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
  – Record one run and Replay 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
...

An unhandled win32 exception occurred in SignatureUpdate.exe [1600].

Possible Debuggers:
New instance of Visual Studio 2008
Faithful Replay

D:/> `set R2_MODE=Replay`
D:/> `R2.exe signatureUpdate.exe srg-tango0`
start check @ 19:12:47.11
... downloading ...
4356 entries downloaded
...
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

Record

OS Kernel

App

interposition interface

read

native read

log output

App

interposition interface

read

return output

Replay Log

Replay
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

<table>
<thead>
<tr>
<th>Problems</th>
<th>R2</th>
<th>Code</th>
</tr>
</thead>
</table>
| 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

```c
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?

Recording Replay

INVALID SOCKET, CRASH!!!
A replay interface must be a call graph cut.
An incorrect cut

- **main**
  - **socket**
  - **recv**
  - **network**
  - **f1**
    - Write: 1 -> 2
  - **g_state**
  - **f2**
    - Read: 1
    - Incorrect Value!!!

Recording Replay
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

\[ cb = \text{(struct iocb *)} \text{malloc}(\ldots); \]
\[ \text{ReadFileEx}(\text{hSocket, } cb\rightarrow\text{buf}, \text{BUFSIZE, (OVERLAPPED *)}&cb, 0); \]

Thread 2

\[ \text{GetQueuedCompletionStatus}(\text{hPort,} \]
\[ \&\text{size}, \ldots, \text{(OVERLAPPED *)}&cb, \ldots); \]
\[ \text{void * buf} = cb\rightarrow\text{buf}; \ldots \]

INVALID ADDRESS, CRASH!!!

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

Network Message

log

0x104CEF00
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

Replay Space

System Space

app
libraries
OS Kernel
Memory Isolation

Replay Space

app

malloc

malloc

replay interface

malloc

libraries

malloc

OS Kernel

System Space

Deterministic Memory Pool

Native Memory Pool

User Memory Address Space
Handle data transfer across interface

`char* getcwd (NULL, 0);`

Diagram:
- **Replay Space**
  - `app`
  - `replay interface`
  - `libraries`
  - `OS Kernel`

- **System Space**
  - `malloc`

- **Native Memory Pool**
  - `block`

- **Deterministic Memory Pool**
  - `block`

- **User Memory Address Space**
Memory footprints are deterministic now

**Thread 1**

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

cb = (struct iocb *)malloc(...);

ReadFileEx(hSocket, cb->buf, BUFSIZE, (OVERLAPPED *)&cb, 0);
```

**Thread 2**

```c
GetQueuedCompletionStatus(hPort, &size, ..., (OVERLAPPED *)&cb, ...);
void * buf = cb->buf;...
```
Deterministic execution order in multi-threaded applications

• What?
  – If function A is executed after function B during recording \( \text{happens-before relation} \), 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**

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

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

**Thread 2**

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

**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

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

- Some keywords are standard, reused from standard annotation language (SAL)
Annotations allow Automated Code Generation

begin_slot

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

```php
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
```

```php
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

<table>
<thead>
<tr>
<th>Category</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web server</td>
<td>Apache, lighttpd, Null HTTPd</td>
</tr>
<tr>
<td>Database</td>
<td>SQLite, Berkeley DB, MySQL</td>
</tr>
<tr>
<td>Distributed system</td>
<td>libtorrent, Nyx, PacificA</td>
</tr>
<tr>
<td>Virtual machine</td>
<td>Lua, Parrot, Python</td>
</tr>
<tr>
<td>Network client</td>
<td>cURL, PuTTY, Wget</td>
</tr>
<tr>
<td>Misc.</td>
<td>zip, MPICH</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Interface</th>
<th>#functions/#reuse</th>
<th>Annotation effort</th>
<th>Kloc (autogen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32</td>
<td>1301 / 1239</td>
<td>~one person week (500+)</td>
<td>110.2</td>
</tr>
<tr>
<td>MPI</td>
<td>191 / -</td>
<td>~2 person days</td>
<td>22.2</td>
</tr>
<tr>
<td>SQLite</td>
<td>153 / -</td>
<td>~2 person days</td>
<td>15.7</td>
</tr>
</tbody>
</table>

**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

SELECT \texttt{COUNT(*)} FROM edge  
GROUP BY src_uid

File I/O (database file, intermediate swapping data)

SQLite API

Application

OS Kernel

Replay Space

System Space
Raising interface reduces overhead

![Log size comparison](image1)

![Time comparison](image2)
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)