Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels

PLDI June 20th, 2017

Timos Antonopoulos, Yale
Paul Gazzillo, Yale
Michael Hicks, UMD
Eric Koskinen, Yale
Tachio Terauchi, JAIST
Shiyi Wei, UMD

Side-channel attacks

Applications contain secrets like passwords

Side channels can leak such secrets indirectly

Side channel of interest: running time

```
Changing password for paul.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      return false;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

Number of loop iterations reveals correct character count

```
Changing password for paul.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

Loop iterations now independent of correct characters

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confi
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      ~~return false;~~ correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

Timing Channel Freedom

$$\forall \pi_1, \pi_2.$$
$$in(\pi_1)[low] = in(\pi_2)[low]$$
$$\Rightarrow$$
$$time(\pi_1) = time(\pi_2) \pm c$$

This means low input values for each traces

- Running time does not depend on secret
- Any two traces have roughly same running time for same low input
- A 2-safety property, i.e., must relate pairs of traces to prove property

Self-Composition

```
boolean chpass(real_pwd_A, real_pwd_B, input_pwd_A, input_pwd_B, ...) {
 assume(input_pwd_A == input_pwd_B);
 assume(new_pwd_A == new_pwd_B);
 ...
 result_A =
```

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

```
 result_B =
```

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

```
  assert(equivalent(time_a, time_b))
}
```

- Describes all pairs of traces
- Analyze composite program
- Use non-relational analysis
- Discovering invariants between distinct programs can be challenging for many verifiers

Reframe Timing Channel Freedom

$$\forall \pi_1, \pi_2.$$
$$in(\pi_1)[low] = in(\pi_2)[low]$$
$$\Rightarrow$$
$$time(\pi_1) = time(\pi_2) \pm c$$

$$\exists f.\ \forall \pi.$$
$$time(\pi) = f\,(in(\pi)[low]) \pm c$$

- Function of public inputs *only*
- Non-relational: in terms of one trace
- Implies timing channel freedom, a relational property

Prove with Running Time Analysis

```
for (int i = 0; i < new_pwd.length; i++) {
  matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
}
```

Static Running Time Analysis

$$\text{time}(\pi) = f\,(\text{in}(\pi)[\text{new\_pwd}]) = \text{new\_pwd.length}$$

- Finds running time function f
- Implies timing channel freedom

Finding f Is Hard

- Programs can have nested conditionals and loops
  - Many branches on public inputs
  - At any program point
  - In loop headers
- f can be *piecewise* with complex cases

$$f = \left\{ \dots \begin{cases} \left\{ \dots \begin{cases} \left\{ \dots \right. \\ \left\{ \dots \right. \end{cases} \right. \\ \left\{ \dots \right. \end{cases} \right.$$

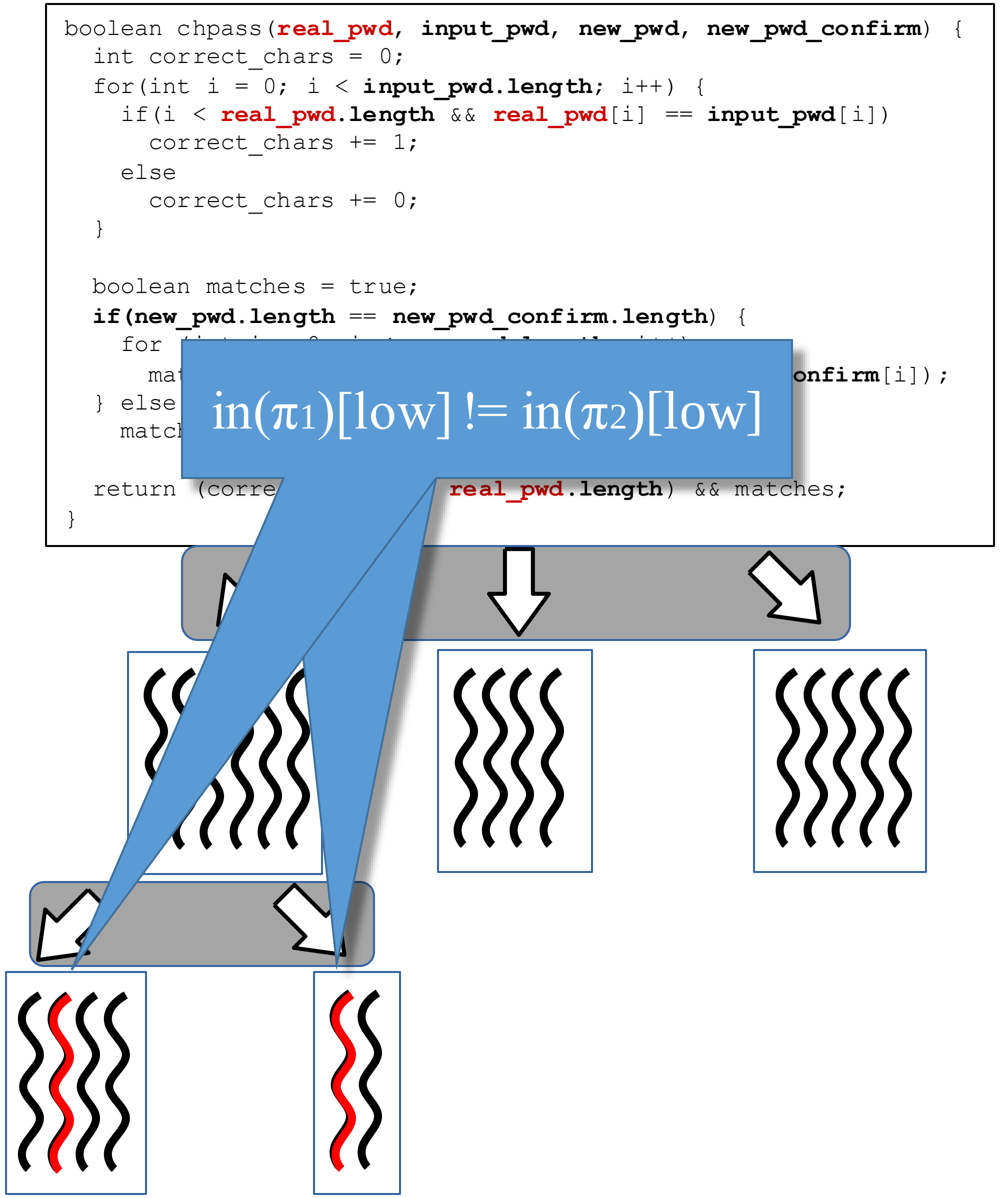
$$f = \begin{cases} \text{input\_pwd.len} * 2 + \text{new\_pwd.len} * 2 + 3 & \textbf{if } \text{new\_pwd.len == new\_pwd\_confirm.len} \\ \text{input\_pwd.len} * 2 + 4 & \textbf{if } \text{new\_pwd.len != new\_pwd\_confirm.len} \end{cases}$$

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

- Piecewise function with complex cases
- Running time analysis can't do this well

Partition the Program

- Prove freedom of partitions alone
- Must choose partitions carefully
- Prove safety of partitions *separately*
- Implies safety of complete program



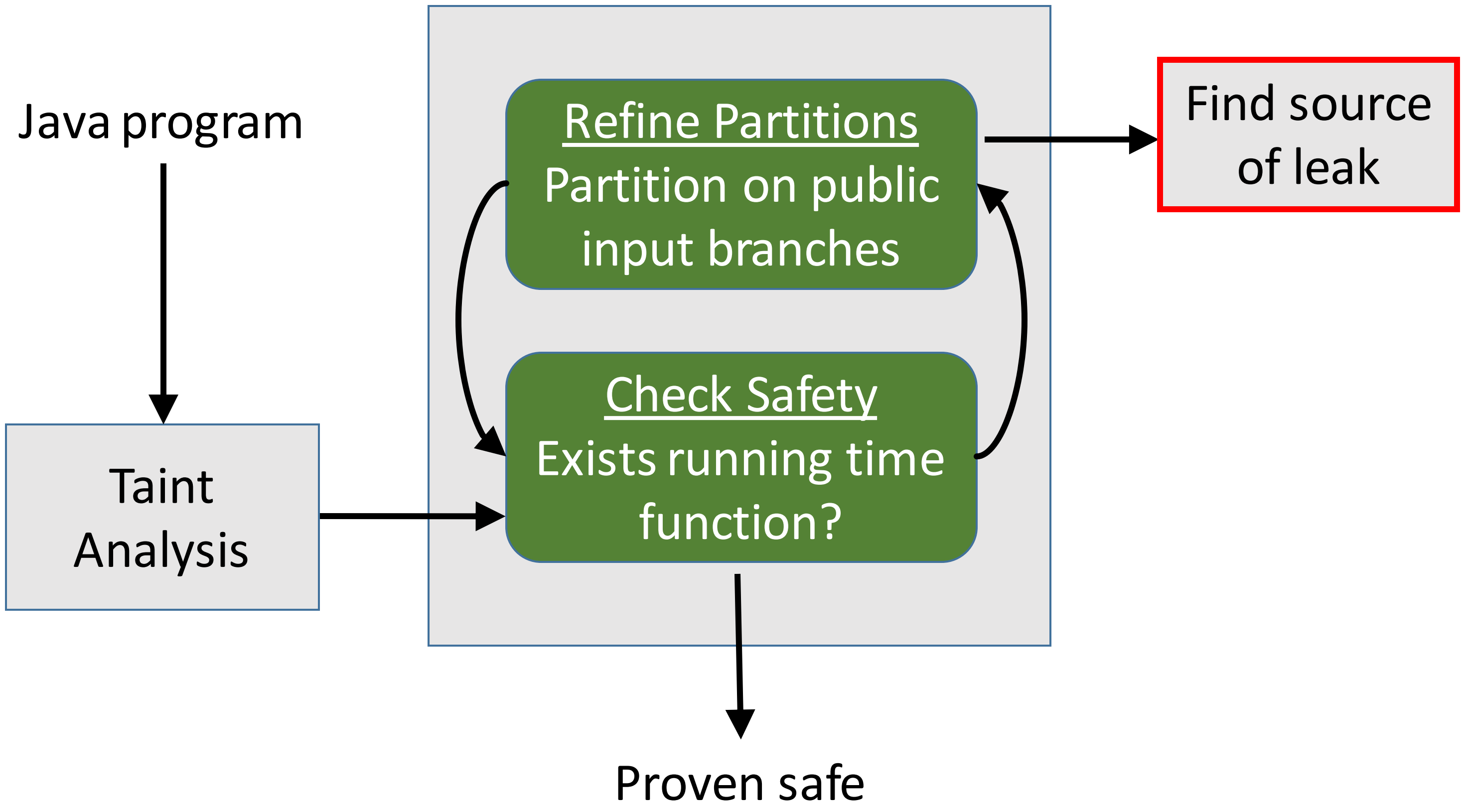
Key Idea

Solve a relational problem by proving a non-relational property about each trace in each partition, for properly chosen partitions.

Contributions

- Technique to prove timing channel freedom by decomposition
- Generalization to k-safety properties
- Implementation of verificaton of timing channel freedom in the Blazer tool with an evaluation

Safety Proving Algorithm



- Use taint analysis to get non-secret branches
- Use static running time bounds analysis
- Iteratively partition and check safety
- Continue partitioning until all partitions safe
- Find source of leak

Algorithm in Action: Check Safety

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_p
    for (int i = 0; i < new_
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

Entire Program

Running time analysis

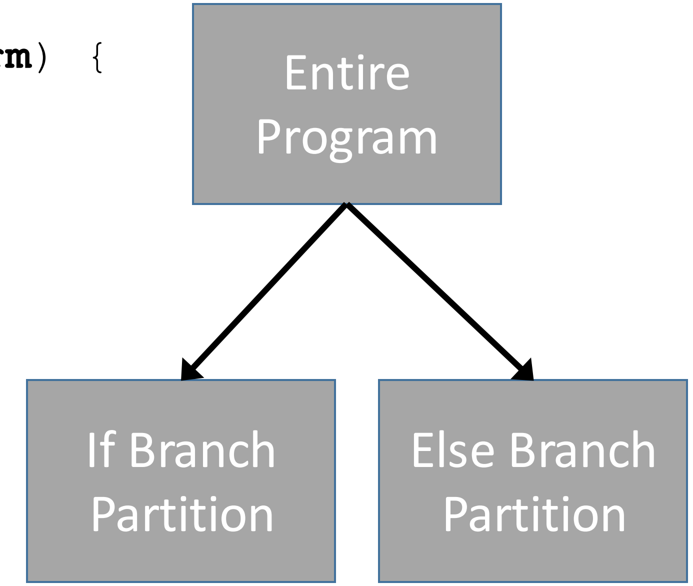
Lower: input_pwd.len * 2 + new_pwd.len * 2 + 3

Upper: input_pwd.len * 2 + 4

Secret-dependent running time could sit between upper and lower

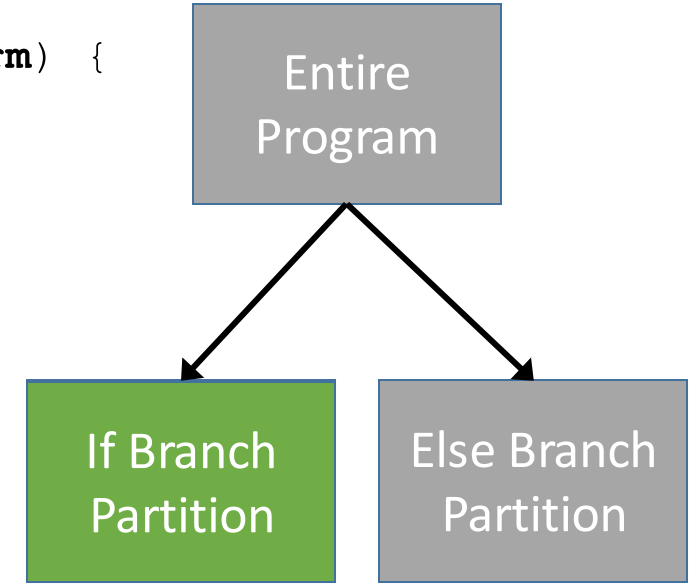
Algorithm in Action: Refine Partitions

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```



Algorithm in Action: Check New Partitions

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = match
  } else
    matches = false;

  return (correct_cha
}
```



Running time analysis

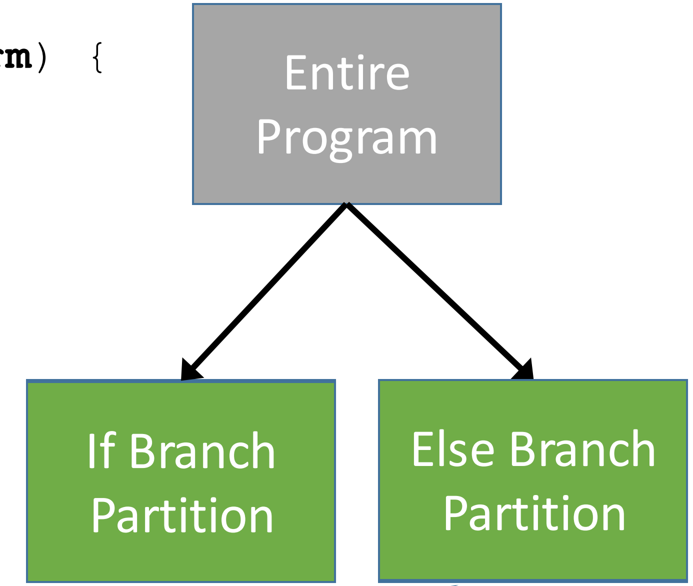
Lower: input_pwd.len * 2 + new_pwd.len * 2 + 3

Upper: input_pwd.len * 2 + new_pwd.len* 2 + 3

Tight running time bounds means safe partition

Algorithm in Action: Check New Partitions

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i
  } else
    matches = false;

  return (correct_chars == real_pwd.l
}
```



Running time analysis

Lower: input_pwd.len * 2 + 4

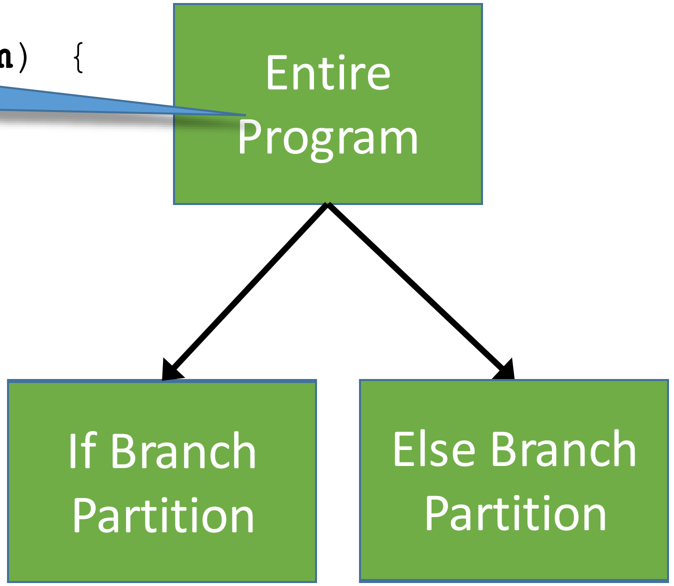
Upper: input_pwd.len * 2 + 4

Tight running time bounds means safe partition

Algorithm in Action: Whole Program Safety

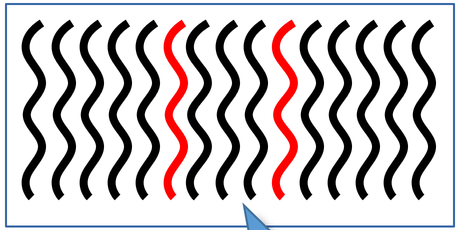
```
boolean chpass(rea ... pwd_confirm) {
  int correct_char ...
  for(int i = 0; i ...
    if(i < real_pw ... input_pwd[i])
      correct_chars += 1;
    else
      correct_chars += 0;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

Safety of all partitions means whole program is safe



General k-Safety Properties

k-Safety Properties



E.g., for 2-safety consider all pairs of traces

$$q(C) \triangleq \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k . \Phi_q(\pi_1, \ldots, \pi_k)$$

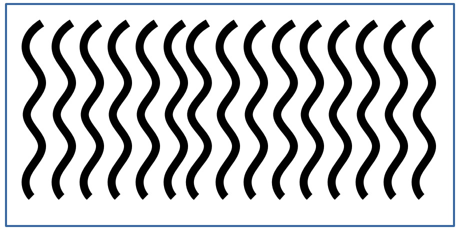
$in(\pi_1)[low] = in(\pi_2)[low]$

$\Rightarrow$

$time(\pi_1) = time(\pi_2) \pm c$

- Reason about combinations of k traces
- To disprove k-safety, you need k traces

Relational by Property Sharing



For a k-safety property q, RBPS(P,q) =

$$\forall \pi_1, \ldots, \pi_k . \bigwedge_{1 \leq i \leq k} P(\pi_i) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)$$

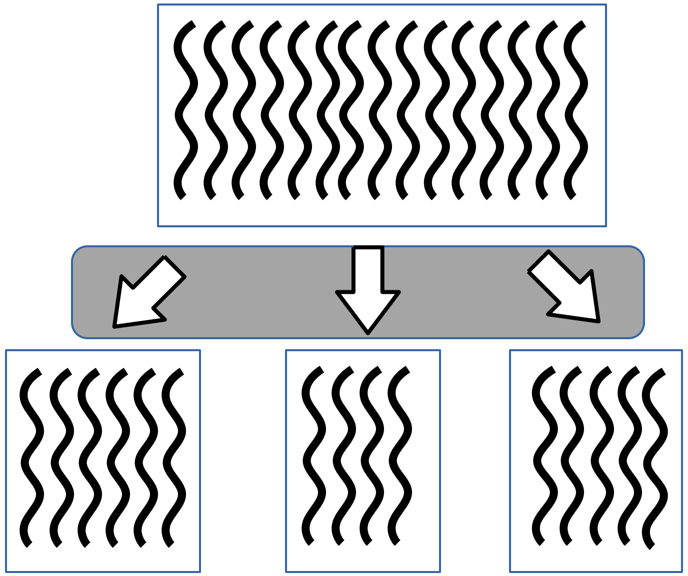
time(π) = f (in(π)[low]) ± c

in(π_1)[low] = in(π_2)[low]

$\Rightarrow$

time(π_1) = time(π_2) ± c

ψ-Quotient Partitioning



$\mathfrak{T} = \{T_1, T_2, T_3\}$

$\mathfrak{T}$ is a ψ-quotient partition provided that:

$$\forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k.$$
$$\psi(\pi_1, \ldots, \pi_k) \Rightarrow \exists T \in \mathfrak{T}. \bigwedge_{1 \leq i \leq k} \pi_i \in T$$

$in(\pi_1)[low] = in(\pi_2)[low]$

$\Rightarrow$

$time(\pi_1) = time(\pi_2) \pm c$

A k-safety property F can be partitioned in this way if

$$\forall C. \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k.$$
$$(\psi(\pi_1, \ldots, \pi_k) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)$$

$in(\pi_1)[low] = in(\pi_2)[low]$

Soundness Theorem

Theorem 3.1 (Soundness). *Suppose that q is ψ-quotient partitionable, and $\mathfrak{T}$ is a ψ-quotient partition for a program C. Then, $q(C)$ holds if for each $T \in \mathfrak{T}$, there exists P such that (i) $\mathsf{RBPS}(P, q)$, and (ii) for each $\pi \in [\![C]\!] \cap T$, $P(\pi)$.*

(Proof in paper)

Other k-Safety Properties

- Determinism

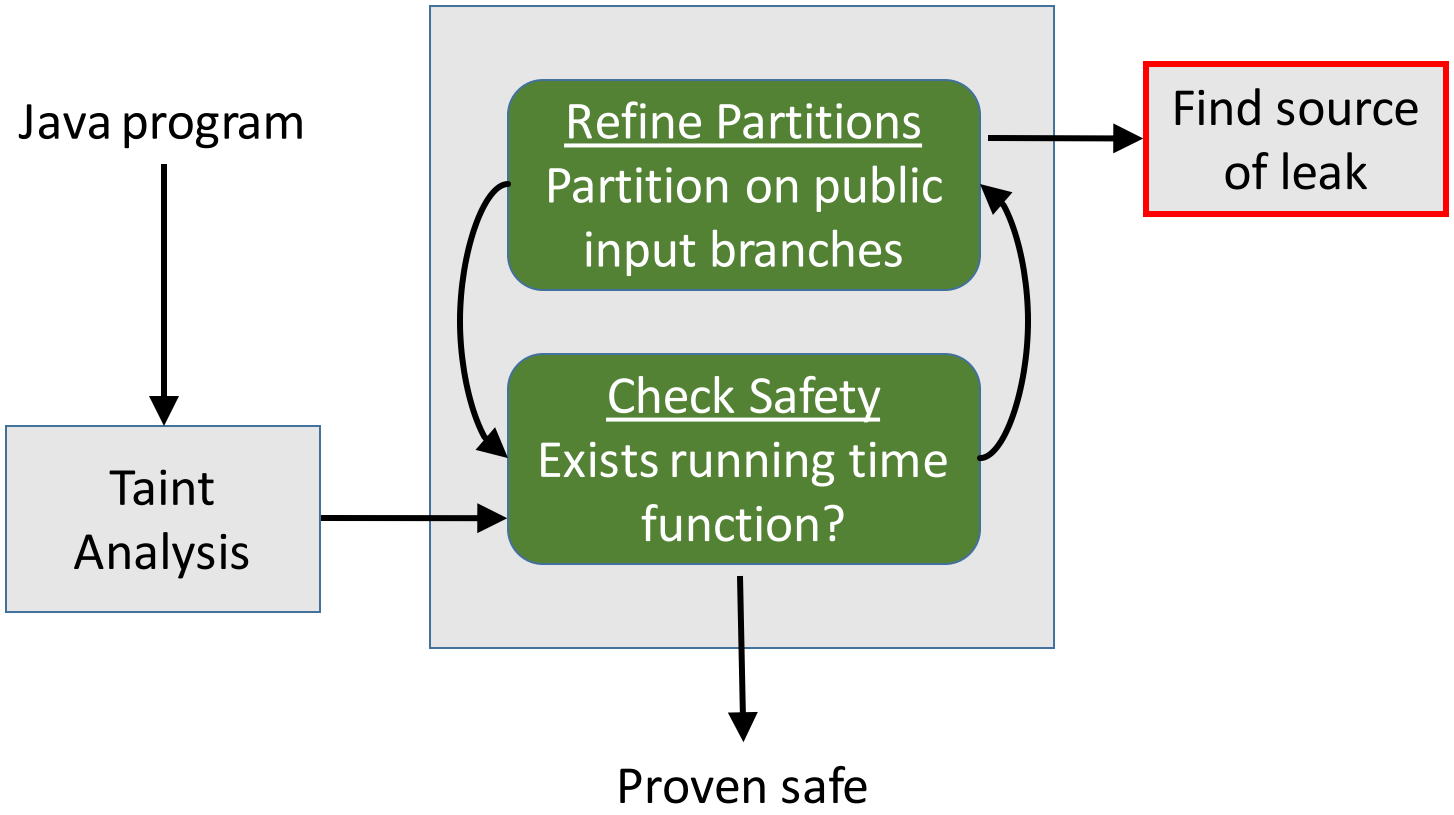
$$det(C) \triangleq \forall\, \pi_1, \pi_2.\\ in(\pi_1) = in(\pi_2) \Rightarrow out(\pi_1) = out(\pi_2).$$

- Relaxed timing channel freedom

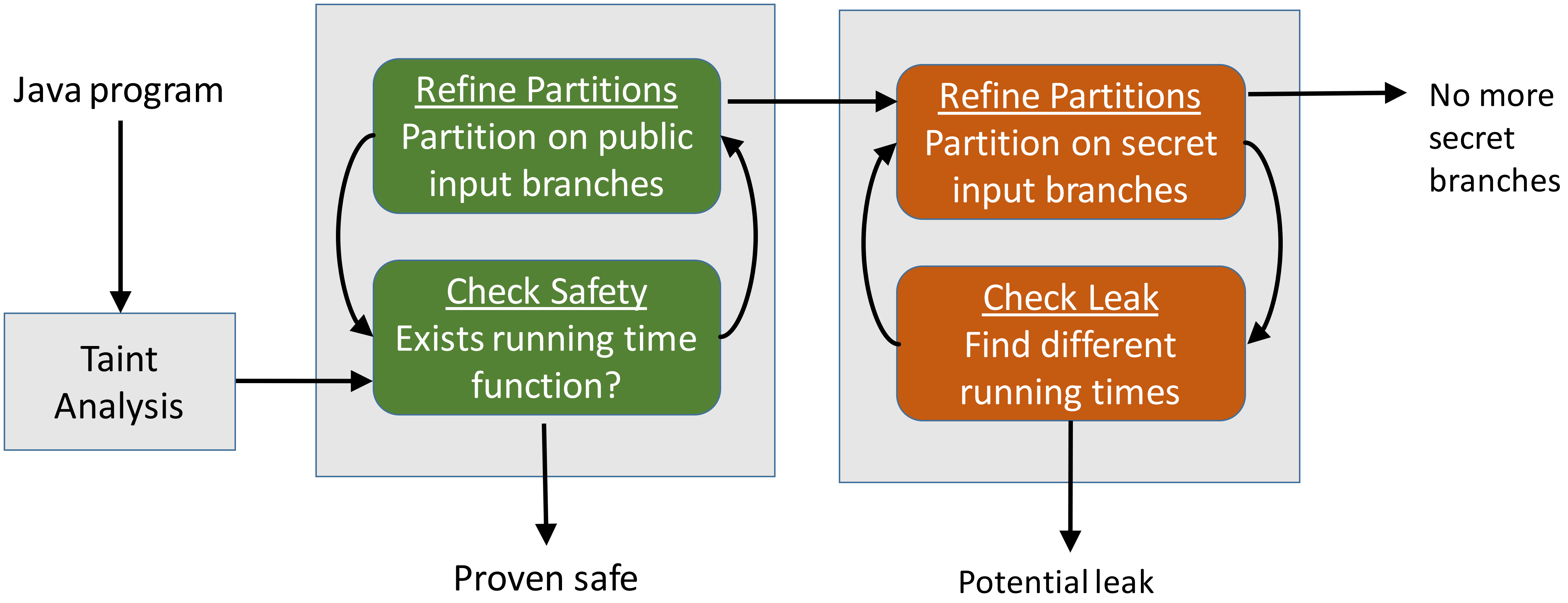
$$ktcf(C) \triangleq \forall\, \pi_1, \pi_2, \pi_3 \in [\![C]\!]^3.\\ \big(in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}] = in(\pi_3)[\boldsymbol{\ell}]\big) \Rightarrow\\ \big(time(\pi_1) \approx time(\pi_2) \vee\\ time(\pi_1) \approx time(\pi_3) \vee\\ time(\pi_2) \approx time(\pi_3)\big).$$

Identifying Leaks

Safety Proving Algorithm



Leak Identification Algorithm



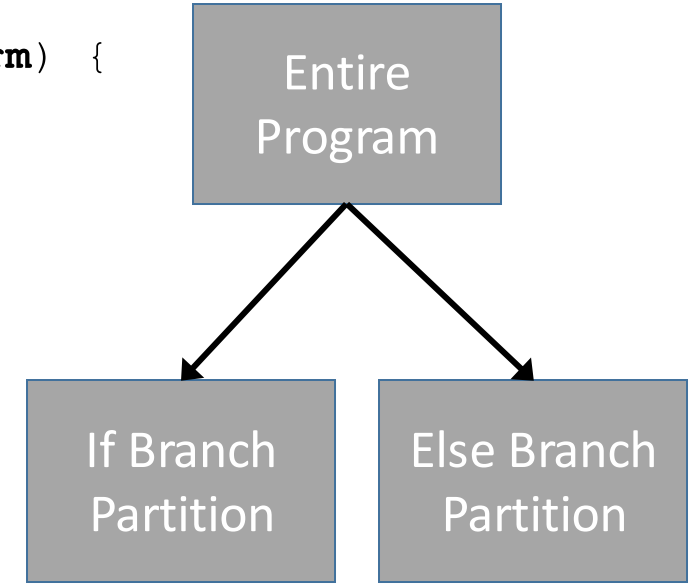
Algorithm in Action

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      return false;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```

Number of loop iterations reveals correct character count

Algorithm in Action: Attempt to Prove Safety

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      return false;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confi
  } else
    matches = false;

  return (correct_chars
}
```



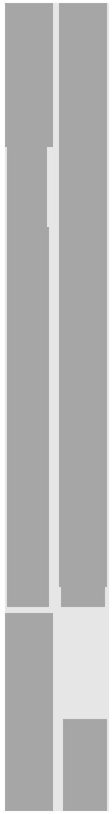
Running time analysis

Lower: 2 + new_pwd.len * 2 + 3

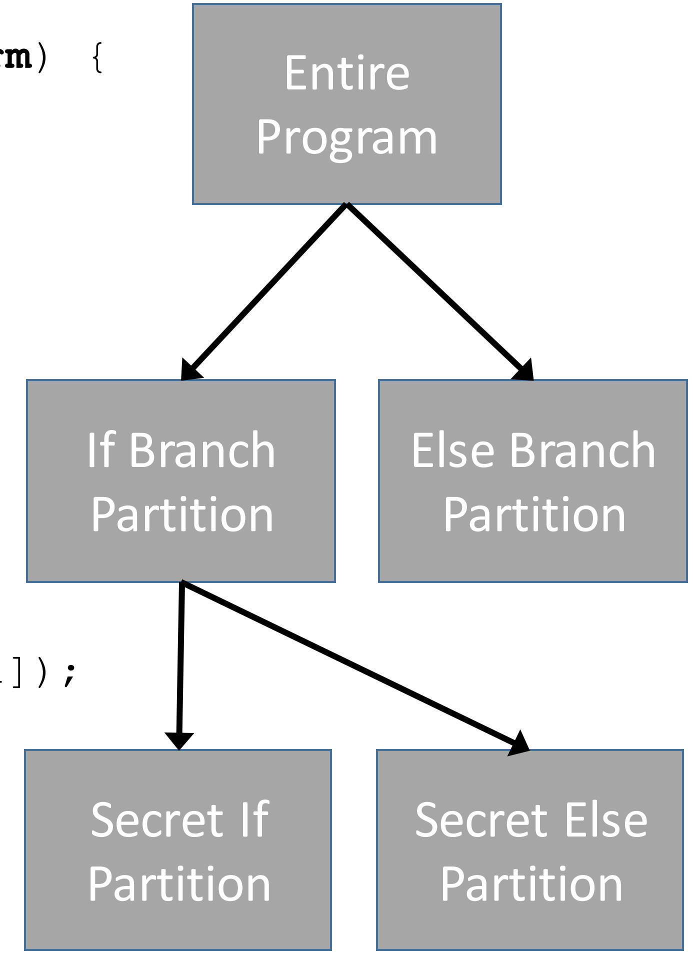
Upper: input_pwd.len * 2 + new_pwd.len * 2 + 3

Could be a secret-dependent difference in running time

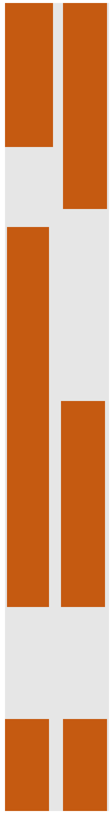
Algorithm in Action: Refine Secret Partitions



```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      return false;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
      matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
  } else
    matches = false;

  return (correct_chars == real_pwd.length) && matches;
}
```



Algorithm in Action: Identify Leak



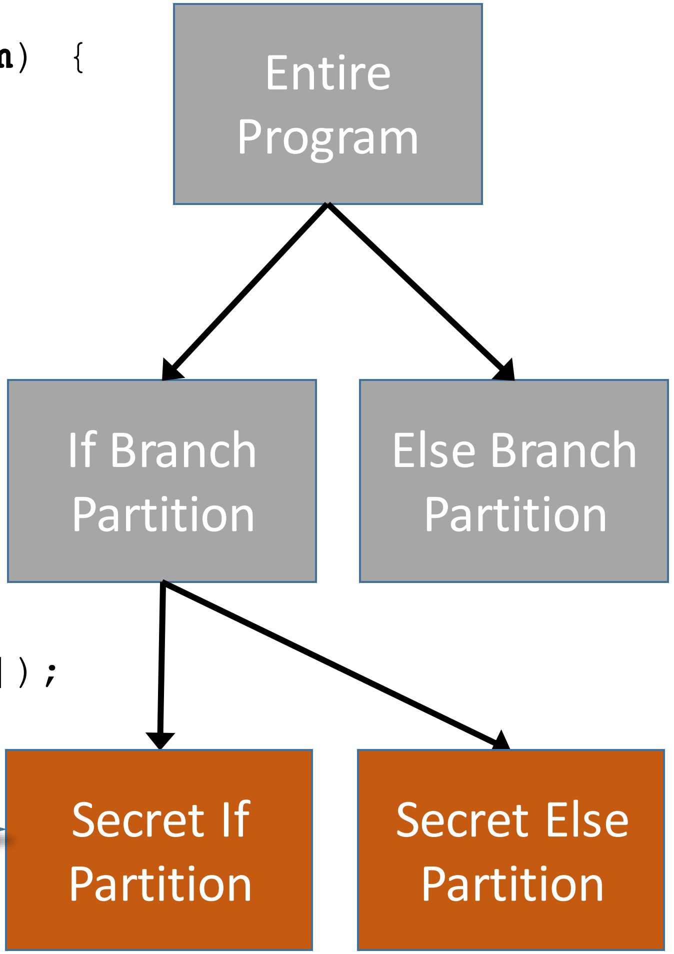
```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {
  int correct_chars = 0;
  for(int i = 0; i < input_pwd.length; i++) {
    if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
      correct_chars += 1;
    else
      return false;
  }

  boolean matches = true;
  if(new_pwd.length == new_pwd_confirm.length) {
    for (int i = 0; i < new_pwd.length; i++)
                                       irm[i]);
  }

  r                                             ;
}
```

Running time analysis

Lower: 2 + new_pwd.len * 2 + 3

Upper: input_pwd.len * 2 + new_pwd.len * 2 + 3

Could be a secret-dependent difference in running time



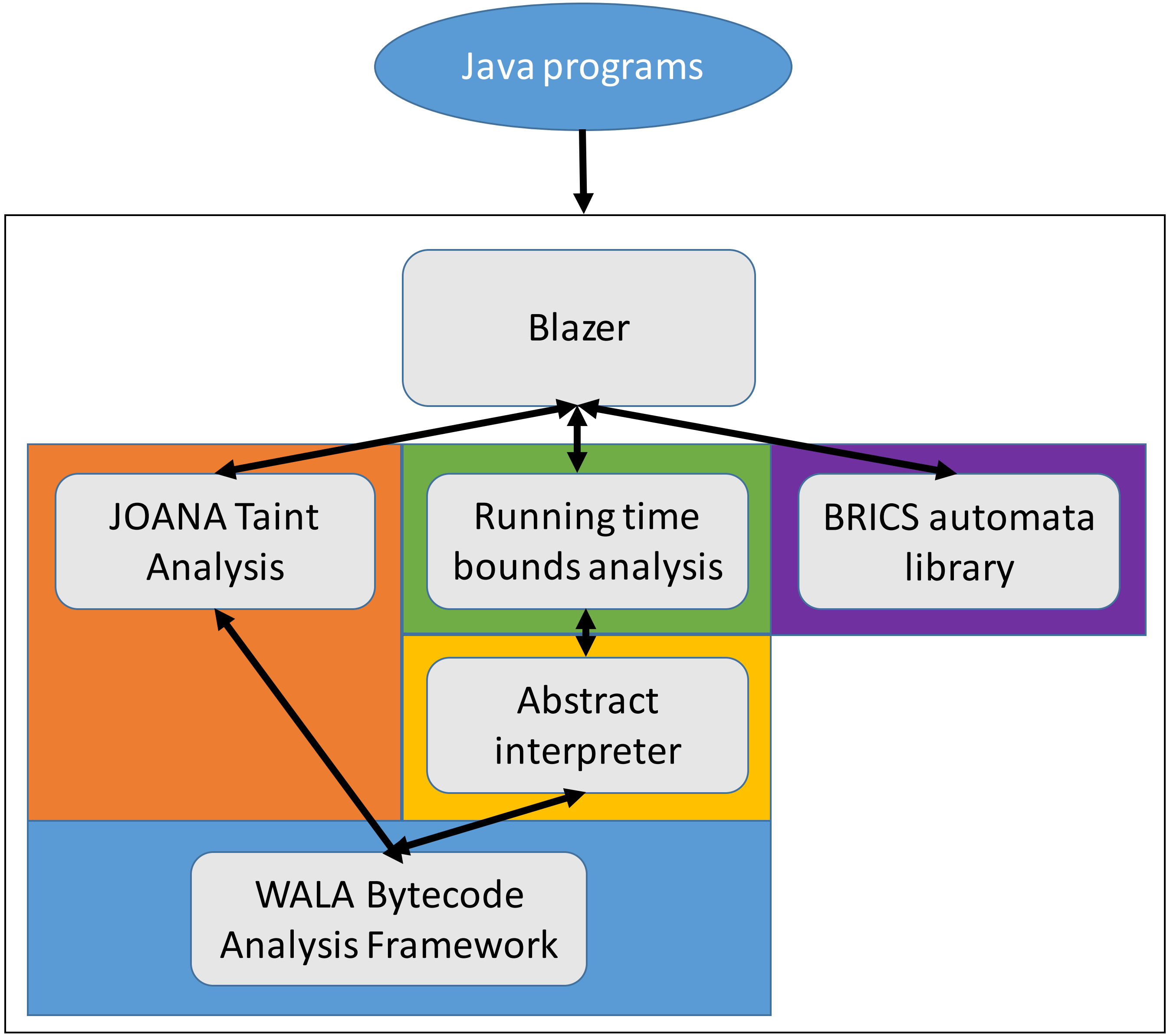
Blazer

An Implementation for Timing Channel Freedom

Architecture

- Taint analysis for public/secret branches
- BRICS library for partitioning
- Running time analysis based on abstract interpreter invariants

Component (Scala)	SLoC
Blazer	4,302
Running time analysis	1,517
Abstract interpreter	4,416



Benchmarks

- Three types of Java benchmarks
  - MicroBench – 12 small, simple examples
  - STAC - 6 from DARPA challenge programs
  - Literature – 6 adapted from literature
- DARPA/STAC benchmarks extracted from large, real-world program
  - Use other techniques to find potentially vulnerable hot methods
- The benchmarks are a few lines to a few dozen
  - One safe version
  - One unsafe version

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
MicroBench			
array_safe	16	1.60	–
array_unsafe	14	0.16	0.70
loopBranch_safe	15	0.23	–
loopBranch_unsafe	15	0.65	1.54
nosecret_safe	7	0.35	–
notaint_unsafe	9	0.28	1.77
sanity_safe	10	0.63	–
sanity_unsafe	9	0.30	0.58
straightline_safe	7	0.21	–
straightline_unsafe	7	22.20	28.49
unixlogin_safe	16	0.86	–
unixlogin_unsafe	11	0.77	1.27
STAC			
modPow1_safe	18	1.47	–
modPow1_unsafe	58	218.54	464.52
modPow2_safe	20	1.62	–
modPow2_unsafe	106	7813.68	31758.92
pwdEqual_safe	16	2.70	–
pwdEqual_unsafe	15	1.30	2.90
Literature			
gpt14_safe	15	1.43	–
gpt14_unsafe	26	219.30	1554.64
k96_safe	17	0.70	–
k96_unsafe	15	1.29	3.14
login_safe	18	6.54	–
login_unsafe	17	4.40	9.10

- Size in basic blocks
- Time to prove safety
  - Average of 5 runs
- If not safe, time to prove attack
  - Average of 5 runs
- A few seconds or less for most benchmarks
  - 22.20s at most for safety proving

Proved safety or leak for all.

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
MicroBench			
array_safe	16	1.60	–
array_unsafe	14	0.16	0.70
loopBranch_safe	15	0.23	–
loopBranch_unsafe	15	0.65	1.54
nosecret_safe	7	0.35	–
notaint_unsafe	9	0.28	1.77
sanity_safe	10	0.63	–
sanity_unsafe	9	0.30	0.58
straightline_safe	7	0.21	–
straightline_unsafe	7	22.20	28.49
unixlogin_safe	16	0.86	–
unixlogin_unsafe	11	0.77	1.27
STAC			
modPow1_safe	18	1.47	–
modPow1_unsafe	58	218.54	464.52
modPow2_safe	20	1.62	–
modPow2_unsafe	106	7813.68	31758.92
pwdEqual_safe	16	2.70	–
pwdEqual_unsafe	15	1.30	2.90
Literature			
gpt14_safe	15	1.43	–
gpt14_unsafe	26	219.30	1554.64
k96_safe	17	0.70	–
k96_unsafe	15	1.29	3.14
login_safe	18	6.54	–
login_unsafe	17	4.40	9.10

Scalability of Leak Identification

- Notable outliers
- Minutes or hours
- Related to block size
- Likely due to many partitions

Future Directions

- More fine-grained partitioning strategies for timing channel freedom
- Scaling Blazer to larger programs
- Using other running time analyses
  - E.g., dynamic running time analysis
- Application to other k-safety properties
  - New non-relational properties
  - New partitioning properties

Conclusion

- Technique to prove timing channel freedom by decomposition
- Generalization to k-safety properties
- Implementation of verification of timing channel freedom in the Blazer tool with an evaluation