LOCKSMITH: Context-sensitive Correlation Analysis for Detecting Races

Polyvios Pratikakis
Jeffrey S. Foster
Michael Hicks

University of Maryland, College Park
Data Races

- Race: two threads access a memory location without synchronization and at least one is a write

- Well known problems caused by races:
  - August 14th 2003, Northeastern Blackout
  - 1985-1987, Therac-25 medical accelerator

- Programs with races are difficult to understand
A way to prevent races

- Shared locations $\rho$, locks $\ell$
- Correlation $\rho \triangleright \ell$:  
  Lock $\ell$ is correlated with pointer $\rho$ if-f $\ell$ 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 $\lambda_\triangleright$, proof of soundness
- **LOCKSMITH**: Implementation for C
```c
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);
```
```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);
```
Correlation

```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);
```
```c
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);
```
Correlation

```c
pthread_mutex_t(l_1) L1 = ...;
int x; // &x: int*<p_x>
void munge(pthread_mutex_t(l) *l, int *<p> 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);
pthread_mutex_t(l_1) L1 = ..., (l_2) 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);
}
...
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; // <\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);
```

Diagram:

- \(\rho_x\):
- \(\rho_y\):
- \(\rho_z\):
- \(\ell_1\):
- \(\ell_2\):
- \(\rho\) pointing to \(\ell\):

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

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

```c
pthread_mutex_t {\ell_1} L1 = \ldots, {\ell_2} L2 = \ldots;
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);  
```
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);
}
...

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

Diagram showing context-sensitive correlation analysis for detecting races.
```c
pthread_mutex_t l1 = ..., l2 = ...;
int x, y, z; // ρ_x, ρ_y, ρ_z
void munge(pthread_mutex_t *l, int *ρ 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; // ρ_x, ρ_y, ρ_z
void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3; // (1)
    pthread_mutex_unlock(l); // (1)
}
...
munge^1(&l1, &x);
munge^2(&l2, &y);
munge^3(&l2, &z);
```
Context Sensitivity

```c
pthread_mutex_t<\ell_1> L1 = ...; 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_1(&L1, &x);
munge_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);
```

```
\begin{tikzpicture}[->]
    \node (rho) at (0,0) {$\rho$};
    \node (l1) at (1,1) {$\ell_1$};
    \node (l2) at (1,-1) {$\ell_2$};
    \node (l) at (0.5,0) {$\ell$};
    \node (rho_x) at (-1,0) {$\rho_x$};
    \node (rho_y) at (0,0) {$\rho_y$};
    \node (rho_z) at (1,0) {$\rho_z$};
    \draw (rho) -- (l1);
    \draw (rho) -- (l2);
    \draw (rho) -- (l);
    \draw (rho_x) -- (rho_y);
    \draw (rho_y) -- (rho_z);
\end{tikzpicture}
```
```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);
}
...
```

```
\rho_x \quad \rho_y \quad \rho_z \quad \ell_1 \quad \ell_2
```

```
munge_1(&L1, &x);
munge_2(&L2, &y);
munge_3(&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);
}

... munge^1(&L1, &x);
munge^2(&L2, &y);
munge^3(&L2, &z);
```

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

\ldots
munge^1\langle \ell_1, \&L1, \&x \rangle;
munge^2\langle \ell_2, \&L2, \&y \rangle;
munge^3\langle \ell_2, \&L2, \&z \rangle;
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);
```

Context-sensitive Correlation Analysis for Detecting Races – p. 8/21
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
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
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
  - $\psi \xrightarrow{(i)} \psi'$: control enters function at call site $i$
  - $\psi \xrightarrow{(i)} \psi'$: function returns control at call site $i$
  - Solve using data-flow analysis
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

```
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

```
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_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^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^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<1>(&L1, &x);
```

\[\Psi_{\text{ret}}\]
\[\Psi_{\text{call}}\]
\[\Psi_3\]
\[\Psi_2\]
\[\Psi_1\]
\[\Psi_{\text{in}}\]

\[\rho_x\]
\[\ell_1\]

\[\rho\]
\[\ell\]

\text{Released}

Context-sensitive Correlation Analysis for Detecting Races – p. 13/21
Example: solving constraints
Example: solving constraints
Example: solving constraints
Example: solving constraints
Example: solving constraints
Example: solving constraints

\[
\begin{align*}
\Psi_{\text{in}} &\rightarrow \Psi_{\text{out}} \\
\Psi_1 &\rightarrow \Psi_2 \\
\Psi_2 &\rightarrow \Psi_3 \\
\Psi_3 &\rightarrow \psi_{\text{call}} \\
\psi_{\text{call}} &\rightarrow \psi_{\text{ret}} \\
\rho &\rightarrow \rho_x \\
\ell &\rightarrow \ell_1
\end{align*}
\]
Example: solving constraints
Often, locks exist in data structures:

```c
struct foo {
    pthread_mutex_t<\ell> lock;
    int*<\rho> 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 { \exists \rho, \exists \ell. \rho \triangleright \ell
    pthread_mutex_t<\ell> lock;
    int*<\rho> data;
    struct foo* next;
  };
  ```

- Alias analysis conflates nodes in data structures
- Can recover precise correlation within individual elements
- Programmer writes existential annotations
- More details in the paper
  - Full description in SAS’06
Experiments

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