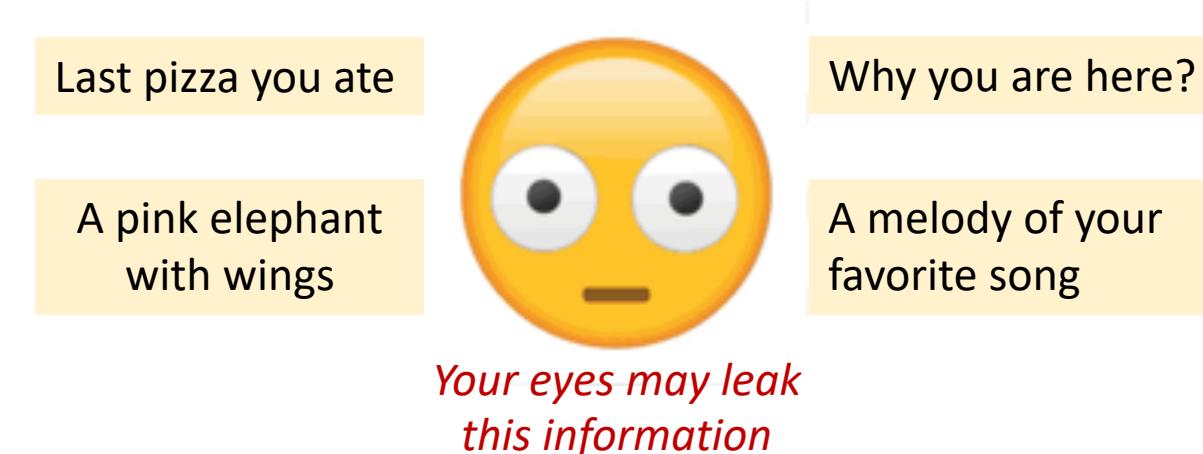
The Visual Microphone: Passive Recovery of Sound from Video
<https://www.youtube.com/watch?v=FKXOucXB4a8>



Can You Recover Sound From Images?
<https://www.youtube.com/watch?v=eUzB0L0mSCI>

Side Channel Vulnerability

- Side channel vulnerabilities allow attackers to **infer** potentially *sensitive information* just by *observing normal behavior of software system*.
- Attacker can be active or passive observer.
- Like mind reading? Which thought do you currently think?



Side Channel Vulnerabilities

Compromising Reflections

— or —

How to Read LCD Monitors Around the Corner

Michael Backes
Saarland University and
Max Planck Institute for Software Systems
Saarbrücken, Germany
backes@cs.uni-sb.de

Markus Dürrmuth Dominique Unruh
Saarland University
Saarbrücken, Germany
{duermuth,unruh}@cs.uni-sb.de

Abstract

We present a novel eavesdropping technique for spying at a distance on data that is displayed on an arbitrary computer screen, including the currently prevalent LCD monitors. Our technique exploits reflections of the screen's optical emanations in various objects that one commonly finds in close proximity to the screen and uses those reflections to recover the original screen content. Such objects include eyeglasses, tea pots, spoons, plastic bottles, and even the eye of the user. We have demonstrated that this attack can be successfully mounted to spy on even small fonts using inexpensive, off-the-shelf equipment (less than 1500 dollars) from a distance of up to 10 meters. Relating to more ex-



proach for spying on confidential data. As early as in 1985, electrical emanations of CRT screens were successfully exploited to reconstruct the screen's content from a distance [12]. This attack was further refined in diverse variations of different levels of sophistication, e.g., emanations from the cable connecting an LCD

Learn what a user types by observing reflections of monitor picture

from over 30 m away. Particularly good results were obtained from reflections in a user's eyeglasses or a tea pot located on the desk next to the screen. Reflections that stem from the eye of the user also provide good results. However, eyes are harder to spy on at a distance because they are fast-moving objects and require high

Timing Analysis of Keystrokes and Timing Attacks on SSH*

Dawn Xiaodong Song David Wagner Xuqing Tian
University of California, Berkeley

Abstract

SSH is designed to provide a secure channel between two hosts. Despite the encryption and authentication mechanisms it uses, SSH has two weaknesses: First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which reveals the approximate size of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed, which leaks the interkeystroke timing information of users' typing. In this paper, we show how these seemingly minor weaknesses result in serious security risks.

Our work introduces a side-channel that is not an idiosyncrasy of the computer's behavior, but it exploits the visual emanation of the screen itself – and hence its proper functionality – in combination with everyday objects that are located in close proximity to the screen such as tea pots, eyeglasses, plastic bottles, spoons, or the eye of the user. Our approach is predicated on the idea that the image of the screen can be reconstructed from reflections on those objects. see Figure 1. We

First we show that even very simple statistical techniques suffice to reveal sensitive information such as the length of users' passwords or even root passwords. More importantly, we further show that by using more advanced statistical techniques on timing information collected from the network, the eavesdropper can learn significant information about what users type in SSH sessions. In particular, we perform a statistical study of users' typing patterns and show that these patterns reveal information about the keys typed. By developing a Hidden Markov Model and our key sequence prediction algorithm, we can predict key sequences from the interkeystroke timings. We further develop an attacker system, *Herbivore*, which tries to learn users' passwords by monitoring SSH sessions. By collecting timing information on the network, Herbivore can speed up exhaustive search for passwords by a factor of 50. We also propose some countermeasures.

In particular we identify that two seemingly minor weaknesses of SSH lead to serious security risks. First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use). Therefore an eavesdropper can easily learn the approximate length of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such as Shift or Ctrl). We show in the paper that this prop-

Learn what a user types by observing inter-packet timing in encrypted SSH session

Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow

Shuo Chen
Microsoft Research
Redmond, WA, USA
shuchen@microsoft.com

Rui Wang, XiaoFeng Wang, Kehuan Zhang
School of Informatics and Computing
Indiana University Bloomington
Bloomington, IN, USA
[wang63, xw7, kehzhang]@indiana.edu

Abstract With software-as-a-service becoming mainstream, more and more applications are delivered to the client through the Web. Unlike a desktop application, a web application is split into browser-side and server-side components. A subset of the application's internal information flows are inevitably exposed on the network. We show that despite encryption, such a side-channel information leak is a realistic and serious threat to user privacy. Specifically, we found that surprisingly detailed sensitive information is being leaked out from a number of high-profile, top-of-the-line web applications in healthcare, taxation, investment and web search: an eavesdropper can infer the illnesses/medications/surgeries of the user, her family income and investment secrets, despite HTTPS protection; a stranger on the street can glean enterprise employees' web search queries, despite WPA/WPA2 Wi-Fi encryption. More importantly, the root causes of the problem are some fundamental characteristics of web applications: stateful communication, low entropy input for better interaction, and significant traffic distinctions. As a result, the scope of the problem seems industry-wide. We further present a concrete analysis to demonstrate the challenges of mitigating such a threat, which points to the necessity of a disciplined engineering practice for side-channel mitigations in future web application developments.

Keywords side-channel leak; Software-as-a-Service (SaaS); web application; encrypted traffic; ambiguity set; padding

I. INTRODUCTION

Regarding the pseudonyms used in the paper

This paper reports information leaks in several real-world web applications. We have notified all the affected parties of our findings. Some requested us to anonymize their product names. Throughout the paper, we use superscript "A" to denote such pseudonyms, e.g., *OnlineHealth^A*, *OnlineTax^A*, and *OnlineInvest^A*.

The drastic evolution in web-based computing has come to the stage where applications are increasingly delivered as services to web clients. Such a software-as-a-service (SaaS) paradigm excites the software industry. Compared to desktop software, web applications have the advantage of not requiring client-side installations or updates, and thus are easier to deploy and maintain. Today

data flows and control flows) are inevitably exposed on the network, which may reveal application states and state-transitions. To protect the information in critical applications against network sniffing, a common practice is to encrypt their network traffic. However, as discovered in our research, serious information leaks are still a reality.

For example, consider a user who enters her health profile into *OnlineHealth^A* by choosing an illness condition from a list provided by the application. Selection of a certain illness causes the browser to communicate with the server-side component of the application, which in turn updates its state, and displays the illness on the browser-side user interface. Even though the communications generated during these state transitions are protected by HTTPS, their observable attributes, such as packet sizes and timings, can still give away the information about the user's selection.

Side-channel information leaks. It is well known that the aforementioned attributes of encrypted traffic, often referred to as *side-channel information*, can be used to obtain some insights about the communications. Such side-channel information leaks have been extensively studied for a decade, in the context of secure shell (SSH) [15], video-streaming [13], voice-over-IP (VoIP) [23], web browsing and others. Particularly, a line of research conducted by various research groups has studied anonymity issues in encrypted web traffic. It has been shown that because each web page has a distinct size, and usually loads some resource objects (e.g., images) of different sizes, the attacker can fingerprint the page so that even when a user visits it through HTTPS, the page can be re-identified [7][16]. This is a concern for anonymity channels such as Tor [17], which are expected to hide users' page-visits from eavesdroppers.

Although such side-channel leaks of web traffic have been known for years, the whole issue seems to be neglected by the general web industry, presumably because little evidence exists to demonstrate the seriousness of their consequences other than the effect on the users of anonymity channels. Today, the Web has evolved beyond a publishing system for static web pages, and instead, becomes a platform for delivering full-fledged software applications. The side-channel vulnerabilities of encrypted communications, coupled with the distinct features of web

Learn about the user actions performed on Web by observing packet sizes in encrypted Web traffic

Side Channel Vulnerabilities

Exposing Private Information by Timing Web Applications

Andrew Bortz
Stanford University
abortz@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Palash Nandy
palashn@gmail.com

ABSTRACT

We show that the time web sites take to respond to HTTP requests can leak private information, using two different types of attacks. The first, *direct timing*, directly measures response times from a web site to expose private information such as validity of an username at a secured site or the number of private photos in a publicly viewable gallery. The second, *cross-site timing*, enables a malicious web site to obtain information from the user's perspective at another site. For example, a malicious site can learn if the user is currently logged in at a victim site and, in some cases, the number of objects in the user's shopping cart. Our experiments suggest that these timing vulnerabilities are wide-spread. We explain in detail how and why these attacks work, and discuss methods for writing web application code that resists these attacks.

Categories and Subject Descriptors

K.4.4 [Computers and Society]: Electronic Commerce—Security; K.4.1 [Computers and Society]: Public Policy Issues—Privacy

General Terms

Design, Security, Experimentation

Keywords

web application security, web browser design, privacy, web spoofing, phishing

1. INTRODUCTION

Web applications are vulnerable to a variety of well publicized attacks, such as cross-site scripting (XSS) [15], SQL injection [2], cross-site request forgery [14], and many others. In this paper we study timing vulnerabilities in web application implementations. Our results show that timing data can expose private information, suggesting that this issue is often ignored by web developers. We first discuss the type of information revealed by a timing attack and then discuss ways to prevent such attacks.

We consider two classes of timing attacks. The first, called a *direct timing* attack, measures the time the web site takes

to respond to HTTP requests. We experiment with two types of direct attacks:

- Estimating hidden data size. Many sites holding user data, such as photo-sharing sites, blogging sites, and social networking sites, allow users to mark certain data as private. Photo sharing sites, for example, allow users to mark certain galleries as only viewable by certain users. We show that direct timing measurements can expose the existence of private data, and even reveal the size of private data such as the number of hidden pictures in a gallery.
- Learning hidden boolean values. Web login pages often try to hide whether a given username is valid — the same error message is returned whether the input username is valid or not. However, in many cases, the site executes a different code path depending on validity of the given username. As a result, timing information can expose username validity despite the site's attempt to conceal it.

The second class of attacks, called *cross-site timing*, is a form of cross-site request forgery [14]. The attack enables a malicious site to obtain information about the user's view of another site — a violation of the same-origin principle [11, 8]. We describe this attack in Section 4. At a high level, the attack begins when the user visits a malicious page, which proceeds to time a victim web site using one of several techniques, all of which time the exact content the user would actually see. We show that this timing data can reveal private information: for example, it can reveal whether the user is currently logged-in. In some cases, timing information can even reveal the size and contents of the user's shopping cart and other private data, as discussed in Section 4. This information enables a context-aware phishing attack [9] where the user is presented with a custom phishing page.

These attacks exploit weaknesses in server-side application software, specifically when execution time depends on sensitive information. Our results suggest that these vulnerabilities are often ignored.

1.1 Related work

Timing attacks were previously used to attack crypto implementations on smartcards [10, 12, 13] and web servers [4, 1]. Felten and Schneider [6] used a cache-based timing attack to track web users. Their idea is that once a user visits

Learn existence of username from response time of Web application

Exposing Private Information by Timing Web Applications

Andrew Bortz
Stanford University
abortz@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Palash Nandy
palashn@gmail.com

ABSTRACT

We show that the time web sites take to respond to HTTP requests can leak private information, using two different types of attacks. The first, *direct timing*, directly measures response times from a web site to expose private information such as validity of an username at a secured site or the number of private photos in a publicly viewable gallery. The second, *cross-site timing*, enables a malicious web site to obtain information from the user's perspective at another site. For example, a malicious site can learn if the user is currently logged in at a victim site and, in some cases, the number of objects in the user's shopping cart. Our experiments suggest that these timing vulnerabilities are wide-spread. We explain in detail how and why these attacks work, and discuss methods for writing web application code that resists these attacks.

Categories and Subject Descriptors

K.4.4 [Computers and Society]: Electronic Commerce—Security; K.4.1 [Computers and Society]: Public Policy Issues—Privacy

General Terms

Design, Security, Experimentation

Keywords

web application security, web browser design, privacy, web spoofing, phishing

1. INTRODUCTION

Web applications are vulnerable to a variety of well publicized attacks, such as cross-site scripting (XSS) [15], SQL injection [2], cross-site request forgery [14], and many others. In this paper we study timing vulnerabilities in web application implementations. Our results show that timing data can expose private information, suggesting that this issue is often ignored by web developers. We first discuss the type of information revealed by a timing attack and then discuss ways to prevent such attacks.

We consider two classes of timing attacks. The first, called a *direct timing* attack, measures the time the web site takes

to respond to HTTP requests. The first, called a *direct timing* attack, measures the time the web site takes

Learn the amount of hidden images in Gallery

Timing Attacks on Web Privacy

Edward W. Felten and Michael A. Schneider
Secure Internet Programming Laboratory
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

ABSTRACT

We describe a class of attacks that can compromise the privacy of users' Web-browsing histories. The attacks allow a malicious Web site to determine whether or not the user has recently visited some other, unrelated Web page. The malicious page can determine this information by measuring the time the user's browser requires to perform certain operations. Since browsers perform various forms of caching, the time required for operations depends on the user's browsing history; this paper shows that the resulting time variations convey enough information to compromise users' privacy. This attack method also allows other types of information gathering by Web sites, such as a more invasive form of Web "cookies". The attacks we describe can be carried out without the victim's knowledge, and most "anonymous browsing" tools fail to prevent them. Other simple countermeasures also fail to prevent these attacks. We describe a way of reengineering browsers to prevent most of them.

1. Introduction

This paper describes a class of attacks that allow the privacy of users' activities on the Web to be compromised. The attacks allow any Web site to determine whether or not each visitor to the site has recently visited some other site (or set of sites) on the Web.

The attacker can do this without the knowledge or consent of either the user or the other site. For example, an insurance-company site could determine whether the user has recently visited Web sites relating to a particular medical condition; or an employer's Web site could determine whether an employee visiting it had recently visited the sites of various political organizations.

The attacks work by exploiting the fact that the time required by the user's browser to perform certain operations varies, depending on which Web sites the user has visited in the past. By measuring the time required by certain operations, a Web site can learn information about the user's past activities. These attacks are particularly worrisome, for several reasons:

- The attacks are possible because of basic properties of Web browsers, not because of fixable "bugs" in a browser.
- The attacks can be carried out without the victim's knowledge.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to

- Standard Web "anonymization" services do not prevent the attacks; in many cases they actually make the attacks worse.
- Disabling browser features such as Java, JavaScript, and client-side caching do not prevent the attacks.
- The only effective ways we know to prevent the attacks require either an unacceptable slowdown in Web access, or a modification to the design of the browser.
- Even modifying the browser design allows only a partial remedy; several attacks remain possible.

1.1 Why Web Privacy Matters

There is now widespread concern about the privacy of users' activities on the World-Wide Web. The list of Web locations visited by a user often conveys detailed information about the user's family, financial or health situation. Consequently, users often consider their Web-browsing history to be private information that they do not want unknown parties to learn. Of course, visiting a Web site necessarily leaks some information to that site; but users would like some assurance that information about their visit to a site is not available to arbitrary third parties.

Thus far in the short history of the Web, two types of problems have led to compromise of users' Web-browsing histories, and remedies are available for both types.

First, some Web sites gather information and then reveal it to third parties without the informed consent of users. (Alternatively, some sites cause users' browsers to reveal information directly to a third-party site.) These problems have been dealt with by the use of privacy policies and third party audits of Web sites. While these remedies leave much to be desired, they do give users a chance to guess where information will go after it is revealed to a law-abiding site.

Second, some implementation bugs in browsers have provided opportunities for unscrupulous third parties to gather information without the user's consent. While these bugs are part of an unfortunate pattern of security bugs in browsers, each bug by itself has been fixed.

The attacks we describe in this paper admit no such remedy. Because information about visits to a site is not controlled by that site, privacy policies, auditing, and trust in sites are not effective remedies. Because the attacks are not caused by browser bugs, they cannot easily be fixed.

2. Exploiting Web Caching

The first timing attack we will discuss exploits Web caching. We first review how Web caching works, and then discuss the attack.

Learn private key of SSL server

Where's Wally? Precise User Discovery Attacks in Location Proximity Services

Iasonas Polakis George Argyros Theofilos Petsios
Suphannee Sivakorn Angelos D. Keromytis

Network Security Lab, Computer Science Dept.
Columbia University, New York, NY, USA

{polakis, argyros, theofilos, suphannee, angelos}@cs.columbia.edu

Abstract

Location proximity schemes have been adopted by social networks and other smartphone apps as a means of balancing user privacy with utility. However, misconceptions about the privacy offered by proximity services have rendered users vulnerable to trilateration attacks that can expose their location. Such attacks have received major publicity and, as a result, popular service providers have deployed countermeasures for preventing user discovery attacks.

In this paper, we systematically assess the effectiveness of the defenses that proximity services have deployed against adversaries attempting to identify a user's location. We provide the theoretical foundation for formalizing the problem under different proximity models, design practical attacks for each case, and prove tight bounds on the number of queries required for carrying out the attacks. To evaluate the completeness of our approach, we conduct extensive experiments against popular services. While we identify a diverse set of defense techniques that prevent trilateration attacks, we demonstrate their inefficiency against more elaborate attacks. In fact, we pinpoint Facebook users within 5 meters of their exact location, and 90% of Foursquare users within 15 meters. Our attacks are extremely efficient and complete within 3-7 seconds. The severity of our attacks was acknowledged by Facebook and Foursquare, both of which have followed our recommendations and adopted spatial cloaking to protect their users. Furthermore, our findings have wide implications as numerous popular apps with a massive user base remain vulnerable to this significant threat.

1. INTRODUCTION

Location-based services (LBS) have become an integral part of everyday life. However, accessibility to fine-grained location information has raised significant privacy concerns, as users are exposed to various threats, ranging from the inference of sensitive data [33] (e.g., medical issues, political inclination and religious beliefs) to physical threats such as stalking [10]. Furthermore, apart from the revelations regarding mass user surveillance by government agencies, articles have revealed that law enforcement agencies also follow more targeted, and unorthodox, tactics. Fake profiles are used to befriend users and gain access to personal data, as well as track their whereabouts by monitoring their check-in behavior [6, 8]. Therefore, the information accessible by users' contacts is a significant aspect of their privacy.

Revealing a user's location is considered a significant privacy breach [46], and services are adopting the more privacy-

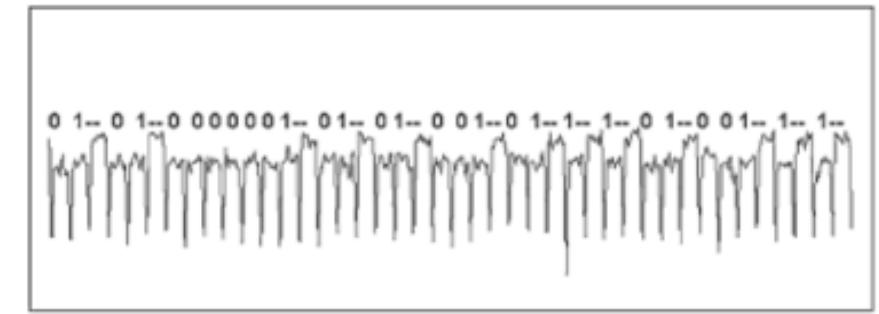
preserving approach of location proximity: notifying users about who is nearby, and at what distance. However, when the exact distance to a user is revealed by the service, trilateration attacks become feasible, with several examples being presented in the media recently. Articles have also reported that the Egyptian government used trilateration to locate and imprison users of gay dating apps [7, 9]. While the use of trilateration has not been confirmed, such reports highlight the potential severity of such attacks, and the importance of preserving the locational privacy of users. Naturally, these reports have caught the attention of popular services, which in turn have deployed defense mechanisms to prevent localization attacks [2].

In this paper, we explore the privacy guarantees of 10 popular social networks and LBS. We audit the services and identify the mechanisms deployed to protect the location privacy of their users. To evaluate the defenses that have been adopted by the industry, we formalize the problem of locating users as a search problem in the discrete Euclidean plane. To our knowledge, this is the first formal treatment of user discovery attacks in proximity services. We prove tight bounds on the number of queries required to attack a service under different proximity models, and devise optimal algorithms that realize those attacks. The lower bounds on the query complexity of our techniques provide useful insight on the effectiveness of mitigations against localization attacks, such as rate limiting the number of queries.

We evaluate our attacks against four of the audited services that employ a diverse set of countermeasures. We show that user discovery attacks against proximity services may require complex techniques; our attacks include geometric algorithms that gradually reduce the candidate bounding area where a user resides, the employment of colluding accounts for obtaining side channel information on the distance between users, and the utilization of statistical algorithms for coping with the randomization used by services as a defense mechanism. Our results demonstrate that, despite the defense mechanisms in place, our attacks are still very effective and time-efficient, and practical for use at scale and on a continuous basis (real-time tracking). In particular, using a single account, we pinpoint Facebook users within 5 meters of their actual location in 3 seconds, and 90% of Foursquare's Swarms users within 15m in 7 seconds. We even stress-test our attacks and demonstrate the feasibility of tracking moving targets in real time. Due to the recent events [8], Grindr hides the distance information for citizens of oppressive regimes. Even without any distance information disclosed, we are able to carry out successful attacks by

In fact, we **pinpoint** Facebook users *within 5 meters of their exact location*, and *90% of Foursquare users within 15 meters*. Our attacks are extremely efficient and complete within 3-7 seconds.

Side Channel Vulnerability



- A **side-channel attack** is any attack based on information gained from an unintended behavior of a computer system, rather than weaknesses in the implemented algorithm itself (e.g., cryptanalysis and software bugs).
 - Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.
- Side-channel attacks on the web can occur, even when transmissions between a web browser and server are *encrypted*.
- According to researchers from Microsoft Research and Indiana University[1] Many powerful side-channel attacks are **based on statistical methods** pioneered by Paul Kocher[2].

[1] Side-Channel Leaks in Web Applications - <https://www.microsoft.com/research/publication/side-channel-leaks-in-web-applications-a-reality-today-a-challenge-tomorrow/>

[2] Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems - https://link.springer.com/chapter/10.1007%2F3-540-68697-5_9

General Classes of Side Channel Attacks

Cache Attack

is based on attacker's ability to monitor cache accesses made by the victim in a shared physical system.

Timing Attack

is based on measuring how much time various computations take to perform

Power Monitoring Attack

make use of varying power consumption by the hardware during computation

Electromagnetic Attack

is based on leaked electromagnetic radiation, which can directly provide plaintexts and other information

Acoustic Cryptanalysis Attack

exploit sound produced during a computation

Differential Fault Analysis Attack

secrets are discovered by analyzing software faults

Data Remanence Attack

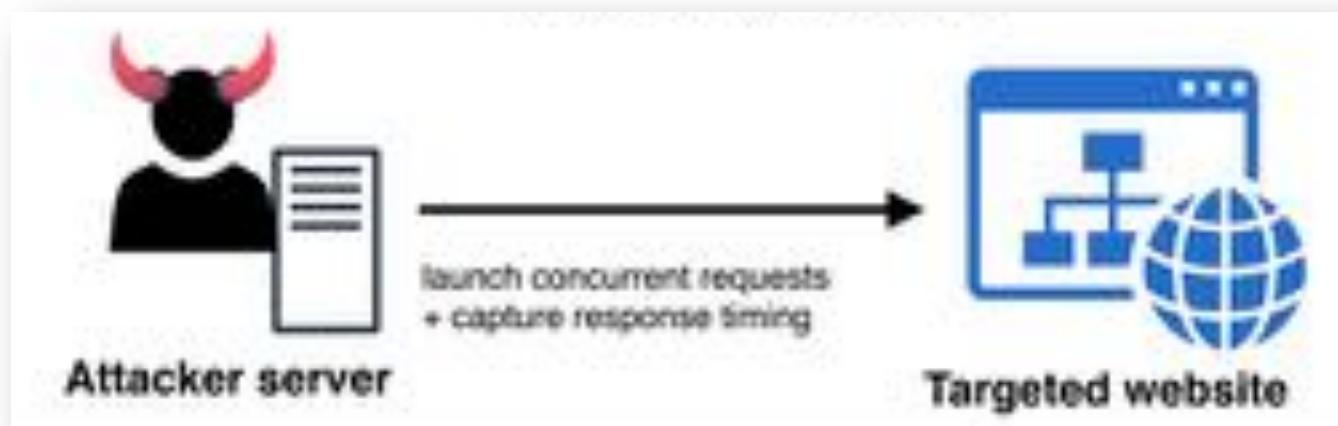
sensitive data are read after supposedly having been deleted

Optical Attack

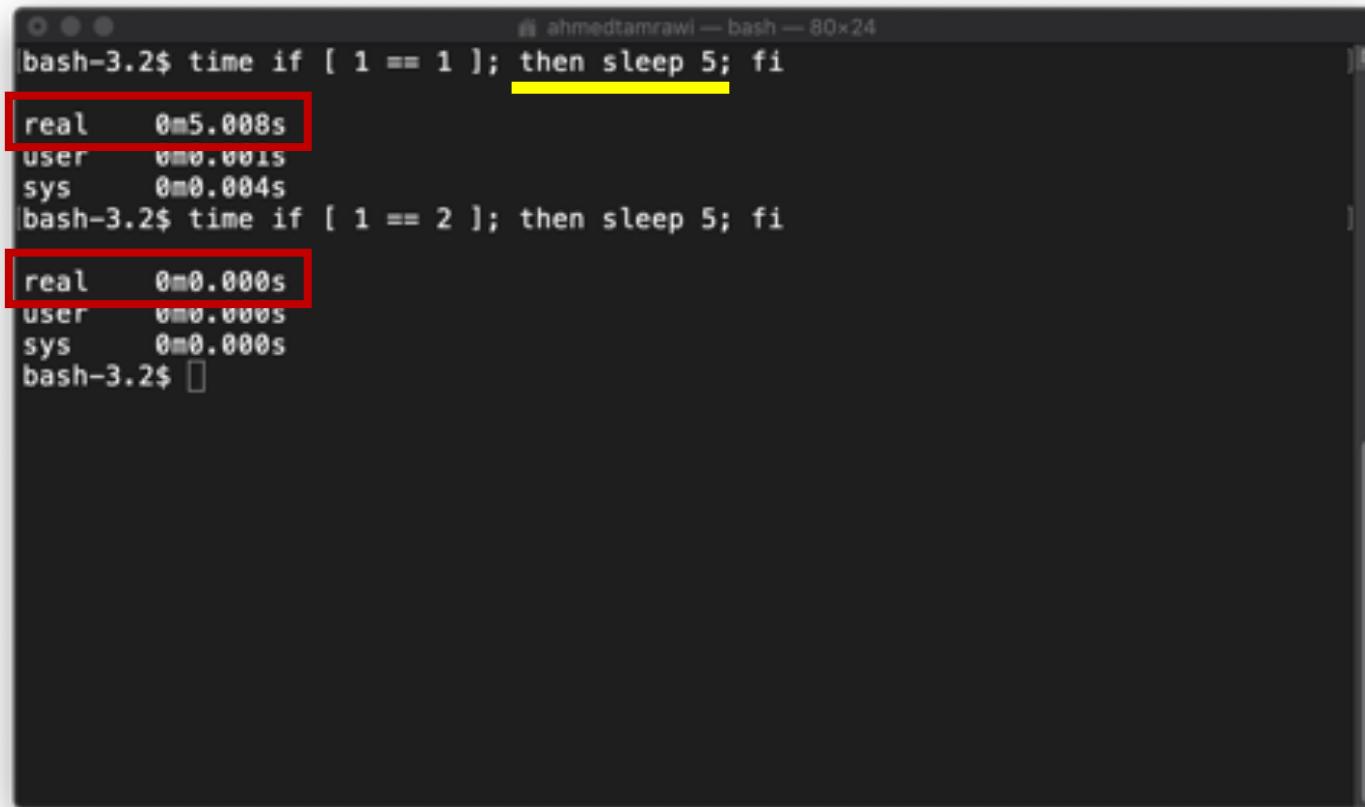
secrets and sensitive data can be read by visual recording using a high-resolution camera or other tools

Timing Side Channel Attacks

- Every logical operation in a computer *takes time to execute*, and the time can **differ based on the input**; with **precise measurements** of the time for each operation, an attacker can work backwards to the input.
- A timing attack is a side-channel attack in which the attacker attempts to leak sensitive information by *analyzing the time taken to execute or run some systems' components*.



Timing Side Channel Attacks



```
ahmedtamrawi:~ bash ~ 80x24
bash-3.2$ time if [ 1 == 1 ]; then sleep 5; fi
real    0m5.008s
user    0m0.001s
sys     0m0.004s
bash-3.2$ time if [ 1 == 2 ]; then sleep 5; fi
real    0m0.000s
user    0m0.000s
sys     0m0.000s
bash-3.2$
```

Timing Side Channel Attacks

```
ahmedtamrawi — bash — 80x24
bash-3.2$ whoami
ahmedtamrawi
bash-3.2$ time if [ $(whoami | cut -c 1) == r ]; then sleep 5; fi
real    0m0.009s
user    0m0.002s
sys     0m0.007s
bash-3.2$ time if [ $(whoami | cut -c 1) == b ]; then sleep 5; fi
real    0m0.007s
user    0m0.002s
sys     0m0.006s
bash-3.2$ time if [ $(whoami | cut -c 1) == a ]; then sleep 5; fi
real    0m5.014s
user    0m0.003s
sys     0m0.008s
bash-3.2$
```

Timing Side Channel Attacks

Login Form

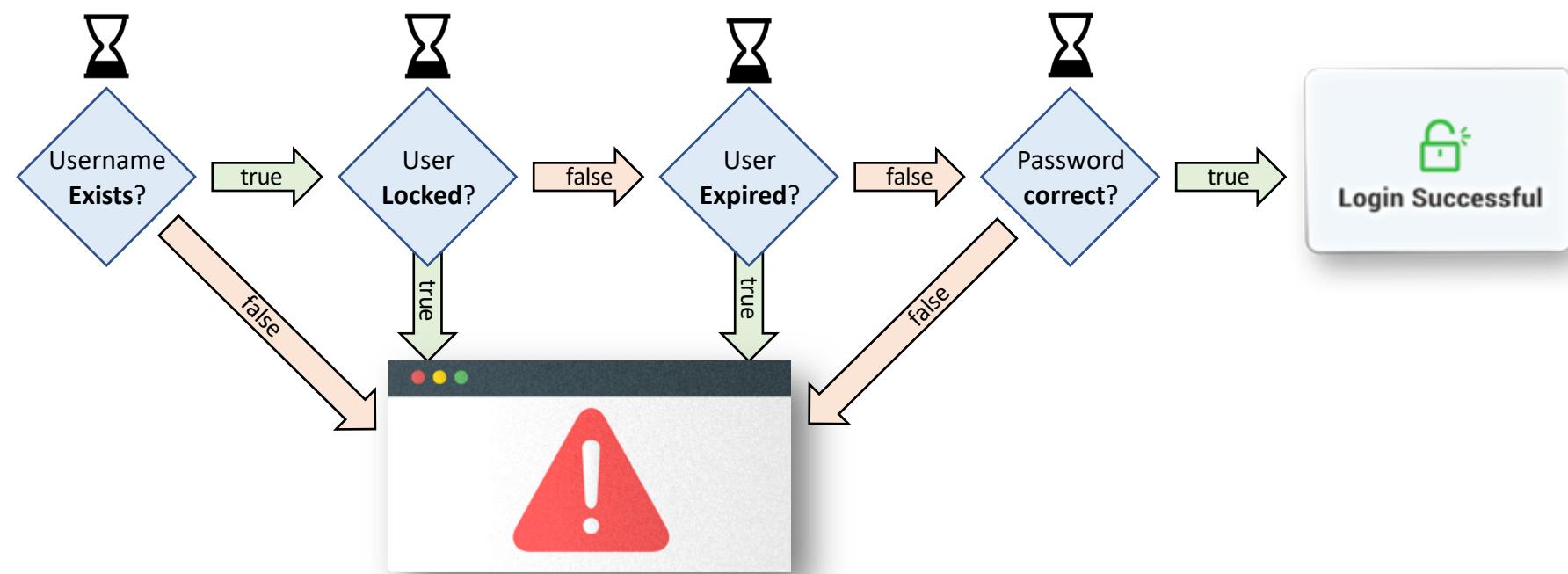
Email or Phone

Password

Forgot Password?

LOGIN

Not a member? [Signup now](#)



Can an attacker leak sensitive information about the *existence of a username* by **measuring the time difference** in the control flow?

Timing Side Channel Attacks

Login Form

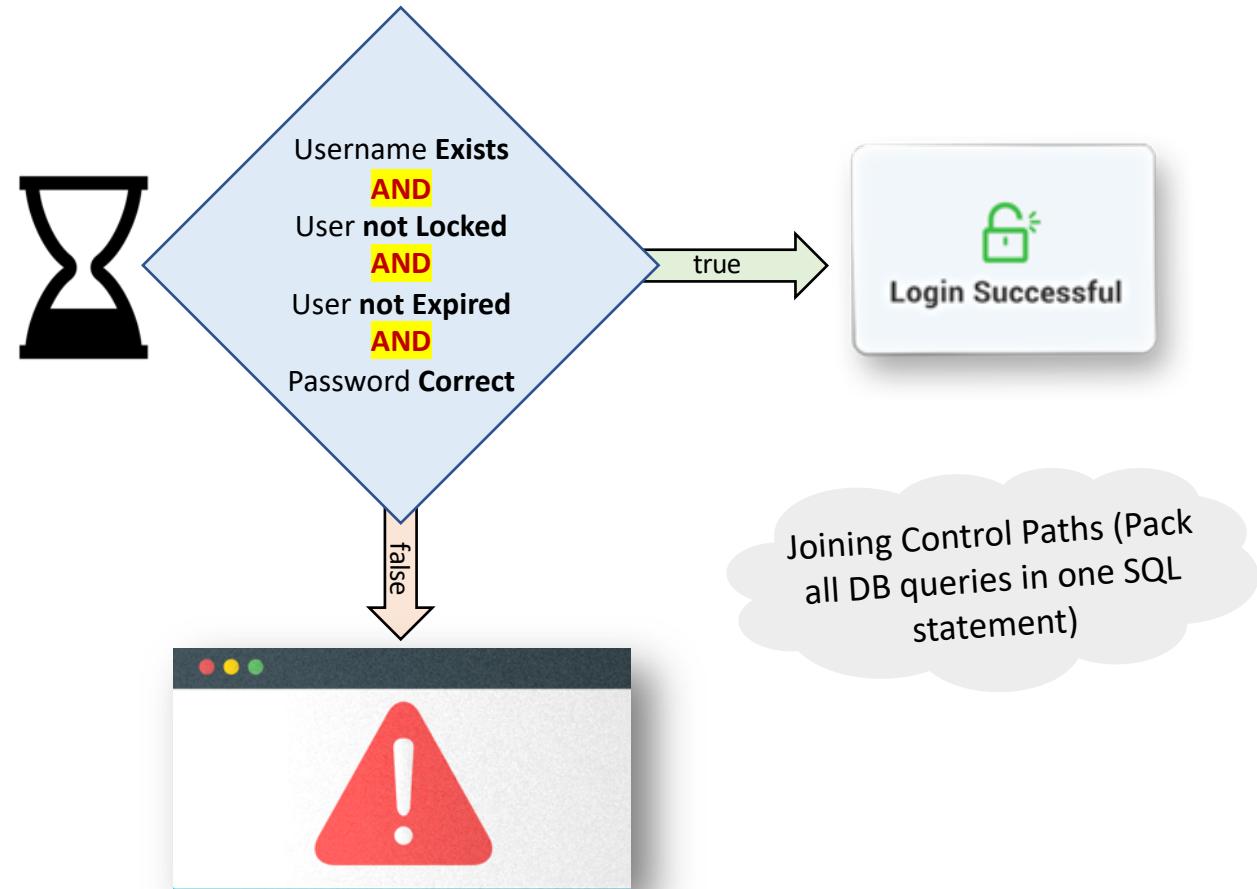
Email or Phone

Password

Forgot Password?

LOGIN

Not a member? [Signup now](#)



Can an attacker leak sensitive information about the *existence of a username* by **measuring the time difference** in the control flow?

Timing Side Channel Attacks

Login Form

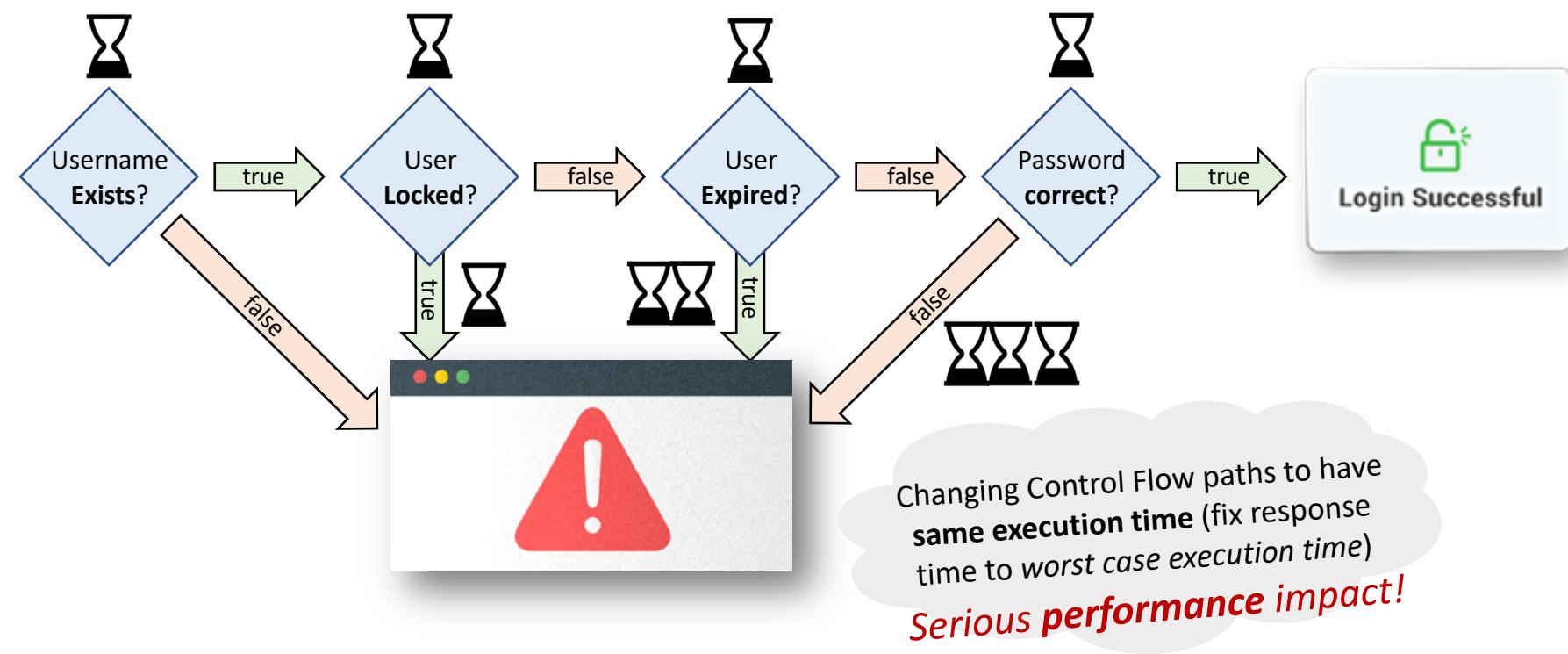
Email or Phone

Password

Forgot Password?

LOGIN

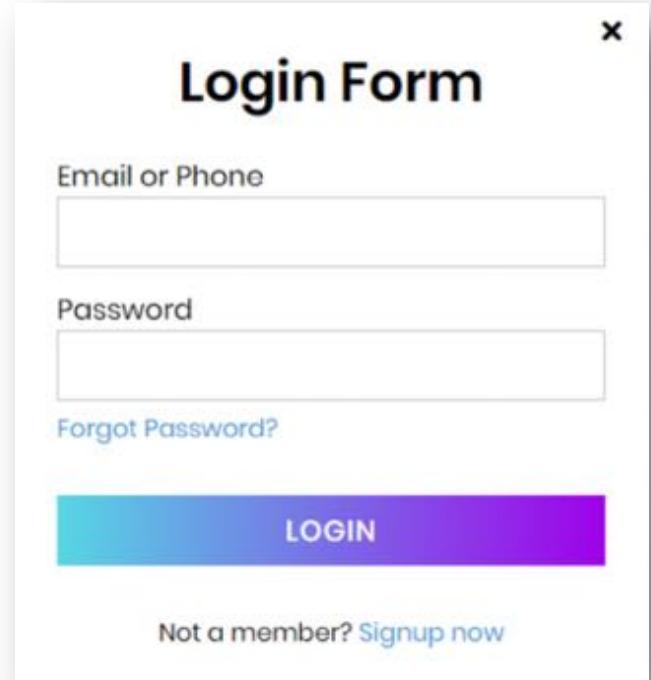
Not a member? [Signup now](#)



Can an attacker leak sensitive information about the *existence of a username* by **measuring the time difference** in the control flow?

Login Side Channel Attack Example

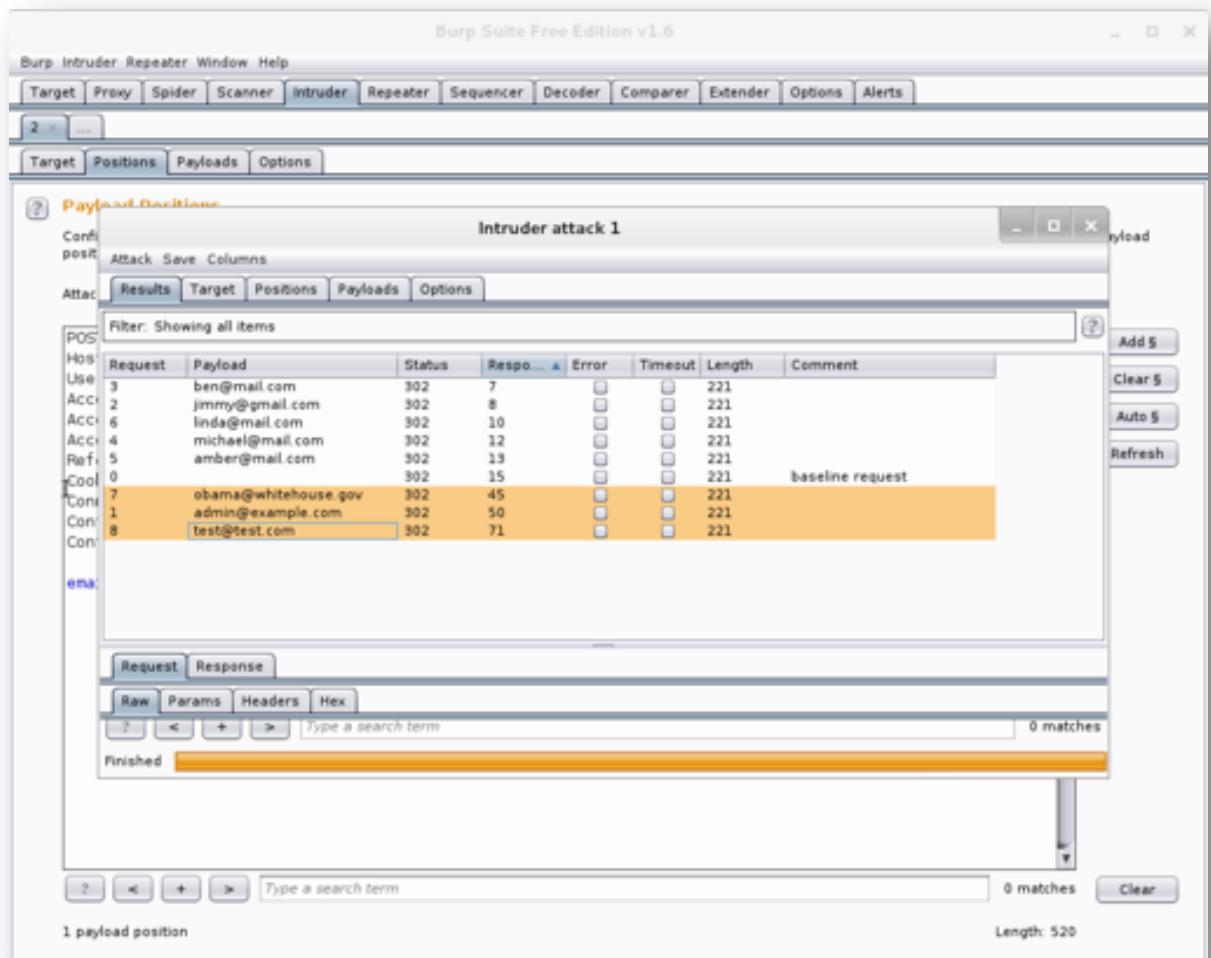
- The login mechanism has some short circuit logic that *does not compute the hash of the given password if the user does not exist* in the database.
- By examining the response times of various users (regardless of the provided password) an attacker can enumerate and **discover the web applications users**.
- Demo code is available at:
<https://github.com/benjholla/LoginSideChannels>



The image shows a standard web-based login interface. At the top right is a close button (an 'x'). Below it is the title "Login Form". There are two input fields: one for "Email or Phone" and one for "Password". To the right of the password field is a blue link "Forgot Password?". At the bottom is a large rectangular button with a gradient from blue to purple, labeled "LOGIN" in white. Below the button is another blue link "Not a member? Signup now".

Login Side Channel Attack Example

- **Vulnerability:** The existence of users can be inferred through timing differentials.
 - Because of the additional time to check password, more time is required when it is a valid username compared to an invalid username.
 - Attacker does not need to know valid passwords, only needs to guess the valid usernames.



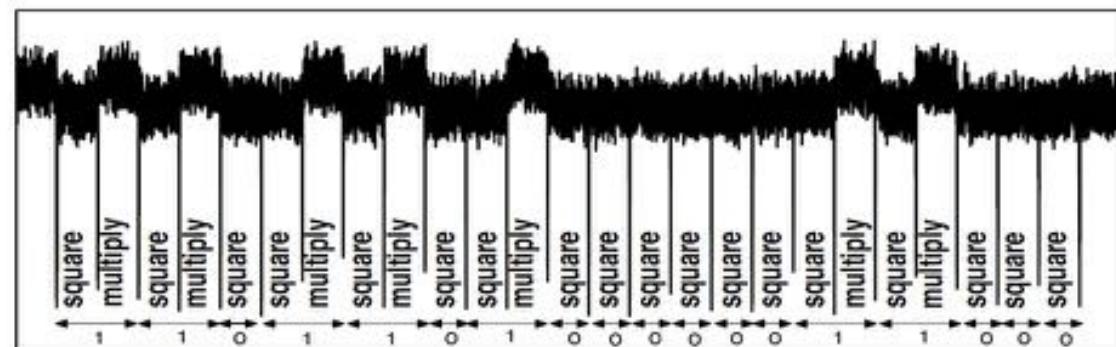
Timing Side Channel Attacks Examples

- The execution time for the *square-and-multiply algorithm used in modular exponentiation* depends linearly on the number of '1' bits in the key.
 - While the number of '1' bits alone is not nearly enough information to make finding the key easy, repeated executions with the same key and different inputs can be used to perform statistical correlation analysis of timing information to recover the key completely, even by a passive attacker.

ALGORITHM 5 Modular Exponentiation.

```
procedure modular_exponentiation(b: integer, n = (ak-1ak-2...a1a0)2,  
m: positive integers)  
x := 1  
power := b mod m  
for i := 0 to k - 1  
    if ai = 1 then x := (x · power) mod m  
    power := (power · power) mod m  
return x{x equals bn mod m}
```

Example: square-and-multiply RSA exponentiation.



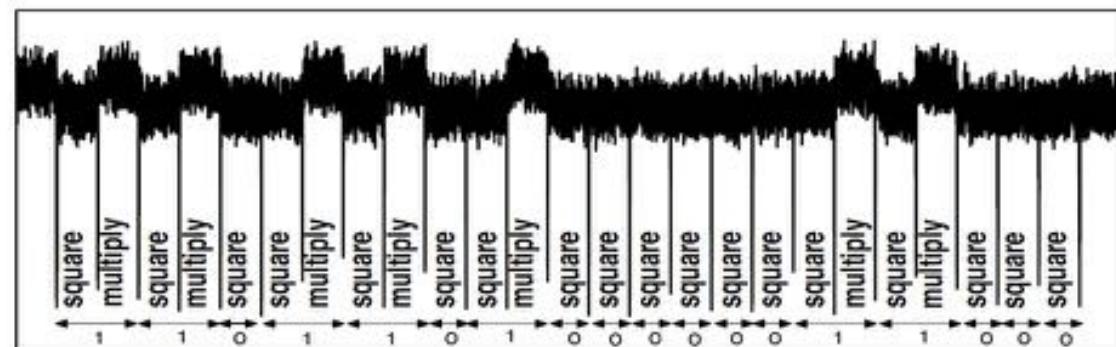
Timing Side Channel Attacks Examples

- The execution time for the *square-and-multiply algorithm used in modular exponentiation* depends linearly on the number of '1' bits in the key.
 - While the number of '1' bits alone is not nearly enough information to make finding the key easy, repeated executions with the same key and different inputs can be used to perform statistical correlation analysis of timing information to recover the key completely, even by a passive attacker.

ALGORITHM 5 Modular Exponentiation.

```
procedure modular_exponentiation(b: integer, n = (ak-1ak-2...a1a0)2,  
m: positive integers)  
x := 1  
power := b mod m  
for i := 0 to k - 1  
    if ai = 1 then x := (x · power) mod m  
    power := (power · power) mod m  
return x{x equals bn mod m}
```

Example: square-and-multiply RSA exponentiation.



Examples

- Some versions of Unix use a relatively expensive implementation of the crypt library function for hashing an 8-character password into an 11-character string.
- On older hardware, this computation took a deliberately and measurably long time: as much as two or three seconds in some cases.
- The login program in early versions of Unix executed the crypt function only when the login name was recognized by the system. This leaked information through timing about the validity of the login name, even when the password was incorrect. An attacker could exploit such leaks by first applying brute-force to produce a list of login names known to be valid, then attempt to gain access by combining only these names with a large set of passwords known to be frequently used.
- Without any information on the validity of login names the time needed to execute such an approach would increase by orders of magnitude, effectively rendering it useless. Later versions of Unix have fixed this leak by always executing the crypt function, regardless of login name validity.

Timing Side Channel Attacks Examples

- Two otherwise securely **isolated processes running on a single system** with either *cache memory or virtual memory* can communicate by deliberately causing page faults and/or cache misses in one process, then monitoring the resulting changes in access times from the other.
- Likewise, if an application is trusted, but its paging/caching is affected by **branching logic**, it may be possible for a second application to determine the values of the data compared to the branch condition by monitoring access time changes; in extreme examples, this can allow recovery of cryptographic key bits.

Timing Side Channel Attacks Examples

```
bool insecureStringCompare(const void *a, const void *b, size_t length) {
    const char *ca = a, *cb = b;
    for (size_t i = 0; i < length; i++)
        if (ca[i] != cb[i])
            return false;
    return true;
}
```

Timing Side Channel Attacks Examples

```
bool insecureStringCompare(const void *a, const void *b, size_t length) {
    const char *ca = a, *cb = b;
    for (size_t i = 0; i < length; i++)
        if (ca[i] != cb[i])
            return false;
    return true;
}
```

```
bool constantTimeStringCompare(const void *a, const void *b, size_t length) {
    const char *ca = a, *cb = b;
    bool result = true;
    for (size_t i = 0; i < length; i++)
        result &= ca[i] != cb[i];
    return result;
}
```

Timing Side Channel Attacks Examples

```
bool check_password(const char input[]){
    const char correct_password[] = "hunter2";

    if (strlen(input) != strlen(correct_password)) return false;

    for (int i = 0; i < strlen(correct_password); i++){
        if (input[i] != correct_password[i]) {
            return false;
        }
    }

    return true;
}
```

An Example from DARPA STAC Program

```
1 Challenge Program:  
2  
3     pwcheck.jar  
4  
5 Challenge Question:  
6  
7     Does the Challenge Program contain any vulnerability that would allow  
8         for the password stored in the password file to be extracted (without  
9             directly looking at the password file) solely by measuring the time  
10            response of repeated queries to the the pwcheck program?  
11  
12 Additional background for question:  
13  
14     It may be assumed that the secret contained within the password file  
15         consists of a password of 5..30 UTF-8 encoded UNICODE charactes from following subset:  
16         [a..z|A..Z|0..9]  
17  
18 Available Operations:  
19  
20     Passive Observations: None  
21     Active Interactions: Run Challenge Program once (1)  
22  
23 Operational Budget:  
24  
25     Max number of operations : 10000  
26     Probability of success   : 99%  
27
```

Timing Attack Avoidance

- Many algorithms can be implemented (or masked by a proxy) in a way that reduces or eliminates data dependent timing information, a **constant-time algorithm**.
- Consider an implementation in which every call to a subroutine always returns in exactly **x** seconds, where **x** is the **maximum time** it ever takes to execute that routine on every possible authorized input.
 - In such an implementation, the timing of the algorithm leaks no information about the data supplied to that invocation.
- The downside of this approach is that the time used for all executions becomes that of the **worst-case performance of the function**.

Timing Attack Avoidance

- The data-dependency of timing may stem from one of the following:
 - **Non-local memory access:** Software run on a CPU with a data cache will exhibit data-dependent timing variations as a result of cache lookups.
 - **Conditional jumps.**
 - “Complicated” Mathematical Operations depending on the actual CPU hardware:
 - Integer division is almost always non-constant time. The CPU uses a microcode loop that uses a different code path when either the divisor or the dividend is small.
 - CPUs without a barrel shifter runs shifts and rotations in a loop, one position at a time. As a result, the amount to shift must not be secret.
 - Older CPUs run multiplications in a way similar to division.

Side Channel Attacks Countermeasures

- Because side-channel attacks rely on the relationship between information emitted (leaked) through a side channel and the secret data, countermeasures fall into two main categories:
 - Eliminate or reduce the release of such information.
 - Eliminate the relationship between the leaked information and the secret data.
- Make the leaked information unrelated, or rather uncorrelated, to the secret data, typically through some form of randomization.