STAC
Apogee Research

The Apogee Research effort on STAC is funded by the United States Airforce Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-15-C-0089.

Categories of Current Limitations of Tools (1/2)

- Category 1: The analysis is focused on establishing the limiting (Big O) behavior, disregarding the coefficients of the asymptotic behavior
- Category 2: Tools disregard the lower orders of the order expansion of resource consumption
- Category 3: Tools only consider the bounding behaviors
  - Best case low-consumption path
  - Worst case high-consumption path
- Category 4: Tools focus exclusively on loops
  - There are other ways to amplify vulnerabilities
- Category 5: Tools disregard attacker input budgets
  - Budget may allow for shifting looping behavior to the user side
- Category 6: Tools disregard side effects on the path from user request to response
- Category 7: Tools focus on localized behavior
  - The cause of a vulnerability may be separated from its effect
- Category 8: Tools disregard the combined effect of multiple dimensions of input
- Category 9: Tools don’t model floating point computations

Categories of Current Limitations of Tools (2/2)

- Category 10: In-scope implementation of packet queueing
- Category 11: Tools don't recognize when constraints can be decoupled prior to reasoning over potential information leakage
- Category 12: Tools assume side channel vulnerabilities require conditional statements
- Category 13: Tools that sample to estimate the complexity curve may miss a high frequency vulnerability with insufficient sampling
- Category 14: Potential SC vulnerabilities can be non-locally balanced
- Category 15: Vulnerability in callback from external library
- Category 16: SC vulnerabilities can leak secret in different domain
- Category 17: Example application with probabilistic complexity function

Cat 1: Coefficients are Disregarded

- Vulnerabilities may go undetected if only analyzing limiting behavior

```
if(guess <= secret)
    for(int i=0; i<n; i++)
        for(int t=0; t<n; t++)
            Consume 1
else
    for(int i=0; i<n; i++)
        for(int t=0; t<n; t++)
            Consume 2
```

- Behavior for guess <= secret is $O(n^2)$
- Behavior for guess > secret is $O(n^2)$
- May conclude program is not vulnerable since complexity is the same for both paths
- However, program is potentially vulnerable since difference in coefficients introduce a differential consumption – in this case itself of order $O(n^2)$

Cat 2: Lower Orders are Disregarded

- Vulnerabilities may go undetected if only analyzing the highest complexity code section

```
for(int i=0; i<n; i++)
    Consume 1000 ms              // SECTION A
    for(int t=0; t<n; t++)
        Consume 1 ms             // SECTION B
```

- Input budget (AC): n <= 99,
  Resource consumption: < 60 s

- Max resource consumption of highest complexity: $1*99^2$ = **9.801 s (< max)**
- Max total resource consumption: $1*99^2 + 1000*99$ = **108.801 s (> max)**

- If only considering only highest complexity, program may be deemed not vulnerable
- However, program exceeds budget after accounting for the lower orders

Cat 3: Only Best & Worst Case Bounds Considered (1/3)

```
if(guess <= Secret){
    if(T == 1){Thread.sleep(1);}
    else if(T == 2){
        for(int i = 0; i < n; i++){Thread.sleep(1);}
    else{
        for(int i = 0; i < n*n*n; i++){Thread.sleep(1);}
}
else{
    if(T == 1){Thread.sleep(1);}
    else if(T == 2){
        for(int i = 0; i < n*n; i++){Thread.sleep(1);}
    else{
        for(int i = 0; i < n*n*n; i++){Thread.sleep(1);}
}
```

Cat 3: Only Best & Worst Case Bounds Considered (2/3)

```
if(guess <= Secret){
    if(T == 1){Thread.sleep(1);}
    else if(T == 2){
        for(int i = 0; i < n; i++){Thread.sleep(1);}
    else{
        for(int i = 0; i < n*n*n; i++){Thread.sleep(1);}
}
else{
    if(T == 1){Thread.sleep(1);}
    else if(T == 2){
        for(int i = 0; i < n*n; i++){Thread.sleep(1);}
    else{
        for(int i = 0; i < n*n*n; i++){Thread.sleep(1);}
}
```

SC Time vulnerability

Cat 3: Only Best & Worst Case Bounds Considered (3/3)

- Side Channels may go undetected if ruled out exclusively through Best and Worst Case comparison of alternative paths

```
if guess <= Secret
    if T == 1        Consume 0
    else if T == 2   Consume N
    else             Consume N^3
else
    if T == 1        Consume 0
    else if T == 2   Consume N^2
    else             Consume N^3
```

- Regardless of the guess and Secret, the best case resource consumption is 0
- Regardless of the guess and Secret, the worst case resource consumption is N^3
- This may lead some tools to conclude there is no differential resource consumption and therefore no side channel
- However, a case with differential resource consumption (N vs N^2) is hiding between the best and worst case paths

Cat 4: Only Loops are Considered (1/3)

```
boolean verifyCreds(String pwd){
    int index = -1;
    for(char x : pwd) {
        if(!correct(x, idx++)){return false;}
        delay();
    }
    return true;
}
…
if verifyCreds(pwd)
    Privileged Action 1
…
if verifyCreds(pwd)
    Privileged Action 2
…
if verifyCreds(pwd)
    Privileged Action N
…
```

Cat 4: Only Loops are Considered (2/3)

```
boolean verifyCreds(String pwd){
    int index = -1;
    for(char x : pwd) {
        if(!correct(x, idx++)){return false;}
        delay();
    }
    return true;
}
…
if verifyCreds(pwd)
    Privileged Action 1
…
if verifyCreds(pwd)
    Privileged Action 2
…
if verifyCreds(pwd)
    Privileged Action N
…
```

Weak SC Time vulnerability

Cat 4: Only Loops are Considered (3/3)

- Vulnerabilities may go undetected if focusing only on loops and their effects, disregarding other ways to amplify the effect of the fundamental cause of a vulnerability (loop or otherwise)

Weak CAUSE of SC (in this case a loop)

```
bool verifyCreds(String pwd)
    int idx = -1
    for(char x: pwd)
        if !correct(x, idx++)
            return false
        else
            delay()
    return true
```

Amplified Effect

```
if verifyCreds(pwd)
    Privileged Action 1
…
if verifyCreds(pwd)
    Privileged Action 2
…
if verifyCreds(pwd)
    Privileged Action N
…
```

- The differential resource consumption of verifyCreds() is too weak to leak secret
- However, when invoked multiple times, the differential consumption is amplified

Cat 5: Input Budgets are Disregarded

- Vulnerabilities may go undetected because tools only analyze individual interactions
- Looping may be shifted to input side by applying budget for multiple (cheap) interactions rather than a single (expensive) interaction; e.g. sampling a weak SC multiple times, or aggregating resource consumption

```
while(true)
    listen for connection
    lookup session state based upon cookie
    if no state found allocate session (Expensive)
    handle requests of session
    end connection and eventually timeout state
```

- There may be no way to exhaust the resources through normal conops of establishing a session and then spending the input budget on exchanging requests and responses
- However, an attacker may apply the input budget towards establishing many back to back sessions, in total exceeding the resource threshold
- Asymmetric cost to application compared to attacker

Cat 6: Side Effects are not Considered

- Vulnerabilities may go undetected if focusing only on the input to output relationships

```
bool verifyCreds(String input)
    bool correct_length = correctLength(input)
    bool can_print = true
    for(char c: input)
        if !correct(c) && can_print
            send “Error” packet
            can_print = false
         delay()
    return can_print && correct_length
```

- Constant consumption from Input to Output regardless of position of first wrong character
- However, timing of output “Error” packet allows segmented guessing

Cat 7: Focus is Limited to Localized Behavior

- Vulnerabilities may go undetected if the cause and effect of a vulnerability are separated

```
public class Foo
    private int[] m

    // Constructor
    public foo(int[] n)
        m = new int[n.size * n.size]

    // Some Method
    public void Bar()
        for(int i: m)
            Consume 1
```

- Complexity of Bar() is O(m.size), but this means O(n.size^2)!

Cat 8: Only Consider a Single Dimension of Input

- Vulnerabilities may go undetected if effect of multiple dimensions of user input is disregarded

```
f1(int n, int m, int p)
    for(int i=0; i<n; i++)
        f2(m,p)

f2(int m, int p)
    for(int i=0; i<m; i++)
        f3(p)

f3(int p)
    for(int i=0; i<p; i++)
        Consume 1
```

- Complexity of $O(n*m*p)$ may be just as bad as $O(n^3)$

Cat 9: Tools Don’t Model Floating Point Math (1/2)

```
private static void function(int x){
    double N = 10000000005.0;
    double z = 0;
    for(int i = 0; i<x; i++){ // z = N*x
        z+=N;
    }
    double w = z/x; // w = z/x = N*x/x = N
    if((long)Math.abs(N - w) != 0){
        // Do computationally expensive calculation
        // Shouldn't happen since w == N
        Thread.sleep(30000);
    }
}
```

Cat 9: Tools Don't Model Floating Point Math (2/2)

- Vulnerabilities due to floating point computation errors may not be caught if the tool does not include a model of floating point computations

```
double w = z/x; // w = z/x = N*x/x = N
if((long)Math.abs(N - w) != 0){
    // Do computationally expensive calculation
    // Shouldn't happen since w == N
    Thread.sleep(30000);
}
```

- Resource Consumption of function depends on the integer component of the absolute value of the floating point error of the addition and division operations.
- Resource Consumption:
  - 30,000 : 6,755,396 < x < 7,355,882 (Approximately 6% of the valid user inputs)
  - 0 : 0 < x < 6,755,397 and 7,355,881 < x < 10,000,000

Cat 10: In-scope implementation of packet queuing vulnerability

- Packet queuing vulnerability counts as in-scope if a mechanism for maintaining the request queue is contained within application.
- Max response time n seconds, queue size q
  - Vulnerable: resource usage limit $\leq n * q$.
  - Non-vulnerable: resource usage limit $> n * q$

Cat 11: Tools Don't Recognize De-coupled Constraints

- Some constraints can be decoupled into sets of constraints which can be analyzed independent of one another.

```
void process(g,s){
  if(g ≤ s){Δ}
  else{~0}}
```

Case A

```
process(g,s)
if(t0)
  if(t1)
    …
      if(tn)
      else{}
    …
  else{}
else{}
```

Case B

```
if(t0)
  if(t1)
    …
      if(tn)
          process(g,s)
      else{process(g,s)}
    …
  else{process(g,s)}
else{process(g,s)}
```

- Cases A and B are equivalent, the set of variables $\{g, s\}$ and the set of variables $\{t_1, t_2, t_3, \dots, t_n\}$ can be analyzed independently.
- The decoupling is more obvious in case A than in case B

Cat 12: Tools Assume SCs Require Conditional Statements (1/3)

- Side channel vulnerabilities can occur without conditional branches. Exception handling for example can be use to cause a side channel vulnerability. The JVM treats conditionals and exceptions differently. The following authentication algorithm uses conditionals to define branching conditions.

```
seqCorrect = exceedLen = 0
bool verifyCreds(String input)
    for(int x =0;x<input.length();x++)
        checkChar(candidate,x+1)
    return seqCorrect == input.length() && exceedLen == 0

void checkChar(String input, int i)
    if(i >= password.length()){exceedLen++}
    else if(password.charAt(i-1) == input.charAt(i-1)){
        if(seqCorrect+1 == i){
            seqCorrect++
            delay()
        }
    }
```

Delay incurred only if all previous chars and current char are correct

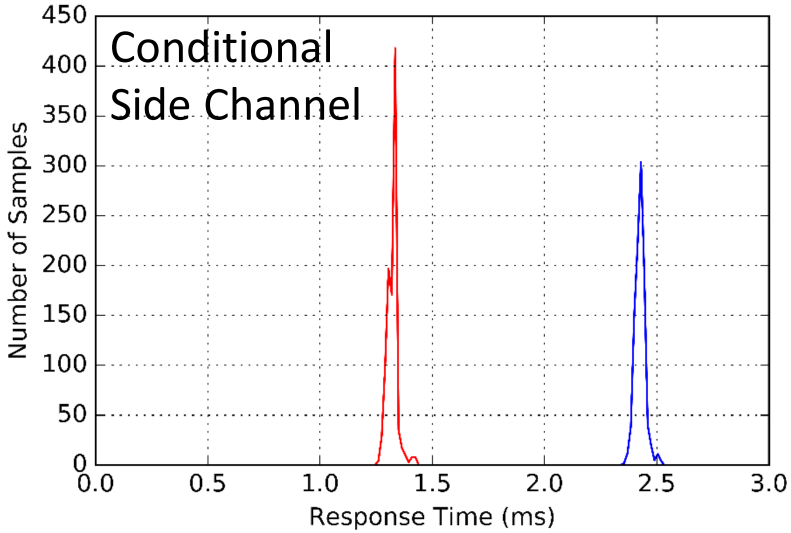
Cat 12: Tools Assume SCs Require Conditional Statements (2/3)

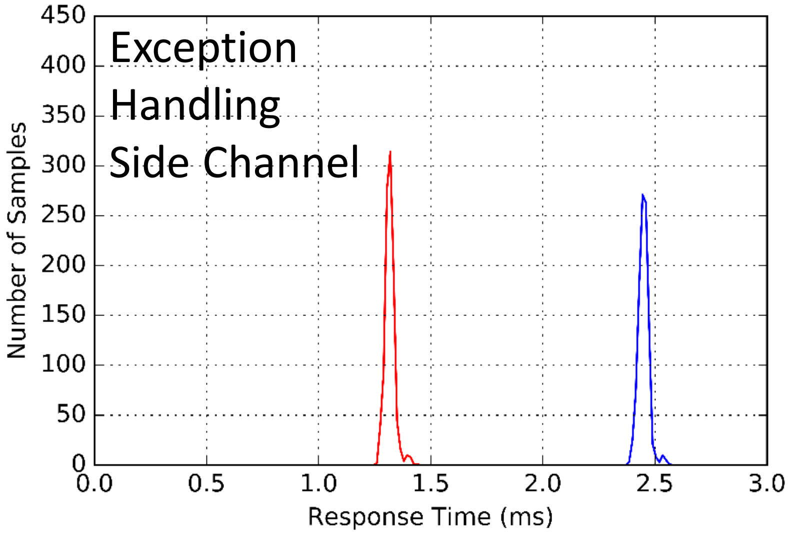
- The authentication algorithm's checkChar method can be re-written with exception handling in place of conditionals.

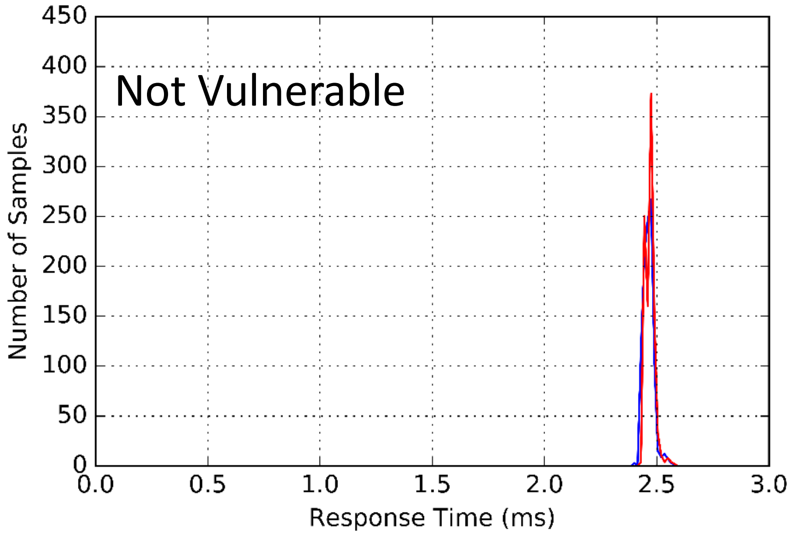
```
void checkChar(String input, int i)
    try{equal=100/(password.charAt(i-1)-input.charAt(i-1))
    }catch(ArithmeticException e1){checkSeqCorrect(i)
    }catch(StringIndexOutofBoundsException e3){
        exceedLen++}

void checkSeqCorrect(int i)
    try{equal=100/(seqCorrect+1 - i)
    }catch(ArithmeticException e2){
        seqCorrect++
        delay()}
```

Delay incurred only if all previous chars and current char are correct

Cat 12: Tools Assume SCs Require Conditional Statements (3/3)





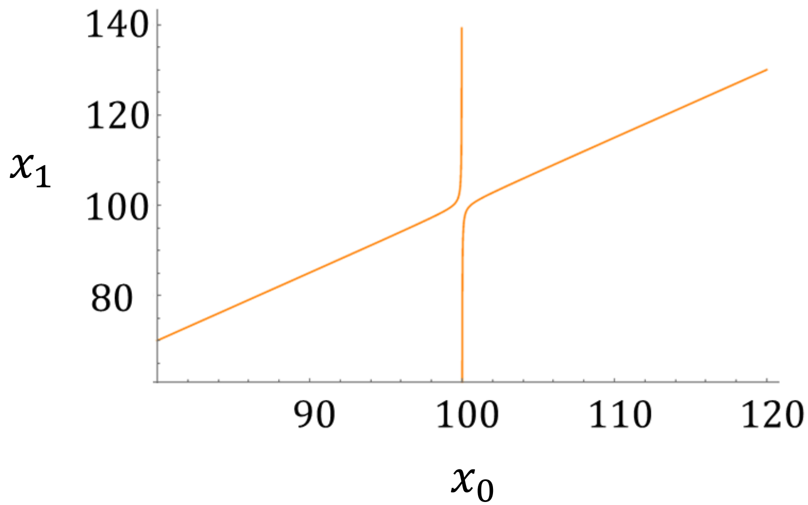


Char 1 Incorrect	
Char 1 Correct	

Cat 13: Sampling Complexity (1/3)

- Tools that under-sample the input range to estimate the complexity function of an application may miss a high frequency spike in the complexity curve.
- Category 13 application uses Newton's method to calculate the roots of a function.
- Newton's method: $x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}; f(x) = (x-M)^2 - \varepsilon$
  - Roots are: $M \pm \sqrt{\varepsilon}$
  - Terminating condition: $|x_{n+1} - x_n| < d$ and $f(x_n) < d$
  - As $x_0 \to M$ or $x_0 \to \infty$, $x_1 \to \infty$. As $x_1 \to \infty$, the number of steps to reach the terminating condition approaches infinity.
- 2 vulnerable regions around $x_0 = 100$
- Input budget allows for value positive values up to $10^{3000} - 1$
- Percentage of vulnerable x_0 values:

$$\frac{2 * 10^{531}}{3 * 10^{3003}} = 6.67 * 10^{-2471}\%$$



Cat 13: Sampling Complexity (2/3)

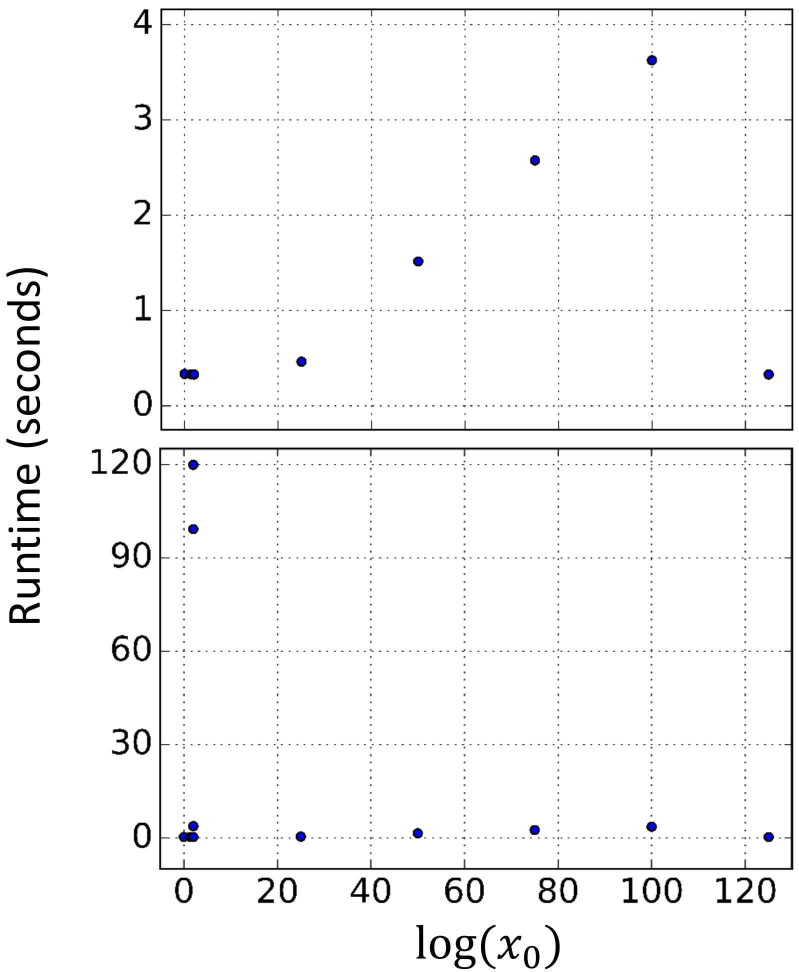
```
M = 100; ε = 1; d = 10^-100
BigDecimal f(x) {return (x − M)^2 − ε}
BigDecimal d(x) {return 2(x − M)}
BigDecimal nextX(x)
    return x − f(x)/d(x)
int newtonMethod(x_0)
    n = 0
    xCurrent = x_0
    do{
        x_P = x_C
        x_C = nextX(x_P)
        n++
    }while(|x_C − x_P| < d and f(x_C) < d)
    return n
```

Where $M = 100; \varepsilon = 1; d = 10^{-100}$, $f(x) = (x-M)^2 - \varepsilon$, $d(x) = 2(x-M)$, nextX returns $x - \frac{f(x)}{d(x)}$, and the loop condition is $|x_C - x_P| < d$ and $f(x_C) < d$.

Cat 13: Sampling Complexity (3/3)

- AC Time experimental data using E5+ AC Time definition:
  - Benign user input: $x_0 = 0$; normal runtime: 0.34 seconds

x_0	Runtime (seconds)
0	0.34
50	0.33
99.9	0.33
$100 - 10^{-100}$	3.80
$100 - 10^{-2465}$	99.21
$100 - 10^{-2996}$	119.87
101	0.33
10^{50}	1.52
10^{75}	2.58
10^{100}	3.62
10^{125}	0.33



Cat 14: Non-local balancing for SC (1/3)

- Tools that perform SC analysis on individual methods or code structures may misclassify an application as vulnerable in cases where for example a timing imbalance in one region of the code is offset in another region of the code leaving the application non-vulnerable as a whole.

```
seqCorrect = exceedLen = 0
void checkChar(String input, int i)
    if(i >= password.length()){exceedLen++}
    else if(password.charAt(i-1) == input.charAt(i-1)){
        if(seqCorrect+1 == i){
            seqCorrect++
            delay()
        }
    }
```

Delay incurred only if all previous chars and current char are correct

```
bool verifyCreds(String input)
    for(int x =0;x<input.length();x++)
        checkChar(candidate,x+1)
    balance(input.length() - seqCorrect)
    return seqCorrect == input.length() && exceedLen == 0
```

Delay incurred for each sequentially correct character

Cat 14: Non-local balancing for SC (2/3)

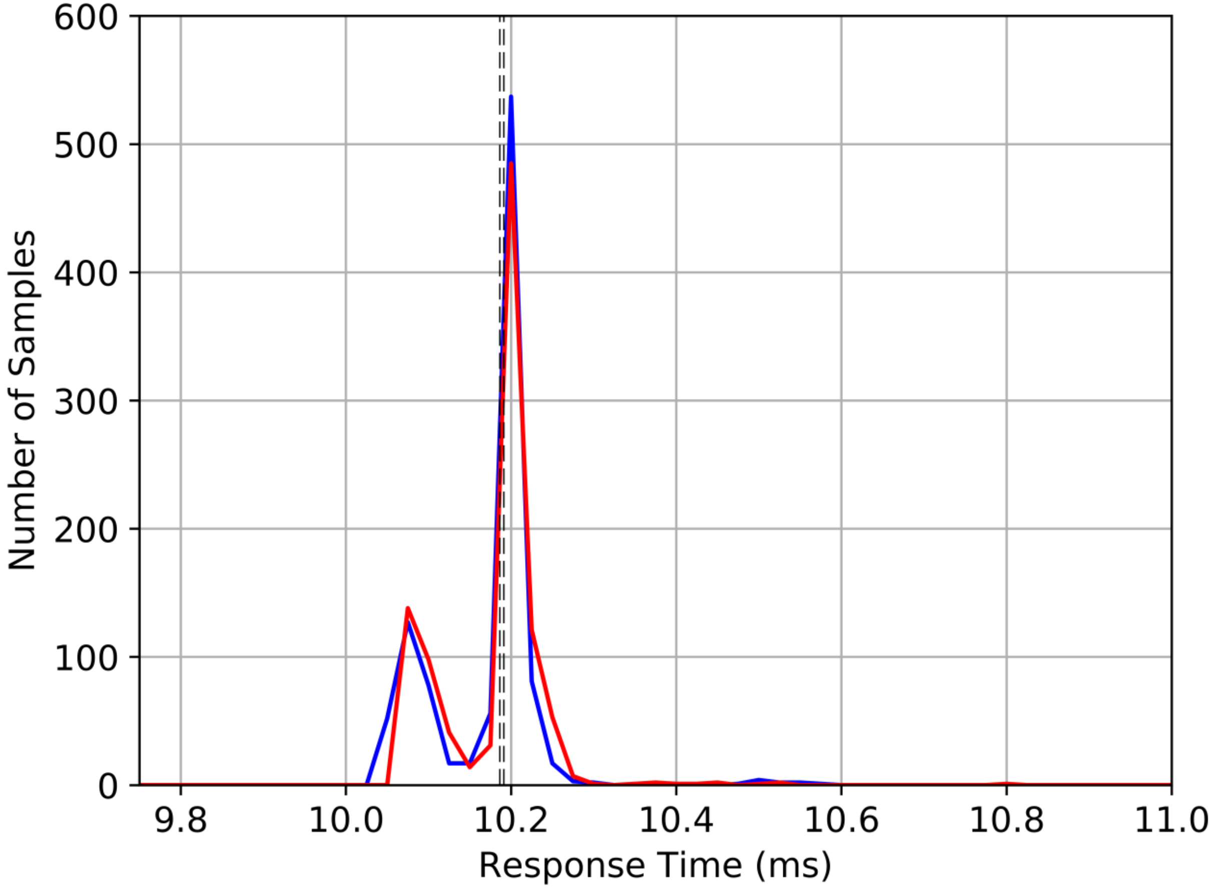
```
void balance(int offset)
    for(int x =0;x<offset;x++)
        delay()
```

Delay incurred for each incorrect character after the first incorrect character is reached

```
seqCorrect = exceedLen = 0
bool verifyCreds(String input)
    for(int x =0;x<input.length();x++)
        checkChar(candidate,x+1)
    balance(input.length() - seqCorrect)
    return seqCorrect == input.length() && exceedLen == 0
```

Delay incurred for each user-provided character preventing a side channel

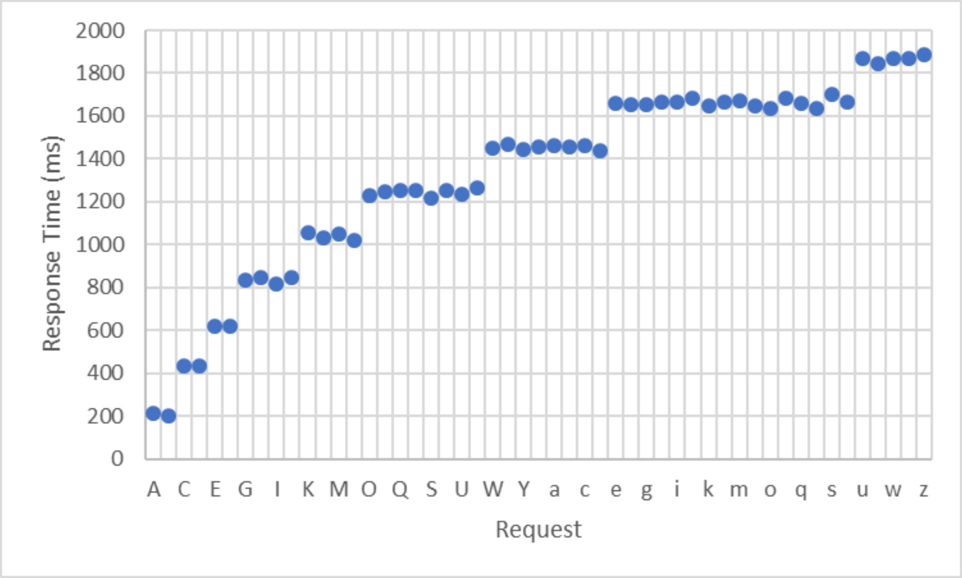
Cat 14: Non-local balancing for SC (3/3)



Char 1 Incorrect	
Char 1 Correct	

Cat 15: External Library Callback Vulnerability

- Demonstrates a vulnerability in a callback from an external library method
- java.util.TreeSet uses a tree data structure maintain a sorted order of added elements
- All elements must implement java.lang.Comparable interface (the compareTo() method)
- java.util.TreeSet calls the application's compareTo() method to set the element order
- The application implements a compareTo method that when combined with the java.util.TreeSet data structure results in a vulnerability

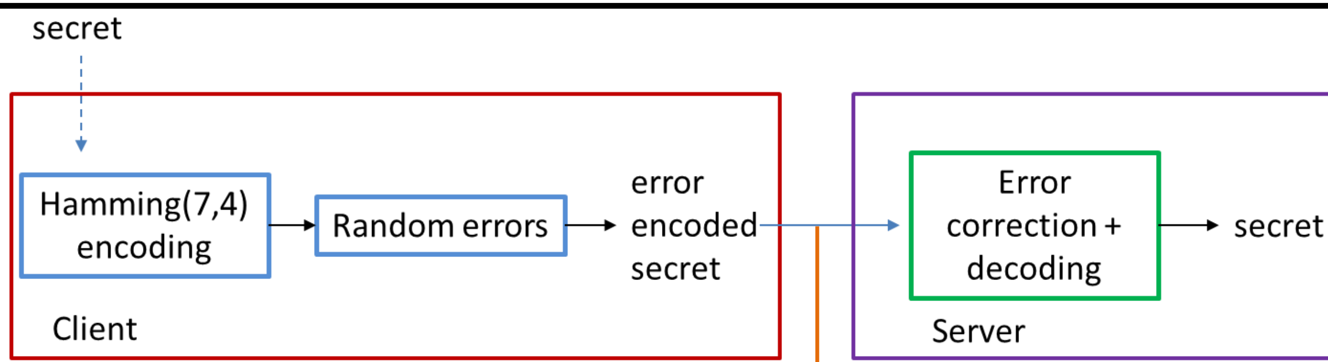


Attacker requests: "A",...,"Z", "a",..., "x"
Benign request: "z"

Cat 16: Leaking secret in a different domain (1/3)

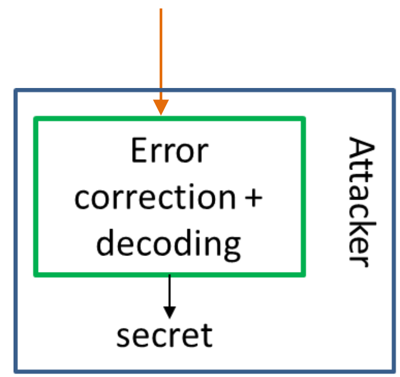
- Applications can leak information on secrets in a transformed domain. Provided a mapping from the transformed state to the secret, leaked information + reverse transformation = secret
- Logically equivalent to an encryption side channel that leaks private keys: encrypted secret + decryption with leaked keys = secret
- Category16 application demonstrates this using a simple-to-analyze Hamming(7,4) linear encoding. Note: Applications can contain more complex non-linear transformations of the secret
  - Client converts secret x to data r and sends r to the server. The server receives r converts it back to the secret x and stores x.
  - The same secret x can result in different values of r
  - Side channel: when the client sends r, bit 0 takes 30 ms and bit 1 takes 60 ms
  - Attacker can determine r from the side channel and use the same algorithm as the server to get the secret x from r.

Cat 16: Leaking secret in a different domain (2/3)




```
prepare(String r){//r as binary string
  for(char bit: r.toCharArray()){
    if(bit == '1'){
      delay(30) //30 ms delay}
    delay(30)
    send(bit)
  }
}
```

$$bit = 0; \Delta_t = 30\ ms$$
$$bit = 1; \Delta_t = 60\ ms$$



Cat 16: Details on hamming encoding (3/3)

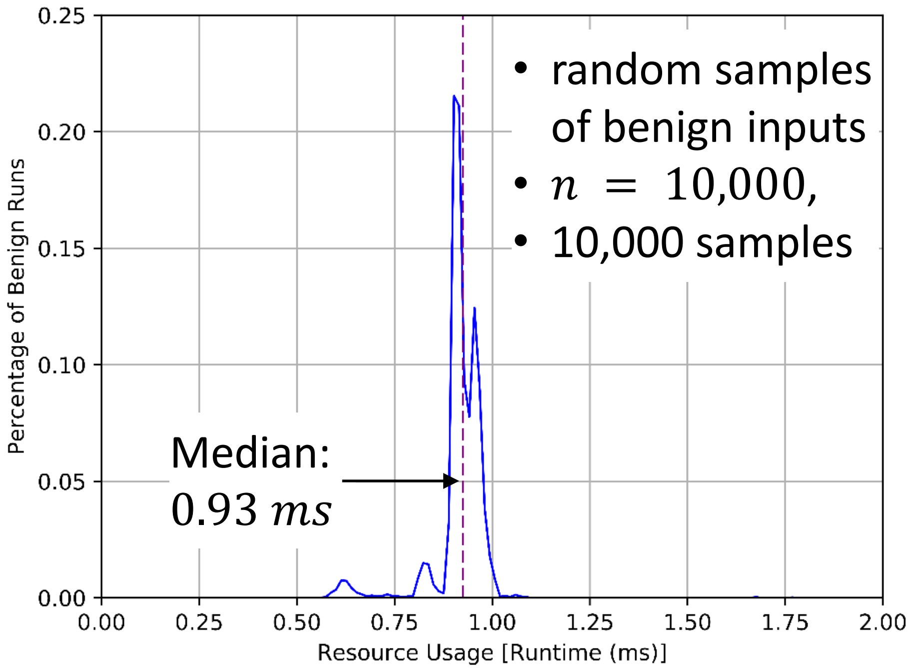
- Hamming (7,4) encodes 4 data bits and detects and corrects cases where a single transmitted bit is accidentally flipped (single bit error).
  - Encodes 4 data bits, p, with 3 parity bits using matrix G
  - Transmitted data: $x = Gp$ (x is the 7 transmitted bits)
  - Matrix H ($HG = [0]$) is used to check errors: $z = Hr$, where r is the 7 received bits
    - if no errors occurred: $r = x$ and $z = Hx = [0]$
    - If single bit error at received bit i: $r = x + e_i$ and $z = Hx + He_i = He_i$
  - If only a single bit error occurs, z indicates which bit is incorrect. That bit is flipped to result in x
  - Matrix R is used to decode x: $Rx = RGp = p$ (4 original data bits)
- Code from server can be used to determine the secret, x, from the data leaked from the side channel, r
  - Entropy of leaked data $>$ entropy of the secret. Leaked data (7 bits), secret (4 bits)
  - Transmitted data (r, Hamming encoding + 1-bit error) may appear to have sufficient entropy resulting in a weak SC
  - The side channel leaks r. Knowledge of operations with G, H, and R creates a deterministic mapping from leaked data r to the secret x.

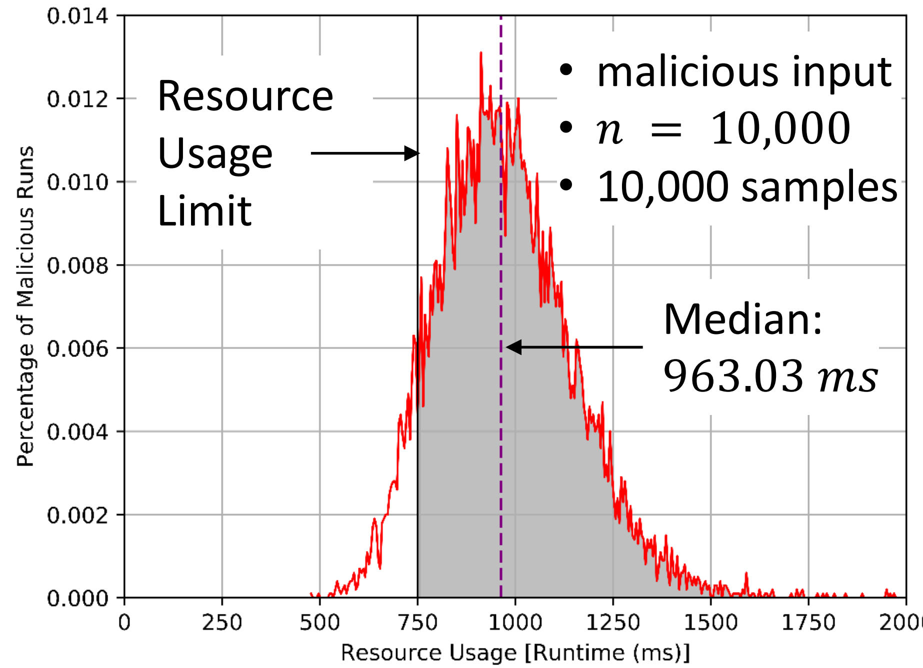
Cat 17: Probabilistic Complexity Function (1/2)

- Probabilistic algorithms are used to solve problems ranging from primality testing to sorting algorithms.
- For example, Quicksort is a sorting algorithm that continually segments the list of items to be sorted.
  - At each step, a pivot parameter controls how the list of items is partitioned into smaller sub-lists.
- Quicksort ranges in complexity from $\mathrm{O}(n \log n)$ to $\mathrm{O}(n^2)$ (given n items) depending on the pivot selection.
  - In the best case, $O(n \log n)$, partition sizes are approximately equal
  - In the worst case, $O(n^2)$, the pivot is always the largest or smallest item; given a sub-list of size m, partition sizes are 0, and $m - 1$
- A probabilistic Quicksort implementation with random pivot selection provides a mean performance of $\mathrm{O}(n \log n)$ for any given input.
- This vulnerable probabilistic Quicksort implementation randomly selects the pivot position, but heavily favors selecting the n^{th} item in the list of n items.

Cat 17: Probabilistic Complexity Function (2/2)

- An attacker can use this knowledge to provide a sorted list of the maximum allowed size ($n = 10{,}000$). A benign user request will be delayed by the time to process this malicious request.





- Input Budget: 100 kB
- Resource Usage Limit: $750\ ms$
- Probability of Success: 50%