CONTEXT-SENSITIVE CORRELATION ANALYSIS FOR RACE DETECTION

Polyvios Pratikakis
Jeff Foster
Michael Hicks

University of Maryland, College Park
Data Races

- Race: two threads access memory without synchronization and at least one is a write
- Famous problems caused by races:
  - August 14th 2004, Northeastern Blackout
  - 1985-1987, Therac-25 medical accelerator
- Programs with races are difficult to understand
A way to prevent races

Intuitively:

- For every dereference, correlate pointer with the acquired locks
- For every shared pointer, intersect locksets of all dereferences
- Verify that all shared pointers are protected.
A way to prevent races

Formally:

- Shared locations $\rho$, locks $\ell$
- Correlation $\rho \triangleright \ell$:
  Lock $\ell$ is correlated with pointer $\rho$ if $\ell$ is held while $\rho$ is accessed
- **Consistent correlation:**
  Location $\rho$ is *always* correlated with lock $\ell$
- Assert that every shared location $\rho$ is *consistently correlated* with a lock $\ell$
**LOCKSMITH: static correlation inference**

- $\rho$ and $\ell$ are static approximations of run-time values
  - Sound, conservative
- Alias analysis:
  - Context-sensitive, flow-insensitive
  - May-alias for locations $\rho$, must-alias for locks $\ell$
- Correlation $\rho \triangleright \ell$ inference
  - Every access creates a $\rho \triangleright \ell$ constraint
  - Infer all other $\rho \triangleright \ell$ relations by closing the constraints
- Consistent correlation
  - Verify consistent correlation for every shared $\rho$, or report a contradiction (race)
Contributions

- Static analysis for inference of *correlation* between locks and pointers

- Context sensitivity
  - Universal polymorphism for function calls
  - Existential polymorphism for data structures

- *Sound* race detection using assertion of *consistent correlation*
  - Formalised for $\lambda^>$, proof of soundness

- **LOCKSMITH**: Implementation for C
Type Based Analysis

- Annotate types with labels:
  - `pthread_mutex_t` → `pthread_mutex_t<\ell>`
  - `\tau^*` → `\tau^*<\rho>`

- Create constraints among labels to capture data flow and correlation
  - Dereferencing `\rho` while `\ell` is held: `\rho \triangleright \ell`
  - Aliasing `\rho` to `\rho'`: `\rho \leq \rho'`
  - Aliasing `\ell` to `\ell'`: `\ell = \ell'`

- Solve constraints to close the relation `\rho \triangleright \ell`

- Verify *consistent correlation* of every shared `\rho` with a single lock `\ell` for all dereferences of `\rho`
pthread_mutex_t L1 = ...;
int x; // &x: int*
void munge(pthread_mutex_t *l, int * p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>
void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
Correlation

```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*\langle\rho_x\rangle
void munge(pthread_mutex_t<\ell> *l, int *\langle\rho\rangle p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```

Correlation

```c
pthread_mutex_t l1 = ...;
int x; // &x: int*ρx
void munge(pthread_mutex_t *l, int *ρp) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```
```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>
void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```
pthread_mutex_t\langle l_1 \rangle \ L1 = ...;
int x; // &x: int*\langle p_x \rangle
void munge(pthread_mutex_t\langle l \rangle *l, int *\langle p \rangle p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
Context Sensitivity

```c
pthread_mutex_t l1 = ..., l2 = ...

int x, y, z; // p_x, p_y, p_z

void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```

\[ \rho_x \quad \rho_y \quad \rho_z \quad \ell_1 \quad \ell_2 \]
pthread_mutex_t \langle \ell_1 \rangle L1 = \ldots, \langle \ell_2 \rangle L2 = \ldots

int x, y, z; // \langle \rho_x \rangle, \langle \rho_y \rangle, \langle \rho_z \rangle

void munge(pthread_mutex_t* \ell, int* \langle \rho \rangle p) {
    pthread_mutex_lock(\ell);
    *p = 3;
    pthread_mutex_unlock(\ell);
}

...  
munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```c
pthread_mutex_t<\ell_1> L1 = ..., <\ell_2> L2 = ...;
int x, y, z; // <\rho_x>, <\rho_y>, <\rho_z>
void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
    munge (&L1, &x);
    munge (&L2, &y);
    munge (&L2, &z);
```
```c
pthread_mutex_t l1 = ..., l2 = ...
int x, y, z; // \rho_x, \rho_y, \rho_z
void munge(pthread_mutex_t l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
... 
munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```
Context Sensitivity

```c
pthread_mutex_t<\ell_1> L1 = ..., <\ell_2> L2 = ...;
int x, y, z; // \langle \rho_x \rangle, \langle \rho_y \rangle, \langle \rho_z \rangle
void munge(pthread_mutex_t<\ell> *l, int *\langle \rho \rangle p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
void munge(NULL, &x);
void munge(NULL, &y);
void munge(NULL, &z);
```
```c
pthread_mutex_t<\ell_1> L1 = ..., \langle \ell_2 \rangle L2 = ...
int x, y, z;  // \langle \rho_x \rangle, \langle \rho_y \rangle, \langle \rho_z \rangle
void munge(pthread_mutex_t<\ell> *l, int *\langle \rho \rangle p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...  
munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```
pthread_mutex_t(L1) L1 = ..., (L2) L2 = ...

int x, y, z; // (ρ_x), (ρ_y), (ρ_z)

void munge(pthread_mutex_t(ℓ) *l, int *ρ p) {
    pthread_mutex_lock(l);
    *(p) = 3;
    pthread_mutex_unlock(l);
}

... munge₁ (&L1, &x);
    munge₂ (&L2, &y);
    munge₃ (&L2, &z);
```c
pthread_mutex_t l1 = ..., l2 = ...;
int x, y, z; // \rho_x, \rho_y, \rho_z
void munge(pthread_mutex_t l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
muenge^1 (&L1, &x);
muenge^2 (&L2, &y);
munge^3 (&L2, &z);
```
Context Sensitivity

```c
pthread_mutex_t l1, l2 = ...;
int x, y, z;  // ρ_x, ρ_y, ρ_z
void munge(pthread_mutex_t l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);
```

![Diagram showing context-sensitive correlation analysis for detecting races]

Context-sensitive Correlation Analysis for Detecting Races – p.10/28
Context Sensitivity

```c
pthread_mutex_t<\ell_1> L1 = ..., <\ell_2> L2 = ...;
int x, y, z; // <\rho_x>, <\rho_y>, <\rho_z>
void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge^1 (&L1, &x);
munge^2 (&L2, &y);
munge^3 (&L2, &z);
```
```c
pthread_mutex_t\langle l_1 \rangle \quad L1 = \ldots, \quad \langle l_2 \rangle \quad L2 = \ldots;
int \quad x, \quad y, \quad z; \quad // \quad \langle \rho_x \rangle, \quad \langle \rho_y \rangle, \quad \langle \rho_z \rangle
void \ munge(pthread_mutex_t\langle l \rangle \quad *l, \quad int \quad *\langle \rho \rangle \quad p) \ \{ \n \quad pthread_mutex_lock(l);
 \quad *p = 3;
 \quad pthread_mutex_unlock(l);
\}

... (3)_3

\mu_nge^1 (\&L1, \ &x);
\mu_nge^2 (\&L2, \ &y);
\mu_nge^3 (\&L2, \ &z);
```

Context-sensitive Correlation Analysis for Detecting Races – p.10/28
pthread_mutex_t ℓ₁ L1 = ..., ℓ₂ L2 = ...
int x, y, z; // ℓ₁ ρₓ, ℓ₂ ρᵧ, ℓ₁ ρᵦ
void munge(pthread_mutex_t ℓ) *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge¹ (&L1, &x);
munge² (&L2, &y);
munge³ (&L2, &z);

\[
\begin{align*}
\rho_x & \rightarrow \rho_y \\
\rho_z & \rightarrow \ell_1 \\
\ell_2 & \rightarrow (3)3
\end{align*}
\]
Context Sensitivity

```c
pthread_mutex_t<\ell_1> L1 = ..., <\ell_2> L2 = ...;
int x, y, z; // <\rho_x>, <\rho_y>, <\rho_z>
void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge^1 (&L1, &x);
munge^2 (&L2, &y);
munge^3 (&L2, &z);
```
Linearity of locks

- Each lock label $\ell$ might represent more than one run-time locks.

- Then:
  - Which one is correlated with $\rho$ in $\rho \triangleright \ell$?
  - Which one gets acquired by `pthread_mutex_lock`?

- So, locks $\ell$ have to be linear (must alias)

- Challenges:
  - Dynamic allocation of locks
  - Want to avoid being overly conservative in loops
Linearity Effects

- Prevent simply unifying every $\ell$ - assert linearity
- Each expression has a linearity effect $\varepsilon$
- Allocating a fresh lock has a fresh singleton effect $\{\ell\}$
- Effect of composite expressions is disjoint union of effects
- Filter effects to remove any $\ell$ that does not escape
Linearity Example

```c
pthread_mutex_t L1, L2, *l;
int x; // &x: int*

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...) l = &L1;
else l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```
Linearity Example

```c
pthread_mutex_t L1<\ell_1>, L2<\ell_2>, *l<\ell>;
int x; // &x: int*<\rho_x>

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...) l = &L1;
else l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```

Context-sensitive Correlation Analysis for Detecting Races – p.13/28
Linearity Example

```c
pthread_mutex_t L1(l_1), L2(l_2), *l(l);
int x; // &x: int* ρ_x

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...) l = &L1;
else l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```
Linearity Example

```
pthread_mutex_t L1(\ell_1), L2(\ell_2), *l(\ell);
int x; // &x: int*(\rho_x)

pthread_mutex_init(&L1); {\ell_1}
pthread_mutex_init(&L2); {\ell_2}
if(...) l = &L1; \emptyset
else l = &L2; \emptyset
pthread_mutex_lock(&l); \emptyset
x = 3;
pthread_mutex_unlock(&l); \emptyset
```
Linearity Example

```c
pthread_mutex_t L1, L2, *l;
int x; // &x: int*

pthread_mutex_init(&L1); {l_1}
pthread_mutex_init(&L2); {l_2}
if(...) l = &L1; 0
else l = &L2; 0
pthread_mutex_lock(&l); 0
x = 3;
pthread_mutex_unlock(&l); 0
```
Linearity Example

```c
pthread_mutex_t L1, L2, *l;
int x; // &x: int*ρx

pthread_mutex_init(&L1); {l1}
pthread_mutex_init(&L2); {l2}
if(...) l = &L1; 0
else l = &L2; 0
pthread_mutex_lock(&l); 0
x = 3;
pthread_mutex_unlock(&l); 0
```

\( l_1 \neq l_2 \)
Soundness

- Formal system for a functional language: $\lambda$
- Proof: type safety in $\lambda$ implies race freedom

- Correlation constraints have other applications:
  - Pointers correlated with allocation regions
  - Arrays correlated with integer lengths
LOCKSMITH: Implementation for C

- Apply consistent correlation inference to the full C language

Challenges:
- Infer the acquired set at every program point
- Locks in data structures
- Increase precision using void * inference
- Thread locality (can be flow sensitive)
- Reduce memory footprint with lazy struct field expansion
- Liveness/uniqueness analysis reduce false positives when an (eventually) shared location variable is provably still thread-local
- Continuation read/write effects used to precisely find shared locations at fork points
Flow sensitive lock state

- Which locks are acquired at each program point?
- Create context sensitive control-flow graph:
  - For every program point create a state variable $\psi$
  - $\psi$ nodes have kinds (Acquire, Release, Newlock, Deref, etc.)
  - $\psi \rightarrow \psi'$: control flow
  - $\psi \leftarrow (i \rightarrow) \psi'$: control enters function at call site $i$
  - $\psi \rightarrow (i \rightarrow) \psi'$: function returns control at call site $i$
  - Solve using data-flow analysis
Example: generating constraints

```
thread_mutex_t l1 = ...;
int x; // &x: int*

void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

... munge(l1, &x);
```
Example: generating constraints

```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>

void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

...  
munge^1(&L1, &x);
```
Example: generating constraints

```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>

void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

munge(&L1, &x);
```

...
Example: generating constraints

```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>

void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

... munge^(1)(&L1, &x);
```
Example: generating constraints

```c
pthread_mutex_t l1 = ...;
int x; // &x: int*<ρ_x>

void munge(pthread_mutex_t l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

munge(&L1, &x);
```

Acquired
Example: generating constraints

```c
pthread_mutex_t\langle\ell_1\rangle L1 = \ldots;
int x; // &x: int*\langle\rho_x\rangle

void munge(pthread_mutex_t\langle\ell\rangle *l, int *\langle\rho\rangle p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

\ldots
munge^l\langle L1, x\rangle;
```

[Dereferenced]

Context-sensitive Correlation Analysis for Detecting Races – p.17/28
Example: generating constraints

```c
pthread_mutex_t<\ell_1> L1 = ...;
int x; // &x: int*<\rho_x>

void munge(pthread_mutex_t<\ell> *l, int *<\rho> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

munge1(&L1, &x);
```

Licensed under the BSD 2-clause license.
Example: solving constraints
Example: solving constraints
Example: solving constraints
Example: solving constraints
Example: solving constraints

\[
\begin{align*}
\psi_{\text{in}} & \quad \psi_1 \quad \psi_2 \quad \psi_3 \quad \psi_{\text{out}} \\
\psi_{\text{call}} & \quad \psi_{\text{ret}} \quad \rho_x \quad \ell_1
\end{align*}
\]

Acq, Deref, Rel
Example: solving constraints
Example: solving constraints
Often, locks exist in data structures:

```c
struct foo {
    pthread_mutex_t lock;
    int* data;
    struct foo* next;
};
```

- Alias analysis conflates nodes in data structures
- Can recover precise correlation within individual elements
- Programmer writes existential annotations
Existential Context Sensitivity

- Often, locks exist in data structures:
  ```c
  struct foo {
    ∃ρ, ℓ. ρ ⊨ ℓ
    pthread_mutex_t<ℓ> lock;
    int*<ρ> data;
    struct foo* next;
  };
  ```

- Alias analysis conflates nodes in data structures
- Can recover precise correlation within individual elements
- Programmer writes existential annotations
struct foo<\ell, \rho> s;
if (...) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell, \rho> s;
if (...) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell, \rho> s;
if(...) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell, \rho> s;
if (...) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell,\rho> s;
if(...) pack^1(s) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo\{l,\rho\} s;
if(...) pack^1(s) {
    s.lock = &L1; s.data = &x;
} else {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell,\rho> s;
if(...) pack^1(s) {
    s.lock = &L1; s.data = &x;
} else pack^2(s) {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo\{\ell, \rho\} s;
if(...) pack^1(s) {
    s.lock = \&L1; s.data = \&x;
} else pack^2(s) {
    s.lock = \&L2; s.data = \&y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
struct foo<\ell, \rho> s;
if(...) pack^1(s) {
    s.lock = &L1; s.data = &x;
} else pack^2(s) {
    s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
Annotate each occurrence of a `void *` pointer with a type list.

Any time a type \( \tau \) is cast to or from a `void *`, we add \( \tau \) to the list.

Track flow and instantiations of `void *` types.

Simple worklist algorithm:
- For every list with \( \geq 2 \) elements, *conflate* all labels.
- Unify type lists if there is flow between `void *` types.
- For `void *` instantiation, unify with an instance of the type list.
- For every *singleton type* `void *`, treat it as \( \tau \).
Lazy struct fields

- Annotate every `struct` type with an empty list of fields
- Whenever a field is used, add it to the list
- Worklist solving algorithm:
  - Unify field lists if there is flow between `struct` types
  - Track instantiations of `struct` types and instantiate field lists
int* shared; /* shared global pointer */
f() {
    int* x = (int*) malloc(sizeof(int));
    *x = 2; /* x is not yet shared */
    shared = x; /* x becomes shared */
}

- Very simple uniqueness analysis “filters” some unnecessary checks
- Reduced number of false positives
- Still sound
Experiments

- Standalone C programs
- Linux device drivers
- Experiments on a dual core Xeon processor, at 2.8MHz, 3.5GB RAM
## Standalone programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (KLOC)</th>
<th>Time</th>
<th>Warn.</th>
<th>Unguarded</th>
<th>Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>aget</td>
<td>1.6</td>
<td>0.8s</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>ctrace</td>
<td>1.8</td>
<td>0.9s</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>pfscan</td>
<td>1.7</td>
<td>0.7s</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>engine</td>
<td>1.5</td>
<td>1.2s</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>smtprc</td>
<td>6.1</td>
<td>6.0s</td>
<td>46</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>knot</td>
<td>1.7</td>
<td>1.5s</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
# Linux Drivers

<table>
<thead>
<tr>
<th>Driver</th>
<th>Size (KLOC)</th>
<th>Time</th>
<th>Warn.</th>
<th>Unguarded</th>
<th>Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>plip</td>
<td>19.1</td>
<td>24.9s</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>eql</td>
<td>16.5</td>
<td>3.2s</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3c501</td>
<td>17.4</td>
<td>240.1s</td>
<td>24</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sundance</td>
<td>19.9</td>
<td>98.2s</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sis900</td>
<td>20.4</td>
<td>61.0s*</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>slip</td>
<td>22.7</td>
<td>16.5s*</td>
<td>19</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hp100</td>
<td>20.3</td>
<td>31.8s*</td>
<td>23</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

(*) Run without lock linearity analysis
Conclusions

Contribution:

- Discover races automatically by inferring consistent correlation
- Formalized correlation inference system with universal and existential context sensitivity
- Proof of soundness
- **LOCKSMITH**: Implementation for C
  - Requires no annotations (minimal annotations when using existential context sensitivity)
  - Found races in existing programs and Linux drivers
LOCKSMITH is available

- Download LOCKSMITH at
  http://www.cs.umd.edu/~polyvios/locksmith

- Analyses are modular, easy to reuse