R2: An Application-Level Kernel for Record and Replay

Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu,
Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang

Microsoft Research Asia, Tsinghua University, MIT CSAIL

What & Why?

- What
  - ***R***ecord one run and ***R***eplay it for debugging
- Why
  - Difficult to reproduce some bugs with re-executing
    - E.g. with specific network message orders
  - Hard to apply comprehensive analysis with no interference at runtime
    - E.g., predicate checking at every write on variable “state”

Record

D:\> *set R2_MODE=Record*

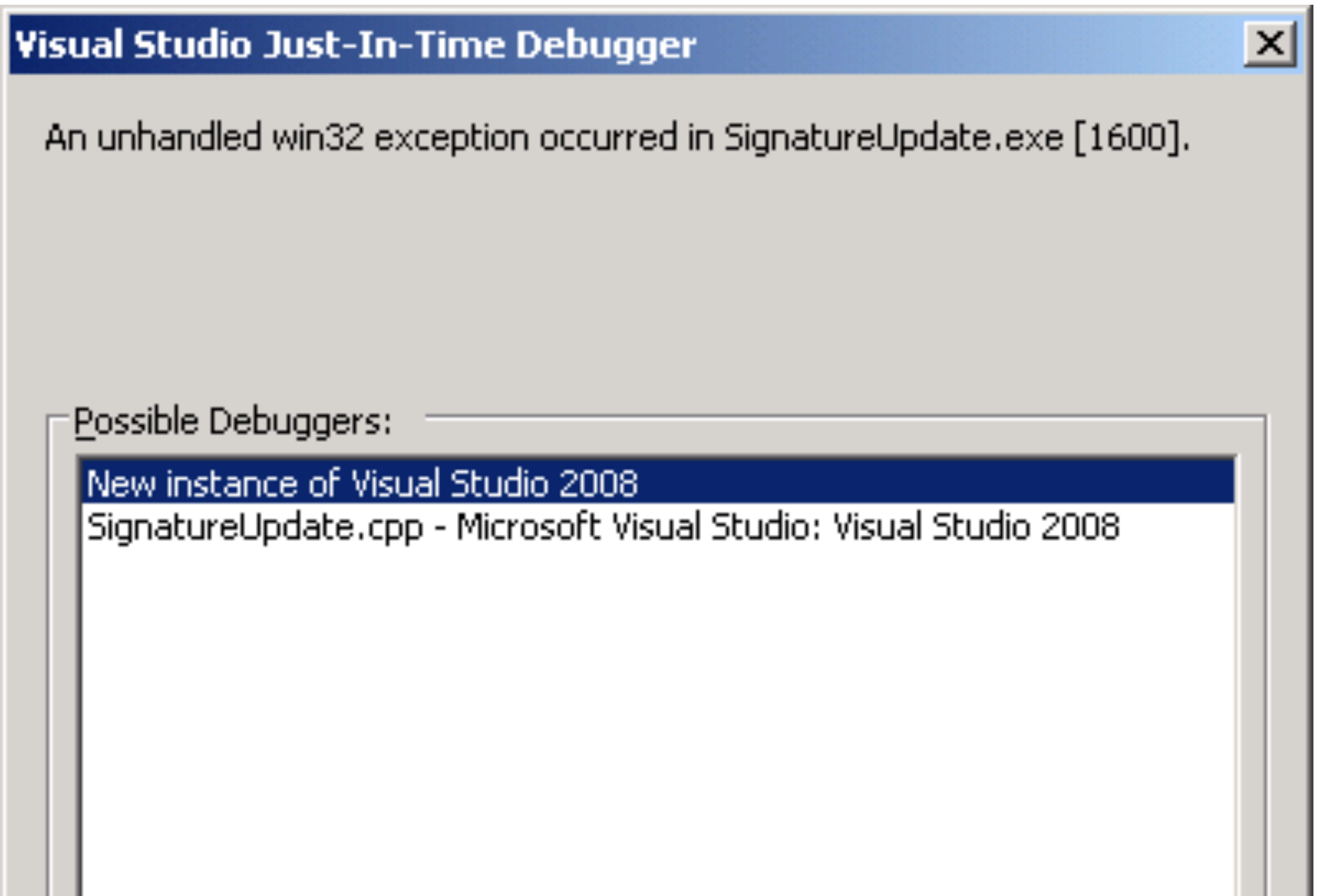
D:\> *R2.exe* signatureUpdate.exe srg-tango0

start check @ 19:12:47.11

… downloading …

4356 entries downloaded

…



Faithful Replay

D:\> *set R2_MODE=Replay*

D:\> *R2.exe* signatureUpdate.exe srg-tango0

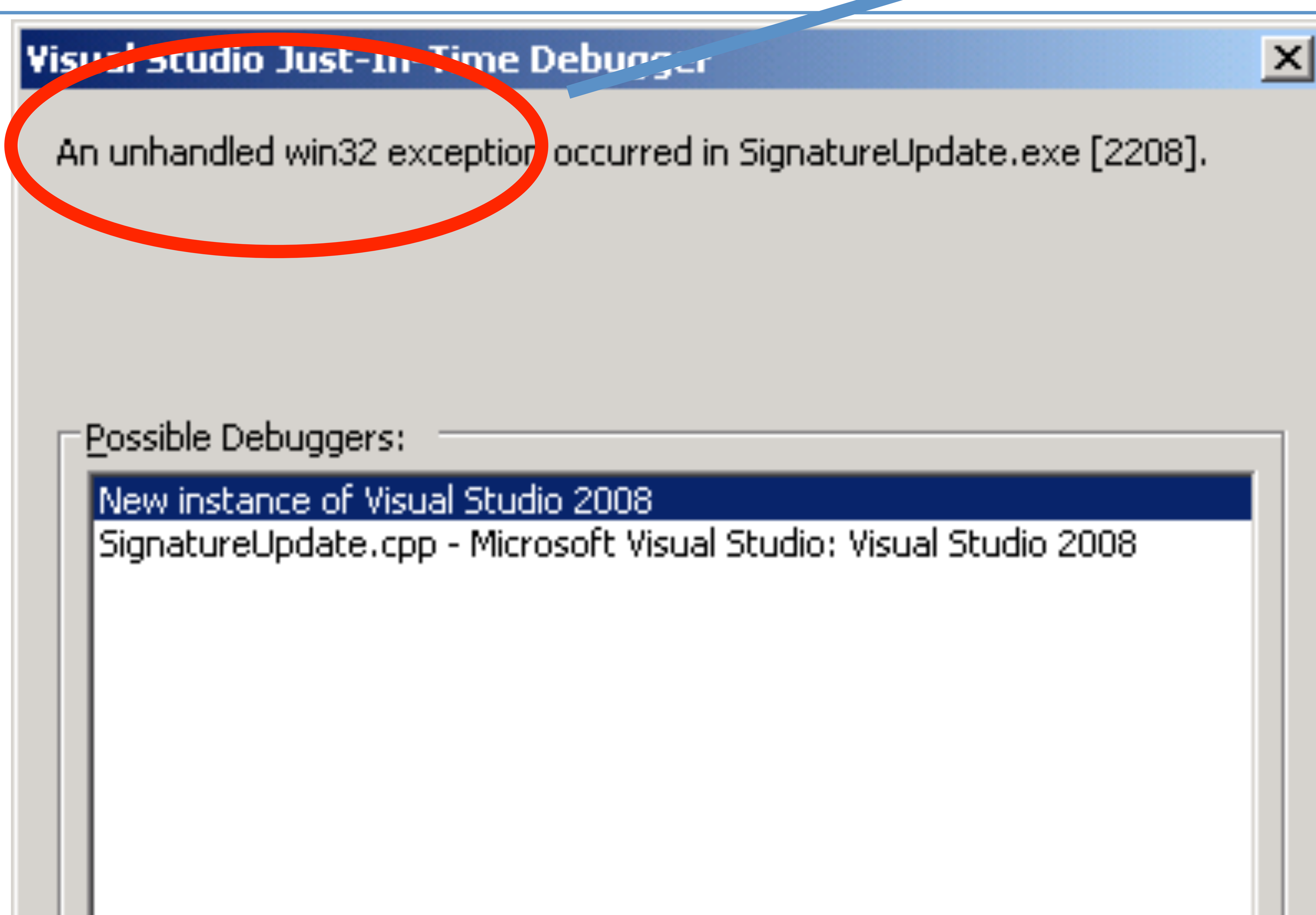
start check @ 19:12:47.11

… downloading …

4356 entries downloaded

…

nondeterminism



State of the art

- Virtual machine approaches
  - George et al, *ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay*, OSDI'02
  - Replay application and the operating system
  - Difficult to deploy
- Library approaches
  - Dennis et al, *Replay debugging for distributed applications*, USENIX'06
  - Replay application only
  - Easy to deploy and lightweight
  - Cannot replay challenging system applications (e.g., with asynchronous I/O)

Library approach

Problems for replaying system applications using library approach

- Cannot record all operations with non-deterministic behavior
  - The code is not a function – *spin lock assembly code*
  - Functions may have unclear semantics – *socketcall, ioctl*

- Can be heavyweight for some applications
  - Log size too large - *read*

- Previous work does not address these problems well

R2's approach

Allow developers to select functions that can be easily & efficiently replayed

Problems	R2	
spin lock assembly code	Find high level function enclosing it	***spin_lock*** (long var) { __asm … }
socketcall	Find functions with clear semantics	***recv***
read	Find function with less I/O	***sqlite_exec***

Overview of R2

Step 1: select a replayable set of functions

spin_lock, recv, sqlite_exec, …

Goal: Capture all nondeterminism

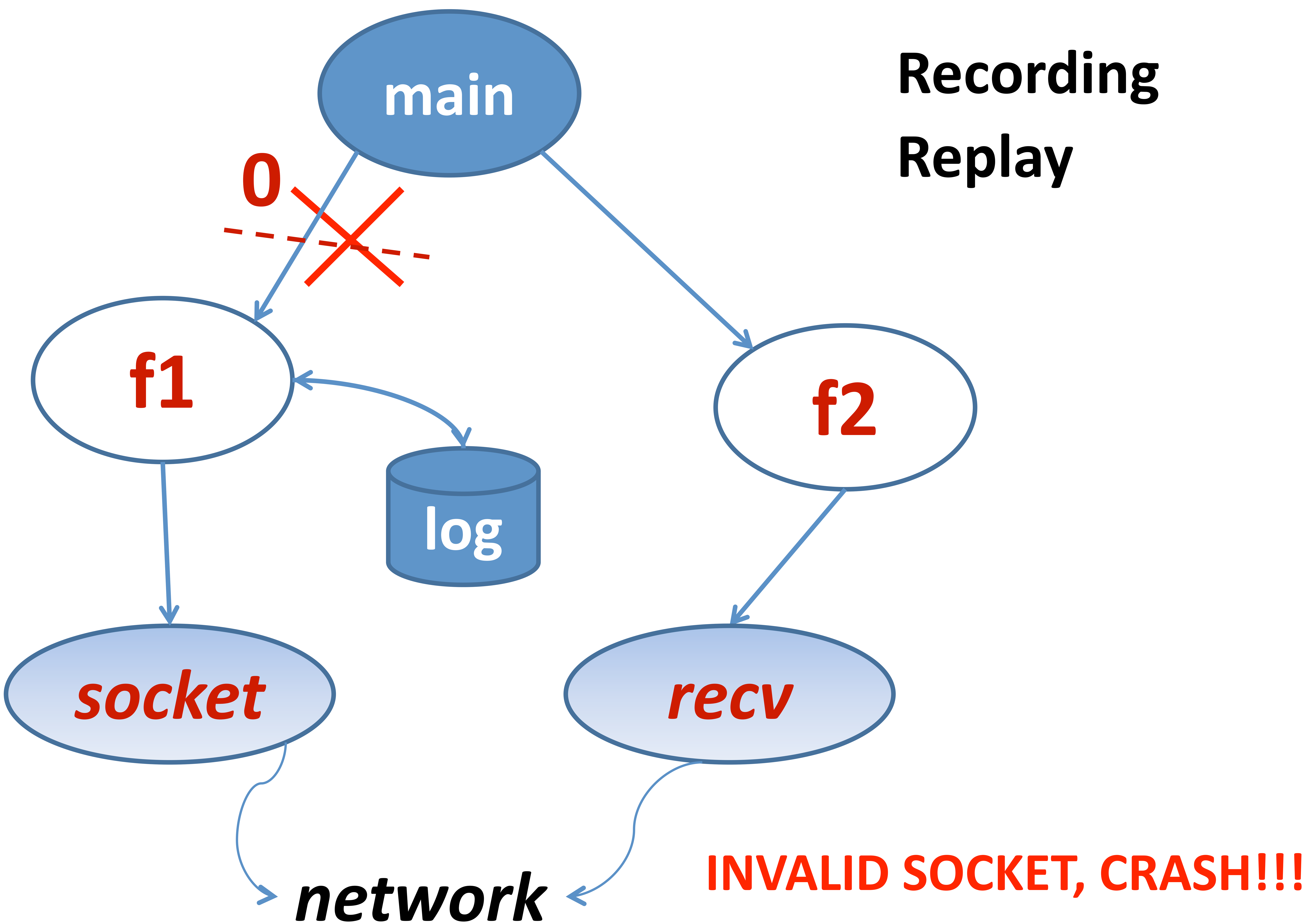
Step 2: annotate functions with R2 supplied keywords so R2 knows what to record

```
int recv ([in] SOCKET socket, [out, bsize(return)] void *buf,
          [in] int nbytes, [in] int flag );
```

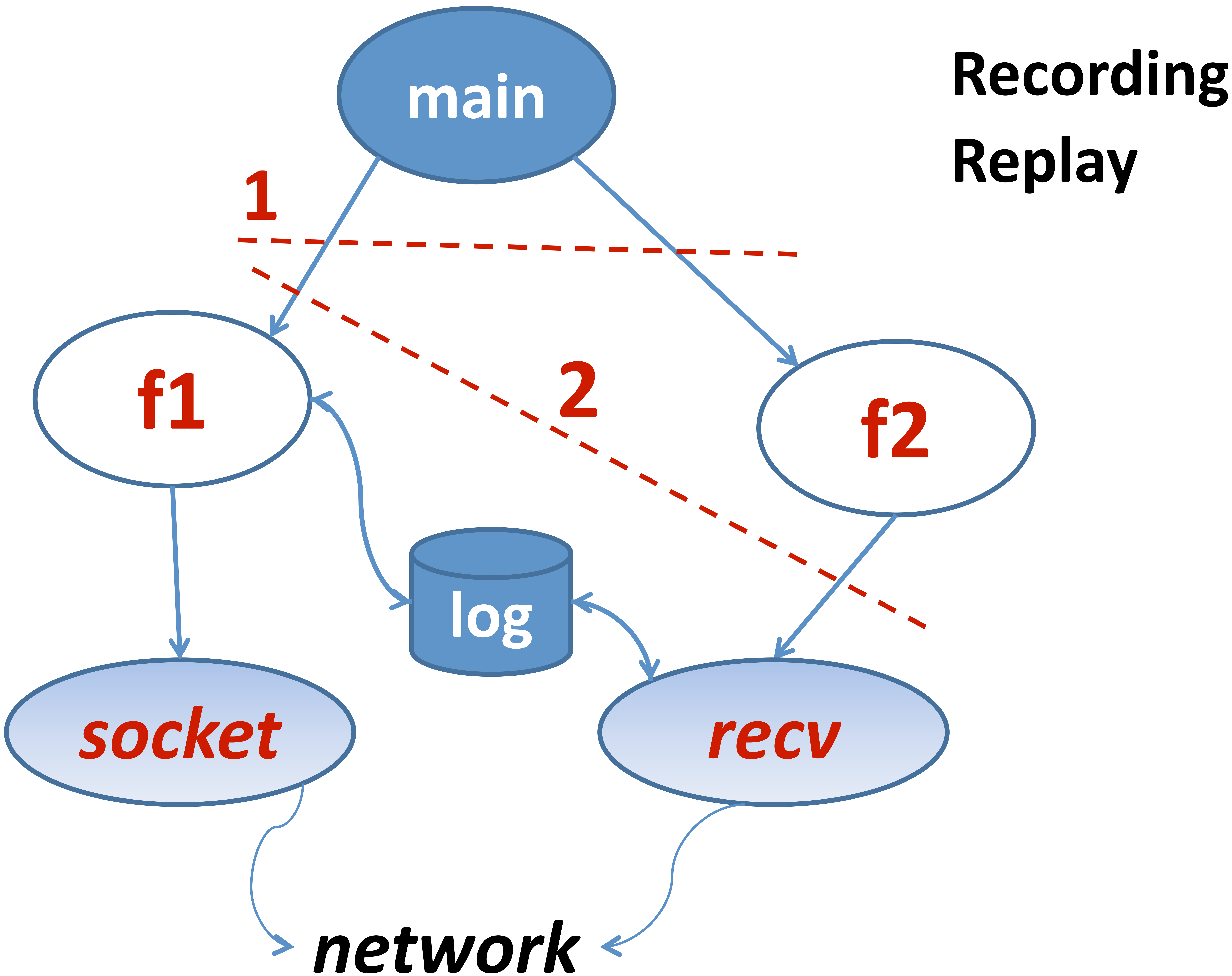
Step 3: R2 generates function stubs for faithful replay automatically

Step 4: compile and run

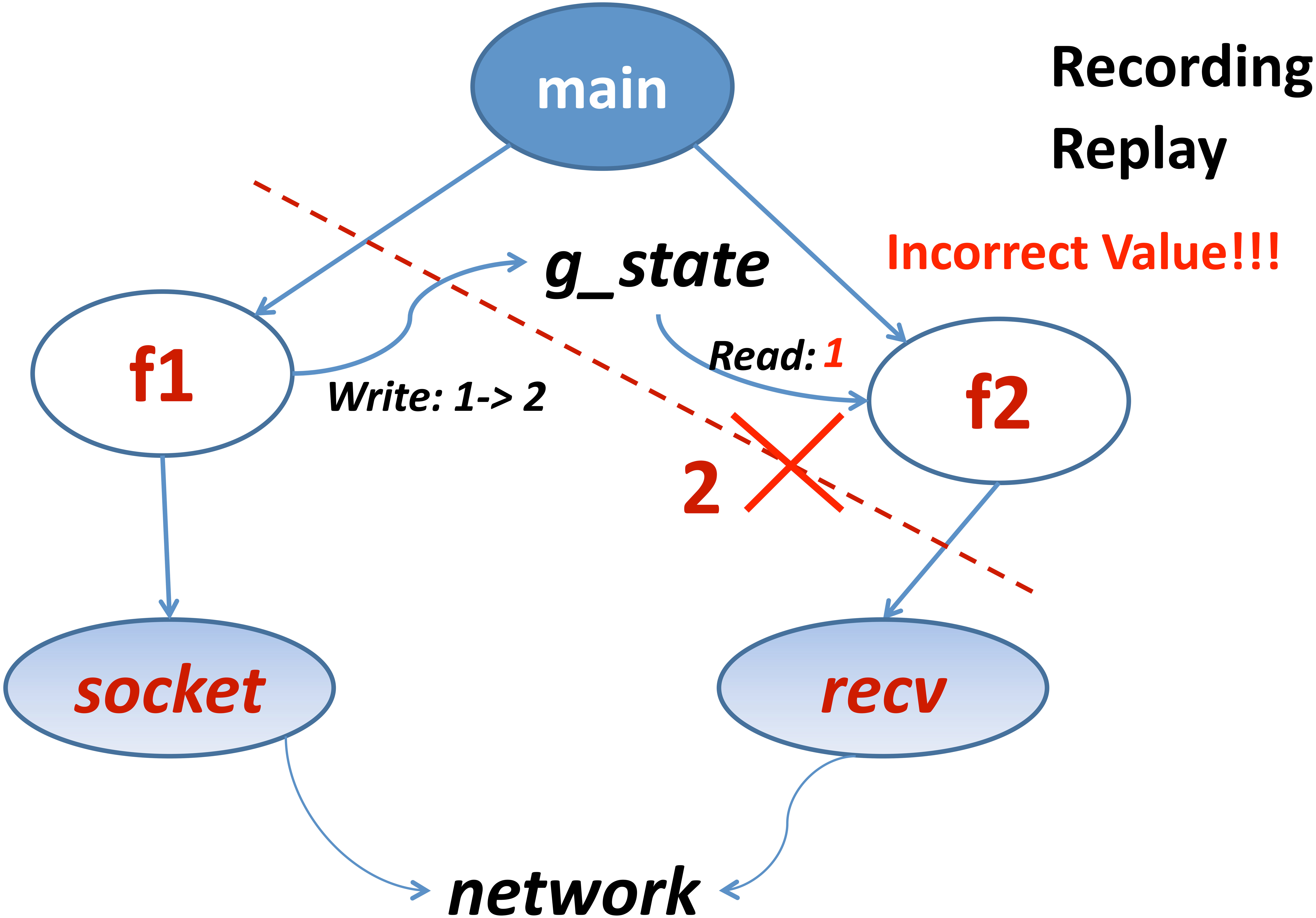
Which functions to select: f1?



A replay interface must be a call graph cut



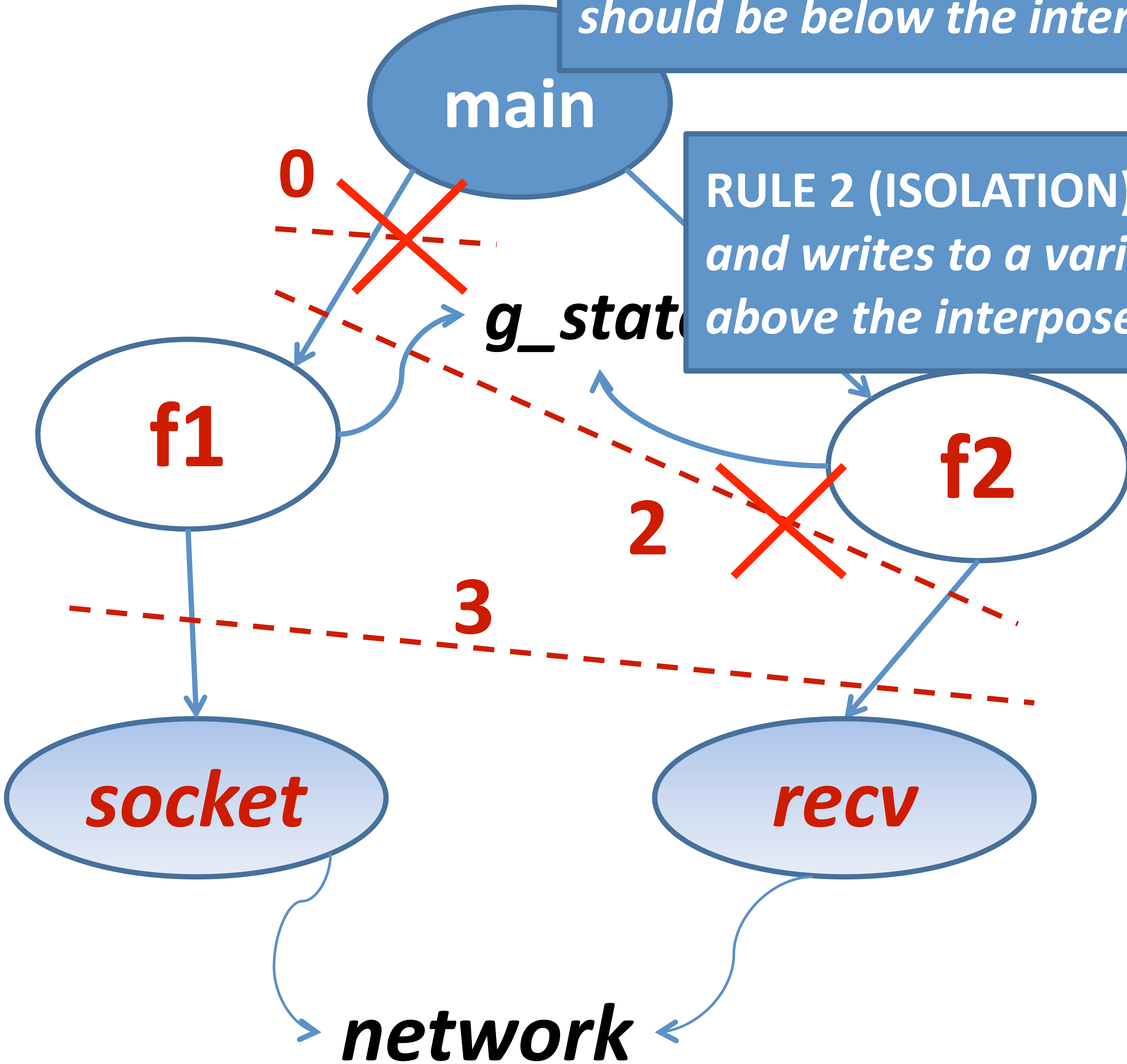
An incorrect cut



Rules for a correct cut (or replay interface)

RULE 1 (NONDETERMINISM) ***Any source of nondeterminism should be below the interposed interface.***

RULE 2 (ISOLATION) ***All instances of unrecorded reads and writes to a variable should be either below or above the interposed interface.***



Find a good cut in practice

- Can be difficult to identify correct cut in complex call graph

- Module APIs are good candidates
  - Encapsulate internal state well

Implementation Challenges

- Deterministic memory footprint
- Execution order in multi-threaded apps
- Reuse intercepted functions
- Identify function side effects
- Threads created by implementation

Deterministic memory footprint

- What?
  - Memory state and its evolution must be the same during recording and replay
- Why?
  - Different memory address may lead to different control flow

Memory problem: a typical network application

Thread 1

```
cb = (struct iocb *)malloc(...);
ReadFileEx(hSocket, cb->buf,
     BUFSIZE,  (OVERLAPPED *)&cb, 0);
```

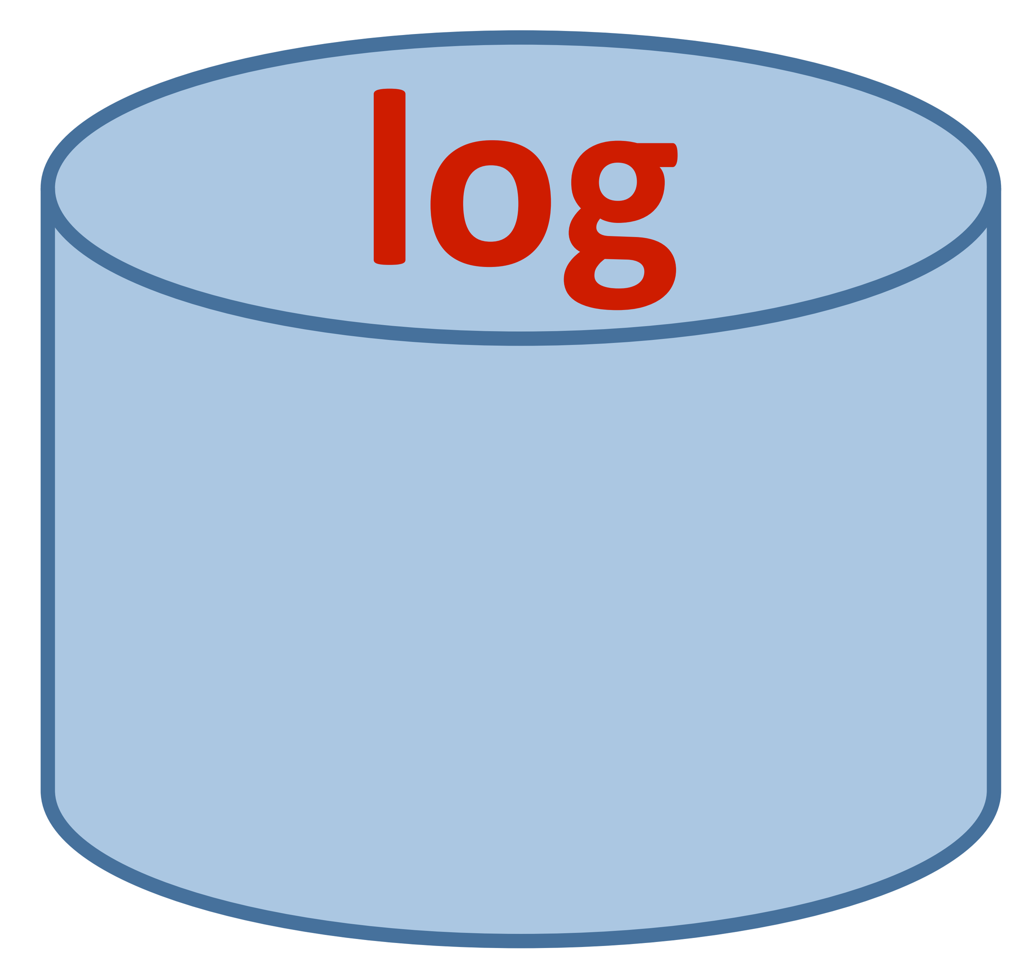
Thread 2

```
GetQueuedCompletionStatus(hPort,
        &size, ..., (OVERLAPPED *)&cb, ...);
void * buf = cb->buf;...
```

INVALID ADDRESS, CRASH!!!

```
struct iocb
{
     OVERLAPPED olp;
     char buf[BUFSIZE];
     ...
}
```

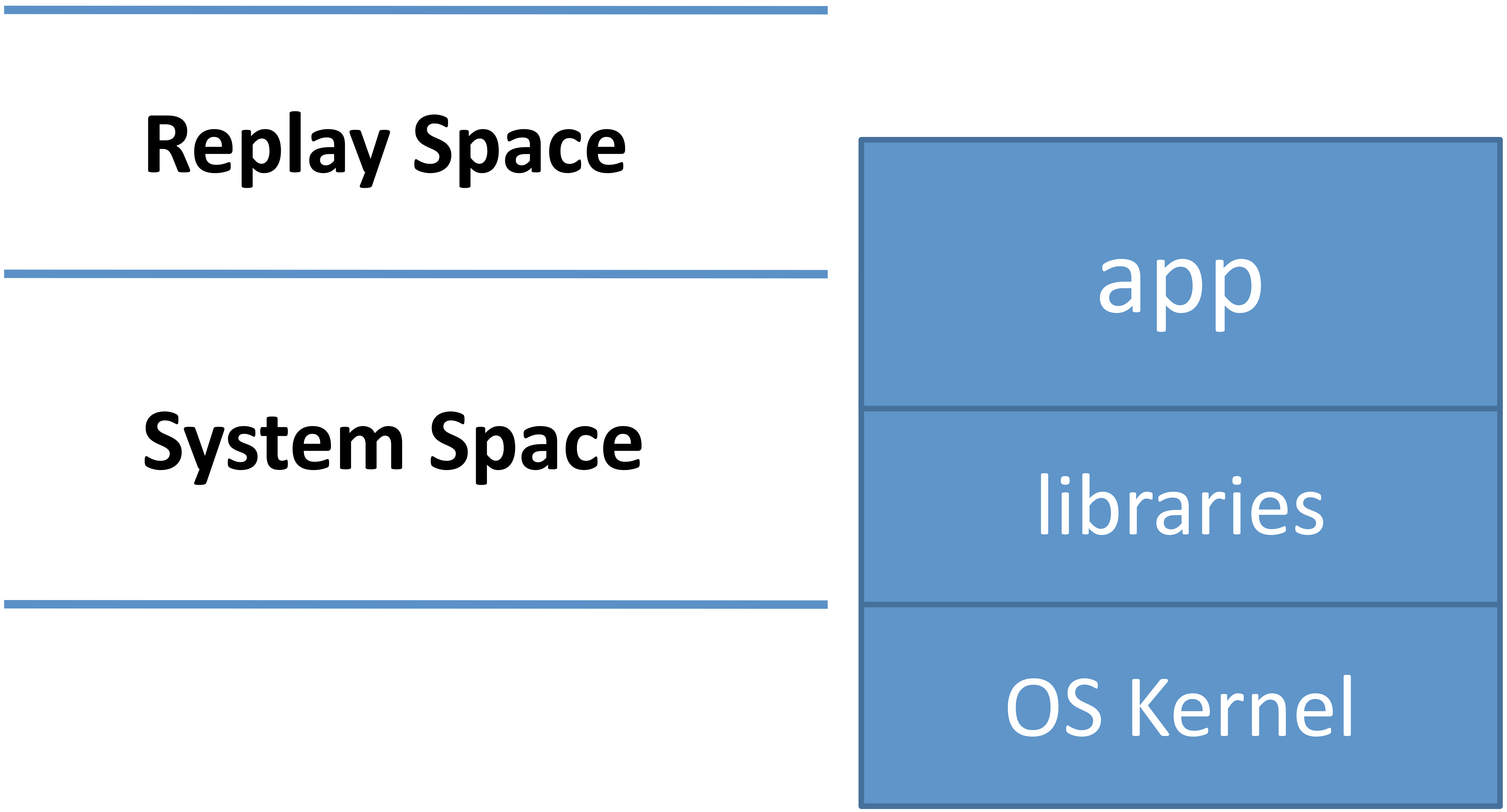
Network Message



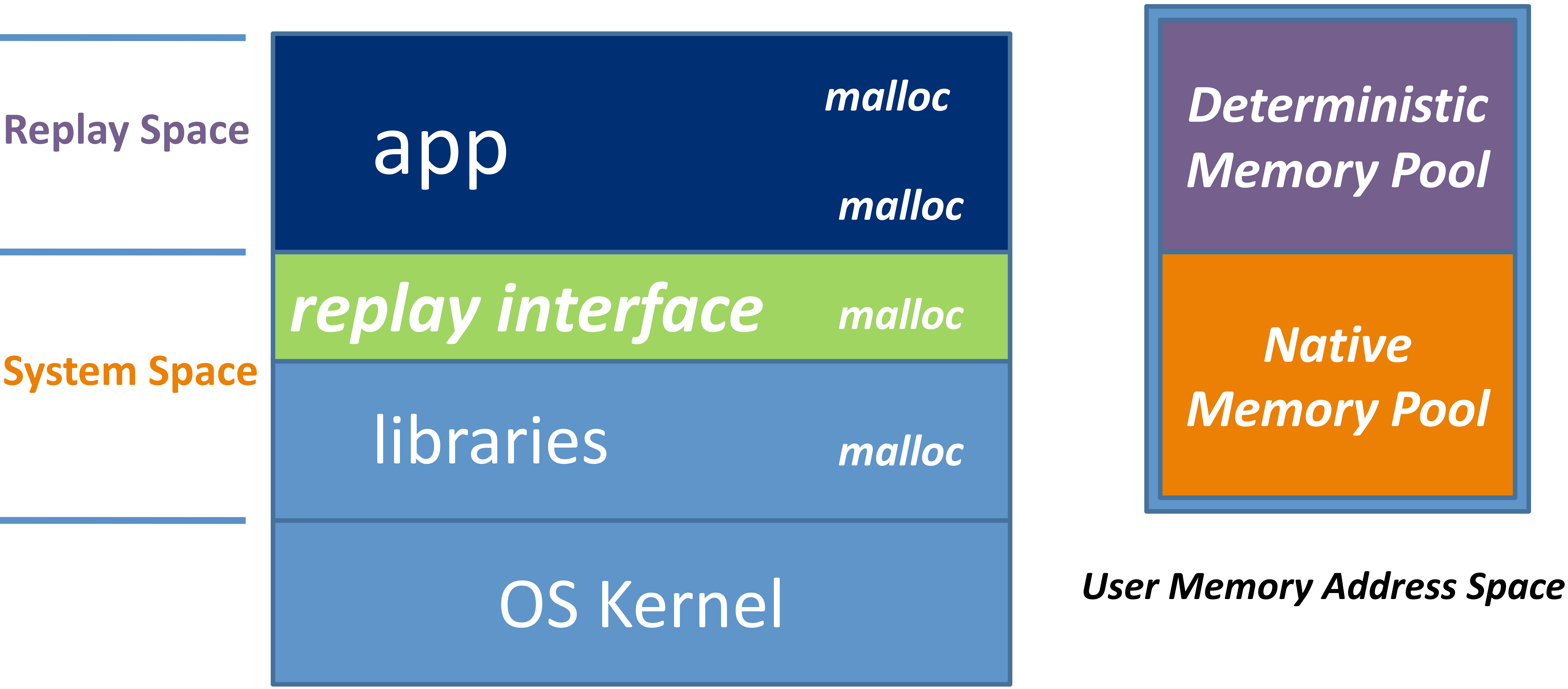
Why different memory addresses?

- The tool and the application are in the same address space
- The tool inherently runs differently during record and replay
- Intercepted functions are not executed during replay
  - Missed memory requests inside the functions
  - Modules not loaded during replay

Isolation using space split

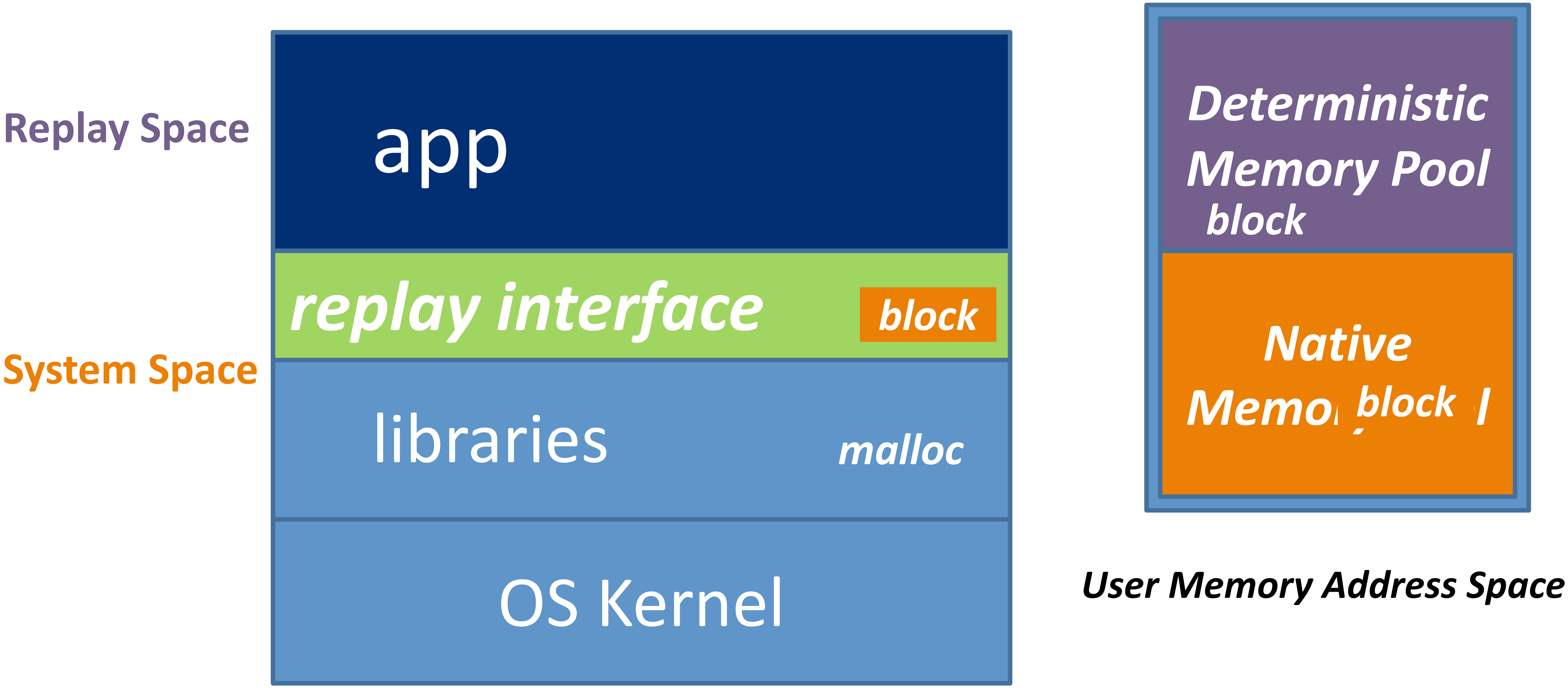


Memory Isolation



Handle data transfer across interface

char* getcwd (NULL, 0);



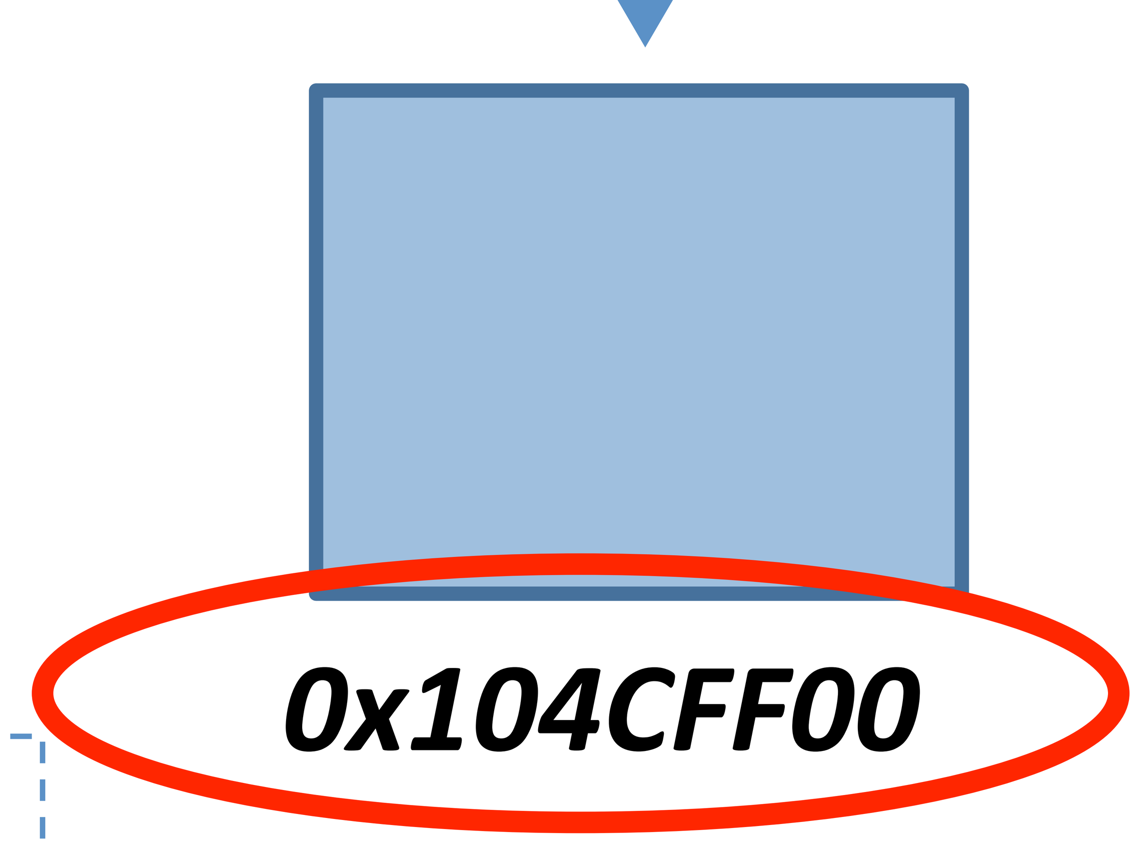
Memory footprints are deterministic now

Thread 1

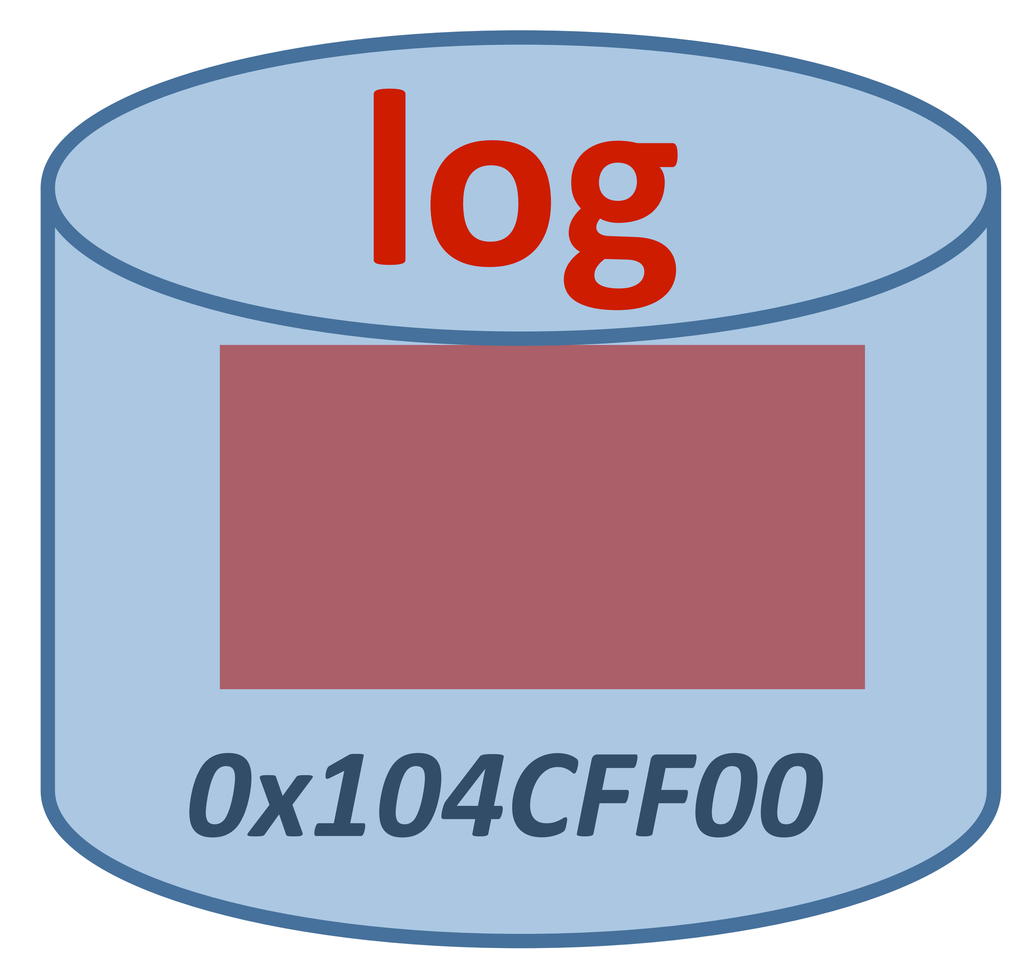
```
cb = (struct iocb *)malloc(...);
ReadFileEx(hSocket, cb->buf,
      BUFSIZE,  (OVERLAPPED *)&cb, 0);
```

Thread 2

```
GetQueuedCompletionStatus(hPort,
      &size, ..., (OVERLAPPED *)&cb, ...);
void * buf = cb->buf;...
```




```
struct iocb
{
      OVERLAPPED olp;
      char buf[BUFSIZE];
      ...
}
```



Deterministic execution order in multi-threaded applications

- What?
  - If function A is executed after function B during recording(*happens-before relation*) , the same order must be enforced during replay.

- Why?
  - May lead to replay failure if it has a different order during replay

An Example

Thread 1

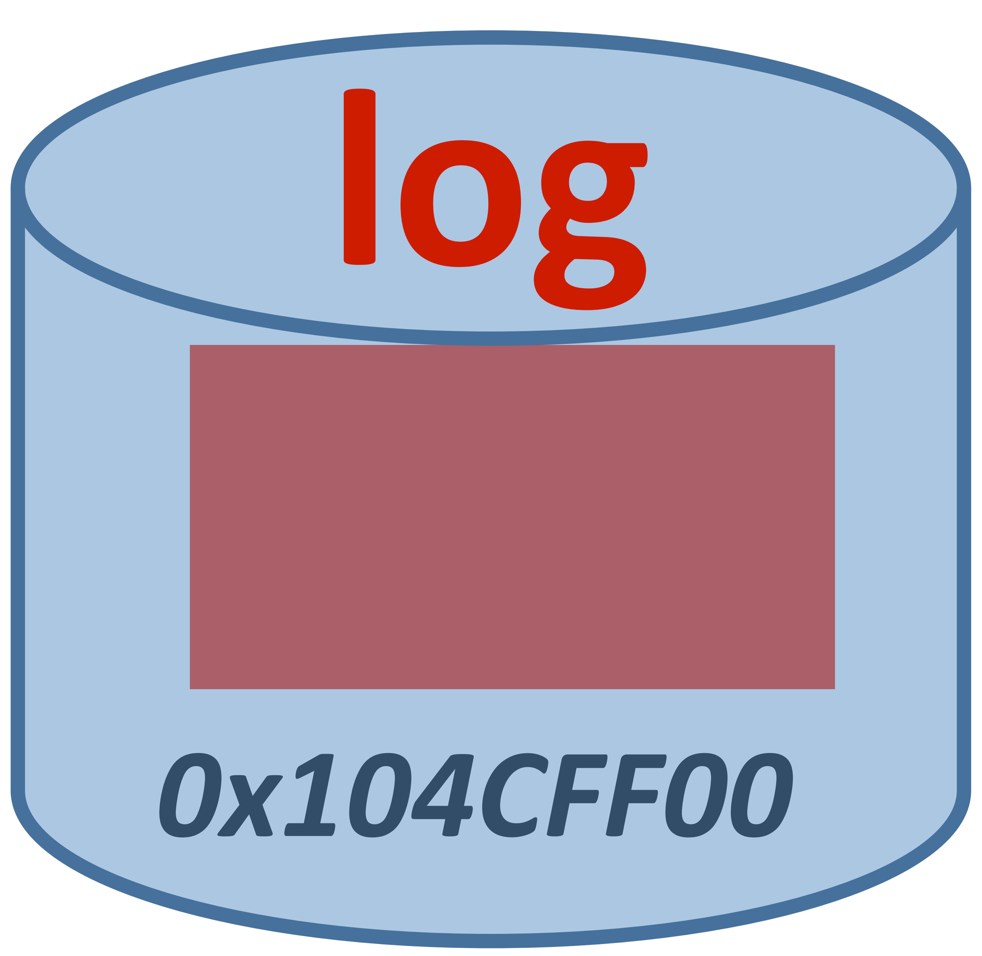
```
cb = (struct iocb *)malloc(...);
ReadFileEx(hSocket, cb->buf,
      BUFSIZE,  (OVERLAPPED *)&cb, 0);
```

Thread 2

```
GetQueuedCompletionStatus(hPort,
        &size, ..., (OVERLAPPED *)&cb, ...);
void * buf = cb->buf;...
```

```
struct iocb
{
     OVERLAPPED olp;
     char buf[BUFSIZE];
     …
}
```

INVALID ADDRESS,
CRASH!!!



Happens-before relations

- Intra-thread
  - always the same during recording and replay

- Inter-thread (Challenges)
  - Callbacks
  - Thread synchronization
  - Resource manipulation
  - Asynchronous I/O

Capture and Maintain happens-before relation

. With the annotations, R2 can capture and enforce these relations

Summary of Annotations

annotation	scope	description
in	param	input (read-only) parameter
out	param	output (mutable) parameter
bsize(val)	param	modified size of an array buffer (val can be any expr)
xpointer(kind)	param	address allocated internally (null , thread, or process)
prepare(key,buf)	func	prepare asynchronous data transfer
commit(key, size)	func	commit asynchronous data transfer
callback	param	callback function pointer (upcall)
sync(key)	func	causality among syscalls/upcalls (key can be any expr)
cache	func	cache for reducing log size
reproduce	func	reproduce I/O for reducing log size

Data Transfer

Execution Order

Optimization

- Some keywords are standard, reused from standard annotation language (SAL)

Annotations allow Automated Code Generation

Annotated Prototype

```
int read (
    [in] int fd,
    [out, bsize(return)] void *buf,
    [in] unsigned int nbytes);
```

PHP Code Template

```
BEGIN_SLOT(record_<?=$f->name?>, <?=$f->name?>)
    logger << <?=$f->name?>_signature << current_tid;
    <?if(is_syscall($f)) {?> logger << return_value;<?}?>
    <?$direction = is_syscall($f) ? 'out' : 'in';?>
    <?foreach($f->params as $p) {
       if ($p->has($direction)) {
        if ($p->has('bsize')) {?>
           logger.write(<?=$f->name?>, <?=$p->val('bsize')?>);
        <?} else {?>
           logger << <?=$f->name?>;
    <?}}}?>
END_SLOT
```

C++ Code Snippet

```
BEGIN_SLOT(record_read, read) // “record_read” after native “read”
    logger << read_signature << current_tid;
    logger << return_value;
    logger.write(buf, return_value);
END_SLOT
```

R2 implementation

- Runs on windows
  - R2 runtime (~20 kloc)
- Annotated three interfaces
  - Win32
  - MPI
  - Sqlite

R2 can replay challenging system applications

Category	Software
Web server	Apache, lighttpd, Null HTTPd
Database	SQLite, Berkeley DB, MySQL
Distributed system	libtorrent, Nyx, PacificA
Virtual machine	Lua, Parrot, Python
Network client	cURL, PuTTY, Wget
Misc.	zip, MPICH

- Replay challenging system software (e.g., those with async IO)
- No modifications to applications but require annotations to the interface

Evaluation

- Questions to answer
  - Annotation effort
  - Overall Performance
  - Effectiveness of customized interface

Experiment Platform

- CPU: 2.0 GHz Xeon dual-core
- Memory: 4 GB
- Disk: two 250 GB, 7200 /s
- Switch: 1 Gbps
- OS: Windows Server 2003 Service Pack 2

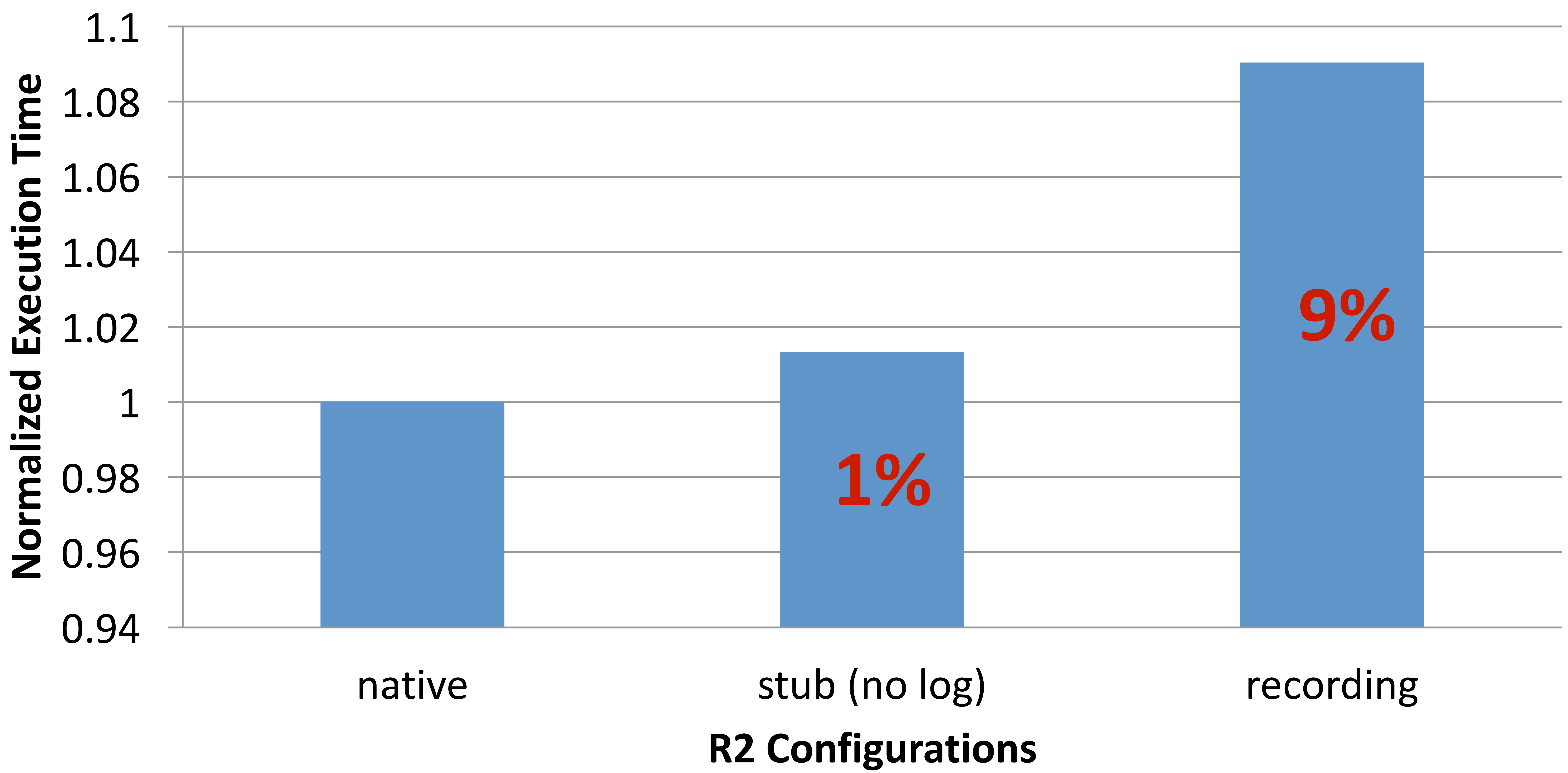
Annotation effort

Interface	#functions/#reuse	Annotation effort	Kloc (autogen)
Win32	1301 / 1239	~one person week (500+)	110.2
MPI	191 / -	~2 person days	22.2
SQLite	153 / -	~2 person days	15.7

Annotate once, used many times!!!

R2 is lightweight in general

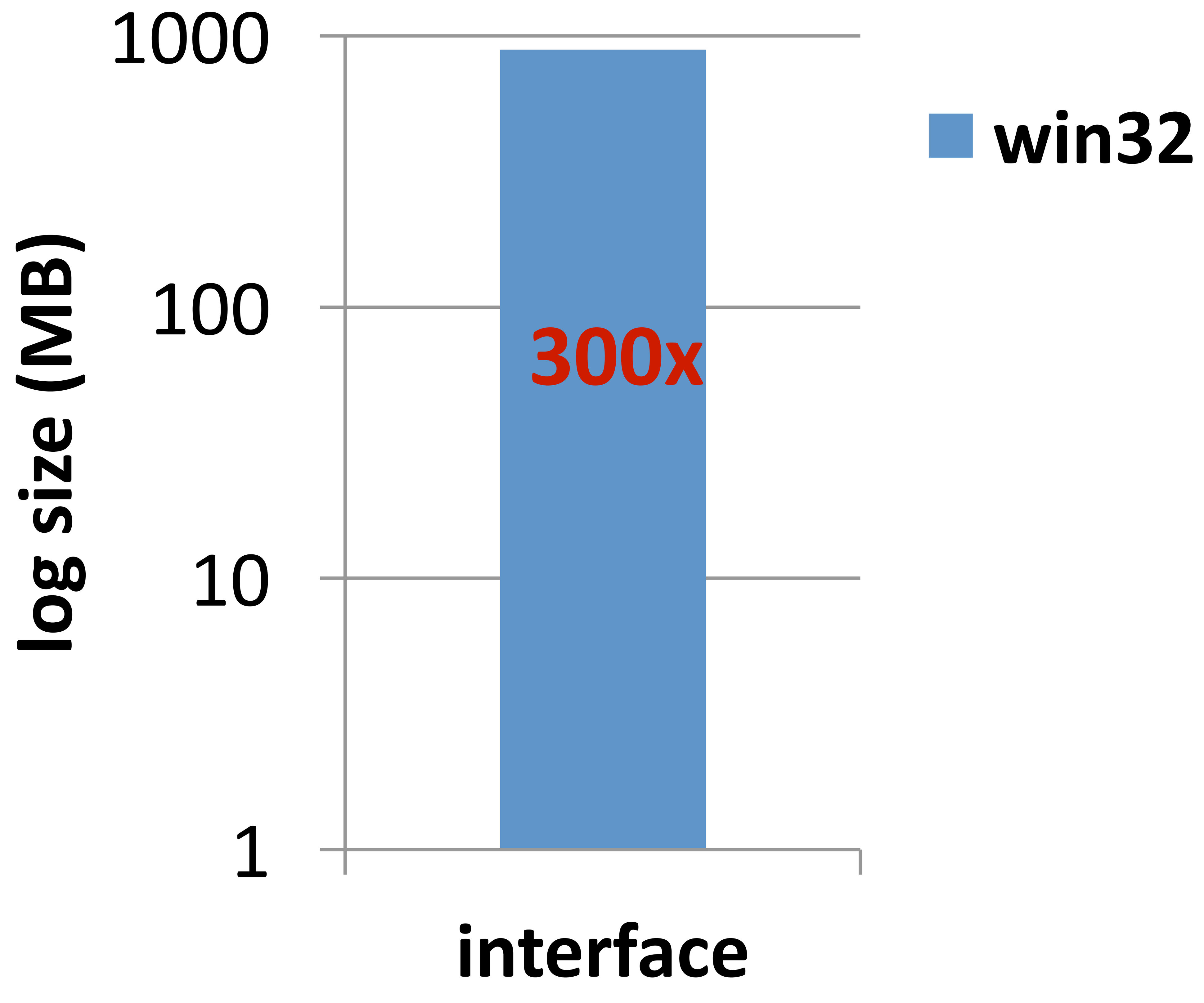
- Apache, filesize = 64 KB, client concurrency = 50, win32 interface

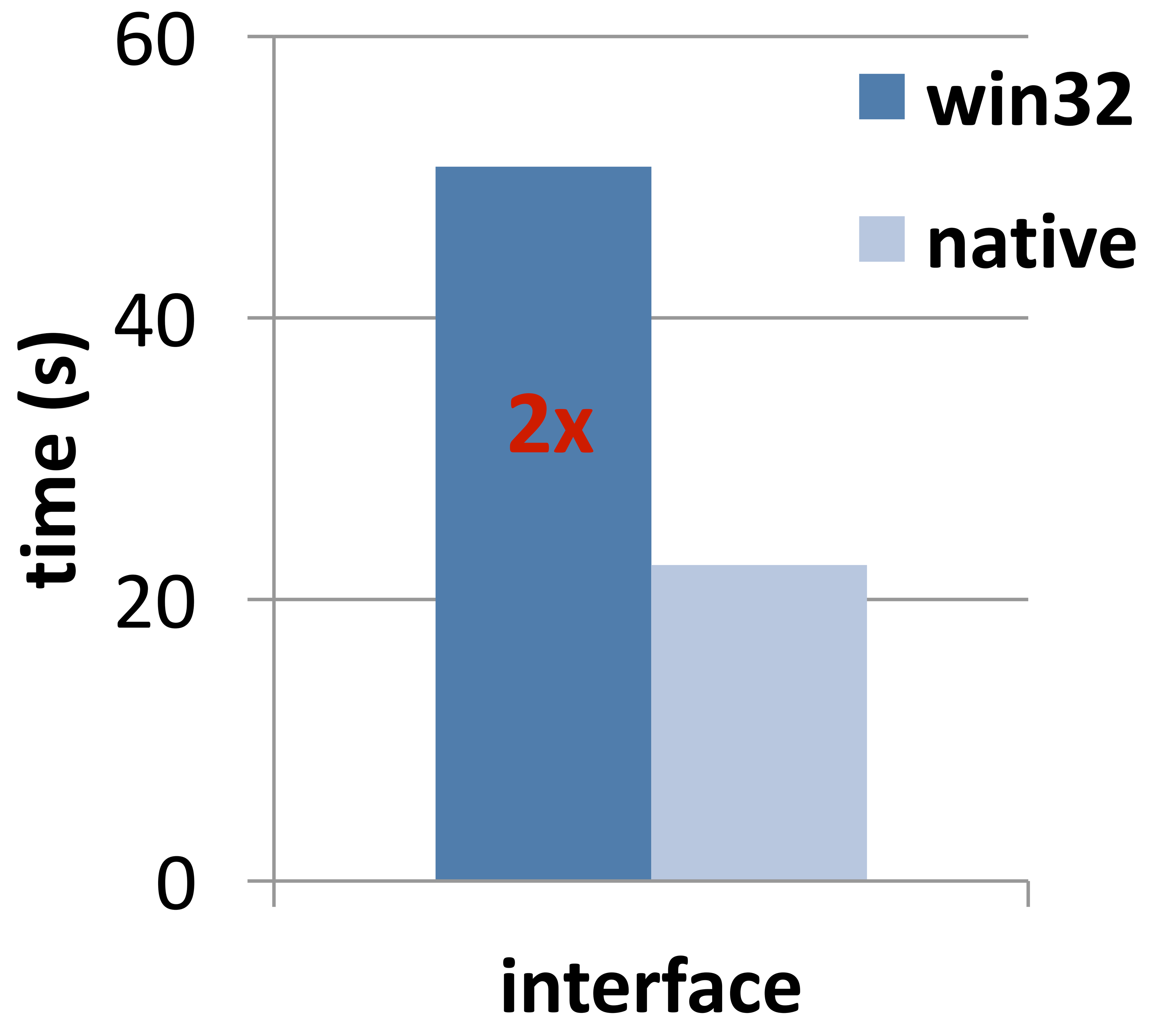


- For some applications, the overhead may be larger

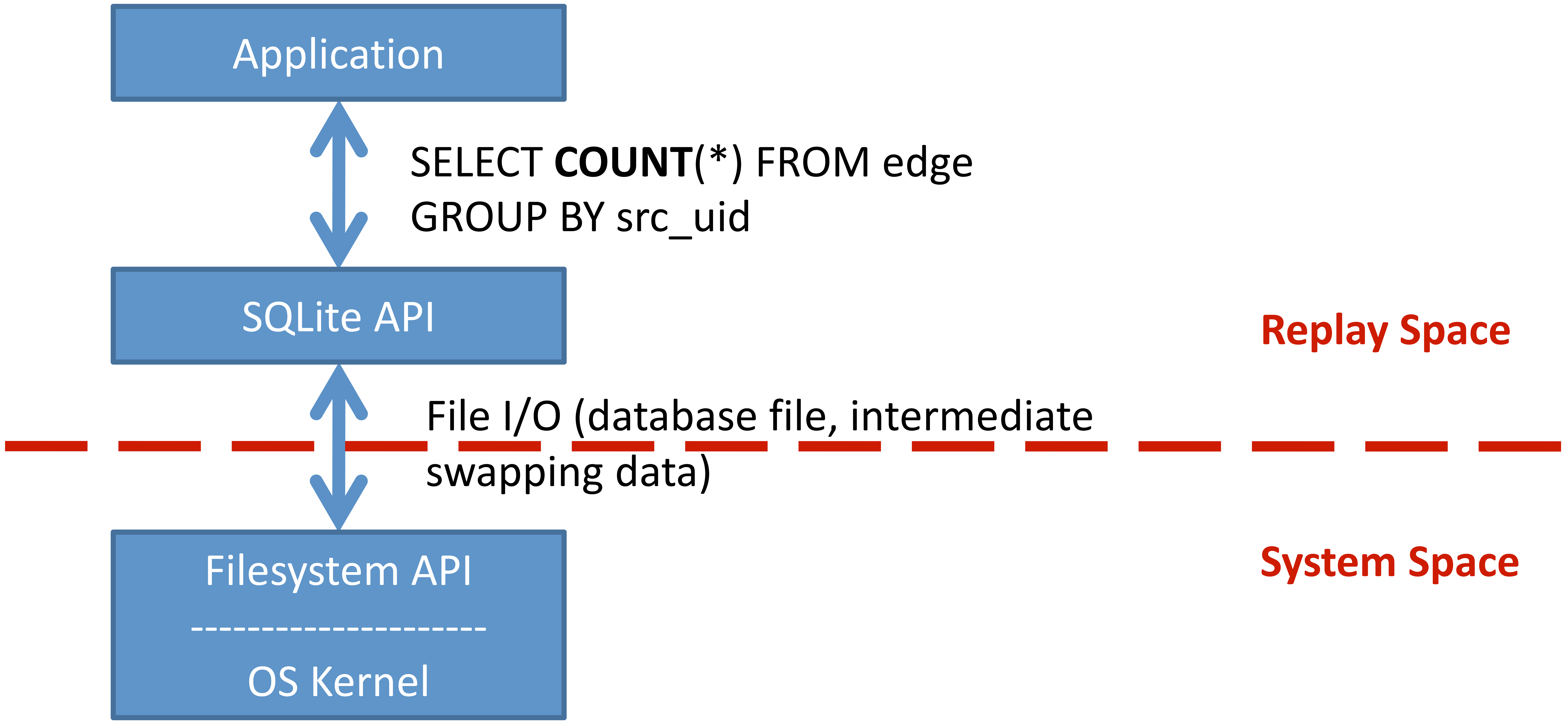
Example: Sqlite

- SELECT **COUNT**(*) FROM edge GROUP BY src_uid
- Filesize = 3 MB, win32 interface

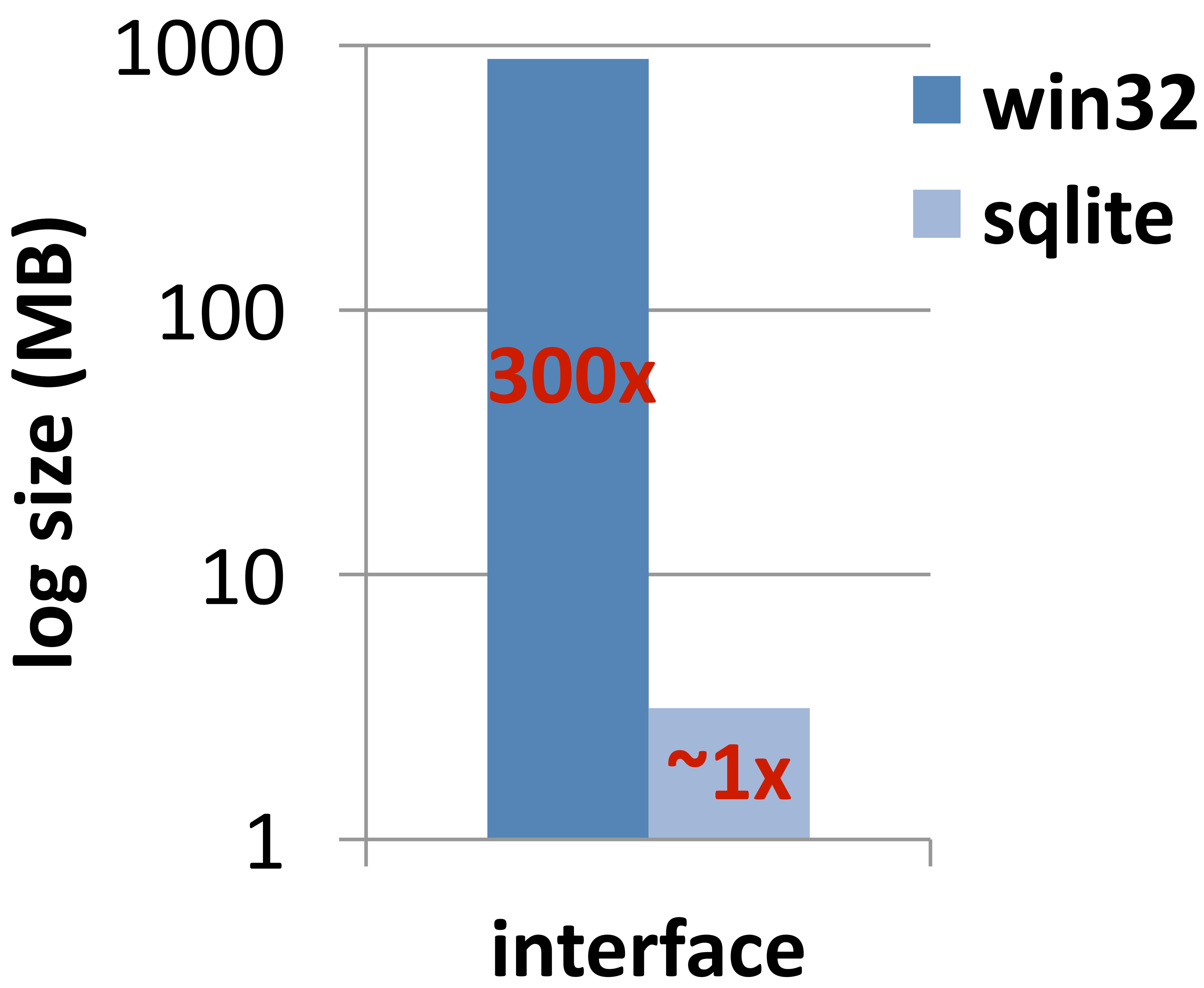


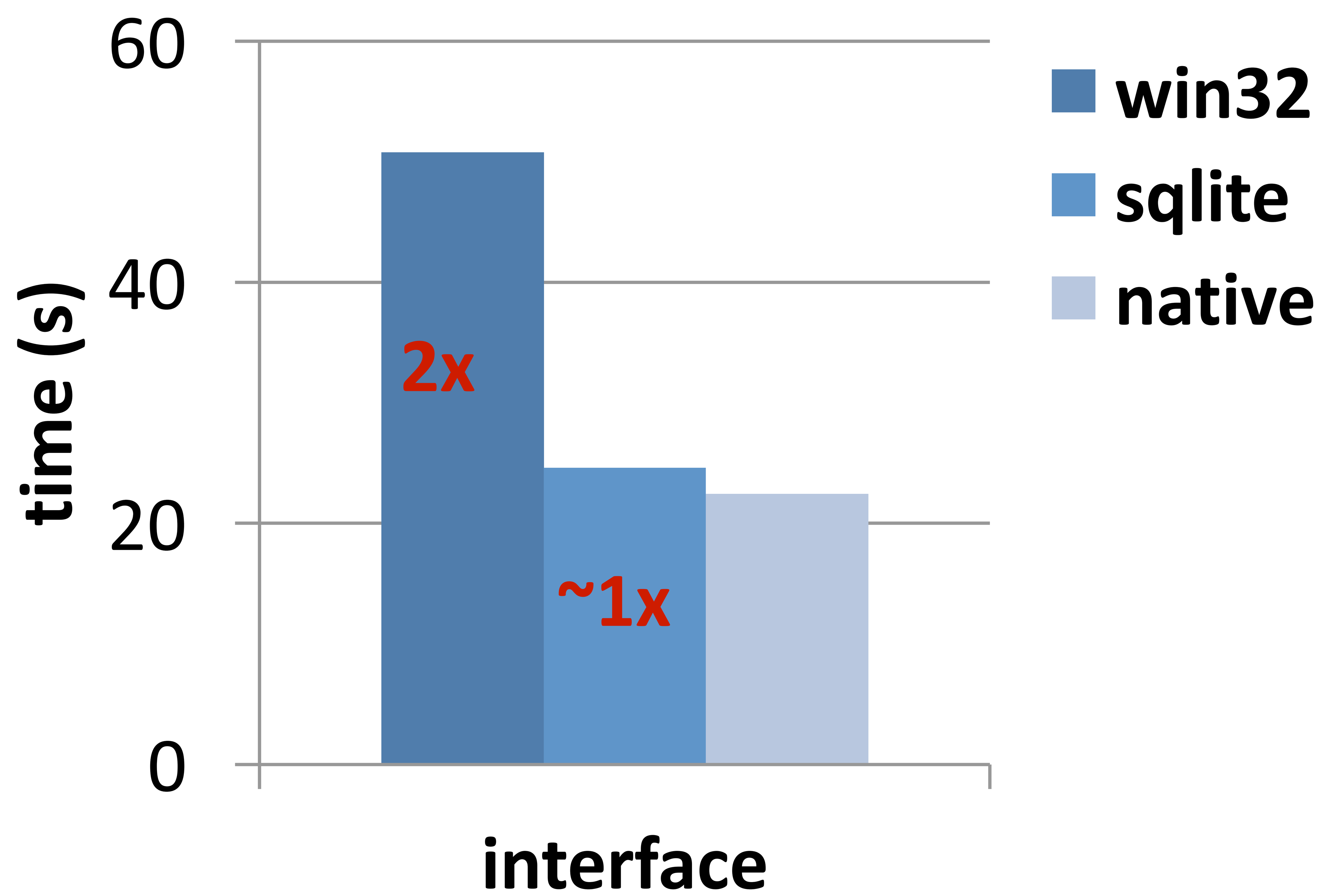


Solution: choose an interface with less I/O



Raising interface reduces overhead





R2 is useful beyond replay

- Space split can help in-process tools
  - E.g., in model checker
    - Define what you do/don't want to check

- Space split + annotation + code generation, a powerful combination that we have applied to many other projects
  - Hang Cure for dynamically curing hang problems (Eurosys'08)
  - Towards Automatic Inference of Task Hierarchies in Complex Systems (HotDep'08)
  - MPIWiz: Subgroup reproducible replay of MPI applications (PPoPP'09)
  - Model checker for distributed systems (Submitted)

Related work

- Library-based replay: liblog (USENIX'06), Jockey (AADEBUG'05), RecPlay (TOCS'99)
- Domain-specific replay: ML, MPI, Java
- Whole system replay: hardware (Strata, ASPLOS'06), VM (Revirt, OSDI'02; iDNA, VEE'06)
- Annotations: SAL (ICSE'06), Deputy (OSDI'06)
- Isolation: XFI (OSDI'06), Nooks (SOSP'03)

- Main distinction: allow developers to select an easy & efficient replay interface

Conclusion

- A set of rules that allows to select an interface that
  - can be made replay faithful (Correctness)
  - cost less record and replay overhead (Performance)
- A set of keywords describing the side effects that helps R2 generate stubs
- A replay/system space split to make the interface replay faithful
- A win32 implementation with low overhead, can replay challenging system applications

Discussion

- How to pick a good interface?
  - API calls
  - Log size
  - Debug visibility
  - Tradeoffs…

Discussion

- Long running execution
  - Checkpointing?
  - What states to save?

Discussion

- Do we need FAITHFUL replay?
  - Probabilistic replay…

Discussion

- Multithreading
  - Data race?
  - Multi core replay

Discussion

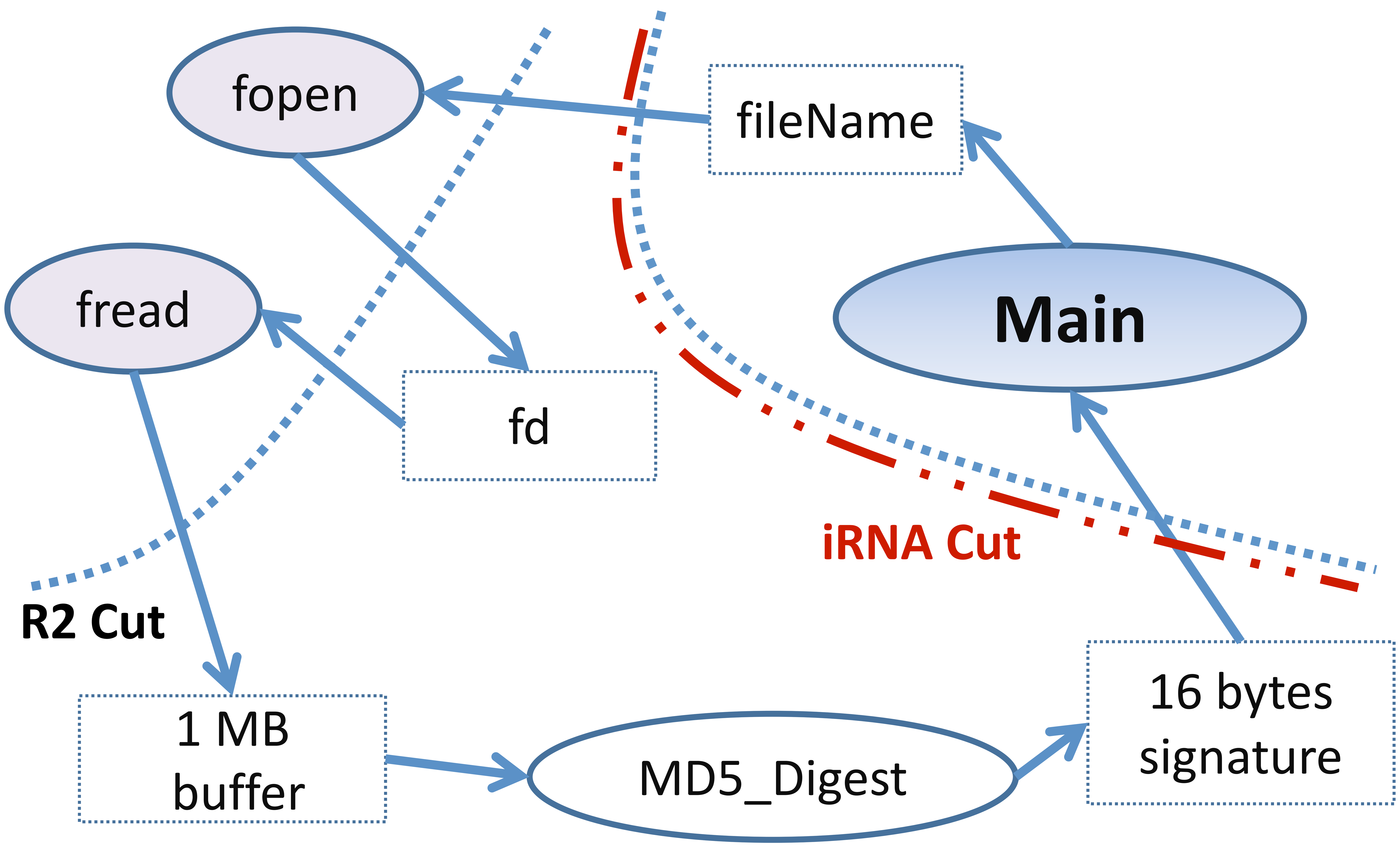
- Many ways to implement R&R
  – Library
  – Virtual machine
  – Mozilla’s rr

Discussion

- Other ways to debug
  - Deterministic execution
  - Model checking
  - Grep TODO

Thanks!

(R2 latest) How to follow the two rules faithfully: static analysis to remove annotation effort (and potential annotation error)!



- Bipartite flow graph between functions and variables
- Static analysis to get this graph
- Dynamic profiling for edge weight
- Min-cut on the graph for the near-optimal replay interface (in terms of log size)