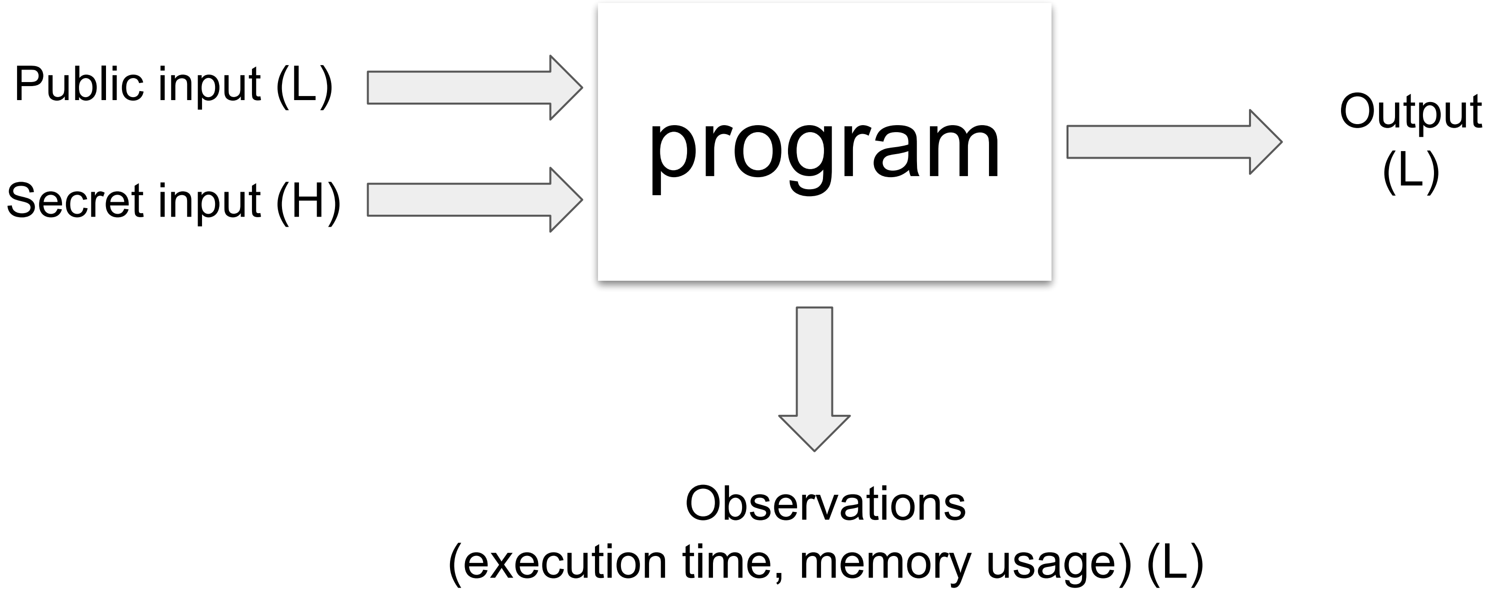
Symbolic Quantitative Information Flow Analysis

Tevfik Bultan

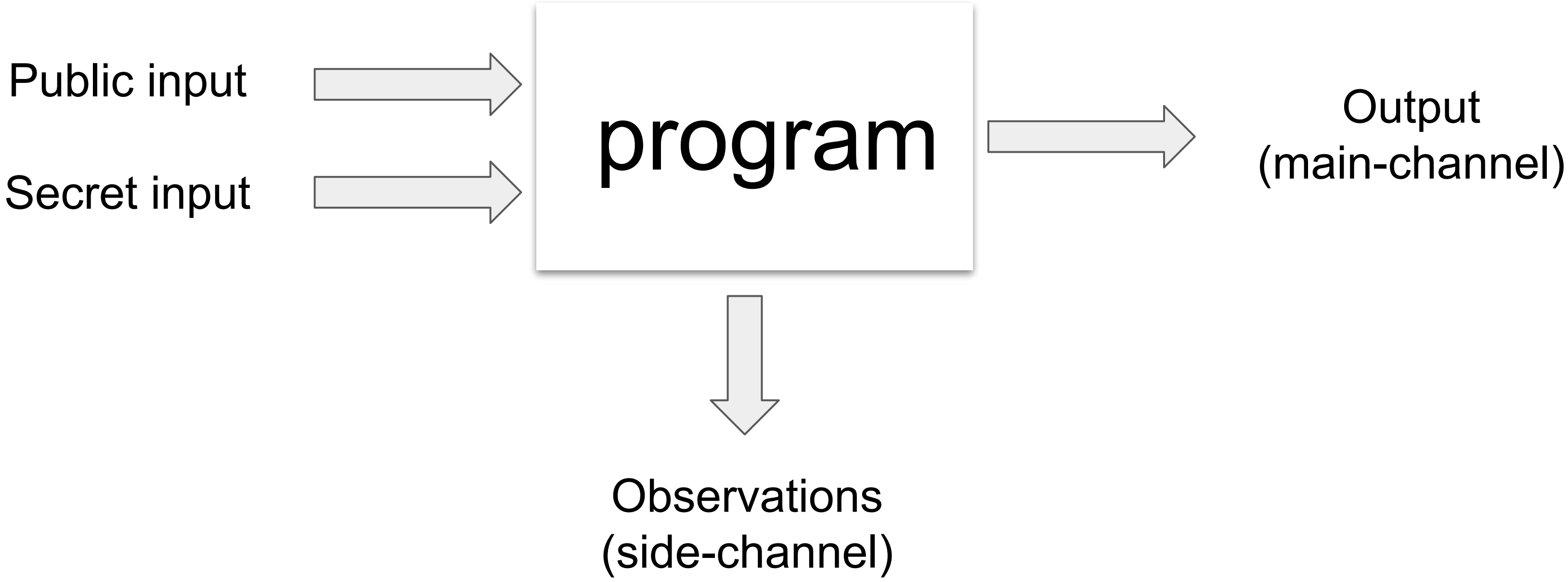
COMPUTER SCIENCE
UC SANTA BARBARA

Side channels

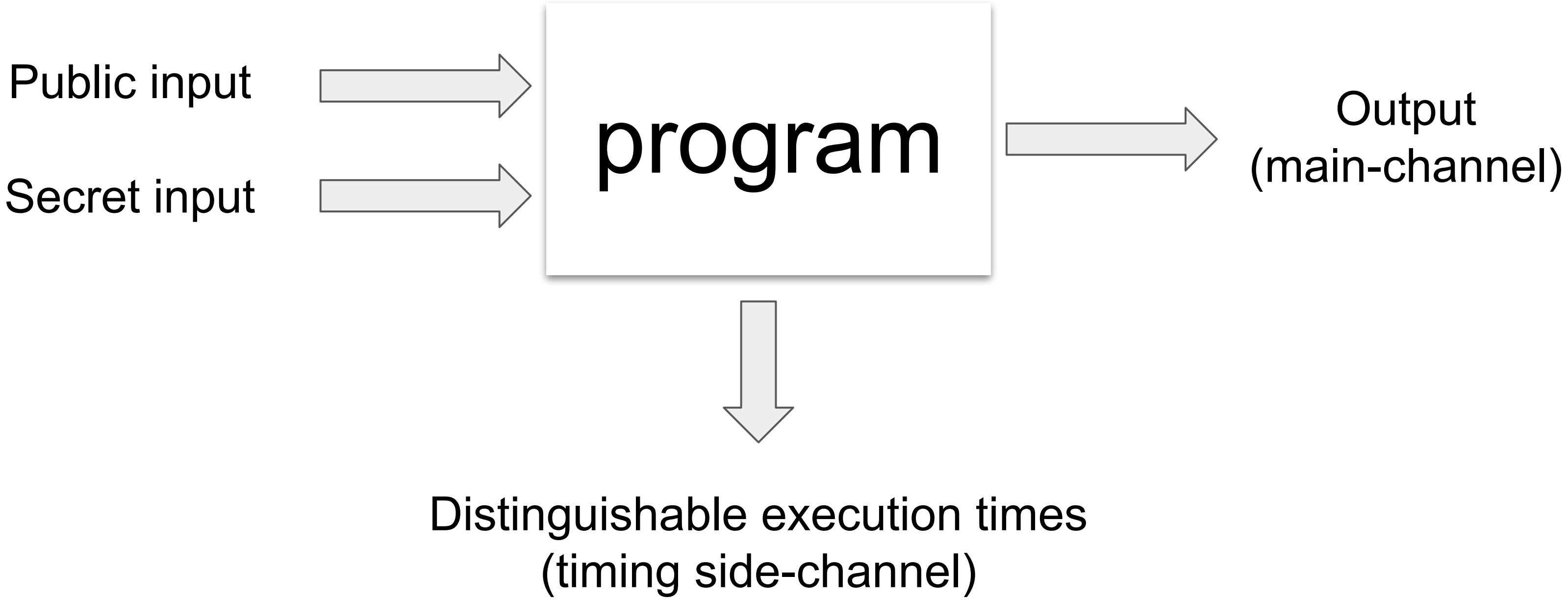
Side Channels



Side channels



Timing side channel



Timing side channel in password checking function

Public input (l)

Secret input (h)

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false/true

Distinguishable execution time

Timing side channel in password checking function

l = "XXXXXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 5 milliseconds

Timing side channel in password checking function

l = "XXXXXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 5 milliseconds

Timing side channel in password checking function

l = "ALAB@UCSB"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 5 milliseconds

Timing side channel in password checking function

l = "VXXXXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 6 milliseconds

(assume one loop iteration takes 1 millisecond)

Timing side channel in password checking function

l = "VLXXXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 7 milliseconds

Timing side channel in password checking function

l = "VLAXXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 8 milliseconds

Timing side channel in password checking function

l = "VLABXXXXX"

h = "VLAB@UCSB"

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

false

Execution time = 9 milliseconds

Timing side channel in password checking function

l = “VLAB@UCSB”

h = “VLAB@UCSB”

```
public boolean passwordChecker(String h,
String l) {

    for (int i = 0; i < h.length(); i++) {

        if (h[i] != l[i])
            return false
    }

    return true
}
```

true

Execution time = 14 milliseconds

Segment oracle side-channel vulnerability

- Segment oracle vulnerability:
  - attacker reveals the secret input segment by segment

- A prefix attack that determines each character one by one starting with the leftmost character can recover the password

Segment oracle side-channel vulnerability

Timing attack in Google Keyczar library

Filed under: Crypto, Hacking, Network, Protocols, python, Security — Nate Lawson @ 11:30 pm

I recently found a security flaw in the Google Keyczar crypto library. The impact was that an attacker could forge signatures for data that was "signed" with the SHA-1

Firstly, I'm really glad to see more high-level libraries being developed so that programmers don't have to work directly with algorithms. Keyczar is definitely a step in t
responding quickly to address this issue after I notified him (Python fix and Java fix).

[security] Widespread Timing Vulnerabilities in OpenID imp

Taylor Nelson taylor at rootlabs.com
Tue Jul 13 20:32:50 UTC 2010

- Next message: [security] Widespread Timing Vulnerabilities in OpenID implementations
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Every OpenID implementation I have checked this far has contained timing
dependent compares in the HMAC verification, allowing a remote attacker
to forge valid tokens.

In JOpenId:
There is a timing vulnerability in thegetAuthentication  function in
trunk/JOpenId/src/org/expressme/openid/OpenIdManager.java
```

Segment oracle side channel vulnerability

```
int memcmp(s1, s2, n)
    CONST VOID *s1; CONST VOID *s2; size t n;
   {
      unsigned char u1, u2;
      for ( ; n- ; s1++, s2++) {
         u1 = * (unsigned char *) s1;
         u2 = * (unsigned char *) s2;
         if ( u1 != u2) { return (u1-u2); }
      }
      return 0;
   }
```

Xbox OS, HMAC signatures compared with memcmp.
Leads to side-channel vulnerability and exploit!

A 4-digit PIN Checker

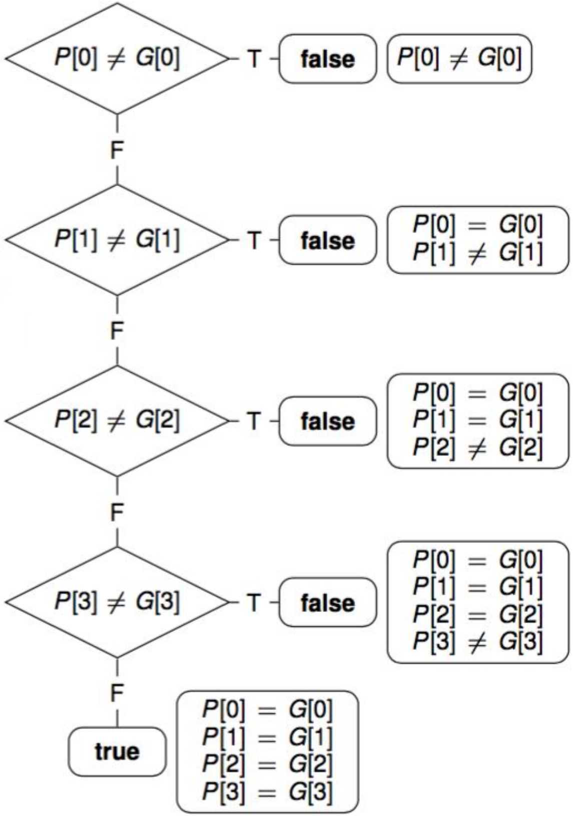
```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess

Symbolic Execution of PIN Checker

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess



Probabilistic symbolic execution

Can we determine the probability of executing a program path?

- Let PC_i denote the path constraint for a program path
- Let $|PC_i|$ denote the number of possible solutions for PC_i
- Let $|D|$ denote the size of the input domain
- Assume uniform distribution over the input domain

- Then the probability of executing that program path is:

$$p(PC_i) = |PC_i| / |D|$$

Probabilistic symbolic execution of PIN checker

- Assume binary 4 digit PIN, P and G each have 4 bits

  $|D| = 2^8 = 256$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$					
p_i					

$p(PC_i) = |PC_i| / |D|$

Probabilistic symbolic execution of PIN checker

- Assume binary 4 digit PIN, P and G each have 4 bits

$$|D| = 2^8 = 256$$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128				
p_i	1/2				

$$p(PC_i) = |PC_i| \,/\, |D|$$

Probabilistic symbolic execution of PIN checker

- Assume binary 4 digit PIN, P and G each have 4 bits

$|D| = 2^8 = 256$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64			
p_i	1/2	1/4			

$p(PC_i) = |PC_i| / |D|$

Probabilistic Symbolic Execution of PIN Checker

- Assume binary 4 digit PIN, P and G each have 4 bits

$|D| = 2^8 = 256$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\lvert PC_i \rvert$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16

Probability that an adversary can guess a prefix of length i in one guess is given by p_i

Extending symbolic execution

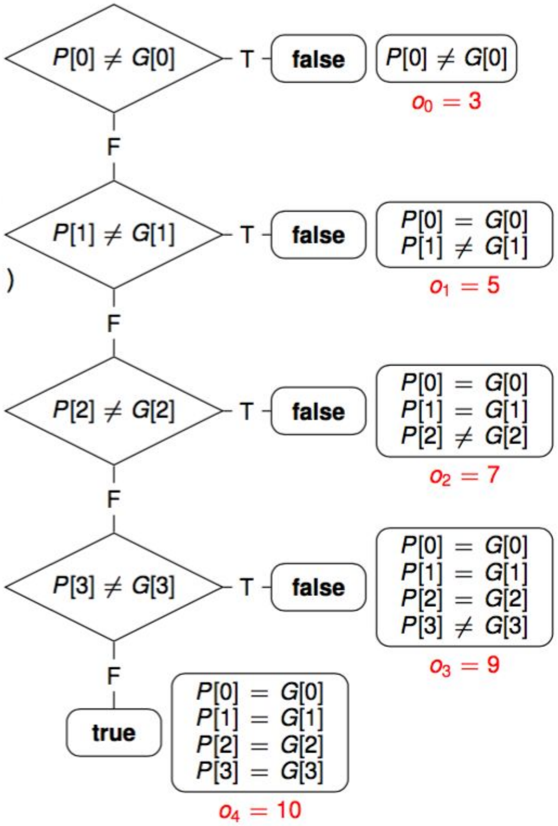
- We need to extend symbolic execution to keep track of observables
- Implement listeners to *collect time/memory costs* for all explored (complete) paths
    - Costs corresponding to the “observables”

Symbolic execution with observable tracking

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, G: guess

o_i = lines of code



Timing side channel:

- Estimate the execution time using the number of instructions executed
- Estimate can be improved with profiling

We call this the ``observable”

- For a space side channel the observable could be amount of memory allocated or size of a file

Probabilistic symbolic execution of PIN checker

- Assume binary 4 digit PIN, P and G each have 4 bits

  $|D| = 2^8 = 256$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\lvert PC_i \rvert$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

Information leakage

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

$$H = \sum p_i \log \frac{1}{p_i} = 1.8750$$

- H: Information leakage or the expected amount of information gain by the adversary

A secure PIN checker

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
          matched = false
    else
          matched = matched
  return matched
```

- Only two observables (just the main channel, no side channel):

  o_0: does not match, o_1: full match

- $p(o_0) = 15/16, p(o_1) = 1/16$
- $H_{secure} = 0.33729$

Secure vs. vulnerable PIN checker

- Given a PIN of length L where each PIN digit has K values
- Secure PIN checker
  - K^L guesses in the worst case
  - Example: 16 digit password where each digit is ASCII

128^{16} tries in the worst case, which would take a lot of time!

Secure vs. vulnerable PIN checker

- Vulnerable PIN checker
  - A **prefix attack** that determines each digit one by one starting with the leftmost digit
  - Example: 16 digit password where each digit is ASCII

128×16 tries in the worst case, which would not take too much time

A case study from DARPA STAC Program: LawDB

- A web service with a law enforcement database that contains
    - Restricted (secret) & unrestricted (public) employee IDs

- Supports SEARCH & INSERT queries
    - Restricted IDs are not visible during SEARCH and INSERT queries

- **Question**: Is there a side channel in time that a third party can determine the value of a single restricted ID in the database?

Code Inspection

- Using code inspection we identified that the SEARCH and INSERT operations are implemented in:

```
class UDPServerHandler
method channelRead0
switch case 1: INSERT
switch case 8: SEARCH
```

Symbolic Path Finder Driver

```
public class Driver {
        public static void main(String[] args){
      BTree tree = new BTree(10);
      CheckRestrictedID checker = new CheckRestrictedID();
      // create two concrete unrestricted ids
      int id1 = 64, id2 = 85;
      tree.add(id1, null, false);
      tree.add(id2, null, false);
      // create one symbolic restricted id
      int h = Debug.makeSymbolicInteger("h");
      Debug.assume(h!=id1 && h!=id2);
      tree.add(h, null, false);
      checker.add(h);
      UDPServerHandler handler = new UDPServerHandler(tree,checker);
      int key = Debug.makeSymbolicInteger("key");
      handler.channelRead0(8,key);  // send a search query with
        }                           // with search range 50 to 100
}
```

SPF Output

```
>>>>> There are 5 path conditions and 5 observables
cost: 9059
(assert (<= h 100))
(assert (> h 85))
(assert (> h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 15
-----------------------
cost: 8713
(assert (<= h 85))
(assert (> h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 20
-----------------------
cost: 7916
(assert (> h 100))
(assert (> h 85))
(assert (> h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 923
-----------------------
cost: 8701
(assert (>= h 50))
(assert (<= h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 14
-----------------------
cost: 7951
(assert (< h 50))
(assert (<= h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 50
-----------------------
***********************************************************
PC equivalence class model counting results.
***********************************************************
Cost: 9059     Count:     15  Probability:  0.014677
Cost: 8713     Count:     20  Probability:  0.019569
Cost: 7916     Count:    923  Probability:  0.903131
Cost: 8701     Count:     14  Probability:  0.013699
Cost: 7951     Count:     50  Probability:  0.048924

Domain Size: 1022
Single Run Leakage: 0.6309758112933285
```

Observation & Proposed Attack

- SEARCH operation:

takes longer when the secret is within the search range (9059, 8713, 8701 byte code instructions)

as opposed to the case when the secret is out of the search range (7916, 7951 byte code instructions)

- ***Proposed attack***: Measure the time it takes for the search operation to figure out if there is a secret within the search range

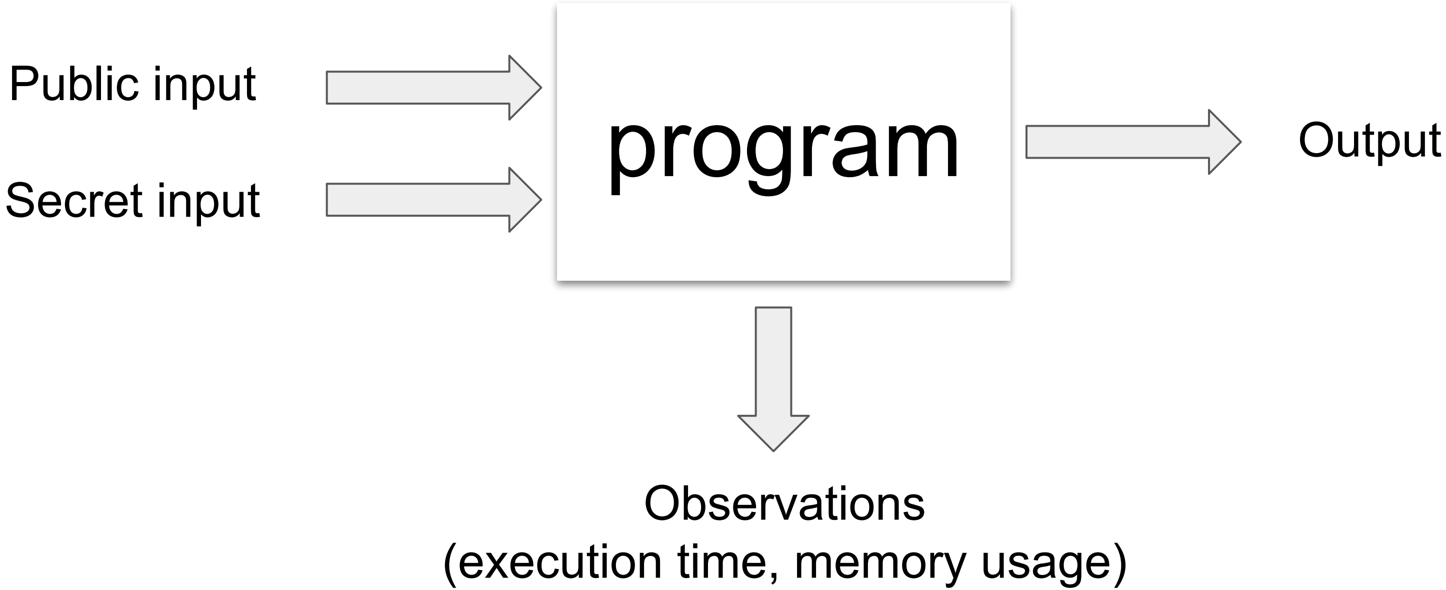
Proposed Attack

- Binary search on the ranges of the IDs
- Send two search queries at a time and compare their execution time
- Refine the search range based on the result

Attack

```
Running [0, 40000000] at 0.
Comparing 467821 vs 612252...
Running [20000000, 40000000] at 2.
Comparing 400377 vs 333665...
Running [20000000, 30000000] at 4.
Comparing 200603 vs 237025...
Running [25000000, 30000000] at 6.
Comparing 163564 vs 115072...
Running [25000000, 27500000] at 8.
Comparing 95736 vs 37388...
Running [25000000, 26250000] at 10.
Comparing 85305 vs 30118...
Running [25000000, 25625000] at 12.
Comparing 22765 vs 72958...
Running [25312500, 25625000] at 14.
Comparing 2147483647 vs 19353...
Running [25312500, 25468750] at 16.
Comparing 517 vs 2147483647...
Running [25390625, 25468750] at 18.
Comparing 317 vs 2147483647...
Running [25429687, 25468750] at 20.
Comparing 2147483647 vs 302...
Running [25429687, 25449218] at 22.
Comparing 2147483647 vs 287...
Running [25429687, 25439452] at 24.
Comparing 336 vs 2147483647...
Running [25434569, 25439452] at 26.
Comparing 300 vs 2147483647...
Running [25437010, 25439452] at 28.
Comparing 2147483647 vs 265...
Running [25437010, 25438231] at 30.
Comparing 2147483647 vs 328...
Running [25437010, 25437620] at 32.
Comparing 280 vs 2147483647...
Running [25437315, 25437620] at 34.
Comparing 293 vs 2147483647...
Running [25437467, 25437620] at 36.
Comparing 2147483647 vs 281...
Running [25437467, 25437543] at 38.
Comparing 2147483647 vs 613...
Running [25437467, 25437505] at 40.
Comparing 2147483647 vs 258...
Running [25437467, 25437486] at 42.
Comparing 2147483647 vs 291...
Running [25437467, 25437476] at 44.
Comparing 362 vs 2147483647...
Running [25437471, 25437476] at 46.
Comparing 311 vs 2147483647...
Running [25437473, 25437476] at 48.
Comparing 2147483647 vs 2147483647...
Checking oracle for: 25437474... true
Checking oracle for: 25437475... false
```

Can we automate attack synthesis?



- ***Which public input values would allow us to learn the secret as fast as possible?***

A Simple Function

```
public int comparison(int i) {

    if(s <= i)
        do something simple; // 1 milisecond
    else
        do something complex; // 2 miliseconds

    return 0;
}
```

A Simple Function

```
public int comparison(int i) {

    if(s <= i)
        do something simple; // 1 milisecond
    else
        do something complex; // 2 miliseconds

    return 0;
}
```

$$O = 1 \Rightarrow s \leq i$$

$$O = 2 \Rightarrow s > i$$

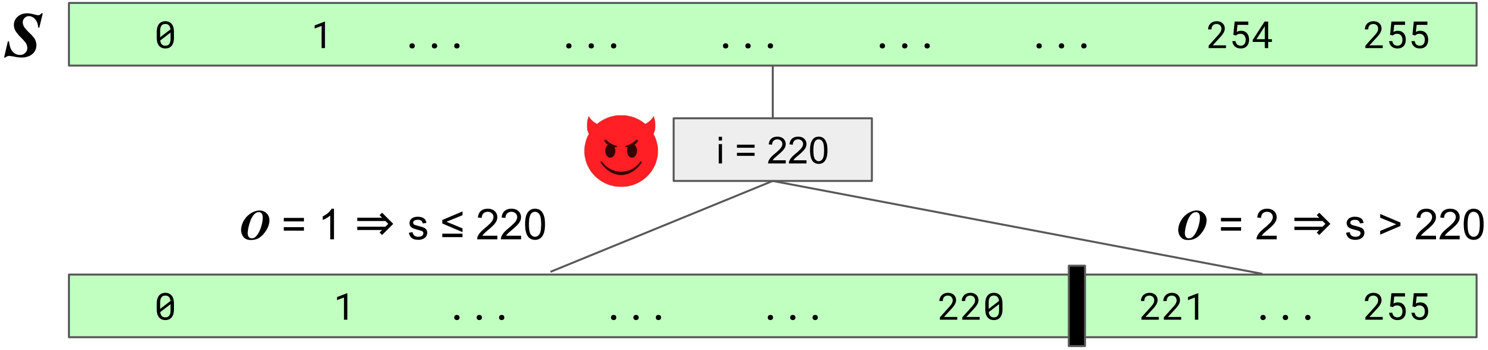
$$O = 1 \Rightarrow s \leq i$$

$$O = 2 \Rightarrow s > i$$

S								
	0	1	...	...	...	...	...	254	255

$O = 1 \Rightarrow s \leq i$

$O = 2 \Rightarrow s > i$



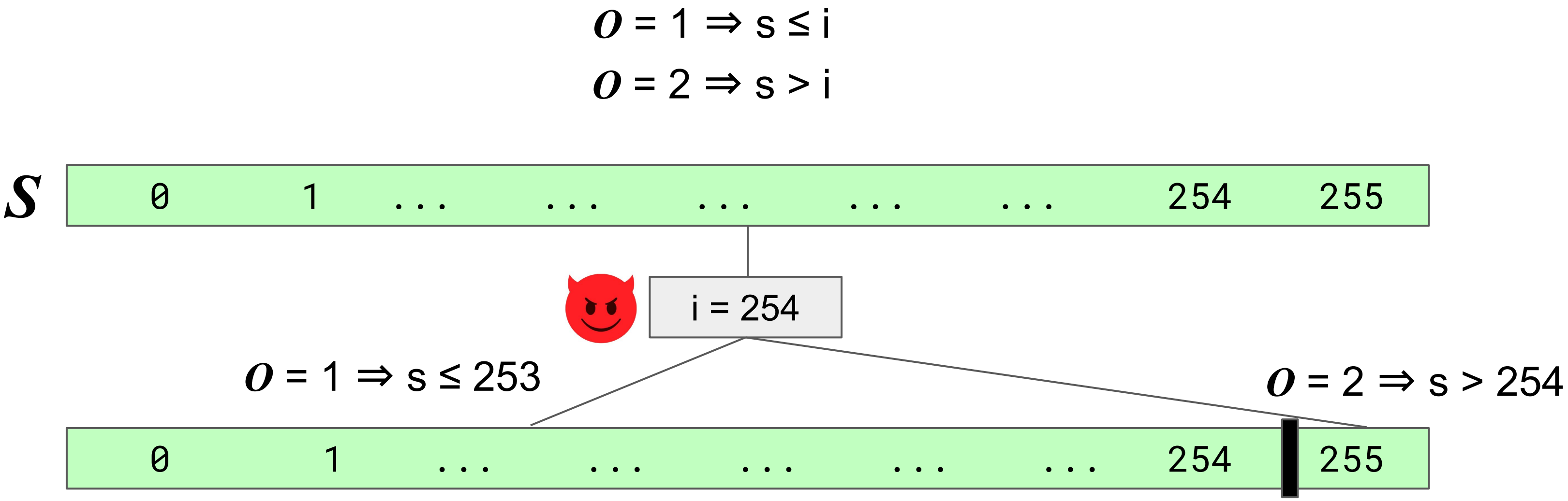
Attacker's input and observation **partitions** domain of S

How should the attacker
choose the inputs
to reveal the secret
as fast as possible?

$O = 1 \Rightarrow s \leq i$

$O = 2 \Rightarrow s > i$

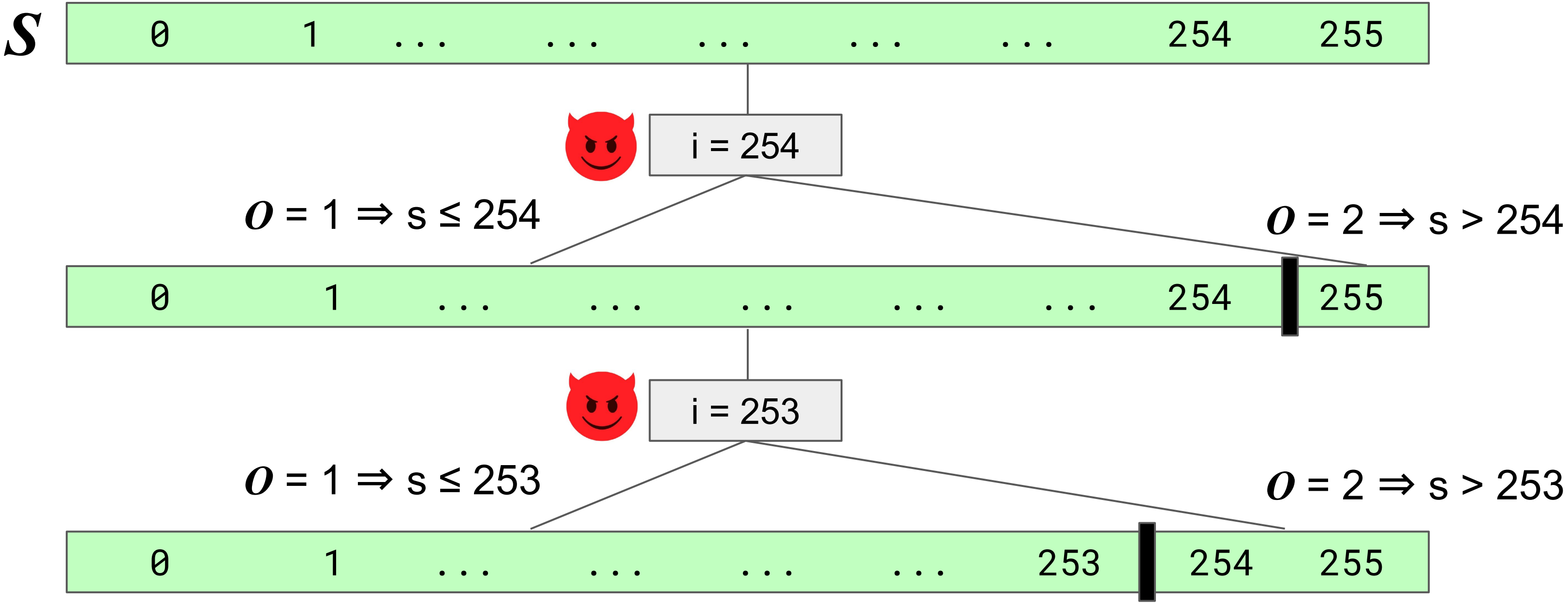
S	0	1	...	...	...	...	...	254	255



𝒪 = 1 ⇒ s ≤ i
𝒪 = 2 ⇒ s > i
S
0 1 ... ... 254 255
i = 254
𝒪 = 1 ⇒ s ≤ 253
𝒪 = 2 ⇒ s > 254
0 1 ... ... 254 255

$O = 1 \Rightarrow s \leq i$

$O = 2 \Rightarrow s > i$



Imbalanced partitions

⇓

Worst case :

number of inputs = domain size = 2^8 = 256

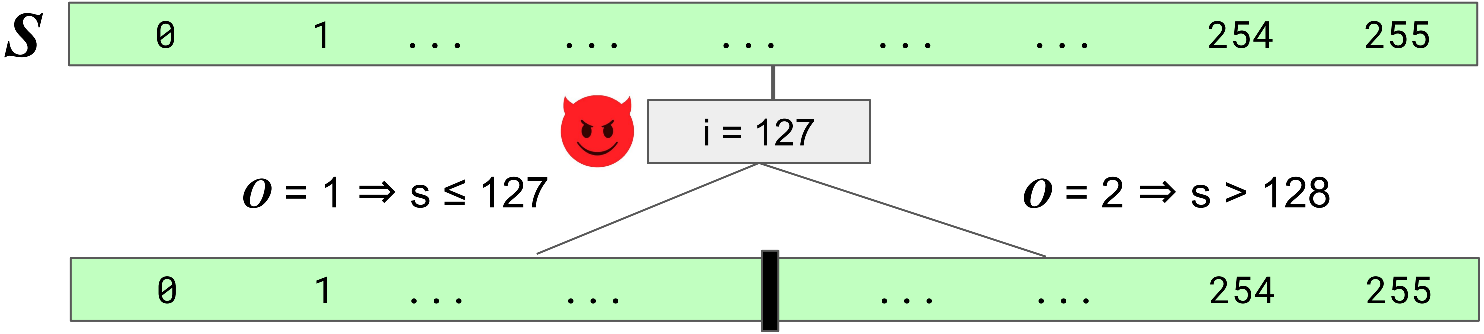
$\boldsymbol{o} = 1 \Rightarrow s \leq i$

$\boldsymbol{o} = 2 \Rightarrow s > i$

$\boldsymbol{S}$	0	1	...	...	...	...	...	254	255

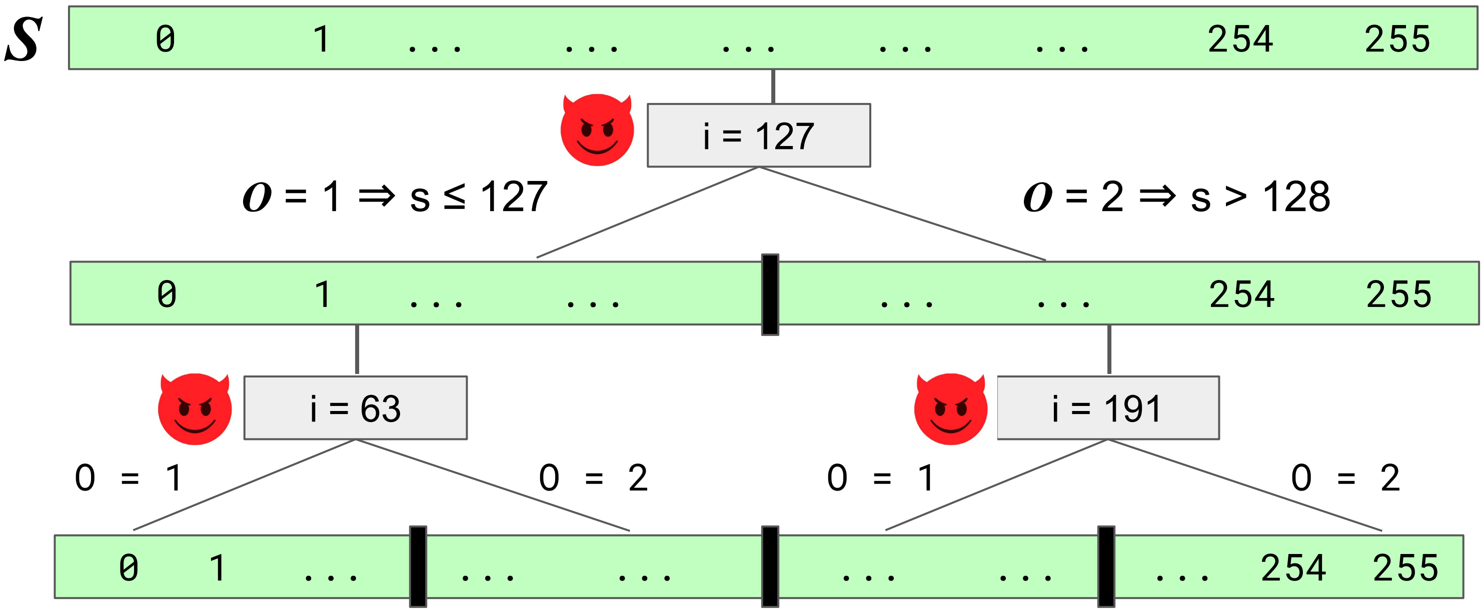
$O = 1 \Rightarrow s \leq i$

$O = 2 \Rightarrow s > i$



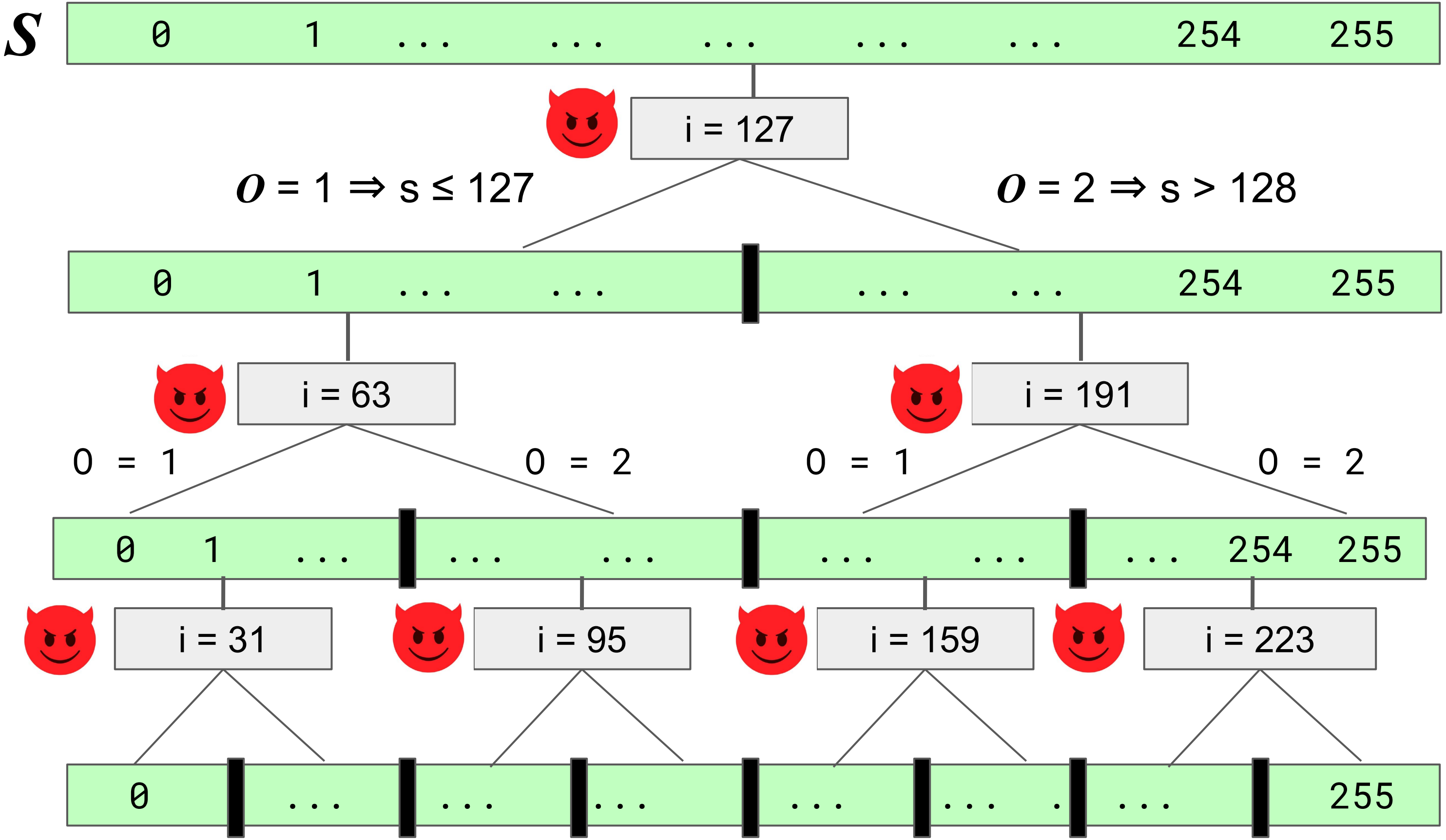
$O = 1 \Rightarrow s \leq i$

$O = 2 \Rightarrow s > i$



$\mathcal{O} = 1 \Rightarrow s \leq i$

$\mathcal{O} = 2 \Rightarrow s > i$



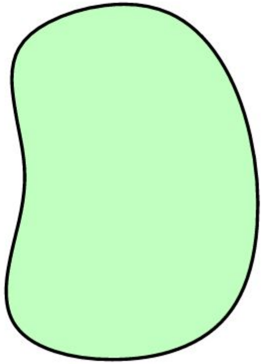
Balanced partitions

Worst case :

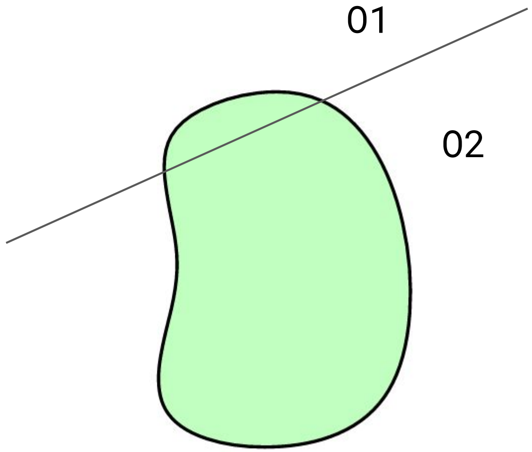
number of inputs = $\log_2(256) = 8$

$i_0 \in I$

secret $s \in S$



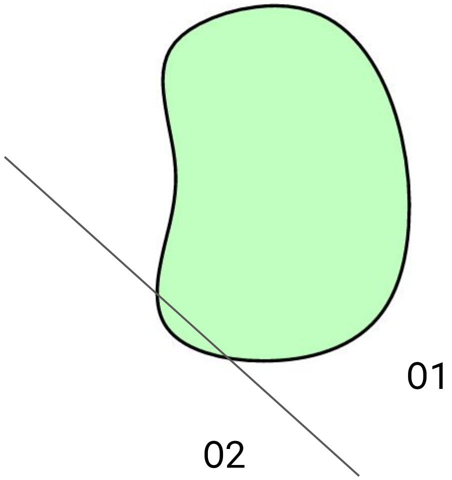
$i_0 \in I$



secret $s \in S$

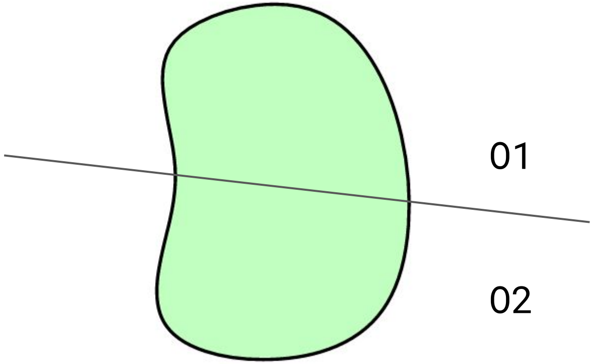
$i_0 \in I$

secret $s \in S$



$i_0 \in I$

secret $s \in S$

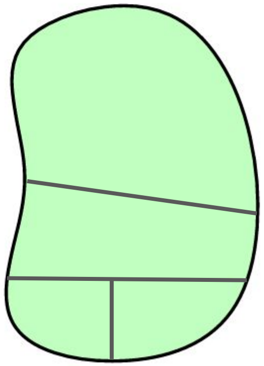


secret $s \in S$

$i_0 \in I$

$i_1 \in I$

$i_2 \in I$



Objective Function

Balanced partitions

$\Downarrow$

Maximizes information gain

Objective Function

$$\boldsymbol{o} = 1 \Rightarrow s \leq i$$
$$\boldsymbol{o} = 2 \Rightarrow s > i$$

Maximize information gain ⇒ Binary Search

Objective Function

$$\boldsymbol{o} = 1 \Rightarrow s \leq i$$
$$\boldsymbol{o} = 2 \Rightarrow s > i$$

Maximize information gain ⇒ Binary Search

Programs in general

Maximize information gain ⇒ Optimal Search

Objective Function

$$\boldsymbol{o} = 1 \Rightarrow s \leq i$$
$$\boldsymbol{o} = 2 \Rightarrow s > i$$

Maximize information gain ⇒ Binary Search

Programs in general

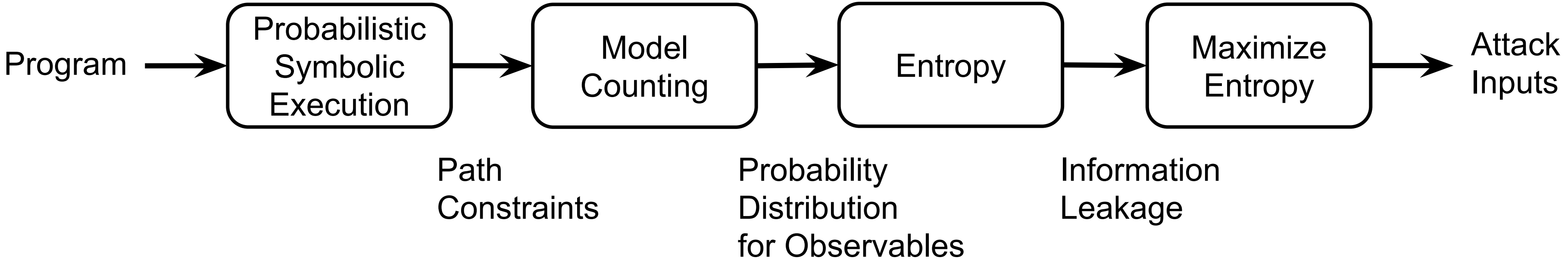
Maximize information gain ⇒ Optimal Attack

Objective Function

information gain

Shannon Entropy

Attack synthesis summary



- The attacks that are synthesized are ***adaptive attacks***
  - Each attack step depends on the results of previous steps
- How to find the input value that maximizes the entropy?
  - Use meta-heuristics such as simulated annealing or genetic algorithm

Two problems with Attack Synthesis

1. It is very expensive to generate the full attack tree statically for all possible secret values
    - At runtime we have a single secret value
2. There is noise at runtime
    - We are using symbolic execution to model the observable values (such as execution time) statically, but there is noise during actual execution

Two problems with Attack Synthesis

1. It is very expensive to generate the full attack tree statically for all possible secret values
    - One idea would be to generate the attack tree at runtime for a particular secret
2. There is noise at runtime
    - We need to model the noise

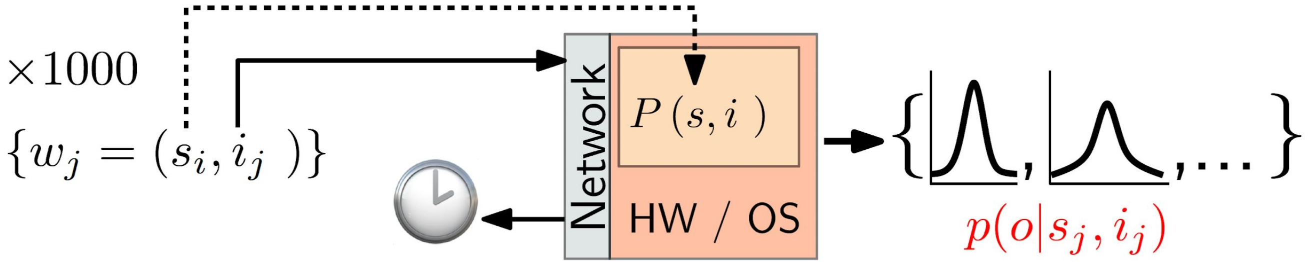
Attack synthesis extension: Online attack synthesis

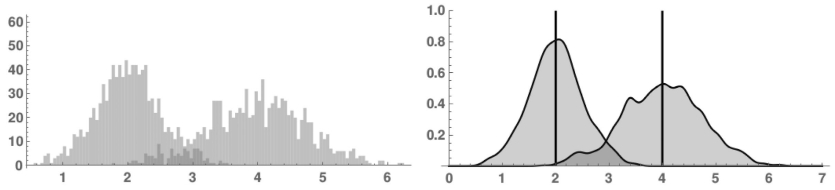
- Generating the full attack tree is expensive
- A full attack tree provides all public input sequences for all possible secret values
  - Full attack tree can be computed offline
  - Exponential blow up with attack depth

- Use online attack synthesis
  - Compute the attack on the fly for a single secret

Attack synthesis extension: Noise modeling

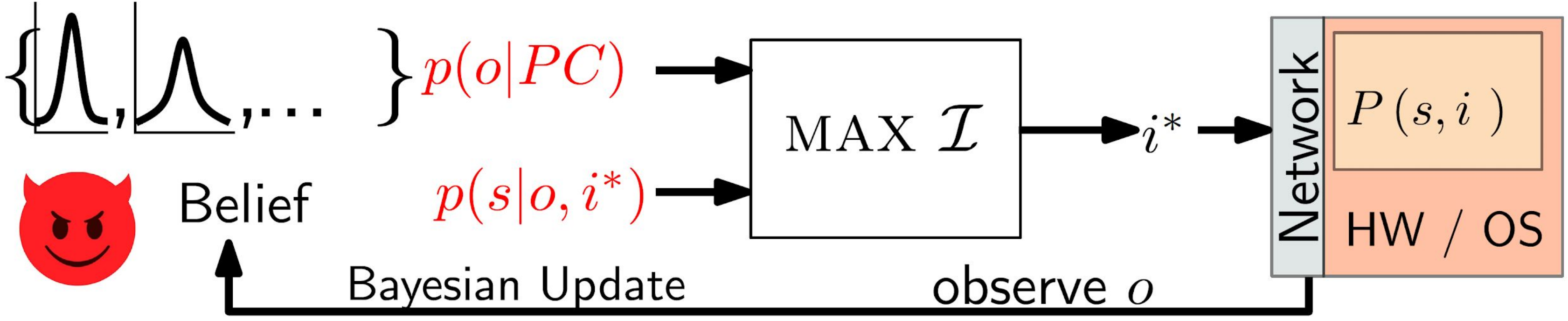
- Use profiling to model the noise
  - Use a witness (a satisfying solution) for each path constraint to profile the observable distribution
  - Generate a noise distribution using smooth kernel density estimation





Attack synthesis extensions: Online attack synthesis

- During attack synthesis, use a probability distribution to model the current belief about the secret
- Use Bayesian inference to update the probability distribution for the secret based on the observations and the noise model



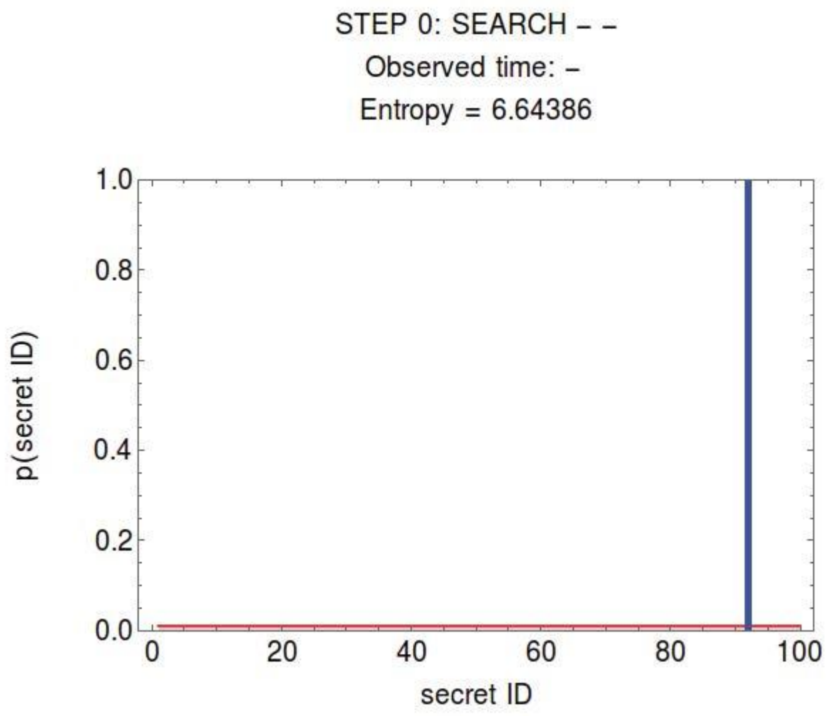
Automatically generated prefix attack against a vulnerable password checker

Phase 0			Phase 1		Phase 2					Phase 3					Phase 4
prefix $= \varepsilon$			prefix $=$ c		prefix $=$ ci					prefix $=$ ciq					prefix $=$ ciqa
ε	fzgk	maau	**c**nte	**c**ved	**ci**ub	**ci**ij	**ci**mq	**ci**tz	**ciq**z	**ciq**i	**ciq**z	**ciq**z	**ciq**u	**ciq**z	**ciqa**
daaz	zgap	vzsc	**c**tdo	**ci**il	**ci**az	**ci**ok	**ci**da	**ci**jw	**ci**hs	**ciq**c	**ciq**z	**ciq**e	**ciq**r	**ciq**r	**ciqa**
uaak	bnza	qyas	**c**vfo	**c**eyu	**ci**gz	**ci**su	**ci**sp	**ci**ne	**ciq**k	**ciq**k	**ciq**d	**ciq**d	**ciq**r	**ciq**z	**ciq**g
ecjq	zmna	asvr	**c**sja	**ci**vf	**ci**fl	**ci**ld	**ci**cz	**ci**le	**ci**eb	**ciq**z	**ciq**q	**ciq**o	**ciq**i	**ciqa**	**ciqa**
tzar	zmna	**c**mxq	**c**wcs		**ci**kt	**ci**pa	**ci**bn	**ci**rx	**ciq**a	**ciq**s	**ciq**z	**ciq**x	**ciq**v		

Secret is “**ciqa**”
Matching characters are shown in **bold**

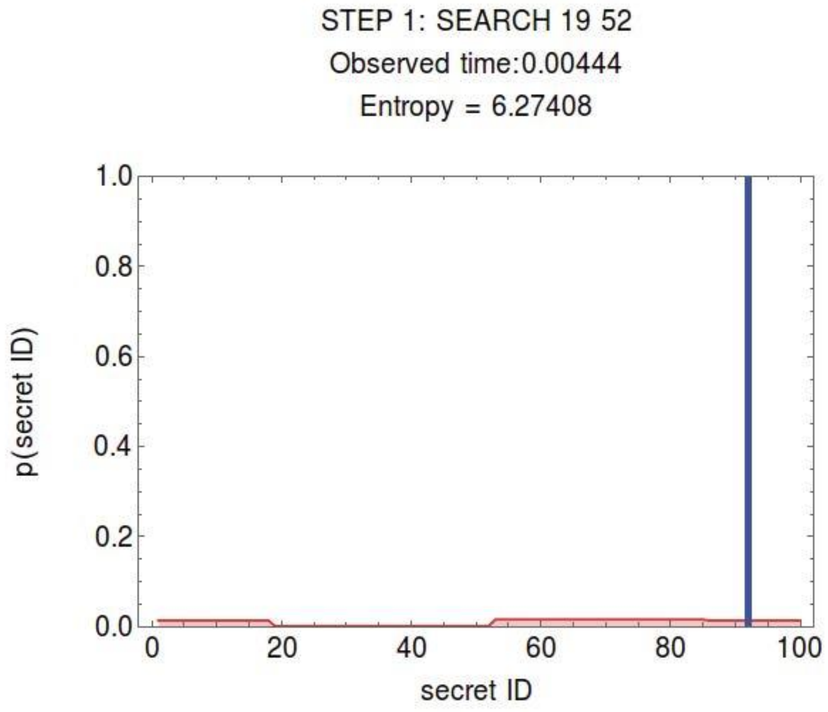
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



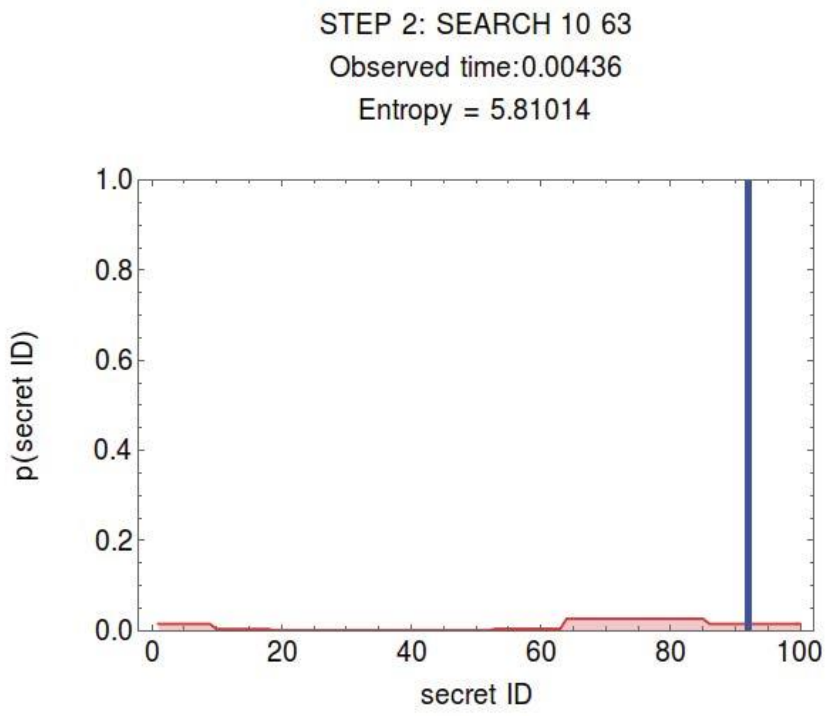
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



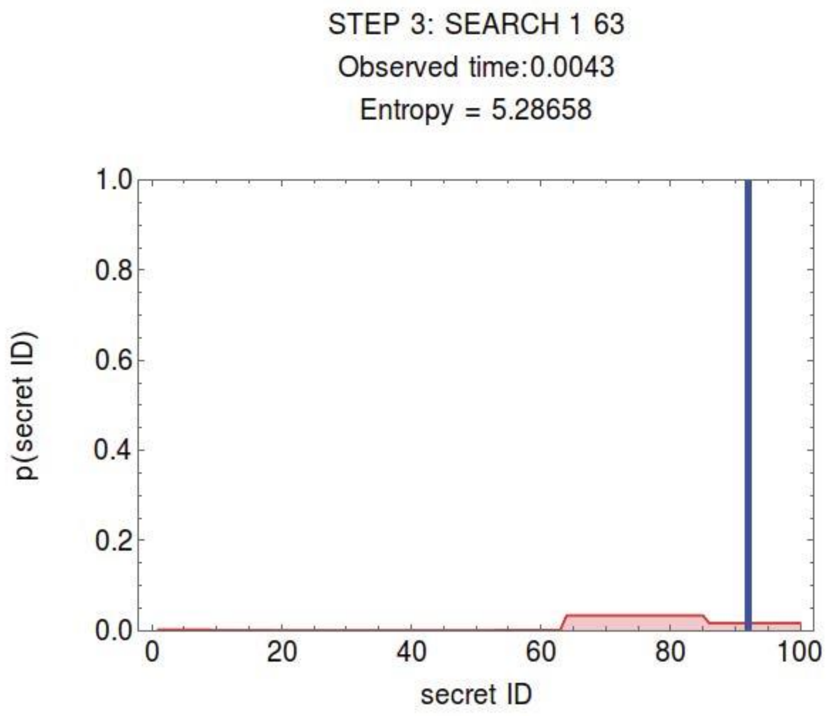
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



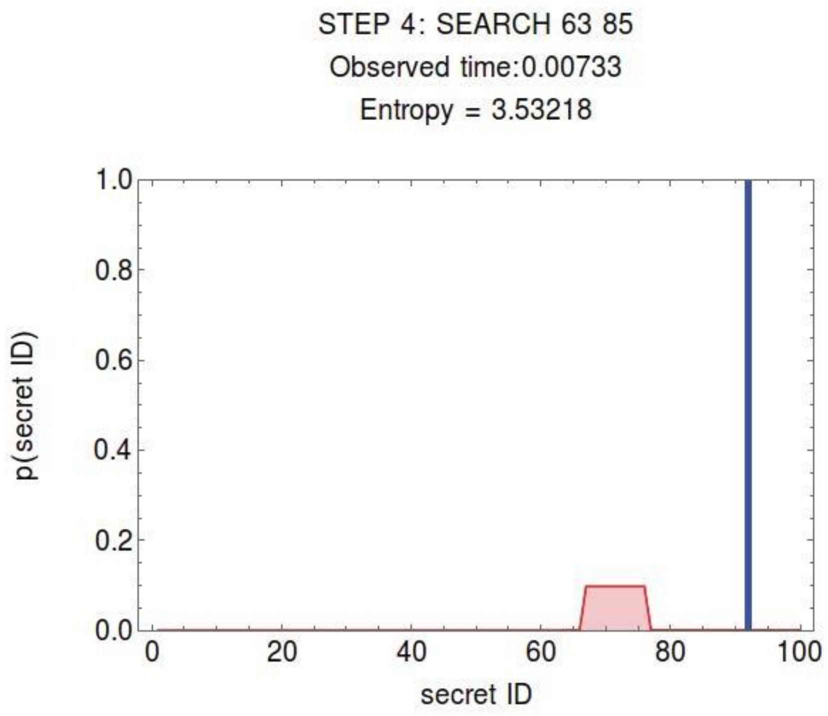
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



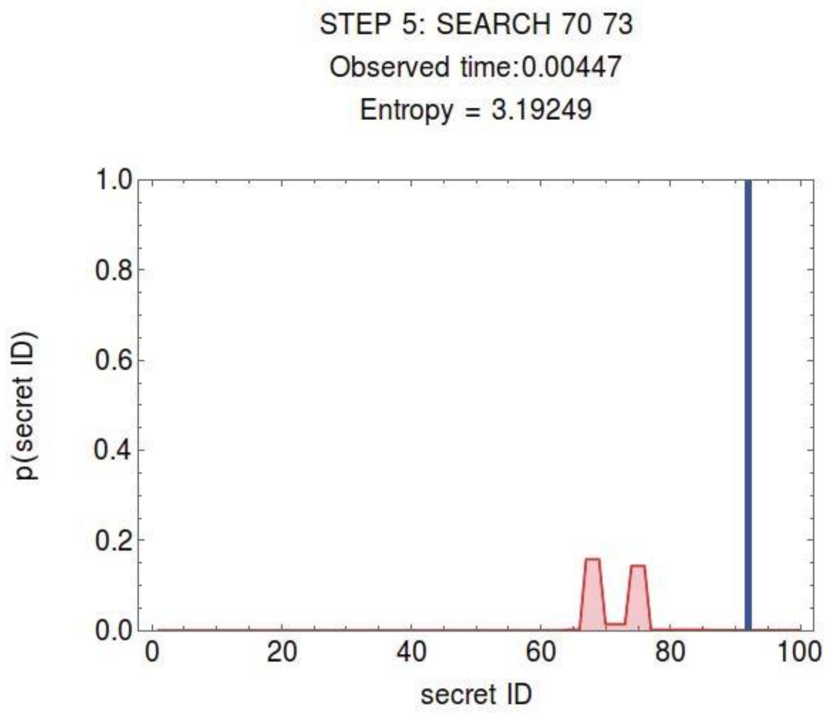
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



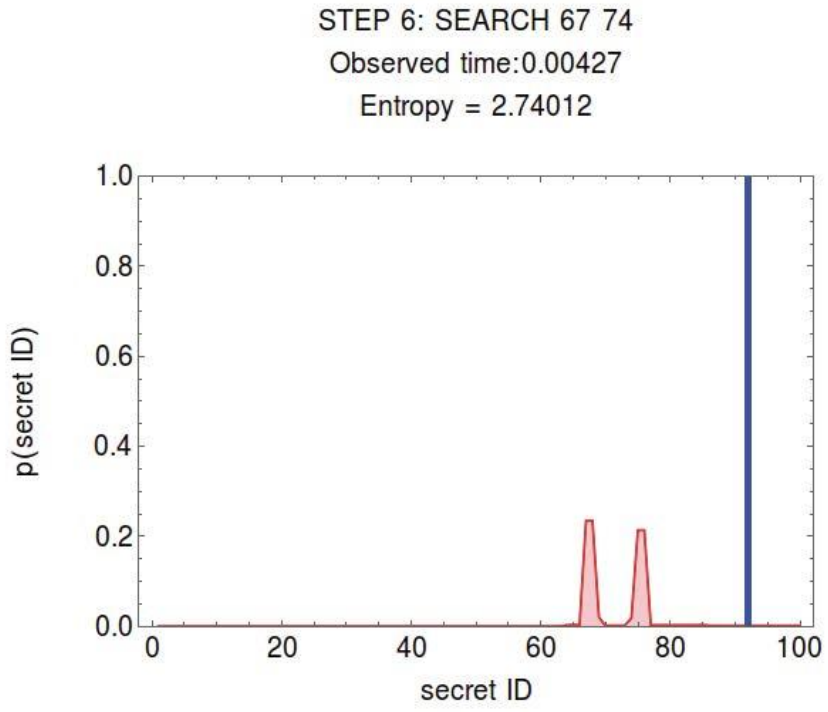
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \qquad ID_1 = 64 \qquad ID_2 = 85 \qquad ID_{res} = 92$$



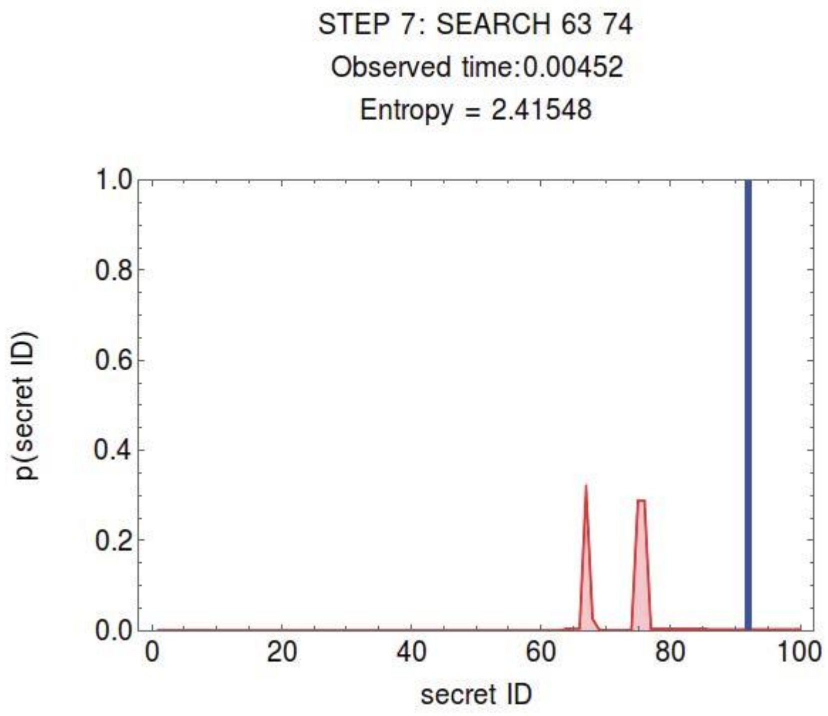
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \qquad ID_1 = 64 \qquad ID_2 = 85 \qquad ID_{res} = 92$$



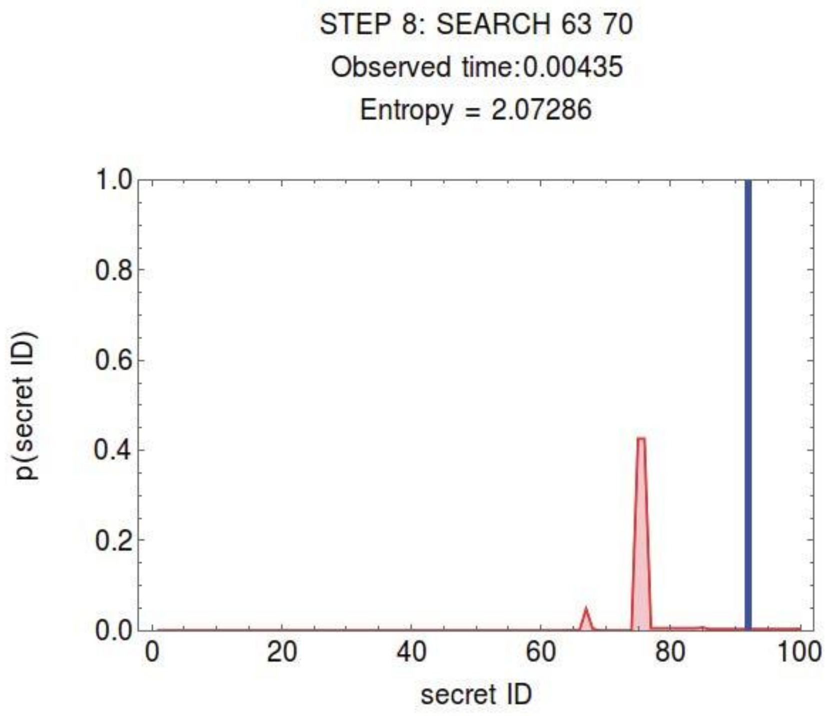
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



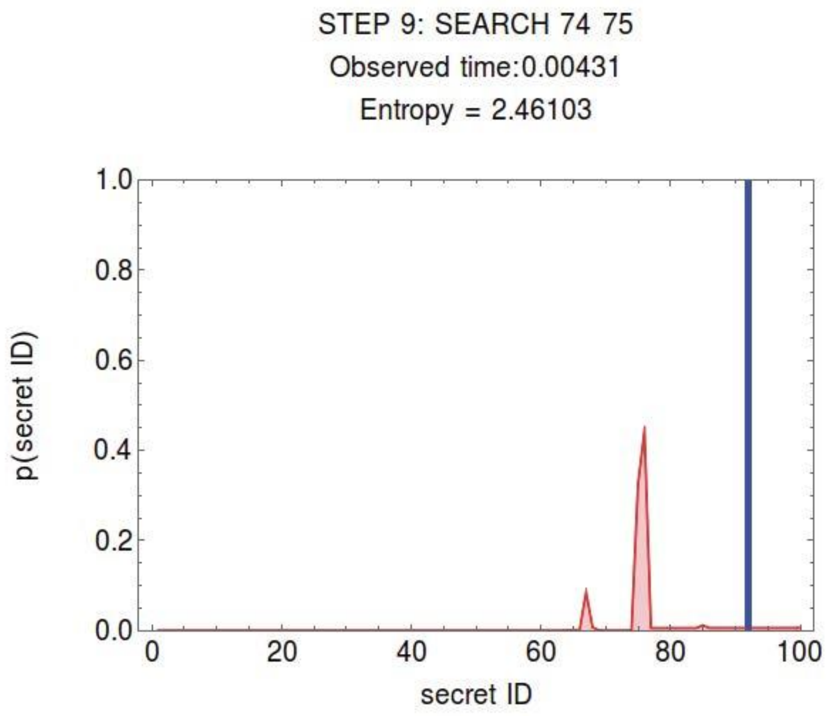
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



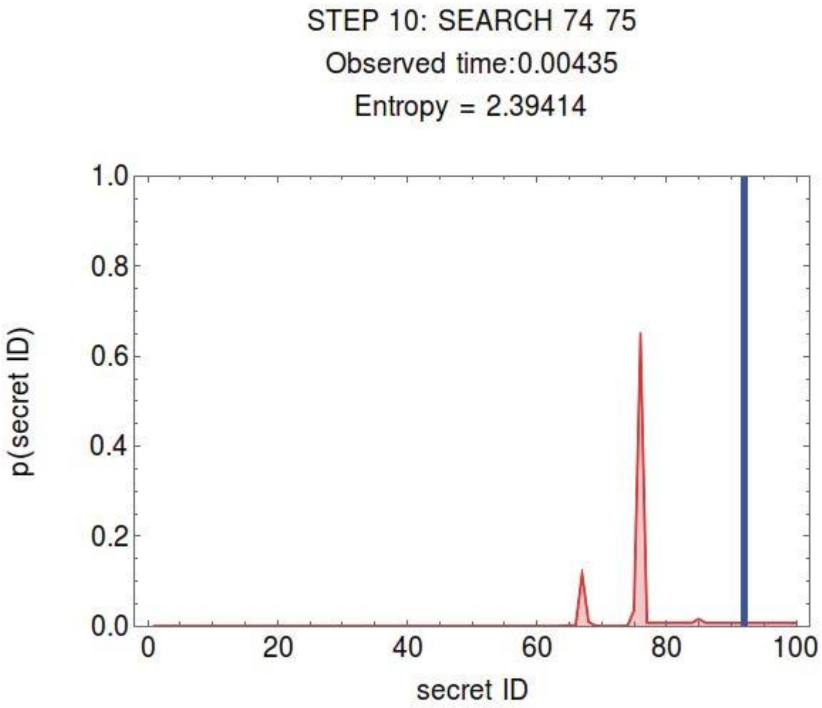
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



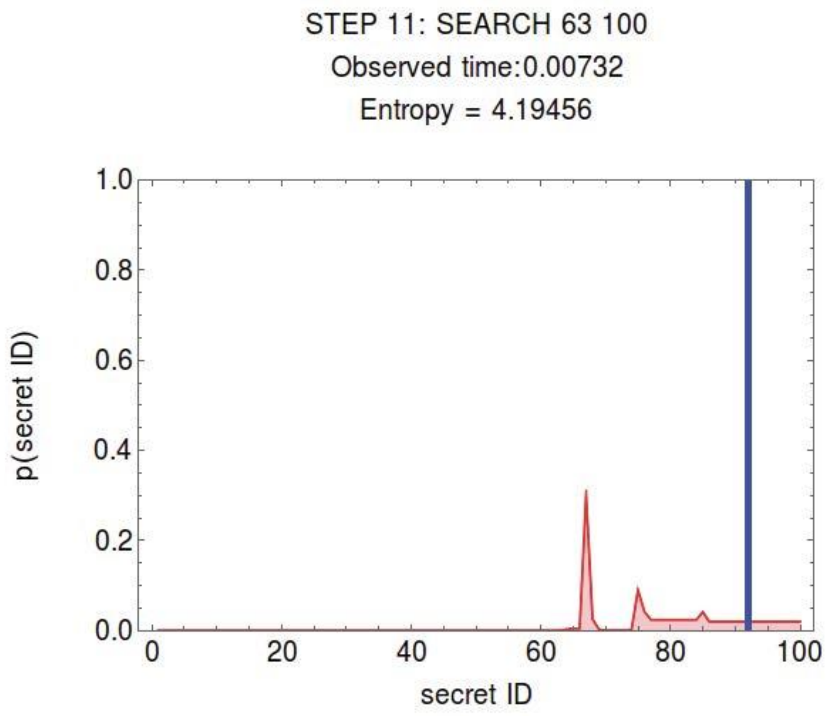
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



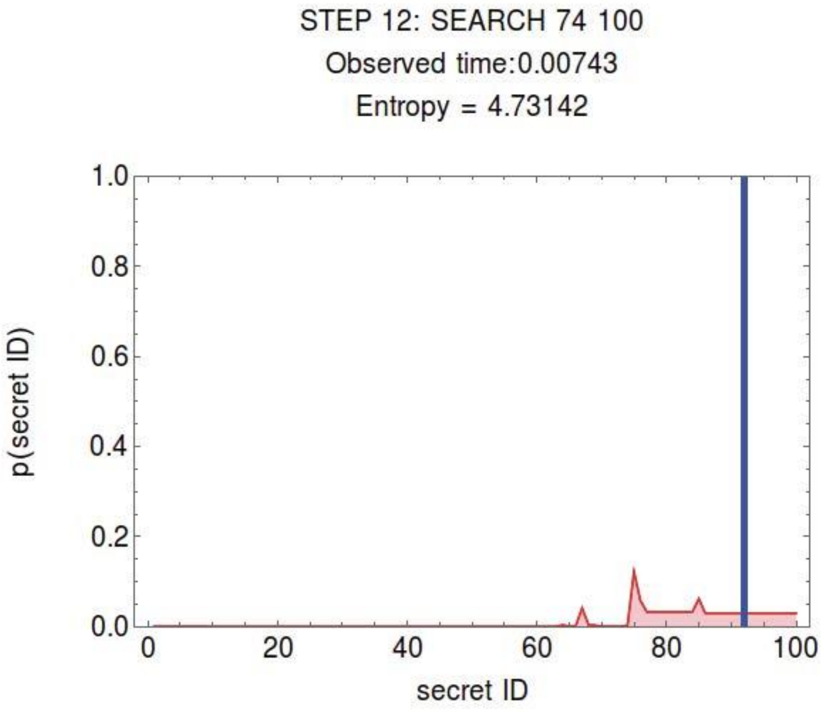
Automatically generated attack against LawDB

$1 \le ID \le 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



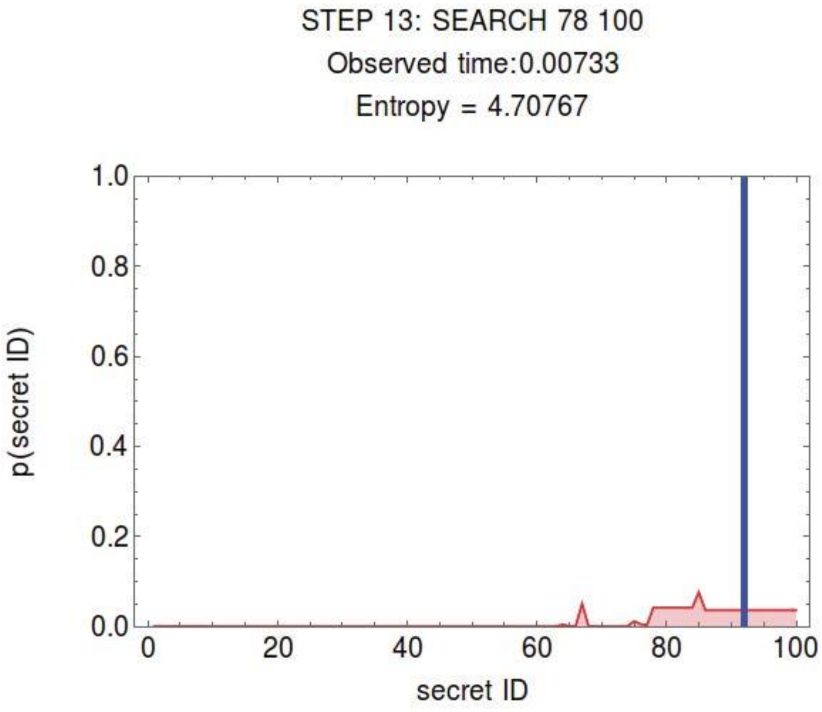
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



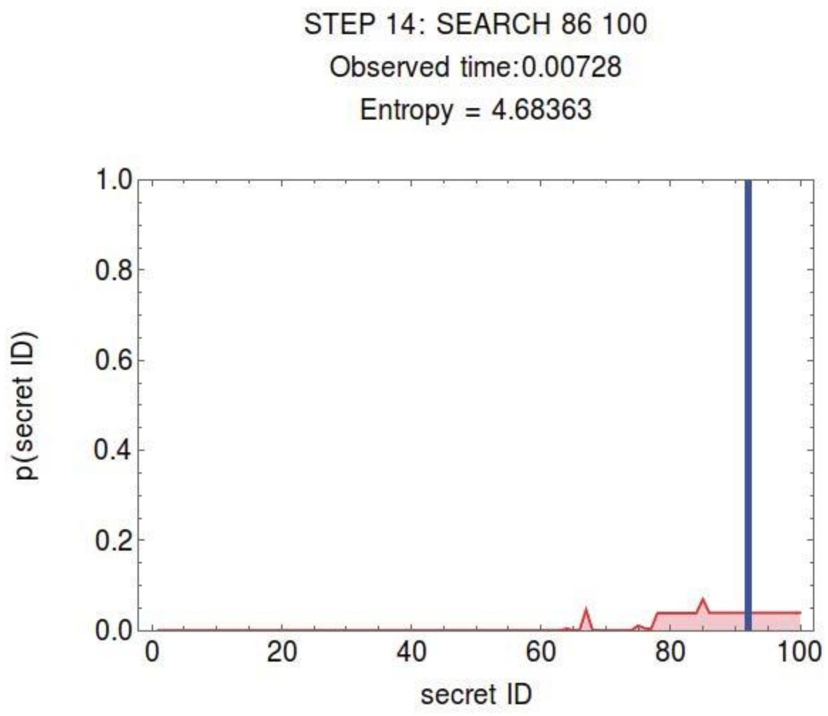
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \qquad ID_1 = 64 \qquad ID_2 = 85 \qquad ID_{res} = 92$$



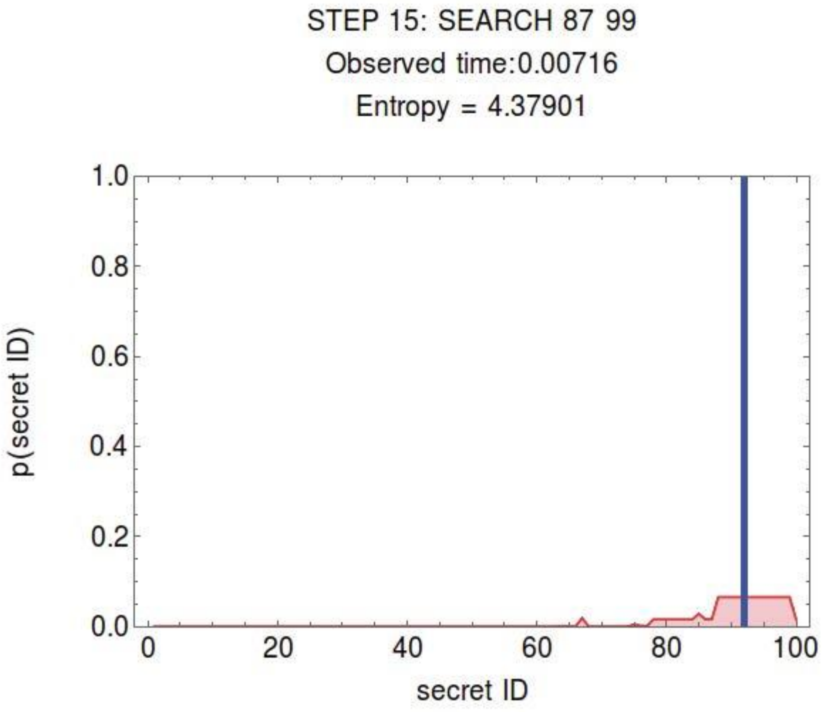
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



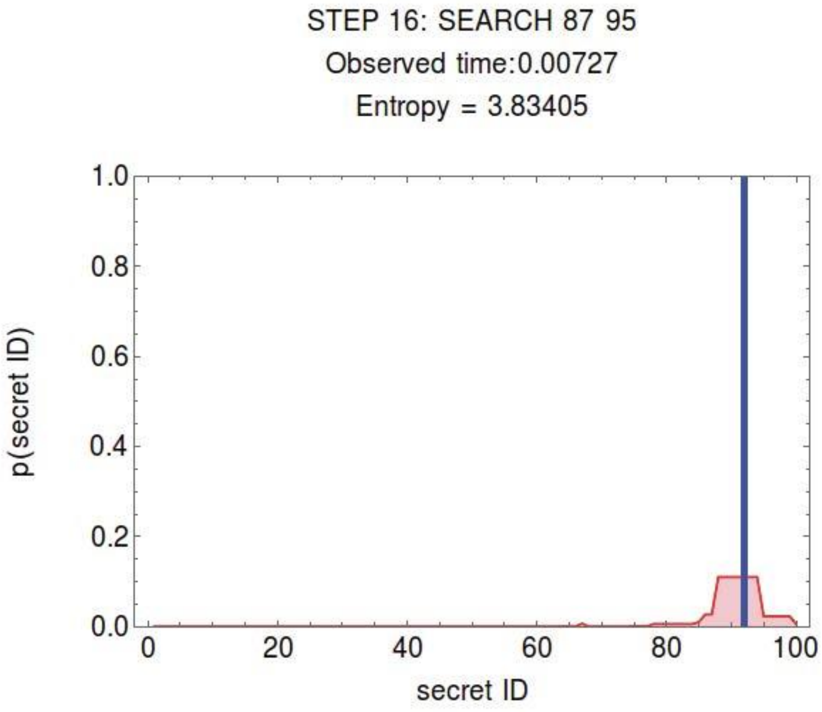
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



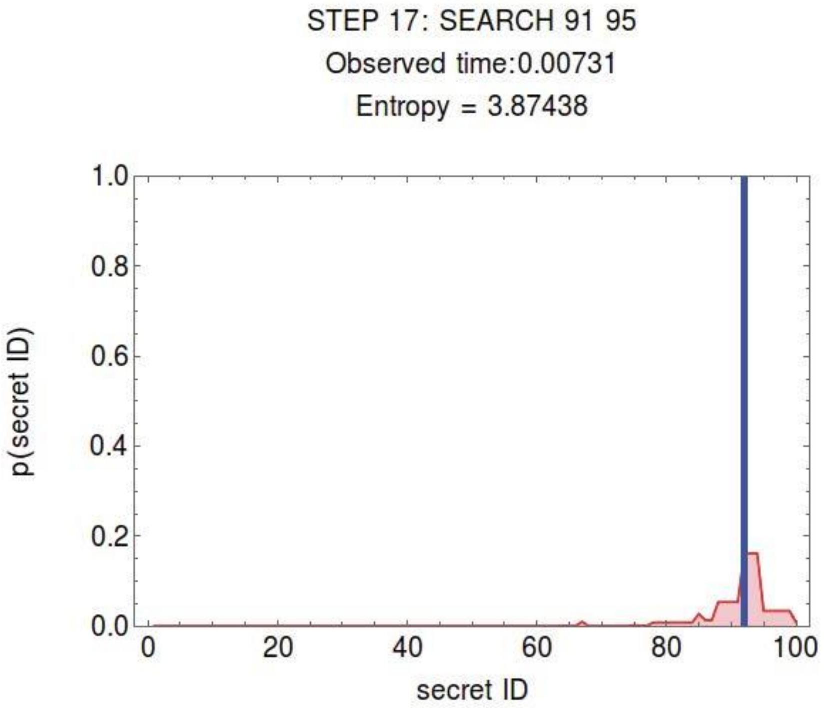
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



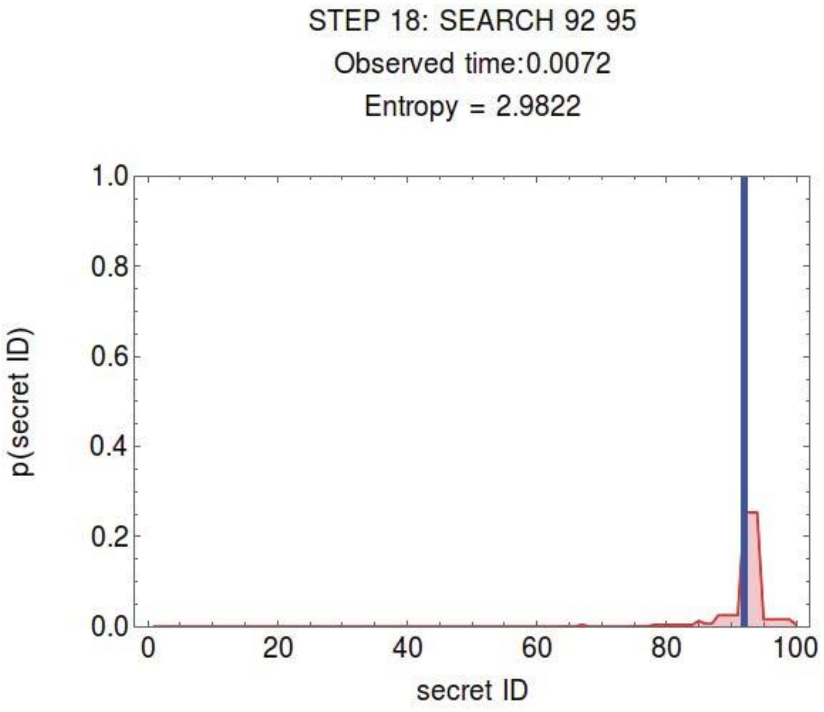
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



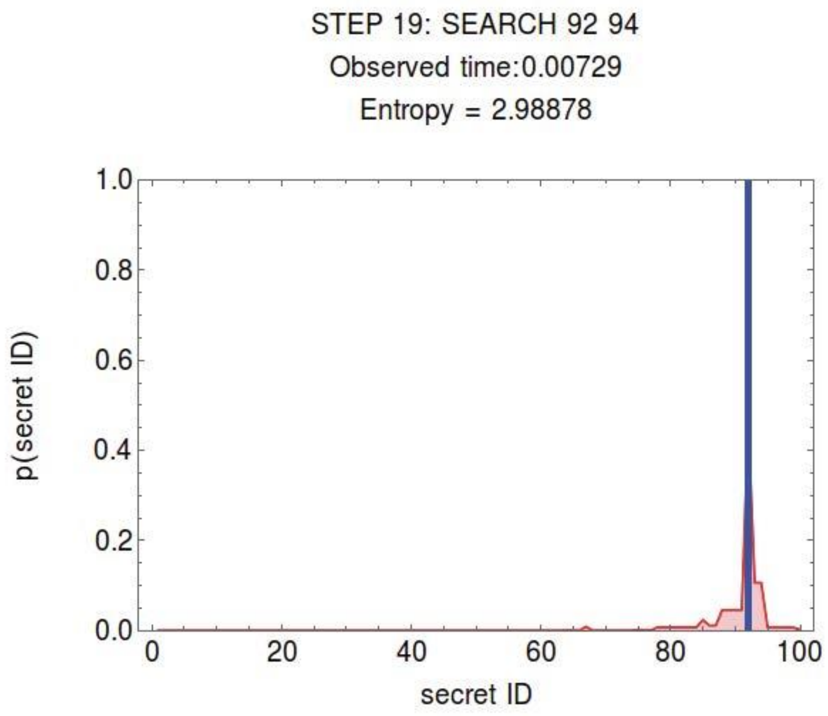
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



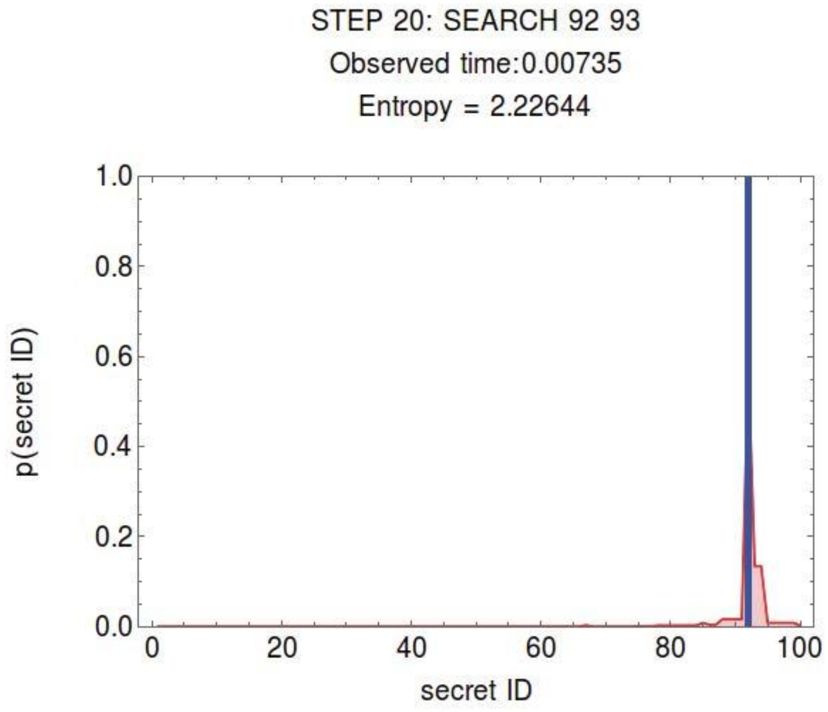
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$



Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



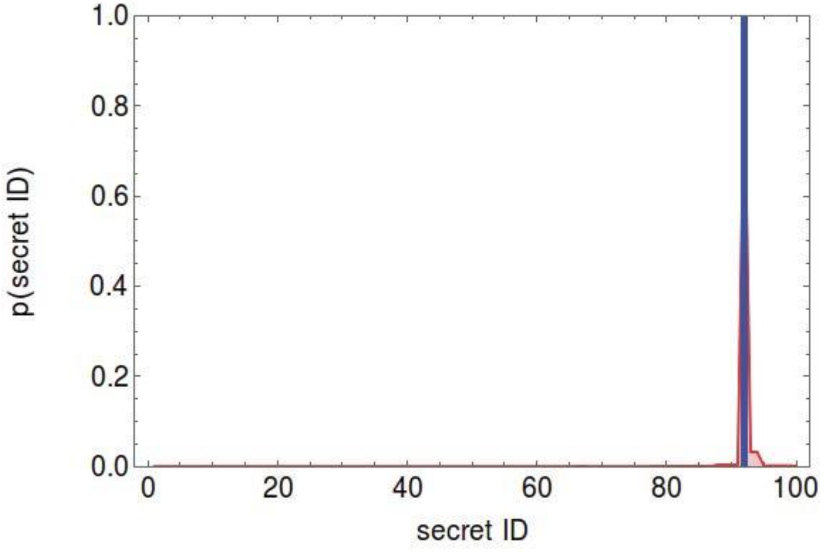
Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$

STEP 21: SEARCH 92 92

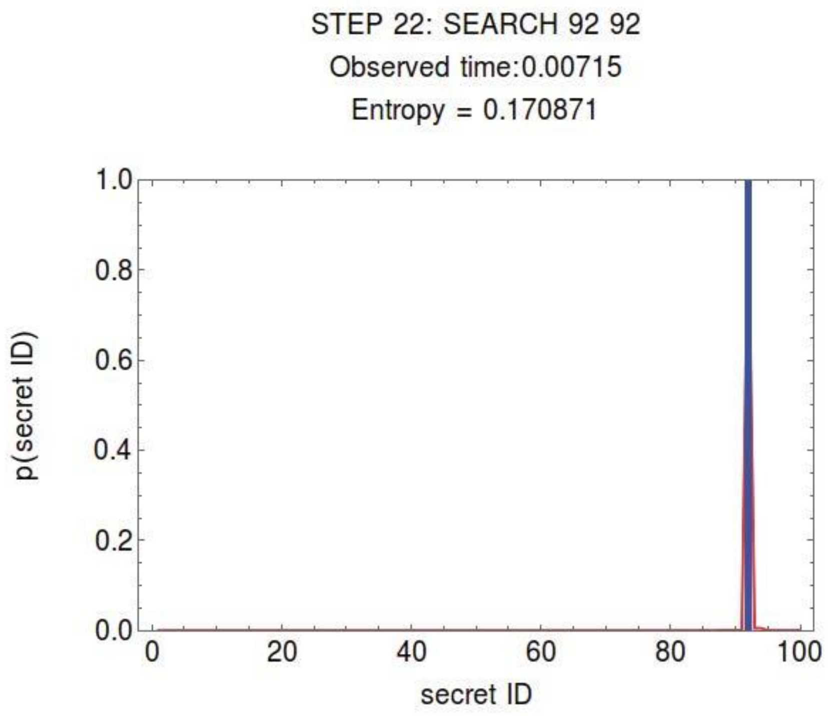
Observed time:0.00739

Entropy = 0.767476



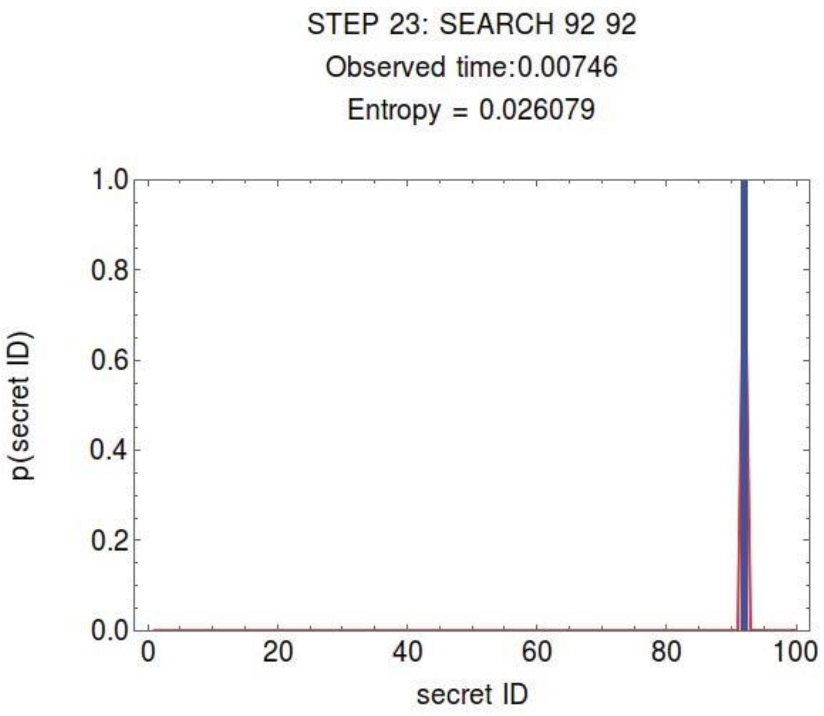
Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



Automatically generated attack against LawDB

$$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$$



Automatically generated attack against LawDB

$1 \leq ID \leq 100 \quad ID_1 = 64 \quad ID_2 = 85 \quad ID_{res} = 92$