String Analysis for Side Channels with Segmented Oracles

Lucas Bang$^1$, Abdulbaki Aydin$^1$, Quoc-Sang Phan$^2$, Corina S. Păsăreanu$^{2,3}$, Tevfik Bultan$^1$

$^1$University of California, Santa Barbara  
Santa Barbara, CA, USA

$^2$Carnegie Mellon University  
Moffet Field, CA, USA

$^3$NASA Ames Research Center  
Moffet Field, CA, USA

ACM Foundations of Software Engineering  
Seattle, Washington, USA  
15 November 2016
Overview
Overview

Program

(Segmented Oracle)
Overview

Program (Segmented Oracle) \rightarrow \text{Symbolic Execution}
Overview

Program (Segmented Oracle) → Symbolic Execution → Path Constraints → Model Counter
Overview

Program
(Segmented Oracle) → Symbolic Execution → Path Constraints → Model Counter → Probability Distribution → Side Channel Analysis
Overview

Program (Segmented Oracle) → Symbolic Execution → Path Constraints → Model Counter → Probability Distribution → Side Channel Analysis → Information Leakage Quantification
Background and Motivation

Software channels:

- **Main Channel**: Output of the program, i.e. return value
- **Side Channel**: Other execution aspects: time, memory, network, ...

Intuitively, Segment Oracles have:

- **Side channels** that reveal information about:
  - **Segments** (single characters, bytes, bits, array slice) of a **secret** program value.
Software channels:

- Main Channel. Output of the program, i.e. return value
Background and Motivation

Software channels:

- **Main Channel.** Output of the program, i.e. return value
- **Side Channel.** Other execution aspects:
  - time, memory, network, ...
Background and Motivation

Software channels:

- Main Channel. Output of the program, i.e. return value
- Side Channel. Other execution aspects: time, memory, network, ...

Intuitively, Segment Oracles have
Software channels:

- Main Channel. Output of the program, i.e. return value
- Side Channel. Other execution aspects: time, memory, network, ...

Intuitively, Segment Oracles have

- **side channels** that reveal information about
Software channels:

- Main Channel. Output of the program, i.e. return value
- Side Channel. Other execution aspects: time, memory, network, ...

Intuitively, Segment Oracles have

- **side channels** that reveal information about
- **segments** (single characters, bytes, bits, array slice) of a
Background and Motivation

Software channels:

- Main Channel. Output of the program, i.e. return value
- Side Channel. Other execution aspects: time, memory, network, ...

Intuitively, Segment Oracles have

- **side channels** that reveal information about
- **segments** (single characters, bytes, bits, array slice) of a
- **secret** program value.
Example

```
1  passcheck(char[] pw, char[] guess)
2    for (int i = 0; i < length; i++)
3      if (pw[i] != guess[i]) return false
4    return true
```
Example

```c
passcheck(char[] pw, char[] guess)
for (int i = 0; i < length; i++)
    if (pw[i] != guess[i]) return false
return true
```

Using the program main channel (true, false), and brute force needs

$$(\text{alphabet size})^L = (128 \text{ ASCII chars})^L$$

guesses in the worst case = thousands of years.
Example

1    passcheck(char[] pw, char[] guess)
2        for (int i = 0; i < length; i++)
3            if (pw[i] != guess[i]) return false
4        return true

What if the adversary can measure execution time? Assume:
  ➤ 1 observable time unit = 1 loop execution.
  ➤ No measurement error, no system noise.
Example

```c
1  passcheck(char[] pw, char[] guess)
2      for (int i = 0; i < length; i++)
3          if (pw[i] != guess[i]) return false
4      return true
```

What if the adversary can measure execution time? Assume:
- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>seatac_airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>User guesses</td>
<td>aaaaaaaaaaaaaaaa</td>
</tr>
</tbody>
</table>
Example

```
1    passcheck(char[] pw, char[] guess)
2        for (int i = 0; i < length; i++)
3            if (pw[i] != guess[i]) return false
4        return true
```

What if the adversary can measure execution time? Assume:

- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>User guesses</th>
<th>Result</th>
<th>Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>seatac_airport</td>
<td>aaaaaaaaaaa</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>saaaaaaaaaaa</td>
<td>false</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

1    passcheck(char[] pw, char[] guess)
2        for (int i = 0; i < length; i++)
3            if (pw[i] != guess[i]) return false
4        return true

What if the adversary can measure execution time? Assume:
- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>seatac_airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>User guesses</td>
<td>aaaaaaaaaaaaaaaa false 1 loop</td>
</tr>
<tr>
<td></td>
<td>saaaaaaaaaaaaaaa false 2 loops</td>
</tr>
<tr>
<td></td>
<td>seaaaaaaaaaaaaaaa false 3 loops</td>
</tr>
</tbody>
</table>
Example

```java
1  passcheck(char[] pw, char[] guess)
2     for (int i = 0; i < length; i++)
3         if (pw[i] != guess[i]) return false
4     return true
```

What if the adversary can measure execution time? Assume:
- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>User guesses</th>
<th>Success</th>
<th>Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>seatac_airport</td>
<td>aaaaaaaaaaaaaa</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>saaaaaaaaaaaaaaa</td>
<td>false</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>seaaaaaaaaaaaaaa</td>
<td>false</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>seatacaaaaaaaaa</td>
<td>false</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Example

1    passcheck(char[] pw, char[] guess)
2        for (int i = 0; i < length; i++)
3            if (pw[i] != guess[i]) return false
4            return true

What if the adversary can measure execution time? Assume:
- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>User guesses</th>
<th>Result</th>
<th>Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>seatac_airport</td>
<td>aaaaaaaaaaaaaaaaaa</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>saaaaaaaaaaaaaaaaa</td>
<td>false</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>sseaaaaaaaaaaaaaa</td>
<td>false</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>seatacaaaaaaaaaa</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>seatac_airport</td>
<td>true</td>
<td>15</td>
</tr>
</tbody>
</table>

Using the program timing channel, adversary needs $128 \times 15$ guesses, which is a few seconds.
Example

```java
1     passcheck(char[] pw, char[] guess)
2         for (int i = 0; i < length; i++)
3             if (pw[i] != guess[i]) return false
4         return true
```

What if the adversary can measure execution time? Assume:

- 1 observable time unit = 1 loop execution.
- No measurement error, no system noise.

<table>
<thead>
<tr>
<th>Secret password</th>
<th>User guesses</th>
<th>Result</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>seatac_airport</td>
<td>aaaaaaaaaaaaaaaaaaa</td>
<td>false</td>
<td>1 loop</td>
</tr>
<tr>
<td></td>
<td>saaaaaaaaaaaaaaaaaa</td>
<td>false</td>
<td>2 loops</td>
</tr>
<tr>
<td></td>
<td>seaaaaaaaaaaaaaaaaa</td>
<td>false</td>
<td>3 loops</td>
</tr>
<tr>
<td></td>
<td>seatacaaaaaaaaaaa</td>
<td>false</td>
<td>7 loops</td>
</tr>
<tr>
<td></td>
<td>seatac_airport</td>
<td>true</td>
<td>15 loops</td>
</tr>
</tbody>
</table>

Using the program timing channel, adversary needs

\[(\text{alphabet size}) \times L = (128) \times 15 \text{ guesses} = \text{a few seconds.}\]
Motivation

Real-life segmented oracle security vulnerabilities:

- Timing Side Channels
Real-life segmented oracle security vulnerabilities:

- Timing Side Channels
  - Authentication keys: Google Keyczar Library, Xbox 360
  - Authorization Frameworks: OAuth, OpenID (Google, Facebook, Microsoft, Twitter)
Motivation

Real-life segmented oracle security vulnerabilities:

- **Timing Side Channels**
  - Authentication keys: Google Keyczar Library, Xbox 360
  - Authorization Frameworks: OAuth, OpenID (Google, Facebook, Microsoft, Twitter)
  - Java’s `Array.equals, String.equals`
  - C’s `memcmp`
  - **Save computation time.**
Motivation

Real-life segmented oracle security vulnerabilities:

- **Timing Side Channels**
  - Authentication keys: Google Keyczar Library, Xbox 360
  - Authorization Frameworks: OAuth, OpenID (Google, Facebook, Microsoft, Twitter)
  - Java’s `Array.equals`, `String.equals`
  - C’s `memcmp`
  - **Save computation time.**

- **Network Packet Size Side Channel**
  - Compression Ratio Infoleak Made Easy (CRIME) [Ekoparty 2012]
  - Browser Recon and Exfiltration via Adaptive Compression (BREACH) [Black Hat 2013]
Motivation

Real-life segmented oracle security vulnerabilities:

- **Timing Side Channels**
  - Authentication keys: Google Keyczar Library, Xbox 360
  - Authorization Frameworks: OAuth, OpenID (Google, Facebook, Microsoft, Twitter)
  - Java’s `Array.equals`, `String.equals`
  - C’s `memcmp`
  - *Save computation time.*

- **Network Packet Size Side Channel**
  - Compression Ratio Infomation Made Easy (CRIME) [Ekoparty 2012]
  - Browser Recon and Exfiltration via Adaptive Compression (BREACH) [Black Hat 2013]
  - Lempel Ziv String Compression. *Save space.*
  - Adversary inject plain text. More compression $\rightarrow$ substring match.
Motivation

Real-life segmented oracle security vulnerabilities:

- **Timing Side Channels**
  - Authentication keys: Google Keyczar Library, Xbox 360
  - Authorization Frameworks: OAuth, OpenID (Google, Facebook, Microsoft, Twitter)
  - Java’s `Array.equals, String.equals`
  - C’s `memcmp`
  - **Save computation time.**

- **Network Packet Size Side Channel**
  - Compression Ratio Infoleak Made Easy (CRIME) [Ekoparty 2012]
  - Browser Recon and Exfiltration via Adaptive Compression (BREACH) [Black Hat 2013]
  - Lempel Ziv String Compression. **Save space.**
  - Adversary inject plain text. More compression → substring match.

**Goal:** quantify information leakage for these types of vulnerabilities.
Overview

Program $\xrightarrow{\text{Symbolic Execution}}$ Path Constraints $\xrightarrow{\text{Model Counting}}$ Probability Distribution $\xrightarrow{\text{Side Channel Analysis}}$ Program Vulnerability Quantification
bool pwcheck(guess[]) 
for(i = 0; i < 4; i++)
  if(guess[i] != pw[i])
    return false
return true

\( P: pw, G: guess \)

\( o_i = \) lines of code
bool pwcheck(guess[]) {
    for(i = 0; i < 4; i++)
        if(guess[i] != pw[i])
            return false;
    return true;
}

P: pw, G: guess

\(o_i\) = lines of code
Segmented Oracle Path Constraints Pattern

\[(o_i, PC_i) : P[0] = G[0] \ldots \land P[i - 1] = G[i - 1] \land P[i] \neq G[i]\]
A criterion for segmented oracles: path constraints grouped by observable are logically equivalent to this pattern (up to reordering).
Multiple Runs of the Program

Adversary learns more with multiple invocations.
Multiple Runs of the Program

Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$’s strategy $S$:

1. $\text{obs} \leftarrow \text{nil}$. Initially observation sequence is empty.
Multiple Runs of the Program

Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$’s strategy $S$:

1. $\text{obs} \leftarrow \text{nil}$. Initially observation sequence is empty.
2. $\mathcal{I} \leftarrow \mathcal{A}(\text{obs})$. Adversary chooses $\mathcal{I}$ based on observations so far.
Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$'s strategy $S$:

1. $\text{obs} \leftarrow \text{nil}$. Initially observation sequence is empty.
2. $\mathcal{I} \leftarrow \mathcal{A}(\text{obs})$. Adversary chooses $\mathcal{I}$ based on observations so far.
3. $o \leftarrow F(\mathcal{I})$. Adversary invokes function, makes observation.
Multiple Runs of the Program

Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$'s strategy $S$:

1. $\text{obs} \leftarrow \text{nil}$. Initially observation sequence is empty.
2. $\mathcal{I} \leftarrow A(\text{obs})$. Adversary chooses $\mathcal{I}$ based on observations so far.
3. $o \leftarrow F(\mathcal{I})$. Adversary invokes function, makes observation.
4. $\text{obs} \leftarrow \text{append}(\text{obs}, \langle \mathcal{I}, o \rangle)$. Update observation record.
Multiple Runs of the Program

Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$’s strategy $S$:

1. $obs \leftarrow nil$. Initially observation sequence is empty.
2. $\mathcal{I} \leftarrow \mathcal{A}(obs)$. Adversary chooses $\mathcal{I}$ based on observations so far.
3. $o \leftarrow F(\mathcal{I})$. Adversary invokes function, makes observation.
4. $obs \leftarrow append(obs, \langle \mathcal{I}, o \rangle)$. Update observation record.
5. Repeat until entire secret revealed.
Multiple Runs of the Program

Adversary learns more with multiple invocations.

Model adversary $\mathcal{A}$’s strategy $S$:

1. $\text{obs} \leftarrow \text{nil}$. Initially observation sequence is empty.
2. $\mathcal{I} \leftarrow \mathcal{A}($obs$)$. Adversary chooses $\mathcal{I}$ based on observations so far.
3. $o \leftarrow F(\mathcal{I})$. Adversary invokes function, makes observation.
4. $\text{obs} \leftarrow \text{append}($obs$, \langle \mathcal{I}, o \rangle)$. Update observation record.
5. Repeat until entire secret revealed.

Symbolic execution of $S$: all possible observable sequences.
How \textit{likely} is a certain program behavior?

What is the probability of a particular program execution path?

\textbf{Computing Path Constraint Probability}
How *likely* is a certain program behavior?

What is the probability of a particular program execution path?

Computing Path Constraint Probability

Probability of $PC = \frac{\text{Number of solutions to } PC}{\text{Total input domain size}}$
How *likely* is a certain program behavior?

What is the probability of a particular program execution path?

**Computing Path Constraint Probability**

Probability of $PC = \frac{\text{Number of solutions to } PC}{\text{Total input domain size}}$

$p(PC) = \frac{|PC|}{|D|}$
How likely is a certain program behavior?

What is the probability of a particular program execution path?

**Computing Path Constraint Probability**

Probability of $PC = \frac{\text{Number of solutions to } PC}{\text{Total input domain size}}$

\[
p(PC) = \frac{|PC|}{|D|}
\]

How do you compute the number of solutions $|PC|$ automatically?
Overview

Program ➔ Symbolic Execution

Path Constraints ➔ Model Counting

Probability Distribution ➔ Side Channel Analysis

Side Channel Analysis ➔ Program Vulnerability Quantification
Symbolic execution for string manipulating programs results in path constraints over string variables.

Count the number of strings consistent with $PC$. 
Model Counting

Symbolic execution for string manipulating programs results in path constraints over string variables.

Count the number of strings consistent with $PC$.

Automata-Based Counter (ABC):

- Constructs an automaton recognizing solutions to $PC$. 

$\text{PC}$ is the number of accepting paths in the automaton.
Symbolic execution for string manipulating programs results in path constraints over string variables.

Count the number of strings consistent with $PC$.

**Automata-Based Counter (ABC):**

- Constructs an automaton recognizing solutions to $PC$.

\[
\begin{array}{c}
0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 1
\end{array}
\]

- $|PC|$ is number of accepting paths in automaton.
Overview

Program → Symbolic Execution → Path Constraints → Model Counting → Probability Distribution → Side Channel Analysis → Program Vulnerability Quantification
Information Leakage

Adversary sees a sequence of observables and PCs:

$$(PC_i, \overrightarrow{o_i}) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)$$

We can compute probabilities:

$$p(\overrightarrow{o_i}) = \frac{|PC_i|}{|D|}$$

Quantify information gain using information entropy:

$$H = \sum p(\overrightarrow{o_i}) \log_2 \left( \frac{1}{p(\overrightarrow{o_i})} \right)$$

Information entropy measures information uncertainty. Initially, $H = \log_2 |D| = \text{number of bits}$. $H$ decreases with increasing observation length. Eventually, $H = 0$, no uncertainty, secret revealed.
Information Leakage

Adversary sees a sequence of observables and PCs:

\((PC_i, \vec{o}_i) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\)

We can compute probabilities:

\[ p(\vec{o}_i) = \frac{|PC_i|}{|D|} \]

Quantify information gain using information entropy:

\[ H = \sum p(\vec{o}_i) \log_2 \frac{1}{p(\vec{o}_i)} \]

Information entropy measures information uncertainty. Initially, \( H = \log_2 |D| = \text{number of bits} \).

Eventually, \( H = 0 \), no uncertainty, secret revealed.
Information Leakage

Adversary sees a sequence of observables and PCs:

\[(PC_i, \overrightarrow{o_i}) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\]

We can compute probabilities:

\[p(\overrightarrow{o_i}) = \frac{|PC_i|}{|D|}\]

Quantify information gain using *information entropy*:

\[H = \sum p(\overrightarrow{o_i}) \log_2 \frac{1}{p(\overrightarrow{o_i})}\]

Information entropy measures information uncertainty. Initially, \(H = \log_2 |D| = \text{number of bits}\). \(H\) decreases with increasing observation length. Eventually, \(H = 0\), no uncertainty, secret revealed.
Information Leakage

Adversary sees a sequence of observables and PCs:

\[(PC_i, \overrightarrow{o}_i) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\]

We can compute probabilities:

\[
p(\overrightarrow{o}_i) = \frac{|PC_i|}{|D|}
\]

Quantify information gain using *information entropy*:

\[
H = \sum p(\overrightarrow{o}_i) \log_2 \frac{1}{p(\overrightarrow{o}_i)}
\]

Information entropy measures information uncertainty.
Information Leakage

Adversary sees a sequence of observables and PCs:

\[(PC_i, \overrightarrow{o_i}) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\]

We can compute probabilities:

\[p(\overrightarrow{o_i}) = \frac{|PC_i|}{|D|}\]

Quantify information gain using *information entropy*:

\[H = \sum p(\overrightarrow{o_i}) \log_2 \frac{1}{p(\overrightarrow{o_i})}\]

Information entropy measures information uncertainty.

Initially, \(H = \log_2 |D| = \) number of bits.
Information Leakage

Adversary sees a sequence of observables and PCs:

\[(PC_i, \overrightarrow{o}_i) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\]

We can compute probabilities:

\[p(\overrightarrow{o}_i) = \frac{|PC_i|}{|D|}\]

Quantify information gain using *information entropy*:

\[H = \sum p(\overrightarrow{o}_i) \log_2 \frac{1}{p(\overrightarrow{o}_i)}\]

Information entropy measures information uncertainty.

Initially, \(H = \log_2 |D| = \text{number of bits}\).

\(H\) decreases with increasing observation length.
Information Leakage

Adversary sees a sequence of observables and PCs:

\[(PC_i, \overrightarrow{o_i}) = (PC_i, \langle o^1, o^2 \ldots o^k \rangle)\]

We can compute probabilities:

\[p(\overrightarrow{o_i}) = \frac{|PC_i|}{|D|}\]

Quantify information gain using information entropy:

\[H = \sum p(\overrightarrow{o_i}) \log_2 \frac{1}{p(\overrightarrow{o_i})}\]

Information entropy measures information uncertainty.

Initially, \(H = \log_2 |D| = \text{number of bits}\).

\(H\) decreases with increasing observation length.

Eventually, \(H = 0\), no uncertainty, secret revealed.
Overview

Program → Symbolic Execution → Path Constraints → Model Counting → Probability Distribution → Side Channel Analysis → Program Vulnerability Quantification
Avoiding Expensive Multirun Symbolic Execution

Do a single run of symbolic execution.
Avoiding Expensive Multirun Symbolic Execution

Do a **single run** of symbolic execution.

Numerically compute multi-run behavior:
Avoiding Expensive Multirun Symbolic Execution

Do a **single run** of symbolic execution.

**Numerically compute multi-run behavior:**

Derive recurrence relating segment sizes $|D_i|$ to $|PC_i|$:

$$\begin{align*}
\prod |D| &= |PC_n| \\
\prod |D| \cdot (|D_i| - 1) \cdot \prod |D|_{i+1:n-1} &= |PC_i|
\end{align*}$$
Avoiding Expensive Multirun Symbolic Execution

Do a single run of symbolic execution.

Numerically compute multi-run behavior:

Derive recurrence relating segment sizes $|D_i|$ to $|PC_i|$:

$$
\begin{align*}
\prod |D| &= |PC_n| \\
\prod |D| \cdot (|D_i| - 1) \cdot \prod |D|_{i+1:n-1} &= |PC_i|
\end{align*}
$$

and probability recurrence:

$$
p(\overrightarrow{o}|D) = p(o^1|D_i) \cdot p(o^2, \ldots, o^k|D_i)
$$
Avoiding Expensive Multirun Symbolic Execution

Do a **single run** of symbolic execution.

**Numerically compute multi-run behavior:**

Derive recurrence relating segment sizes $|D_i|$ to $|P_{C_i}|$:

\[
\begin{align*}
\prod |D| &= |P_{C_n}| \\
\prod |D| \cdot (|D_i| - 1) \cdot \prod |D|_{i+1:n-1} &= |P_{C_i}|
\end{align*}
\]

and probability recurrence:

\[
p(\vec{o} | D) = p(o^1 | D'_i) \cdot p(\langle o^2, \ldots, o^k \rangle | D'_i)
\]

Efficiently compute $p(\vec{o})$ using standard dynamic programming and memoization techniques.
Implementation

- Java Symbolic Pathfinder (JPF / SPF), symbolic execution.
- Specialized listeners for tracking observables.
- ABC and Latte for model counting path constraints.
- SPF packages to quantify information leakage.
Figure: Time for multi-run and single-run SE.
Experiments

Figure: Information leakage and remaining entropy for password checking function. Length = 3, alphabet size = 4.
Experiments

Analysis of the CRIME attack.

- Symbolically execute LZ77 compression. 60 lines of complex code. Nested loops, multiple buffers, complex compression conditions.
Experiments

Analysis of the CRIME attack.

- Symbolically execute LZ77 compression. 60 lines of complex code. Nested loops, multiple buffers, complex compression conditions.
- Length 3 and alphabet size 4 generates 187 path conditions leading to 4 different observables.
- Use Z3 to prove equivalence to segmented oracle PC pattern.
- Leaks all information after 10 executions by the adversary.
Experiments

Analysis of the CRIME attack.

- Symbolically execute LZ77 compression. 60 lines of complex code. Nested loops, multiple buffers, complex compression conditions.
- Length 3 and alphabet size 4 generates 187 path conditions leading to 4 different observables.
- Use Z3 to prove equivalence to segmented oracle PC pattern.
-Leaks all information after 10 executions by the adversary.
- Running time: 8.695 seconds
Conclusions

In this talk:

- Segmented oracles.
- Multi-run symbolic execution of adversary model to get leakage.
- Infer multi-run leakage from a single run of symbolic execution.
- Model counting for string manipulating programs.
- Experimentally validated our approach.

Future work:

- Extend analysis to more general oracles.
- Incorporate model of system noise.
- Automatically generate adversary strategies.
Where do segment oracle side channels come from?

Algorithmic optimizations:

- Saving time and space whenever possible...
Where do segment oracle side channels come from?

Algorithmic optimizations:

- Saving time and space whenever possible...
- early loop termination, text compression...
Where do segment oracle side channels come from?

Algorithmic optimizations:

- Saving time and space whenever possible...
- early loop termination, text compression...
- might reveal some properties of secure data.
Where do segment oracle side channels come from?

Algorithmic optimizations:
- Saving time and space whenever possible...
- early loop termination, text compression...
- might reveal some properties of secure data.

“Premature optimization is the root of all evil.” -Tony Hoare

**Important tradeoff:** efficiency vs. security.

**Important problem to address:** we need tools for automatically measuring this tradeoff.
Questions?

Thank you.
Multi-Run Symbolic Execution

Model “the best” adversary.

- Keep making inputs and observations.
- Iterate over segment alphabet until matched prefix gets longer.
- Search the next segment.
Multi-Run Symbolic Execution

Model “the best” adversary.

- Keep making inputs and observations.
- Iterate over segment alphabet until matched prefix gets longer.
- Search the next segment.

```
procedure \( S = (A_B, F) \)
vars
  \( s \): the current segment of \( h \) being searched
  \( b \): the first time \( s \) is searched
  \( o^0, o^1, \ldots o^k \): observations of the adversary
begin
  \( s \leftarrow 1 \), \( b \leftarrow 1 \), \( o^0 \leftarrow 0 \)
  \text{for all } i \in [1..k] \{ \\
    \text{for all } j \in [b..i] \{ \ \text{assume} \ (l_i^j[s] \neq l_i^j[s]) \} \\
    o^i \leftarrow F(h, l_i^i) \\
    \text{if } (o^i = \|h\|) \{ \ \text{return} \} \\
    \text{if } (o^i > o^{i-1}) \{ \\
      \text{for all } j \in [i + 1..k] \{ \\
        \text{for all } n \in [s..o^i] \{ \ \text{assume} \ (l_i^j[n] = l_i^i[n]) \} \\
      \} \\
      s \leftarrow o^i + 1 \), \( b \leftarrow i + 1 \)
    \}
  \}
end
```
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} \]
Information Theory Intuition

**Information Entropy:**

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]
Information Theory Intuition

**Information Entropy:**

\[
H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right]
\]

The expected amount of information gain.
Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain.
The expected amount of “\textit{surprise}”. 
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain.
The expected amount of “surprise”.

Seattle Weather, Always Raining
\[ p_{\text{rain}} = 1, p_{\text{sun}} = 0 \]

Costa Rica Weather, Coin Flip
\[ p_{\text{rain}} = \frac{1}{2}, p_{\text{sun}} = \frac{1}{2} \]
\[ H = 1 \]

Santa Barbara Weather, Almost Always Sunny.
\[ p_{\text{rain}} = \frac{1}{10}, p_{\text{sun}} = \frac{9}{10} \]
\[ H = 0.4960 \]
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain.
The expected amount of “surprise”.

**Seattle Weather, Always Raining**

\( p_{\text{rain}} = 1, p_{\text{sun}} = 0 \quad H = 0 \)
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain. The expected amount of “surprise”.

Seattle Weather, Always Raining
\[ p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \quad H = 0 \]

Costa Rica Weather, Coin Flip
\[ p_{\text{rain}} = \frac{1}{2}, \ p_{\text{sun}} = \frac{1}{2} \]
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain. The expected amount of “surprise”.

Seattle Weather, Always Raining
\[ p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \quad H = 0 \]

Costa Rica Weather, Coin Flip
\[ p_{\text{rain}} = \frac{1}{2}, \ p_{\text{sun}} = \frac{1}{2} \quad H = 1 \]
Information Theory Intuition

Information Entropy:

\[ H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right] \]

The expected amount of information gain.
The expected amount of “surprise”.

Seattle Weather, Always Raining
\( p_{\text{rain}} = 1, p_{\text{sun}} = 0 \quad H = 0 \)

Costa Rica Weather, Coin Flip
\( p_{\text{rain}} = \frac{1}{2}, p_{\text{sun}} = \frac{1}{2} \quad H = 1 \)

Santa Barbara Weather, Almost Always Sunny.
\( p_{\text{rain}} = \frac{1}{10}, p_{\text{sun}} = \frac{9}{10} \)
Information Theory Intuition

Information Entropy:

$$H = \sum p_i \log \frac{1}{p_i} = E \left[ \log \frac{1}{p_i} \right]$$

The expected amount of information gain. The expected amount of “surprise”.

Seattle Weather, Always Raining
$$p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \quad H = 0$$

Costa Rica Weather, Coin Flip
$$p_{\text{rain}} = \frac{1}{2}, \ p_{\text{sun}} = \frac{1}{2} \quad H = 1$$

Santa Barbara Weather, Almost Always Sunny.
$$p_{\text{rain}} = \frac{1}{10}, \ p_{\text{sun}} = \frac{9}{10} \quad H = 0.4960$$