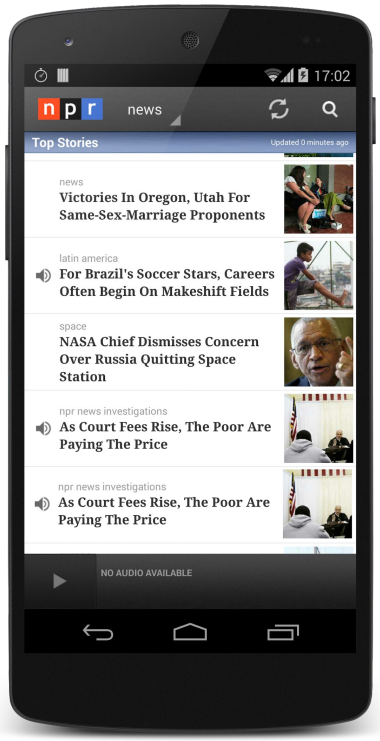
Scalable Race Detection for Android Applications

Pavol Bielik, Veselin Raychev, Martin Vechev

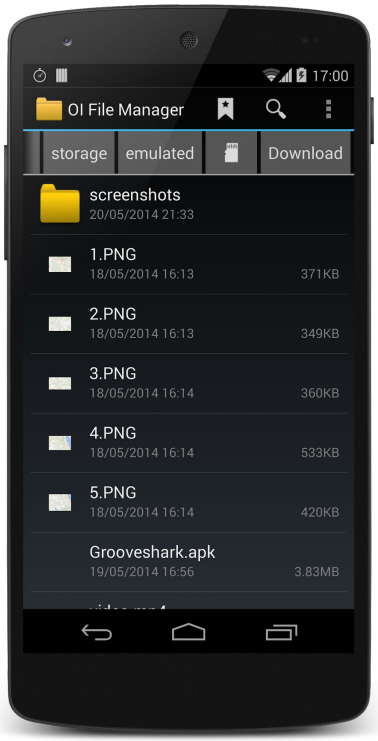
Software Reliability Lab
Department of Computer Science
ETH Zurich

SPLASH
PITTSBURGH · 2015
OCTOBER 25-30

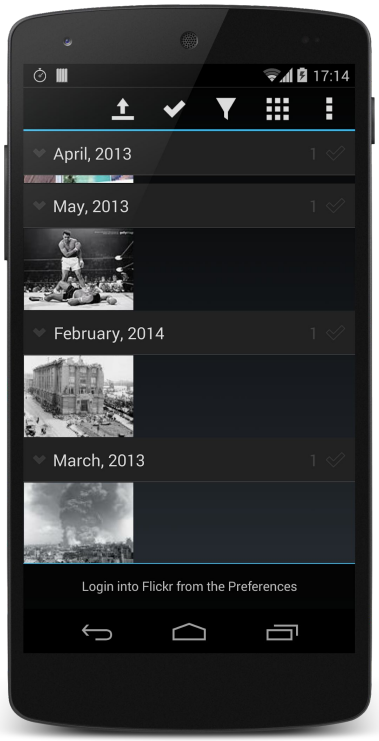
Errors Caused by Concurrency



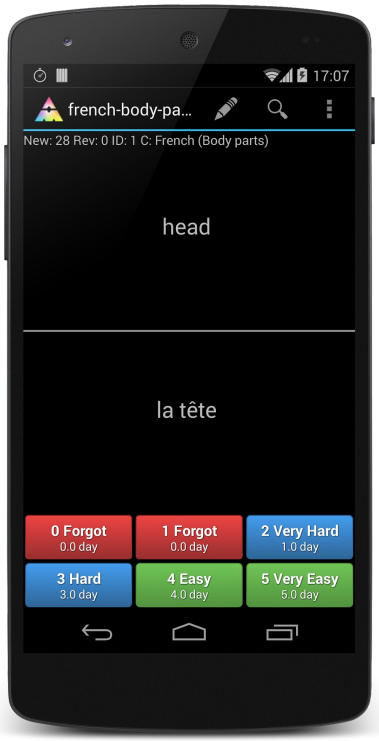
Display article twice



Display wrong directory

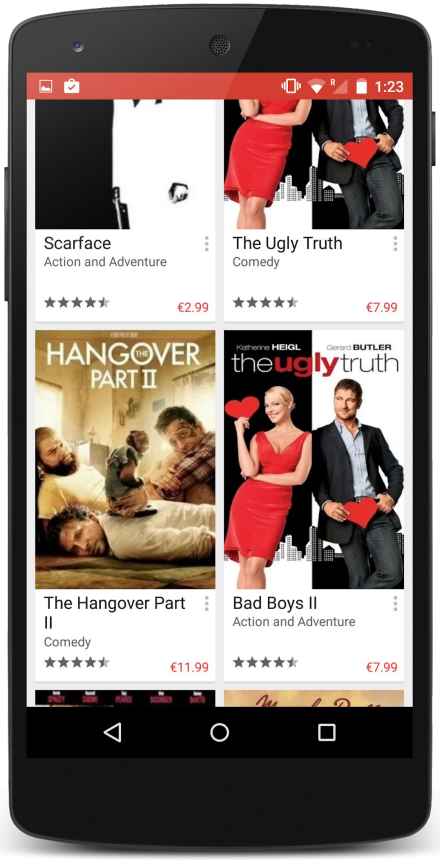


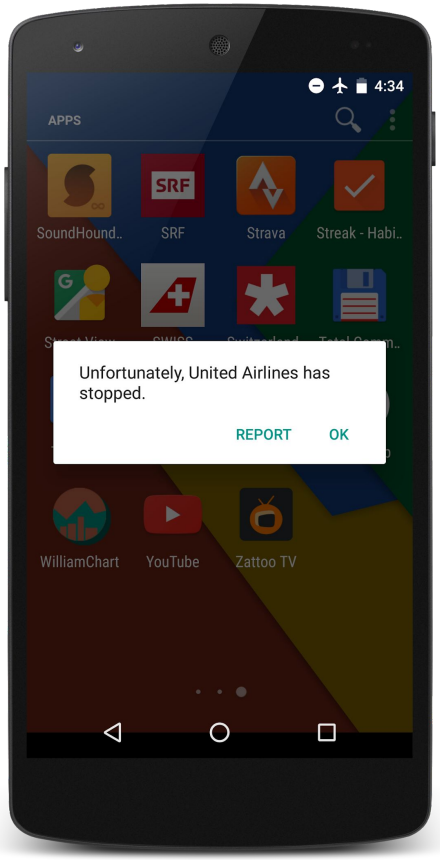
Display wrong order



Rate wrong card

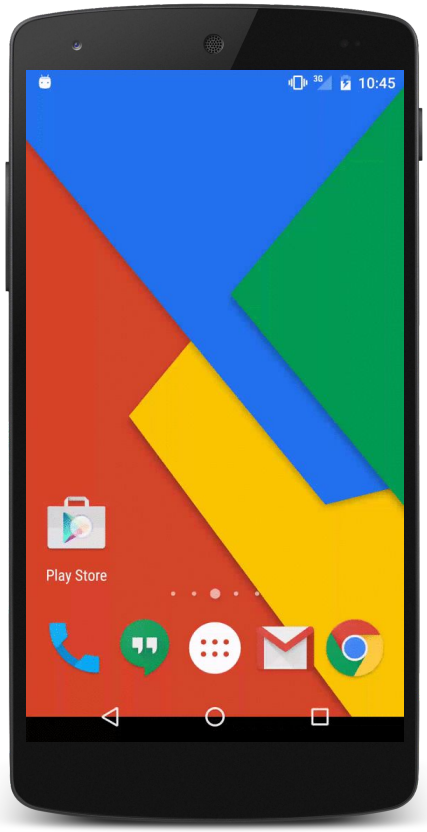
Errors Caused by Concurrency





Android Asynchrony

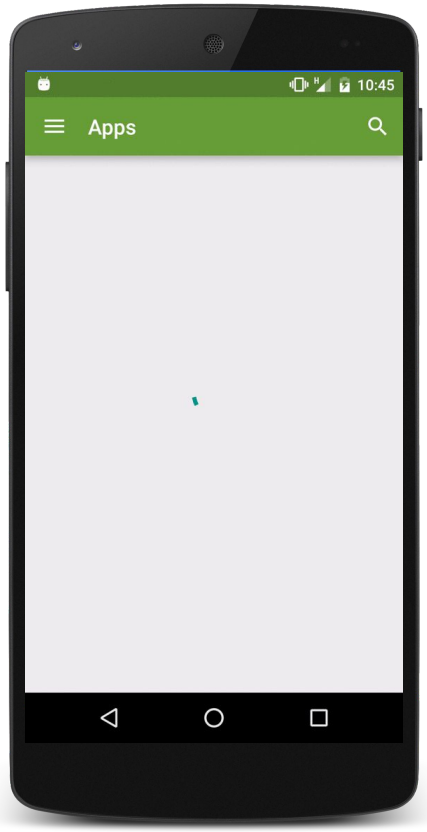
Android Asynchrony



UI Thread

```
onCreate()
```

Android Asynchrony



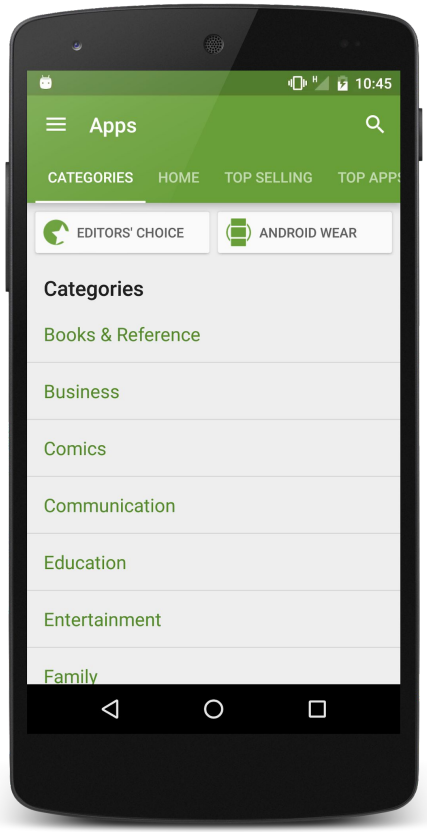
UI Thread

```
onCreate()
```

Background Thread

```
downloadData()
```

Android Asynchrony



UI Thread

```
onCreate()
```

Background Thread

```
downloadData()
```

```
onPostExecute()
```

Android Asynchrony

UI Thread

```
onCreate()
```

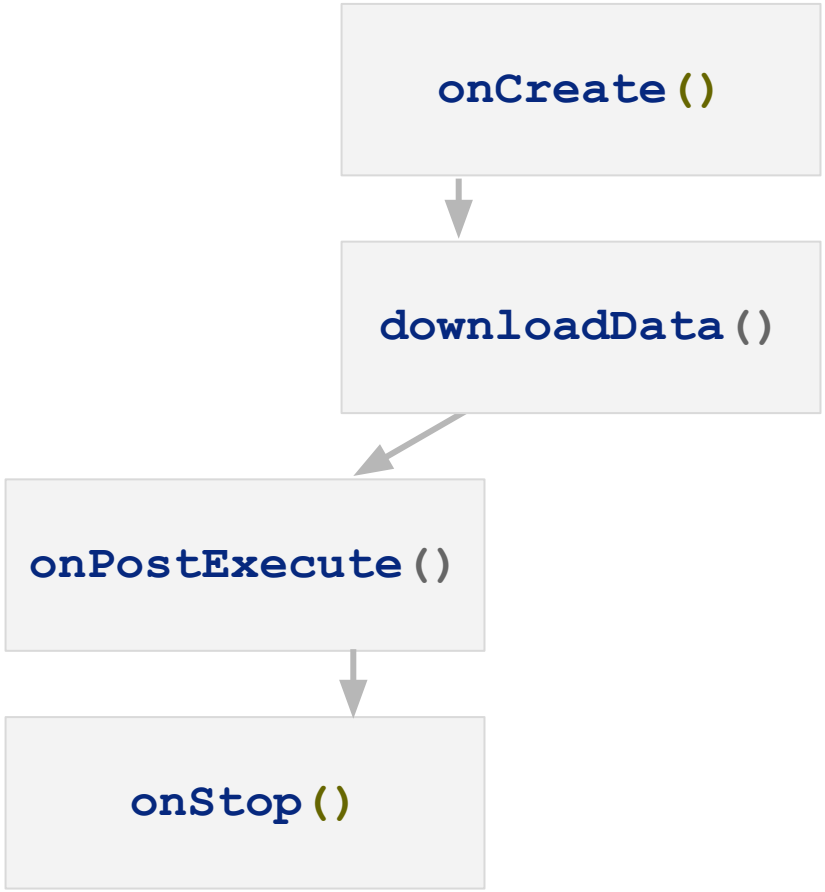
Background Thread

```
downloadData()
```

```
onPostExecute()
```

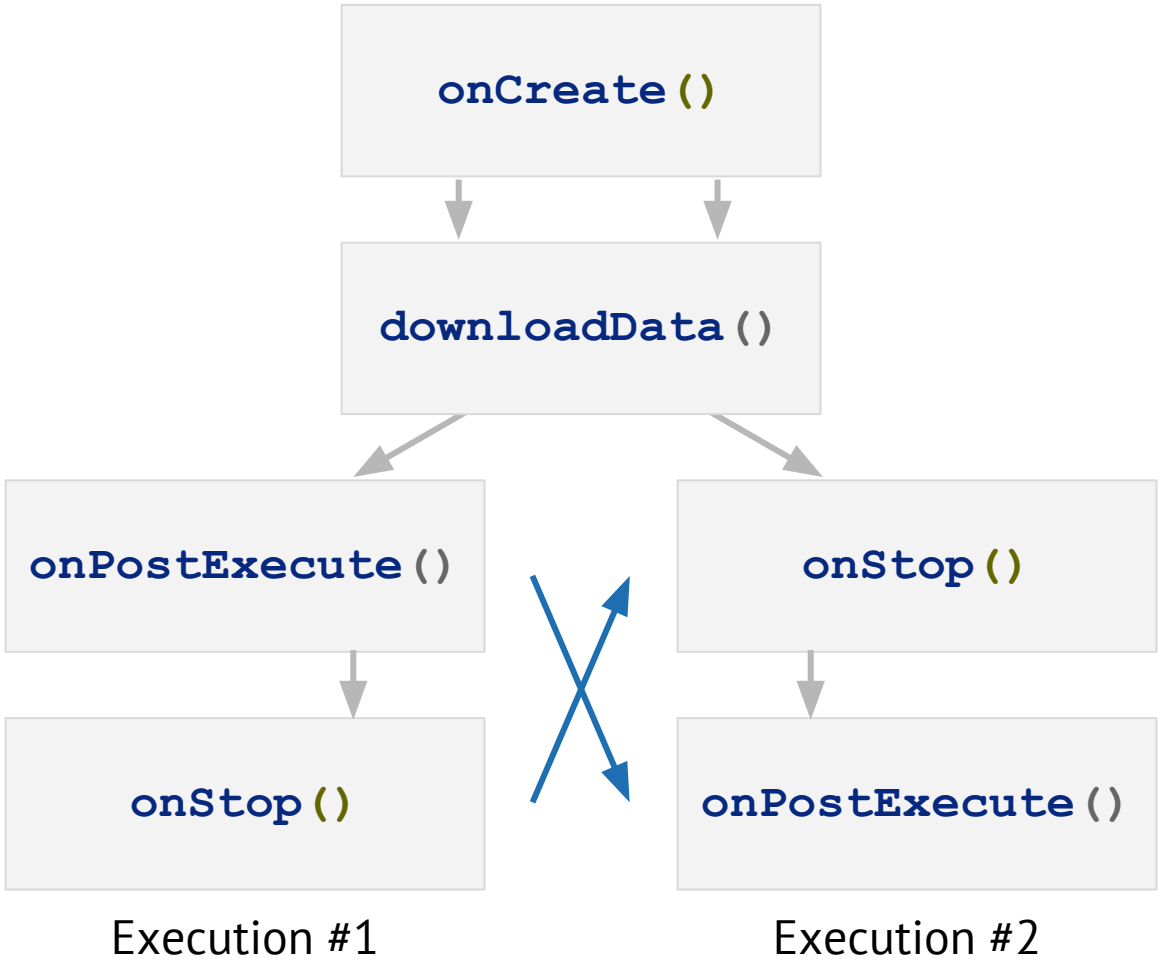
```
onStop()
```

Android Asynchrony



Execution #1

Android Asynchrony



Online analysis tool

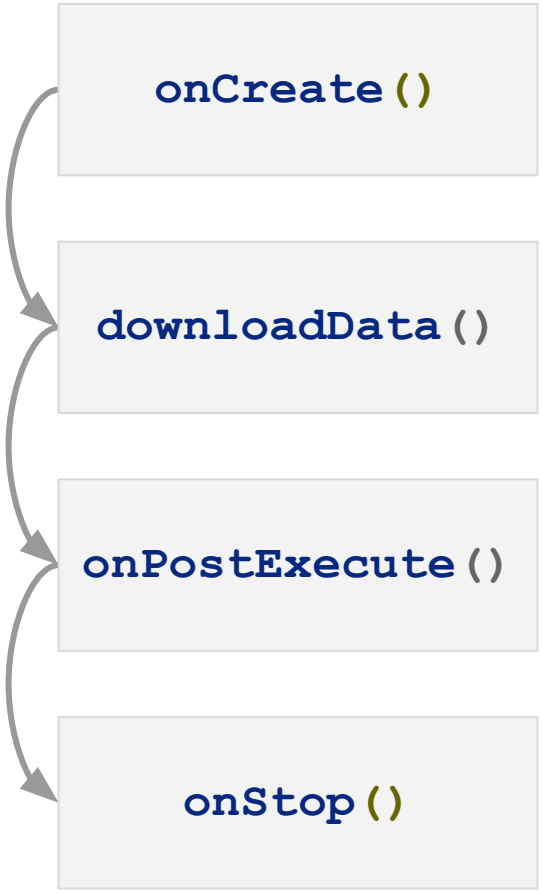
Select Android application APK file for analysis

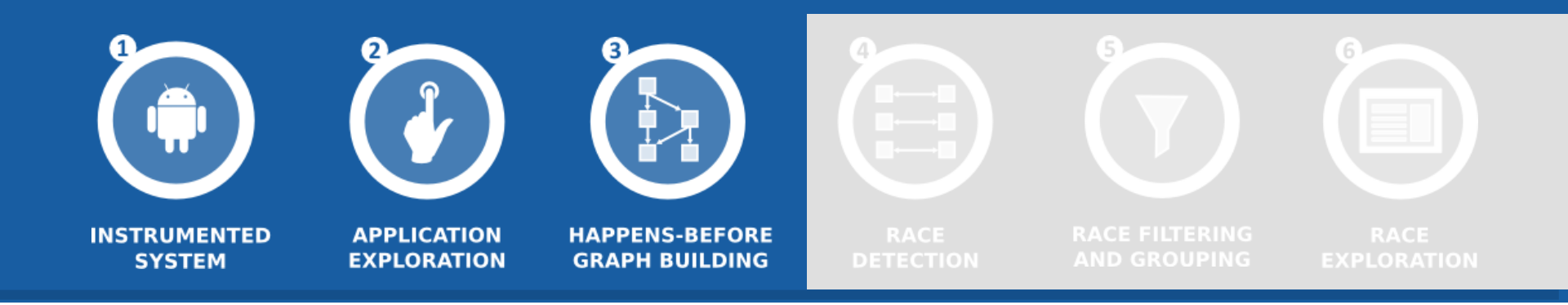
Choose File No file chosen

FIND RACES

Trace Order

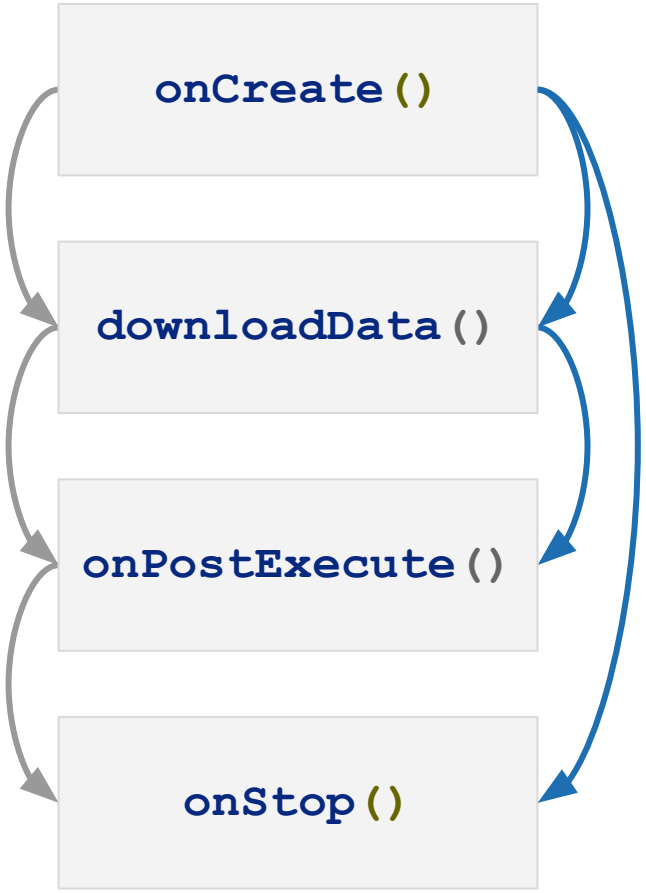
Captures ***single trace*** observed during the dynamic analysis





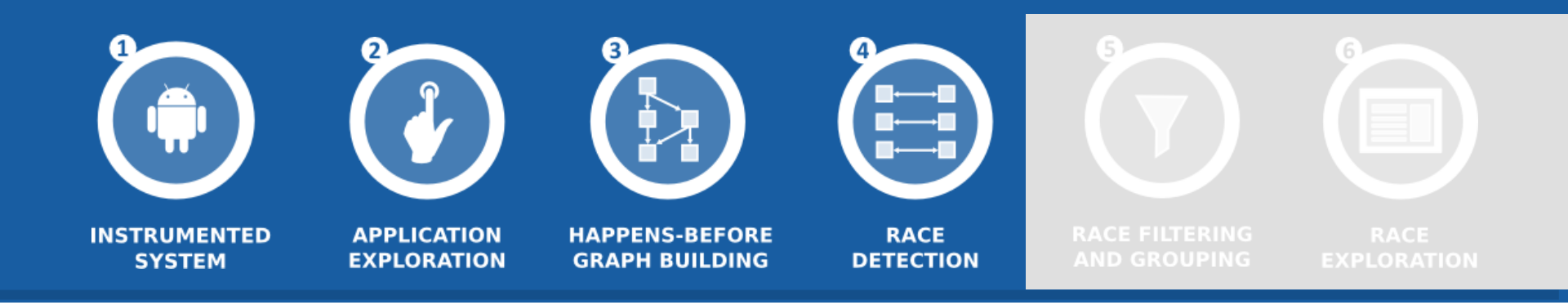
Trace Order

Captures ***single trace*** observed during the dynamic analysis



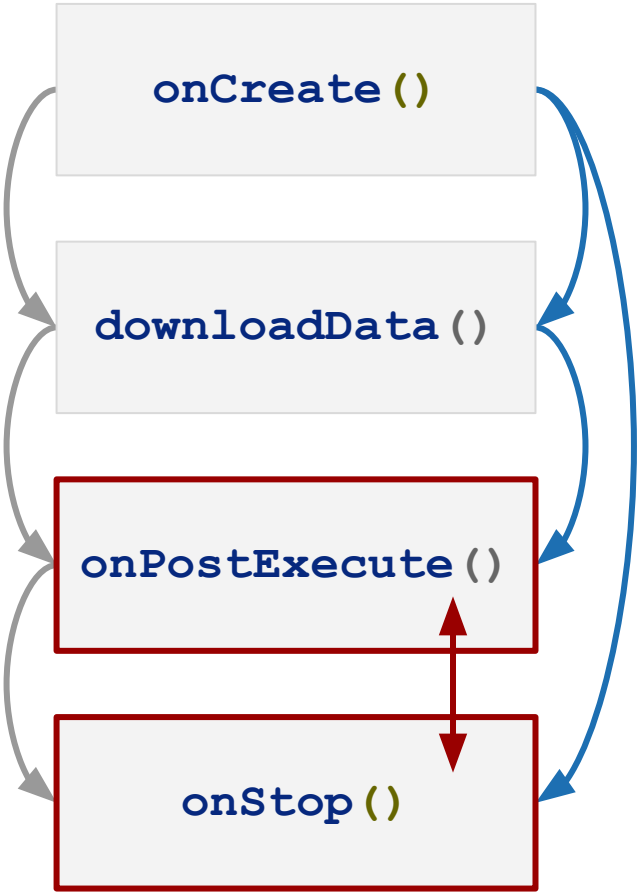
Happens-before Order

Defines ***set of traces*** that are possible interleavings of the original trace



Trace Order

Captures ***single trace*** observed during the dynamic analysis

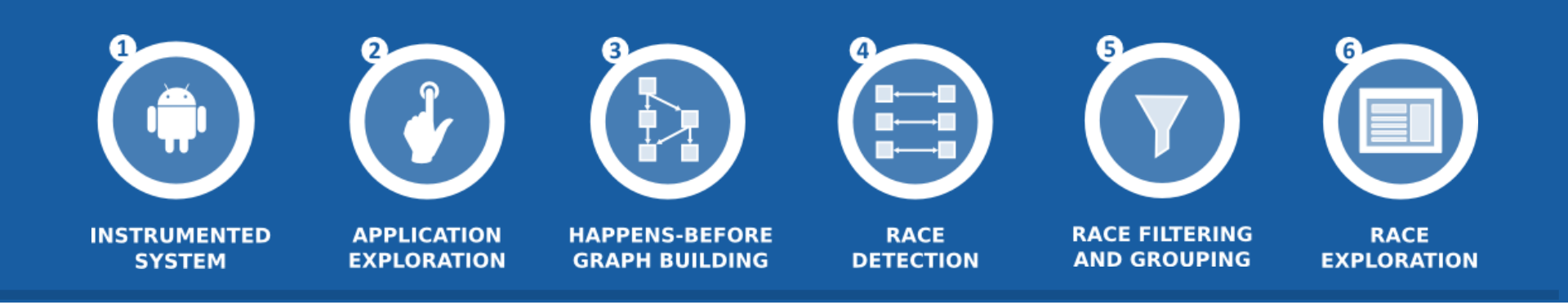


Happens-before Order

Defines ***set of traces*** that are possible interleavings of the original trace

Concurrency Interference

unordered conflicting accesses to the same memory location

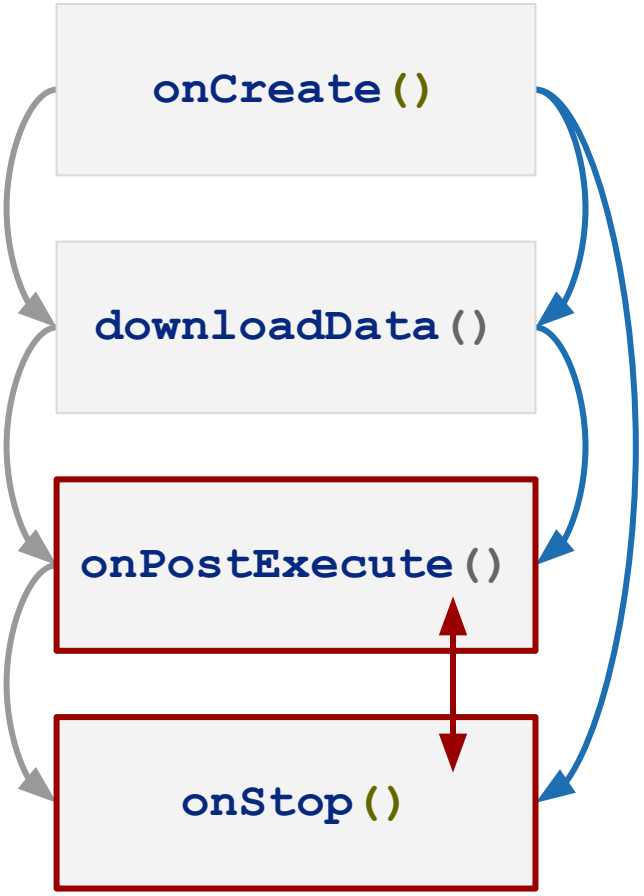


Trace Order

Captures ***single trace*** observed during the dynamic analysis

Filtering & Grouping

order of magnitude reduction of reported conflicts

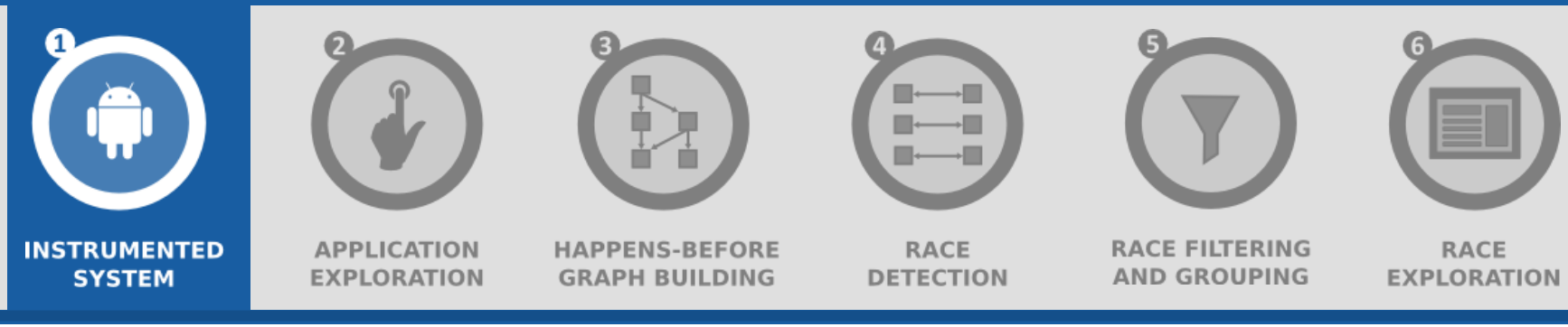


Happens-before Order

Defines ***set of traces*** that are possible interleavings of the original trace

Concurrency Interference

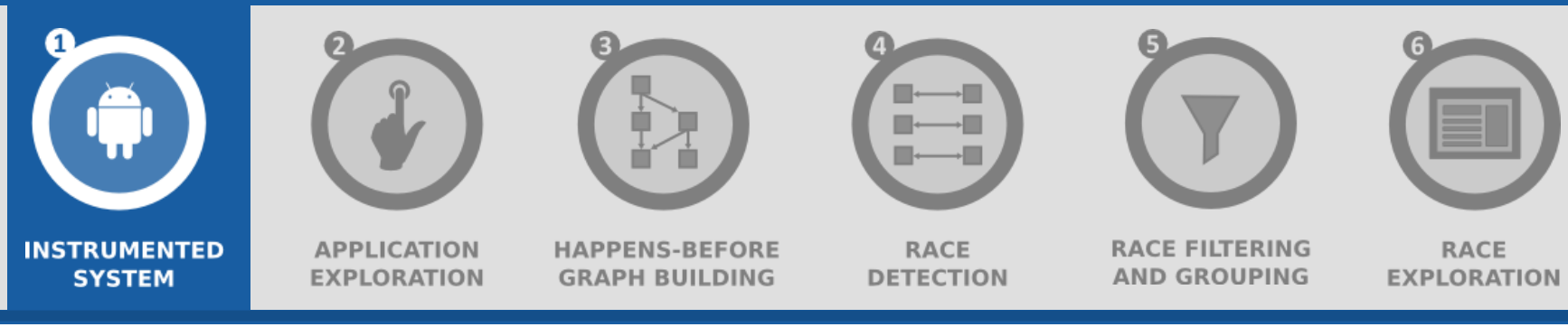
unordered conflicting accesses to the same memory location



What are the operations capturing essential features of event-driven applications?

```
void onCreate() {
  new Thread(…).start();
}
```

```
void downloadData() {
  postDelayed(…, 100);
}
```



What are the operations capturing essential features of event-driven applications?

begin(…)

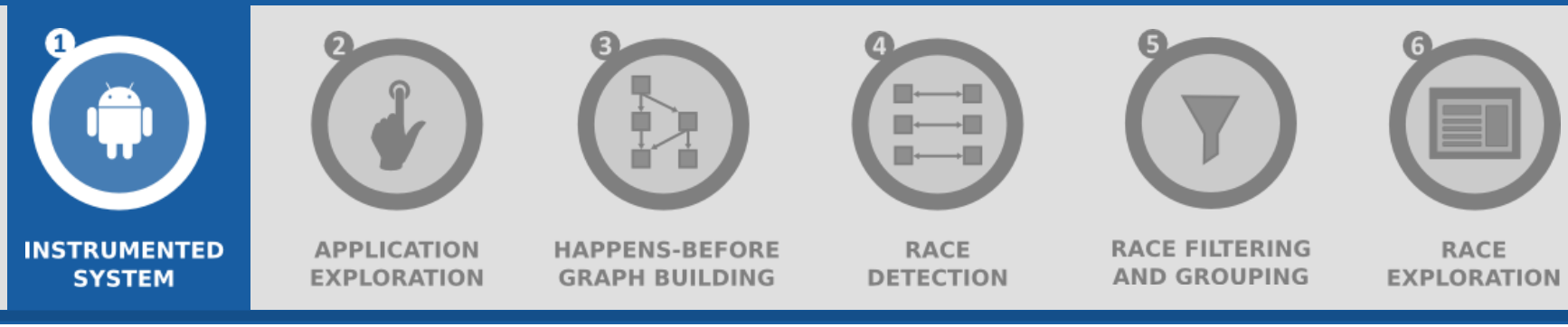
```
void onCreate() {
  new Thread(…).start();
}
```

end(…)

begin(…)

```
void downloadData() {
  postDelayed(…, 100);
}
```

end(…)

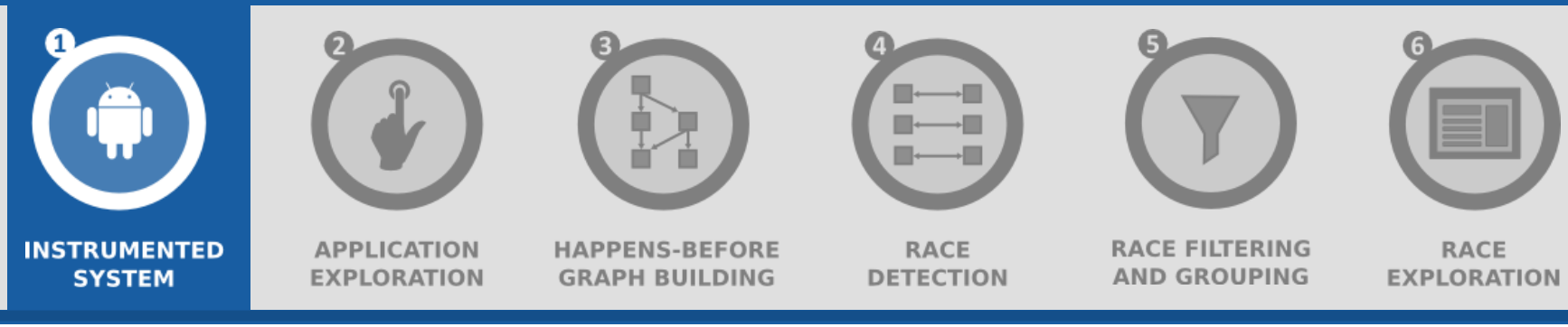


What are the operations capturing essential features of event-driven applications?

```
begin(…)
fork(…)
end(…)
begin(…)
end(…)
```

```
void onCreate() {
  new Thread(…).start();
}
```

```
void downloadData() {
  postDelayed(…, 100);
}
```

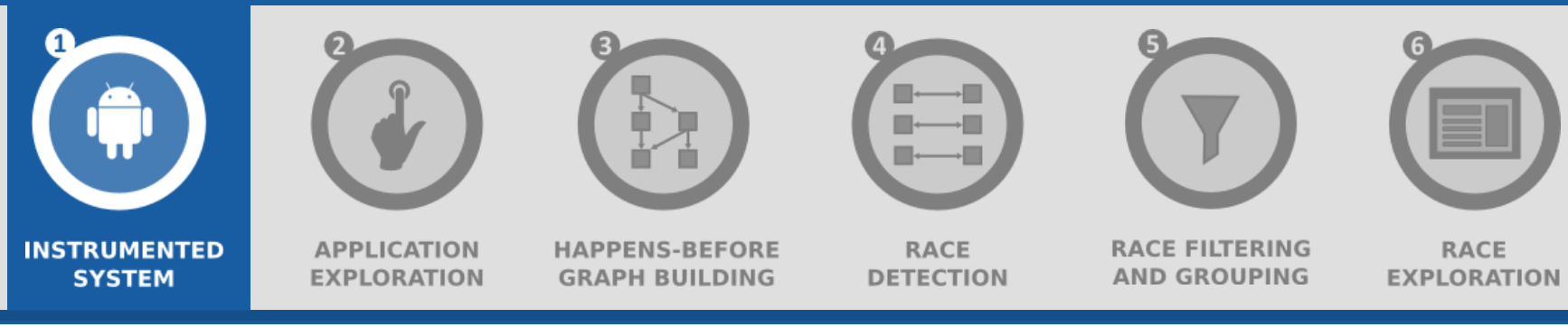


What are the operations capturing essential features of event-driven applications?

```
begin(…)
fork(…)
end(…)
begin(…)
enqueue(…)
end(…)
```

```
void onCreate() {
  new Thread(…).start();
}
```

```
void downloadData() {
  postDelayed(…, 100);
}
```



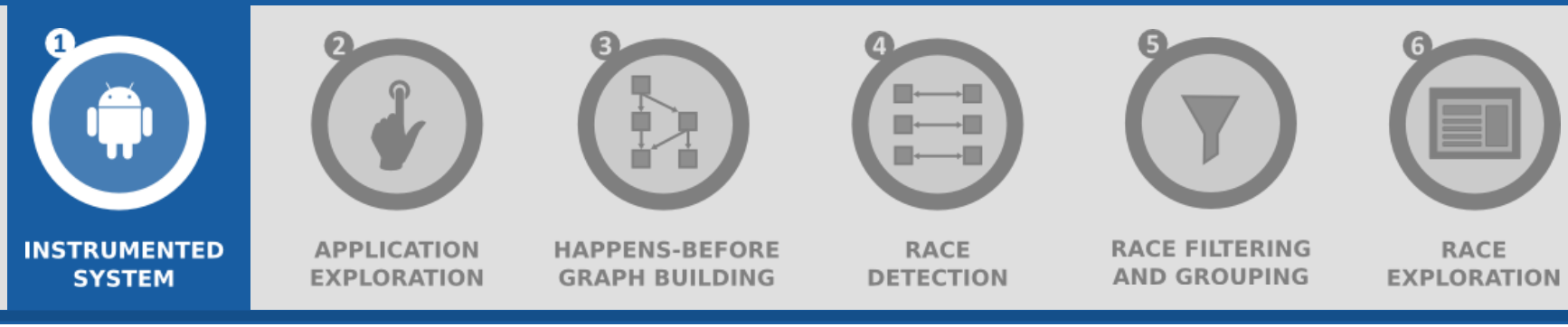
What are the operations capturing essential features of event-driven applications?

mapping of ***all Android APIs*** into 11 operations

```
begin(…)
fork(…)
end(…)
begin(…)
enqueue(…)
end(…)
```

```
void onCreate() {
  new Thread(…).start();
}
```

```
void downloadData() {
  postDelayed(…, 100);
}
```

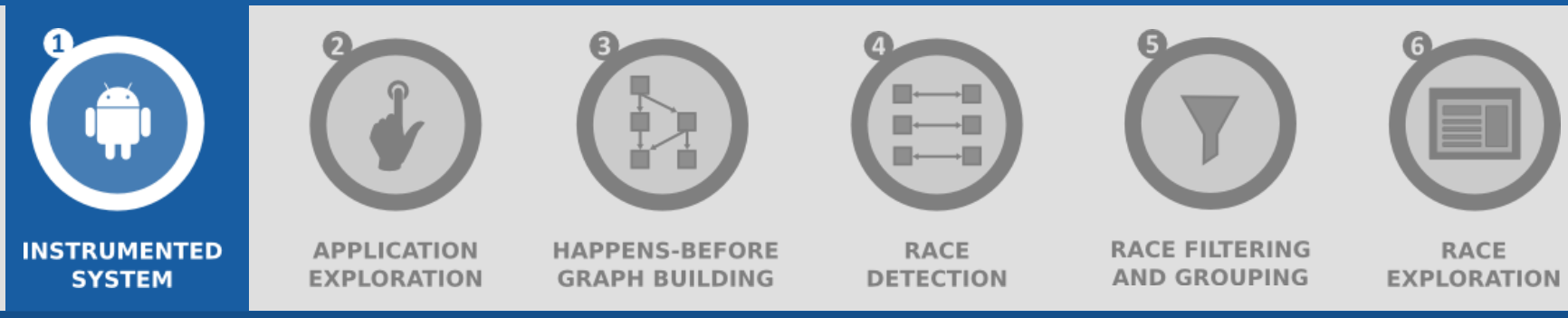


What are the memory locations on which events can interfere?

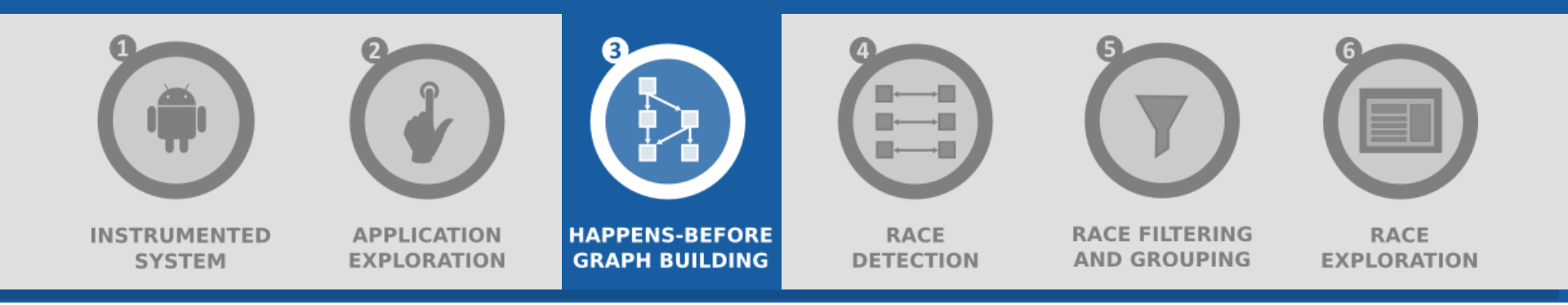
- ➔ Object and Class fields
- ➔ High level operations

```
void onPostExecute() {
  mDatabase.insert();
}
```

```
READ 23867 mDbHelper
WRITE TABLE:Users ID:2
WRITE TABLE:Users ID:3
...
```



Instrumentation of
both application and framework
with overhead only ***~300%***



What is the event happens-before?

Fork-join model

Threads

```
fork()
join()
wait()
notify()
lock()
unlock()
```

+

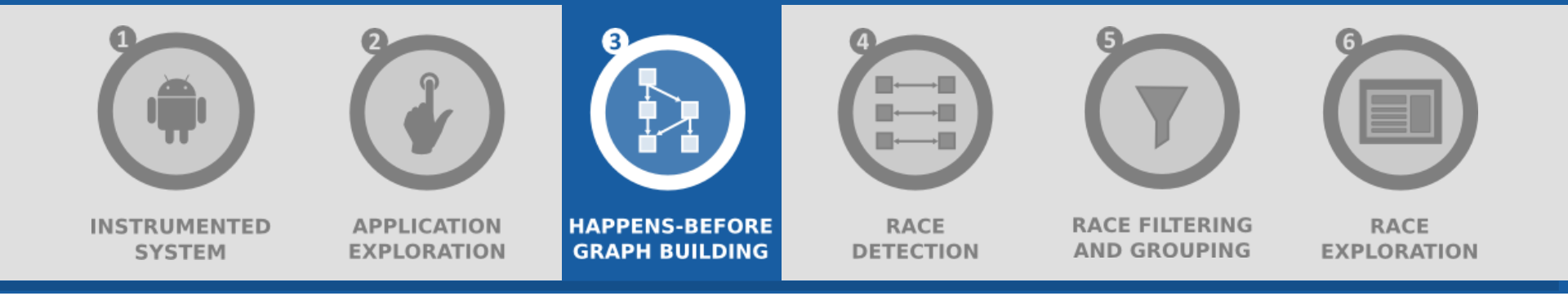
Rich Event-Based model

Message Queue

```
postDelayed(delay)
postAtTime(time)
postFront()
postIdle()
remove()
```

$$\frac{\alpha = fork(\_, tid) \qquad \beta = thread\_init(tid)}{\alpha \prec \beta}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta \prec \gamma \\ looper_{ord}(\eta, \gamma) \qquad \eta.dispatcher = \gamma.dispatcher \\ \eta.type \in \{Delayed, AtTime, Front, Idle\} \\ \gamma.type \in \{Delayed, AtTime, Idle\} \\ \eta.barrier \vee \neg\gamma.barrier\end{array}}{\alpha \prec \beta}$$



$$\frac{\alpha.event \neq \bot \qquad \alpha.event = \beta.event \qquad \alpha <_\pi \beta}{\alpha \prec \beta} \text{ (EventOp)}$$

$$\frac{\alpha = register(c) \qquad \beta = invoke(c, \_)}{\alpha \prec \beta} \text{ (CallbackReg\#1)}$$

$$\frac{\alpha = register(c) \qquad \beta = unregister(c)}{\alpha \prec \beta} \text{ (CallbackReg\#2)}$$

$$\frac{\alpha = invoke(c, \_) \qquad \beta = unregister(c)}{\alpha \prec \beta} \text{ (CallbackUnreg)}$$

$$\frac{\alpha, \beta = invoke(c, sync) \wedge \alpha <_\pi \beta}{\alpha \prec \beta} \text{ (CallbackInv)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta \prec \gamma \\ looper_{ord}(\eta, \gamma) \qquad \eta.dispatcher = \gamma.dispatcher \\ \eta.type \in \{Delayed, AtTime, Front, Idle\} \\ \gamma.type \in \{Delayed, AtTime, Idle\} \\ \eta.barrier \vee \neg\gamma.barrier\end{array}}{\alpha \prec \beta} \text{ (MsgBegin\#1)}$$

$$\frac{\begin{array}{c}\alpha = end(mid') \\ \gamma = enqueue(mid') \qquad \beta = begin(mid) \\ \eta = enqueue(mid) \qquad \eta \prec \gamma \prec \beta \\ \eta.dispatcher = \gamma.dispatcher \qquad \gamma.type = Front \\ \eta.type \in \{Delayed, AtTime, Front, Idle\} \\ \gamma.barrier \vee \neg\eta.barrier\end{array}}{\alpha \prec \beta} \text{ (MsgBegin\#2)}$$

$$\frac{\begin{array}{c}\eta = begin(mid) \qquad \alpha = end(mid) \\ \beta = begin(mid') \qquad \gamma = end(mid') \\ \eta \prec \gamma \qquad \alpha.dispatcher = \beta.dispatcher \\ \alpha.dispatcher_{type} = \texttt{Looper}\end{array}}{\alpha \prec \beta} \text{ (LooperAtomic)}$$

$$\frac{\alpha = enqueue(mid) \qquad \beta = begin(mid)}{\alpha \prec \beta} \text{ (MsgEnqueue)}$$

$$\frac{\alpha = fork(\_, tid) \qquad \beta = thread\_init(tid)}{\alpha \prec \beta} \text{ (ThreadFork)}$$

$$\frac{\alpha = thread\_exit(tid) \qquad \beta = join(\_, tid)}{\alpha \prec \beta} \text{ (ThreadJoin)}$$

$$\frac{\alpha = thread\_init(tid) \qquad \beta \in \pi_{tid} \setminus \alpha}{\alpha \prec \beta} \text{ (ThreadInit)}$$

$$\frac{\beta = thread\_exit(tid) \qquad \alpha \in \pi_{tid} \setminus \beta}{\alpha \prec \beta} \text{ (ThreadExit)}$$

$$\frac{\alpha = notify(id) \qquad \beta = wait(id)}{\alpha \prec \beta} \text{ (NotifyWait)}$$

$$\frac{\begin{array}{c}\alpha = begin(mid) \\ \beta = remove(mid) \qquad \gamma = enqueue(mid) \qquad \gamma \prec \beta\end{array}}{\alpha \prec \beta} \text{ (MsgRemove)}$$

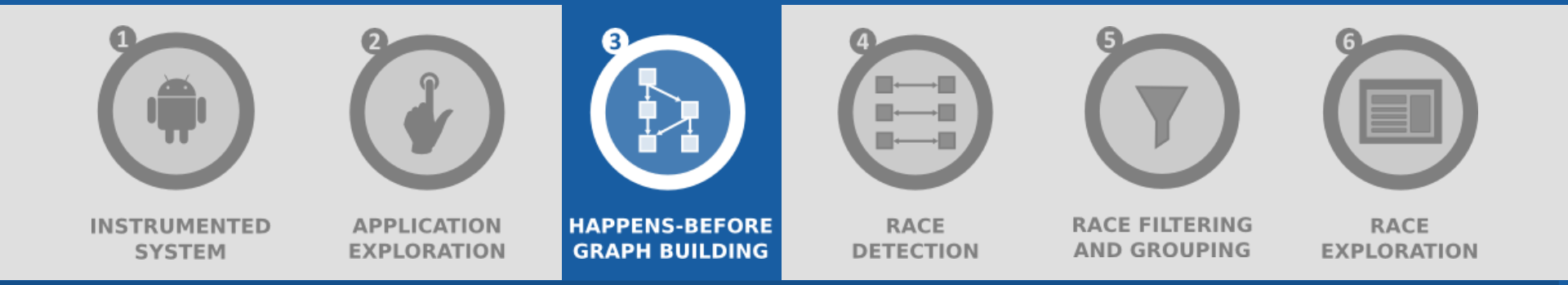
$$\frac{\alpha = end(mid) \qquad \beta = blocking\_enqueue\_end(mid)}{\alpha \prec \beta} \text{ (MsgBlocking)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) <_\pi \beta = begin(mid') \\ \alpha.type, \beta.type \in \{Input, Display\} \\ \alpha.type = \beta.type \qquad \alpha.dispatcher = \beta.dispatcher\end{array}}{\alpha \prec \beta} \text{ (Native)}$$

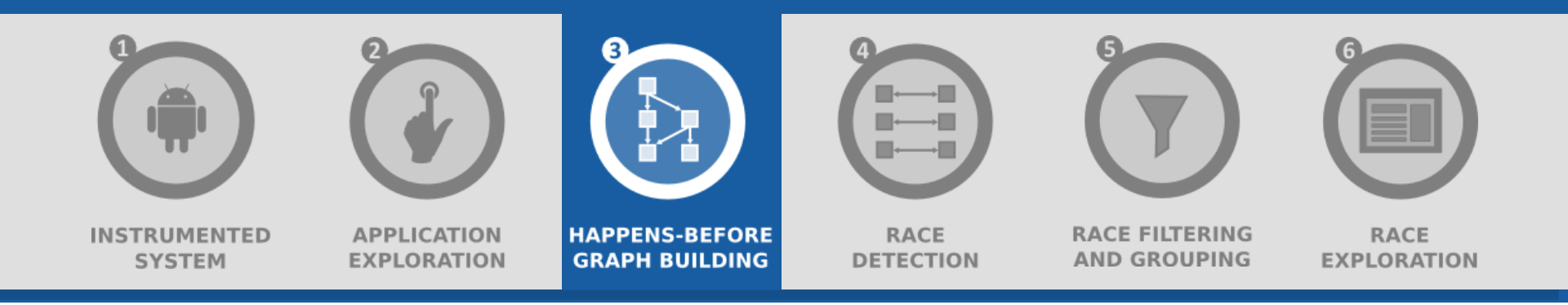
$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta <_\pi \gamma \\ \eta.sync \vee \neg\gamma.sync \qquad \eta.type = \gamma.type = IPC \\ \eta.pid = \gamma.pid \qquad \eta.dispatcher = \gamma.dispatcher\end{array}}{\alpha \prec \beta} \text{ (IpcHandle)}$$

$$\frac{\alpha, \beta \in \pi_{tid} \qquad \alpha <_\pi \beta \qquad \alpha.event = \beta.event = \bot}{\alpha \prec \beta} \text{ (ThreadOp)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta \prec \gamma \\ \eta.type = \gamma.type = IPC \qquad \neg\eta.sync \qquad \neg\gamma.sync \\ \eta.tid = \gamma.tid \qquad \eta.dispatcher = \gamma.dispatcher\end{array}}{\alpha \prec \beta} \text{ (IpcAsync)}$$

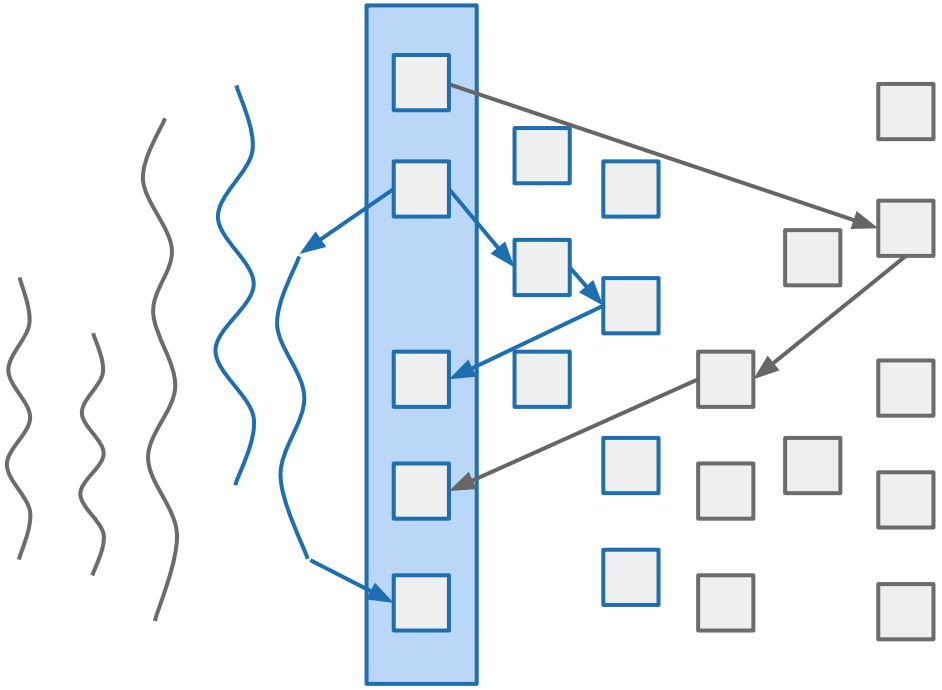


Thorough and precise
happens-before model which captures Android concurrency

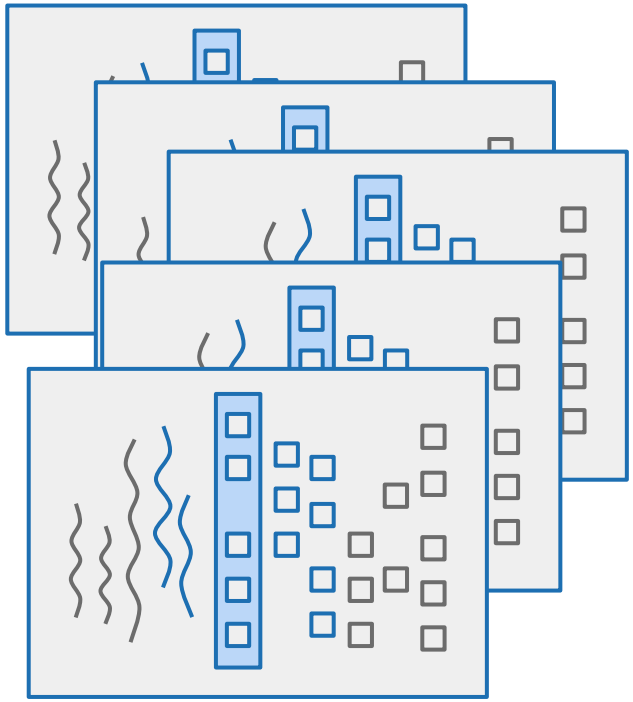


Standard Threads

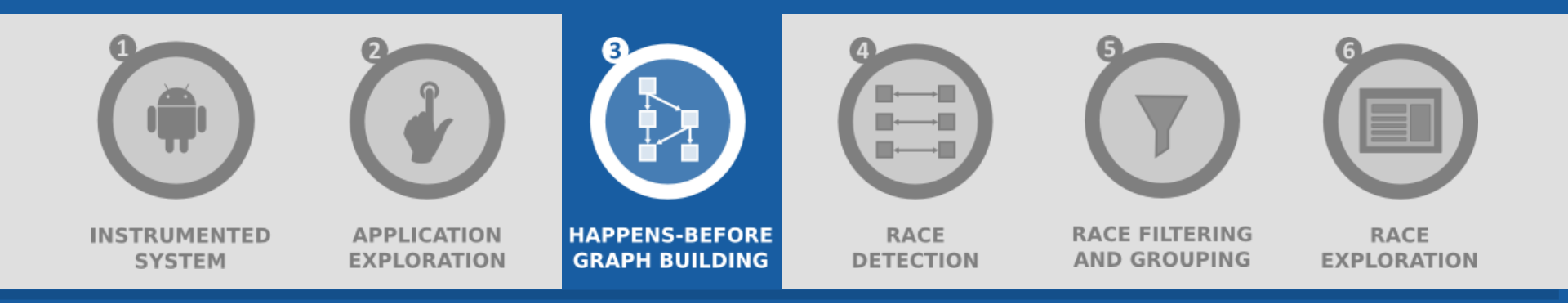
Event Threads



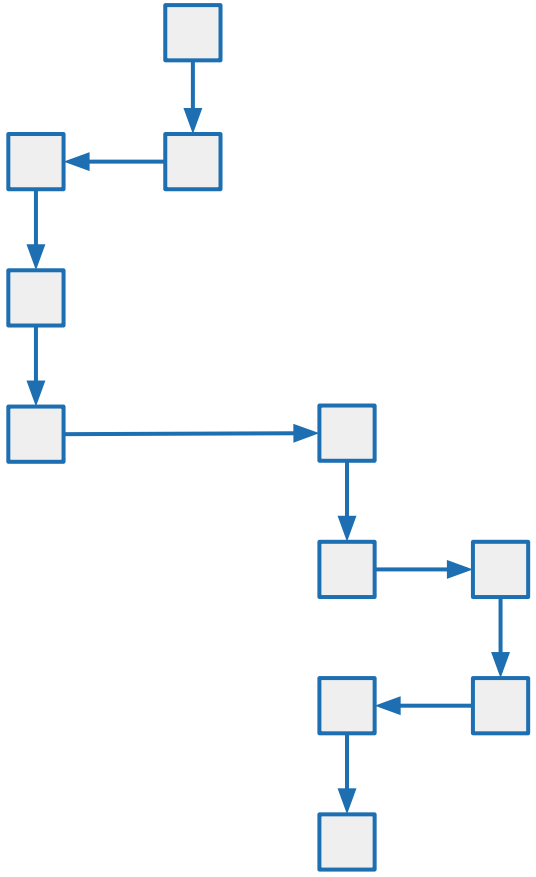
Application



Framework &
Other Applications



How to efficiently build the happens-before graph?



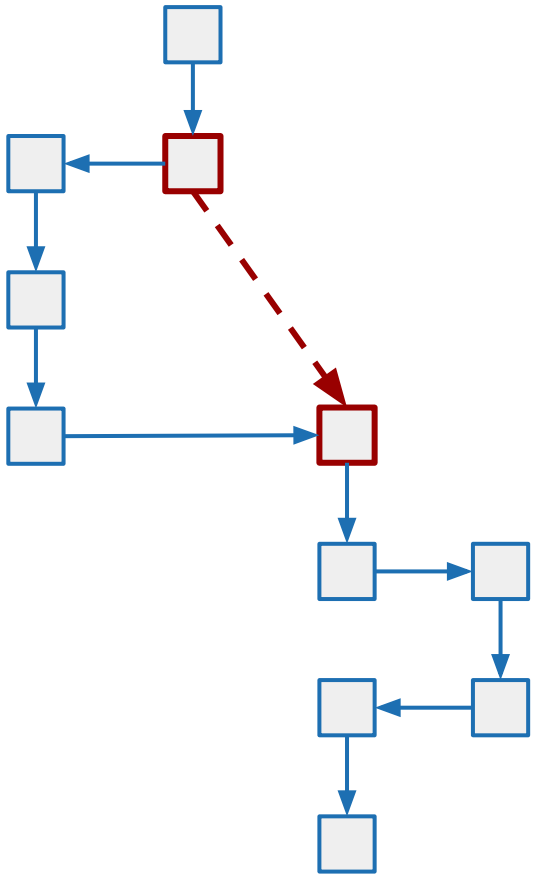
Key Scalability Ingredients:

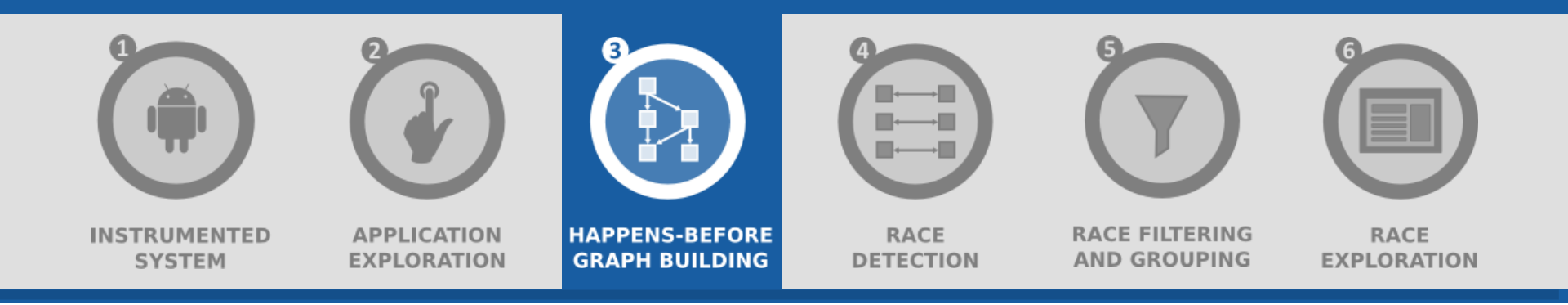
- ➔ Efficient Rule Matching
- ➔ Sparse Graph
- ➔ Fast connectivity queries
- ➔ Evaluating rules only once
- ➔ Trace optimization
- ➔ Graph traversal pruning

How to efficiently build the happens-before graph?

Key Scalability Ingredients:

- ➔ Efficient Rule Matching
- ➔ Sparse Graph
- ➔ **Fast connectivity queries**
- ➔ Evaluating rules only once
- ➔ Trace optimization
- ➔ Graph traversal pruning

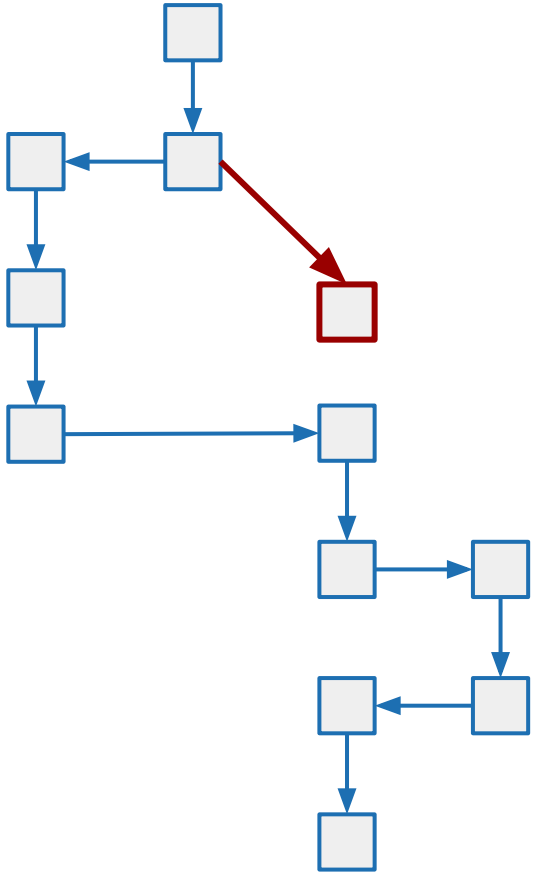




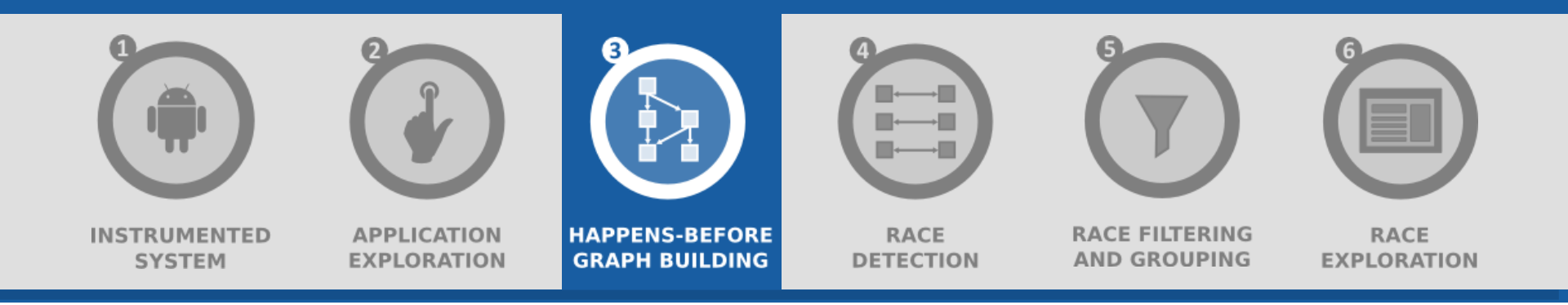
How to efficiently build the happens-before graph?

Key Scalability Ingredients:

- ➔ Efficient Rule Matching
- ➔ Sparse Graph
- ➔ Fast connectivity queries
- ➔ **Evaluating rules only once**
- ➔ Trace optimization
- ➔ Graph traversal pruning



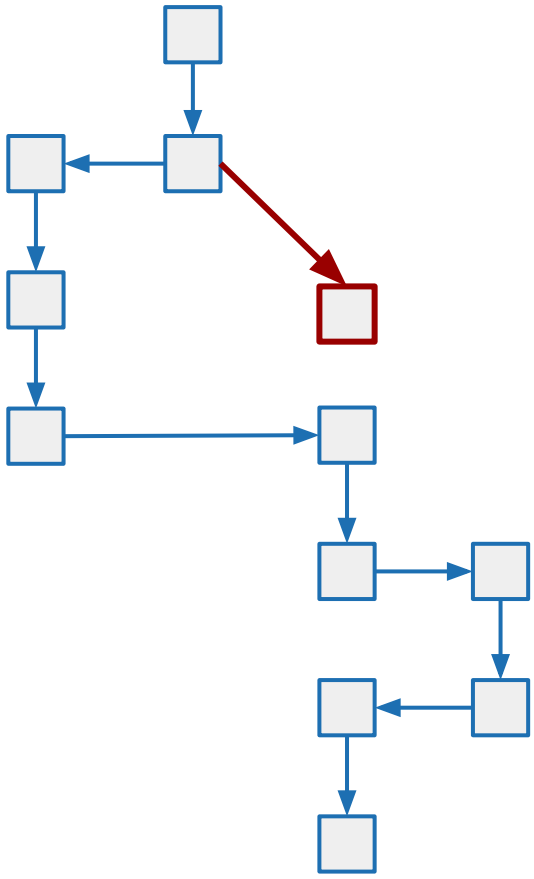
```
EventOp
CallbackReg#1
CallbackReg#2
CallbackUnreg
CallbackInv
MsgBegin#1
MsgBegin#2
LooperAtomic
MsgEnqueue
ThreadFork
ThreadJoin
ThreadInit
ThreadExit
NotifyWait
MsgRemove
MsgBlocking
Native
IpcHandle
ThreadOp
IpcAsync
```



How to efficiently build the happens-before graph?

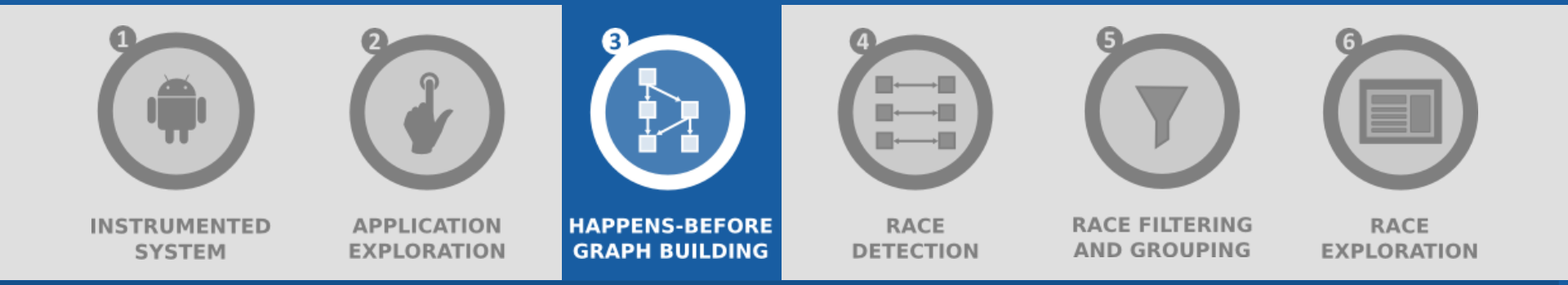
Key Scalability Ingredients:

- ➔ Efficient Rule Matching
- ➔ Sparse Graph
- ➔ Fast connectivity queries
- ➔ **Evaluating rules only once**
- ➔ Trace optimization
- ➔ Graph traversal pruning



re-evaluate rules

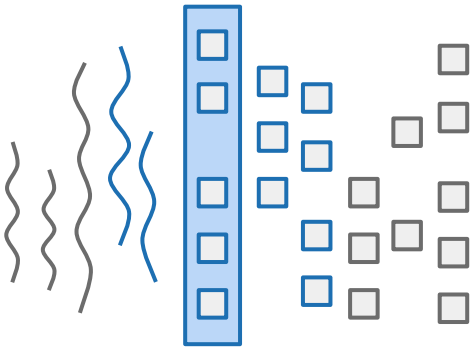
```
EventOp
CallbackReg#1
CallbackReg#2
CallbackUnreg
CallbackInv
MsgBegin#1
MsgBegin#2
LooperAtomic
MsgEnqueue
ThreadFork
ThreadJoin
ThreadInit
ThreadExit
NotifyWait
MsgRemove
MsgBlocking
Native
IpcHandle
ThreadOp
IpcAsync
```



first scalable algorithm for building rich happens-before graph for whole Android system

How to make scalable race detection in event-based setting?

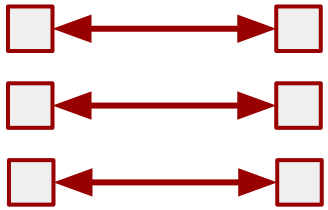
happens-before graph
+
memory locations



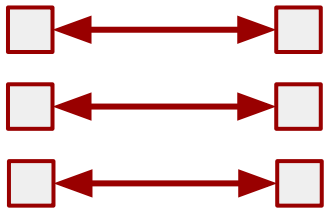
Run state-of-the-art race detector for event driven setting

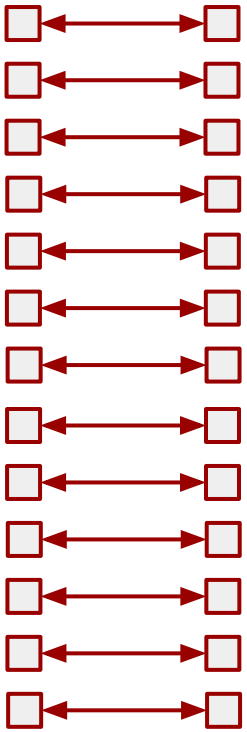
Raychev et. al. [OOPSLA'13]

atomicity violations

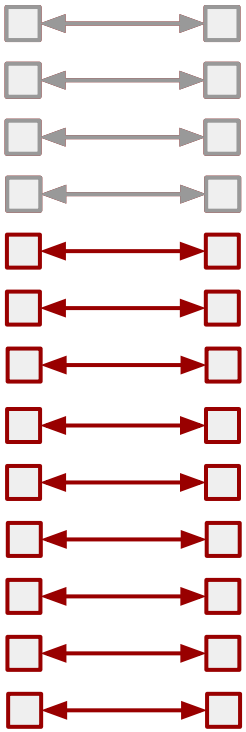


ordering violations





1328

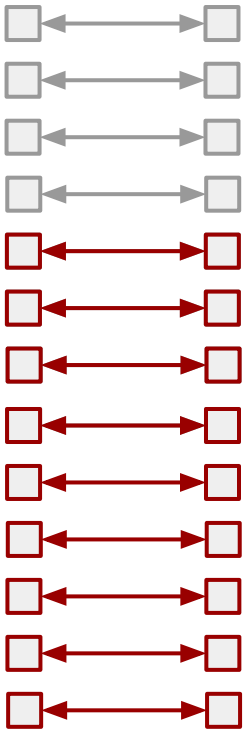


1328

harmless races

```
mCount++;
```

```
mCount++;
```



1328

harmless races

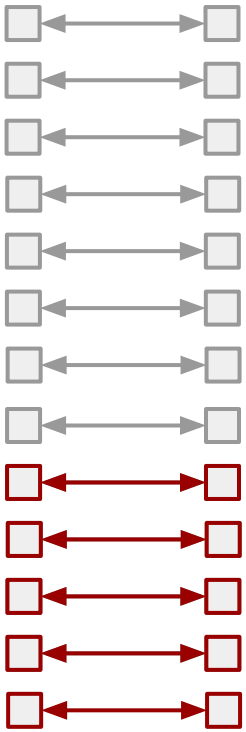
```
mCount++;
```

```
mCount++;
```

commutative races

```
mMap[1] = "OOPSLA";
```

```
mMap[2] = "SPLASH";
```



1328

harmless races

```
mCount++;
```

```
mCount++;
```

commutative races

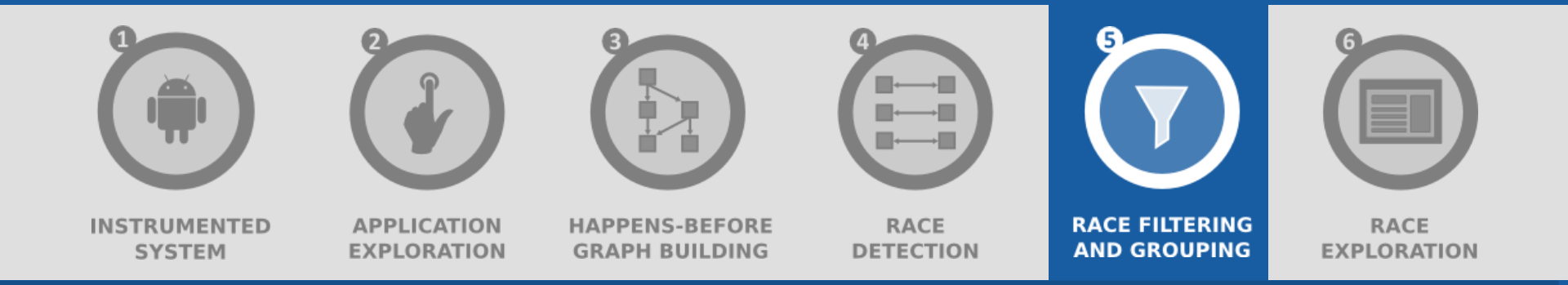
```
mMap[1] = "OOPSLA";
```

```
mMap[2] = "SPLASH";
```

synchronization races

```
if (mActive) return;
```

```
mActive = true;
```



100x reduction of reported concurrency conflicts (1328 → 13)

Manual evaluation

Analysis Scalability

354 Play Store Applications

#events	~28 000
#happens-before operations	~590 000
#memory locations	~5 140 000
analysis runtime	**70s - 130s**

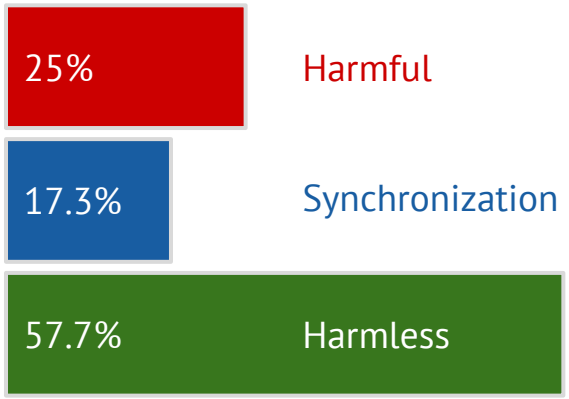
10 minutes application usage

Manual evaluation

Usability

8 Play Store Applications

Main Application Thread
(10625 races → 104 reports)



Other Threads
(2804 races → 135 reports)

Related Work

CAFA [Hsiao et.al, PLDI'14] & DroidRacer [Maiya et.al PLDI'14]

Analysis Scalability

Metric	Our Work	CAFA & DroidRacer
Exploration time	**10 min**	10 - 30 s
Analysis time	**70 - 130 s**	30 min - 1 day

More precise happens-before model:

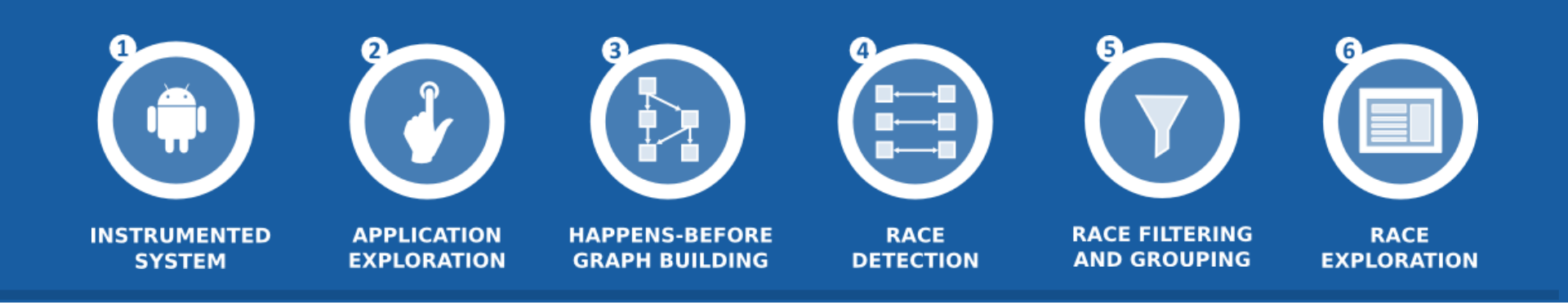
- ➔ Complete handling of message types
- ➔ Message removal
- ➔ Effect of barriers
- ➔ More precise IPC communication

Error Coverage & Usability

➔ [CAFA]
Null pointer dereference
+ usability
- missed bugs

➔ [DroidRacer]
Application code without filtering
+ better bug coverage
- poor usability (too many races reported)

➔ **[our work]**
User + Framework code with filtering
+ complete bug coverage
+ usability (100x report reduction)

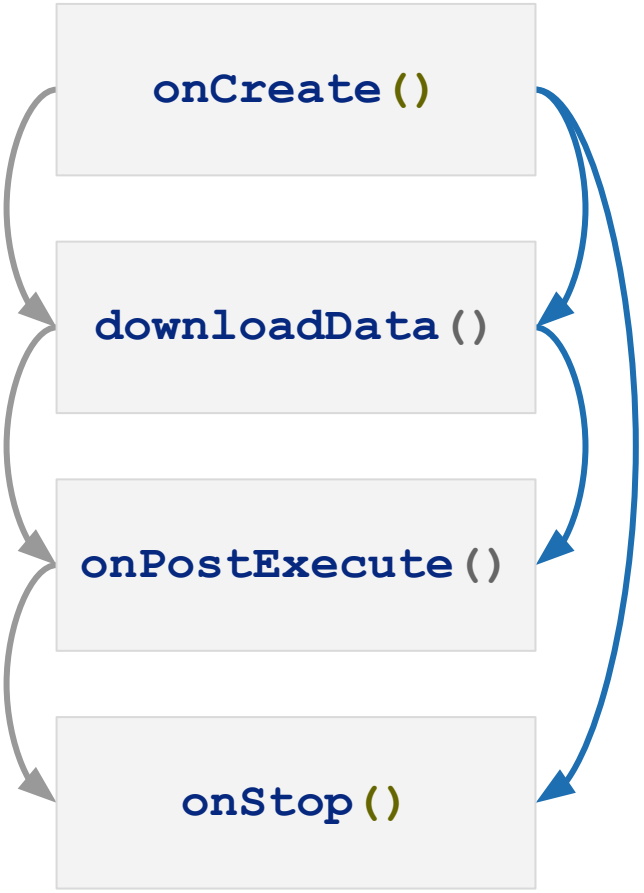


Trace Order

Captures ***single trace*** observed during the dynamic analysis

Filtering & Grouping

order of magnitude reduction of reported conflicts



Happens-before Order

Defines ***set of traces*** that are possible interleavings of the original trace

Concurrency Interference

unordered conflicting accesses to the same memory location