

ME AND YOU  
AND EVERYONE  
WE DEBUG





*I want to debug back and forth.*

with the same trace

)><(<>((  
forever

```
$ id  
uid=65534(nobody)  
gid=65534(nogroup)  
groups=65534(nogroup)
```

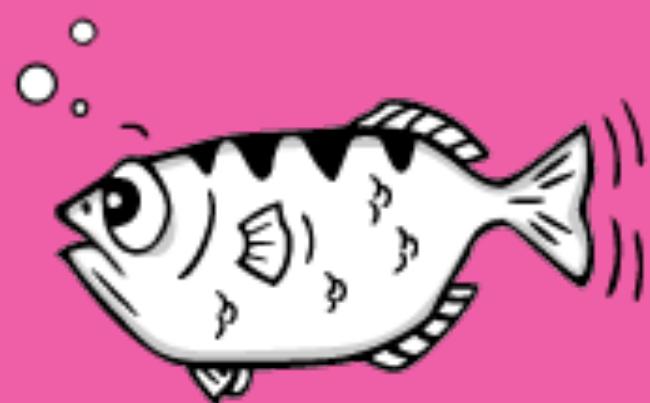
- only 🇺🇸 r2 dev?
- air djibouti 🇯🇪 passenger 
- recent apple convert
- fuzzing every day
- wishing for better debugger tools
- macOS, x86\_64 only, to make things easy
- radare2 user/dev for 6+ years
  - computer user for 10 years
- stupidly told my manager i could build a reversible debugger in a weekend

(d  r)

-



# existing tech/strategies



The screenshot shows the Immunity Debugger interface with the following details:

- Registers:** AX: 0x0, ECX: 0xffffffff, EDX: 0xfbfbb5b8, EBX: 0xfofb0ff4, ESP: 0xfbfccfc0, EBP: 0xfffffefa8, ESI: 0x0, EDI: 0x0, EIP: 0x8048446.
- Stack Dump:** Shows memory starting at address 0x00400000, containing the string "Ox00400000 <-- 0x0".
- Call Stack:** Shows the current call stack with addresses 0x8048446, 0x8048442, 0x8048438, 0x8048434, 0x8048430, 0x804842f, 0x804842e, 0x804842d, 0x804842c, 0x804842b, 0x804842a, 0x8048429, 0x8048428, 0x8048427, 0x8048426, 0x8048425, 0x8048424, 0x8048423, 0x8048422, 0x8048421, 0x8048420, 0x8048419, 0x8048418, 0x8048417, 0x8048416, 0x8048415, 0x8048414, 0x8048413, 0x8048412, 0x8048411, 0x8048410, 0x8048409, 0x8048408, 0x8048407, 0x8048406, 0x8048405, 0x8048404, 0x8048403, 0x8048402, 0x8048401, 0x8048400.

The screenshot shows a Microsoft Visual Studio 2010 debugger interface during a session. The Command window at the bottom left displays assembly code and time travel stack traces. The Locals window shows variable values for MyDog and MyDogs objects. The Stack window shows the call stack. The Breakpoints window is visible on the left.

Breakpoints

File Home View Breakpoints Model Scripting Command Command Memory Source Source

Step Out Step Into Step Over Step Into Back Step Over Back Go Back Restart Stop Debugging Detach Local Help

Break: 30

Flow Control Reverse Flow Control End Source Mode Help

debugbase: \Debug\MyDog\_Console

25 class Simple1DArray  
26 {  
27 public:  
28 ULONG64 m\_size = 5;  
29 int intArray[5] = { 1,2,3,4,5 };  
30 int \*m\_pValues = &intArray[0];  
31 Simple1DArray()  
32 {  
33 }  
34  
35 }  
36  
37  
38  
39 int main()  
40 {  
41 CDog MyDog;  
42 CDogs MyDogs;  
43 Simple1DArray g\_array10;  
44 printf\_s("Array Information \n");  
45 printf\_s("%I64d, %d \n", g\_array1D.m\_size  
46 printf\_s("Array size Dog Object Information  
47 printf\_s("%d, %d\n", MyDog.m\_size, MyDog.m\_weight);  
48 // Commenting out for TDD Testing 9-8-201  
49 getchar();  
50 return 0;

ntdll!TpAllocTimer+0xc5:  
00007ff9'31e06695 488b5c2440 mov rbx,qword ptr [rsp+40h] ss:000  
0:000: p  
Time Travel Position: 12:1E  
ntdll!RtlInitializeHeapGC+0x67:  
00007ff9'31efc4b 83c0 test eax,eax  
0:000: p  
Time Travel Position: 12:23  
ntdll!RtlInitializeHeapGC+0x6b:  
00007ff9'31efc4f 488b4518 mov rax,qword ptr [rbp+18h] ss:000  
0:000: p  
Time Travel Position: 12:23  
could step in/over inline function frames ...  
01 00000092'397df670 00007ff9'31ef41d7 rtdll!RtlInitializeHeapGC+0x76  
00007ff9'31efc5a c705d4060e0001000000 mov dword ptr [ntdll!RtlpHpcCTime  
0:000: g  
Breakpoint hit  
Time Travel Position: 3E:12E  
CDog\_Console!main:  
00007ff6'5ce1ia00 4855 push rbp  
Time Travel Position: 3E:12A  
CDog\_Console!main+0x2f:  
00007ff6'5ce1ia2f 488d4d08 lea rcx,[rbp+8]  
0:000:

Stack

CDog\_Console!main + 0x41  
0x62597d19E8

0x20000000000000C  
0x20000000000000C  
0x20000000000000C  
0x20000000000000C

Locals

Name	Value	Type
MyDog	8	CDog
n_age	8	long
n_weight	30	long
MyDogs	30	CDogs
MyDog1	8	CDog
m_age	8	long
m_weight	30	long

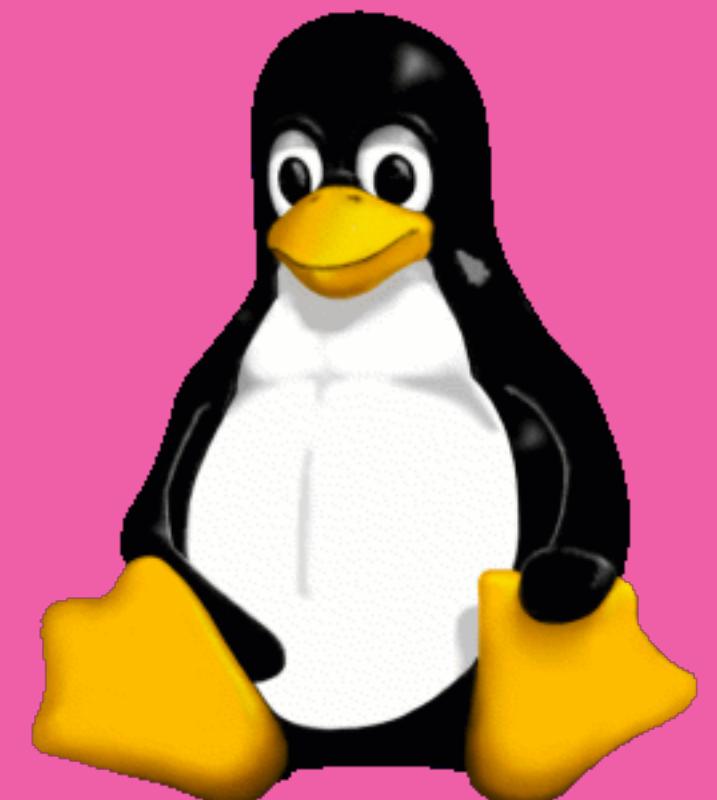
Threads Stack Breakpoints

**rr**

A large black graphic element is positioned in the upper left corner of the page. It features a solid black circle on the left, followed by a vertical bar, and then two dark gray triangles pointing towards the right.

# rr (almost like r2 ;) )

- "rr records nondeterministic executions and debugs them deterministically" : awesome!
- mozilla project from 2014 for debugging difficult to reproduce firefox bugs.
- built on several clever linux kernel hacks, performance counters for replay, etc. There's a whitepaper, it's open source.
- requires linux kernel, not really portable.



# microsoft ttt + windbg



windows only ofc... i asked some msftie friends, but didn't do much digging myself

# radare2

- ren kimura's gsoc project in 2017
- record and replay (rnr), with diffs, checkpoints, and all sorts of great features.
- <https://asciinema.org/a/nR8jzJwHj9bkzo8q6hTwHOZL4>
- unfortunately it's either WAY too slow, or doesn't work on macOS.
  - several minutes for a single `dsb`
  - i didn't really want to go digging (sorry!)



# gdb

- on some platforms "process record" exists.
- if an instruction is in the record log, take the effect of the log, otherwise execute. perfect for both forward and reverse debugging!
- macOS is not one of these platforms
- i ported this in a weekend, super easy to do.
  - 500 line patch to gdb for basic support.
  - many instructions are unsupported, not a big deal, right?



# gdb

- xsave/xrstor not supported. used heavily in libSystem. Everything links against libSystem...
- xrstor required a similar length patch. as did a few other instructions before i gave up
- doing this so many times sucks, it's really tough to make sure you do it right, intel manual is very very dense.
- (ask me and i'll give you the patches), don't think they apply cleanly anymore

```
445+     case 4: /* xsave */
446+     {
447+         if (ir.mod == 3) {
448+             opcode = (opcode << 8) | ir.modrm;
449+             goto no_support;
450+         }
451+         ULONGEST rfbm_eax, rfbm_edx;
452+
453+         regcache_raw_read_unsigned (regcache, I386_EAX_REGNUM, &rfbm_eax);
454+         rfbm_eax &= (1ULL << 32) - 1;
455+
456+         regcache_raw_read_unsigned (regcache, I386_EDX_REGNUM, &rfbm_edx);
457+         rfbm_edx &= (1ULL << 32) - 1;
458+
459+         uint64_t rfbm = tdep->xcr0 & (rfbm_eax | rfbm_edx << 32);
460+
461+         if (rfbm & ~X86_XSTATE_ALL_MASK) {
462+             printf_unfiltered (_("Process record does not support XSAVE "
463+                                 "instruction with RFBM=%" PRIx64 ".\n"),
464+                                 rfbm);
465+             opcode = (opcode << 8) | ir.modrm;
466+             goto no_support;
467+         }
468+
469+         uint64_t tmpu64;
470+         if (i386_record_lea_modrm_addr (&ir, &tmpu64)) {
471+             return -1;
472+         }
473+
474+         uint64_t legacy_size = 160; /* x87 xsave size */
475+         if (rfbm & X86_XSTATE_SSE) {
476+             if (ir.regmap[X86_RECORD_R8_REGNUM])
477+                 legacy_size = 416; /* 64 bit sse xsave size */
478+             else
479+                 legacy_size = 288; /* 32 bit sse xsave size */
480+         }
481+         if (record_full_arch_list_add_mem (tmpu64, legacy_size))
482+             return -1;
483+
484+         uint64_t size = X86_XSTATE_SIZE (rfbm) - 512;
485+         if (record_full_arch_list_add_mem (tmpu64 + 512, size))
486+             return -1;
487+     }
488+     break;
489+
```

# gdb

- i gave this demo to my team at work, only showed code that was supported, won my "bet", ignored this topic for a few months.

**Warrior:** hi

**ed:** Hi friend.

**Warrior:** hi

**ed:** I've been thinking about the "back and forth."

# qira

- qemu-user powered trace recorder and replayer
- some suboptimal webui with crazy colors i don't quite understand, but not a big dealbreaker
- qemu-user only works on linux applications on linux hosts, so that doesn't really work.
- but...
  - there's an outdated pin tracer, i386 only, but pin is cross platform. perfect place to start. my work is mostly built by reading qira sources

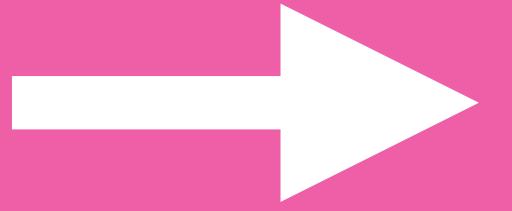


# so, what'd i build?



- plugin for r2 for deterministically replaying a single execution trace.

# so, what'd i build?



- replay trace back on any machine, on any architecture
- (but must be recorded on intel x86)

# so, what'd i build?



- full record and replay, the entire program execution is recorded and nothing is emulated
- support of sse/avx/mmx/other fancy instruction with zero effort

# pin

- intel's tool for binary instrumentation
- support linux, macOS, and windows
- pin runtime lacks rtti, tools cannot use exceptions
- non-free PinCRT, but tools can be open source
- i'd prefer dynamorio, but doesn't work on macOS, so let's dig in



# how does it work

- actually very simple
- ∀ images loaded -> record base addresses
- ∀ instruction executed -> record register set
- ∀ memory write -> record address and value (qword granularity)

# how does it work

- all register/memory access/writes logged to text file.
- single execution of /bin/ls on macOS -> ~20mb log
- totally not-optimized, but overhead is quite high on the initial execution. once the data has been preprocessed, it's not so bad to work with though

# hooking at runtime

- Pin allows you to insert hooks on certain events, so I insert several.

# hooking at runtime (loader)

```
IMG_AddInstrumentFunction(ImageLoad, 0);
```

- if it's the first image, mark the highest and lowest addresses. we can use this for later filtering\*
- otherwise just list all sections mapped, high and low addresses
- macOS has non-contiguous image loading for some reason (performance maybe?)

# hooking at runtime (loader)

```
1 0bb78000-0bb82000 0 /bin/ls
2 0d293000-0d35e000 0 /usr/lib/dyld
3 66206000-66208000 0 /usr/lib/libSystem.B.dylib
4 9f1be000-9f1be2d8 38fb8000 /usr/lib/libSystem.B.dylib
5 d1eca000-dfd023d6 6bcc4000 /usr/lib/libSystem.B.dylib
6 66449000-6649d000 0 /usr/lib/libc++.1.dylib
7 9f1ea000-9f1f1cf8 38da1000 /usr/lib/libc++.1.dylib
8 d1eca000-dfd023d6 6ba81000 /usr/lib/libc++.1.dylib
9 6649d000-664b3000 0 /usr/lib/libc++abi.dylib
10 9f1f2000-9f1f5210 38d55000 /usr/lib/libc++abi.dylib
11 d1eca000-dfd023d6 6ba2d000 /usr/lib/libc++abi.dylib
12 67487000-674b8000 0 /usr/lib/libncurses.5.4.dylib
13 9f4cd000-9f4d0e4c 38046000 /usr/lib/libncurses.5.4.dylib
14 d1eca000-dfd023d6 6aa43000 /usr/lib/libncurses.5.4.dylib
15 67a3d000-681c3000 0 /usr/lib/libobjc.A.dylib
16 9f66a000-9f7f61d0 37c2d000 /usr/lib/libobjc.A.dylib
17 d1eca000-dfd023d6 6a48d000 /usr/lib/libobjc.A.dylib
18 6878a000-6878e000 0 /usr/lib/libutil.dylib
19 9f973000-9f973410 371e9000 /usr/lib/libutil.dylib
20 d1eca000-dfd023d6 69740000 /usr/lib/libutil.dylib
21 690a2000-690a7000 0 /usr/lib/system/libcache.dylib
22 9fa67000-9fa67128 369c5000 /usr/lib/system/libcache.dylib
23 d1eca000-dfd023d6 68e28000 /usr/lib/system/libcache.dylib
24 690a7000-690b2000 0 /usr/lib/system/libcommonCrypto.dylib
25 9fa68000-9fa694b0 369c1000 /usr/lib/system/libcommonCrypto.dylib
26 d1eca000-dfd023d6 68e23000 /usr/lib/system/libcommonCrypto.dylib
27 690b2000-690ba000 0 /usr/lib/system/libcompiler_rt.dylib
28 9fa6a000-9fa6b0b0 369b8000 /usr/lib/system/libcompiler_rt.dylib
29 d1eca000-dfd023d6 68e18000 /usr/lib/system/libcompiler_rt.dylib
30 690ba000-690c4000 0 /usr/lib/system/libcopyfile.dylib
31 9fa6c000-9fa6c6f0 369b2000 /usr/lib/system/libcopyfile.dylib
```

# hooking at runtime (loader)

- this allows us to load each dependency into the r2 session during the trace replay for visualization.
- o [file]; om fd vaddr [size] [paddr] [rwx] [name]
  - do this for each library loaded
- (note, there's the downside if an image gets unmapped and then something else gets mapped back into our section, then, we're totally fucked [workarounds planned])

# hooking at runtime (instruction)

```
INS_AddInstrumentFunction(Instruction, 0);
```

- callback on every instruction called. the overhead isn't actually that bad as pin uses a JIT (beyond the scope of my understanding of how pin works...)
- trace lots of stuff...

# hooking at runtime (instruction)

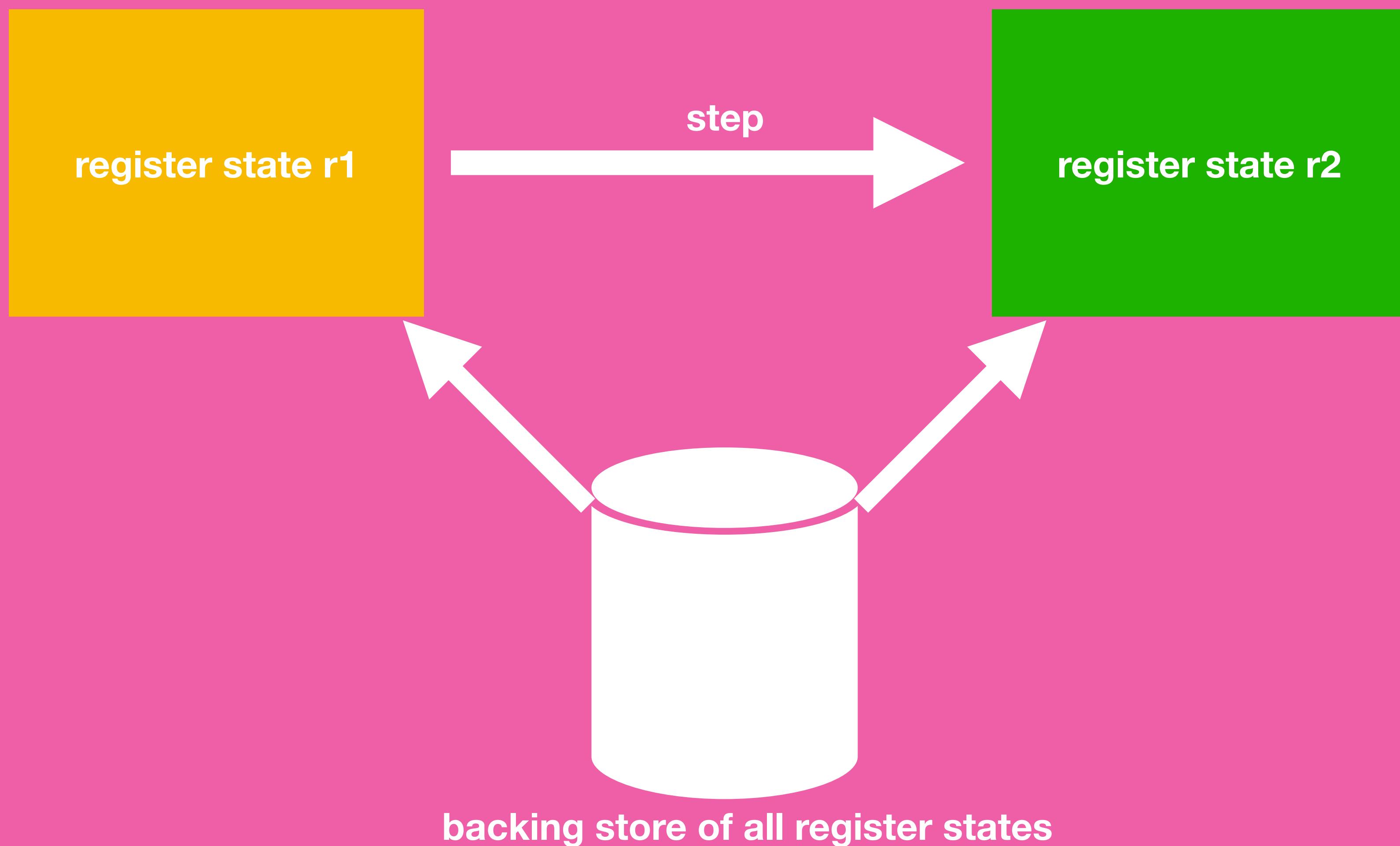
```
if (!filtered) INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)DumpRegsI,  
    IARG_INST_PTR,  
    IARG_REG_VALUE, REG_RAX,  
    IARG_REG_VALUE, REG_RBX,  
    IARG_REG_VALUE, REG_RCX,  
    IARG_REG_VALUE, REG_RDX,  
    IARG_REG_VALUE, REG_RSI,  
    IARG_REG_VALUE, REG_RDI,  
    IARG_REG_VALUE, REG_RBP,  
    IARG_REG_VALUE, REG_RSP,  
    IARG_REG_VALUE, REG_R8,  
    IARG_REG_VALUE, REG_R9,  
    IARG_REG_VALUE, REG_R10,  
    IARG_REG_VALUE, REG_R11,  
    IARG_REG_VALUE, REG_R12,  
    IARG_REG_VALUE, REG_R13,  
    IARG_REG_VALUE, REG_R14,  
    IARG_REG_VALUE, REG_R15,  
    IARG_END);  
  
    fprintf(pFile2, "I %llx  
%llx %llx\n", (ADDRINT)ip, rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r  
14, r15);
```

- for every instruction executed, log all registers
  - literally just logging them out to a file
  - every register, on every instruction

# hooking at runtime (instruction)

- this is second iteration. first was delta diffing,
- stored all diffs (only changed registers) in a messagepack blob, smaller, but actually took up a lot more compute time later, so abandoned.
- can filter to just one particular binary, in case we only want to trace in a certain lib or the main image (probably only want to record/replay in the crashing image or two)

# hooking at runtime (instruction)



BUT IT ISN'T  
ENOUGH!

# hooking at runtime (memory)

- dumping the stack and heap is also important to see how heap or stack could get corrupted in the case of a fuzzing crash.
- it's stupid/impossible to dump and restore the entire mapped memory space on each instruction executed.
- too expensive in terms of both disk and memory for sure
- can we do better?

# hooking at runtime (memory)

```
INS_InsertPredicatedCall(
    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite1, IARG_THREAD_ID,
    IARG_MEMORYOP_EA, i,
    IARG_REG_VALUE, writeea_scratch_reg,
    IARG_RETURN_REGS, writeea_scratch_reg,
    IARG_END
);
```

- track all memory writes, do it at the qword level
- in pseudocode explanation
- if (memory\_is\_written) {  
 if (size > 8) pad\_with\_rest\_of\_value();  
 store\_to\_database(address, value);  
}

```
if (flags & (IS_MEM|IS_WRITE)) {
    fprintf(pFile2, "M %llx %llx\n", addr, data);
}
```

# so what do we have now

- a kind of super useless giant file 20+MB on single execution of /bin/ls on my system



The screenshot shows a terminal window with a dark background and light-colored text. It displays a list of file entries from a command-line interface. The entries are mostly identical, showing paths like '/tmp/98574' through '/tmp/98623' followed by file metadata (permissions, size, and modification time). The terminal window includes status bars at the bottom showing the file name ('myfile2.txt'), the number of lines ('1:zshZ 2:zsh\* 3:zsh- 4:zsh'), memory usage ('9271/16384MB 33.1% up 1 Sat 1:45:18 AM 09-07-2019'), and the current line number ('98623:1').

```
98574 M 7ffee4085dc0 7ffee4086220
98575 M 7ffee4085dc0 7ffee4086220
98576 M 7ffee4085db8 7ffee4086c01
98577 M 7ffee4085db0 10d2ef31c
98578 M 7ffee40861c8 0
98579 M 7ffec4085dc8 10d29c579
98580 M 7ffee4085dc8 10d29c58a
98581 M 7ffee4086058 10d29c5a2
98582 M 7ffee4086bf8 7fff6921f3d5
98583 I 10bb791e8 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086c08 7ffee4086bf8 0 0 0 0 0 0 0 0
98584 M 20 7ffee4086bf0
98585 M 7ffee4086bf0 7ffee4086c08
98586 I 10bb791e9 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086c08 7ffee4086bf0 0 0 0 0 0 0 0 0
98587 M 28 7ffee4086bf0
98588 I 10bb791ec 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086bf0 0 0 0 0 0 0 0 0
98589 M 20 7ffee4086be8
98590 M 7ffee4086be8 0
98591 I 10bb791ec 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086be8 0 0 0 0 0 0 0 0
98592 M 20 7ffee4086be0
98593 M 7ffee4086be0 0
98594 I 10bb791f0 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086be0 0 0 0 0 0 0 0 0
98595 M 20 7ffee4086bd8
98596 M 7ffee4086bd8 0
98597 I 10bb791f2 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086bd8 0 0 0 0 0 0 0 0
98598 M 20 7ffee4086bd0
98599 M 7ffee4086bd0 0
98600 I 10bb791f4 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086bd0 0 0 0 0 0 0 0 0
98601 M 20 7ffee4086bc8
98602 M 7ffee4086bc8 0
98603 I 10bb791f5 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086bc8 0 0 0 0 0 0 0 0
98604 M 20 7ffee4086b0
98605 I 10bb791fc 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 0
98606 M 78 7ffee4086c18
98607 I 10bb791ff 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 0
98608 M 70 1
98609 I 10bb79202 10bb791e8 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 1 7ffee4086c18
98610 M 0 7ffee4086b0
98611 I 10bb79209 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98612 M 7ffee4086bc0 7ffee4086b0
98613 I 10bb7920d 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98614 I 10bb7920f 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98615 I 10bb79216 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 7ffee4086c18 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98616 M 30 10bb7cae8
98617 I 10bb7921d 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 10bb7cae8 1 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98618 M 38 0
98619 I 10bb7921f 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 10bb7cae8 0 7ffee4086bf0 7ffee4086b0 0 0 0 0 0 0 0 1 7ffee4086c18
98620 M 20 7ffee4086a8
98621 M 7ffee4086a8 10bb79224
98622 I 10bb7c56a 7ffee4086b0 0 7ffee4086e78 7ffee4086c28 10bb7cae8 0 7ffee4086bf0 7ffee4086a8 0 0 0 0 0 0 0 1 7ffee4086c18
98623 M 10bb7d1f0 7fff6929ce8b
```

NORMAL | myfile2.txt [1:myfile2.txt ] unix | utf-8 | text 7% 98623:1  
busytown.local ~ 18. 1:zshZ 2:zsh\* 3:zsh- 4:zsh 9271/16384MB 33.1% up 1 Sat 1:45:18 AM 09-07-2019

- over ~1.4M lines long (filtered to main binary!)

# let's do something useful

- convert it into a database for fast lookups! using sqlite3 here, so it's easy/portable

```
sqlite> .tables
memory    registers
sqlite> .schema registers
CREATE TABLE registers (
    idx int,
    rip int,
    rax int,
    rbx int,
    rcx int,
    rdx int,
    rsi int,
    rdi int,
    rbp int,
    rsp int,
    r8 int,
    r9 int,
    r10 int,
    r11 int,
    r12 int,
    r13 int,
    r14 int,
    r15 int
);
```

- each register state

# let's do something useful

- but memory writes aren't as easy, there can be multiple memory writes in a single instruction
- or writes between unfiltered instructions

```
sqlite> .schema memory
CREATE TABLE memory(
    lastidx int,
    address int,
    value int
);
```

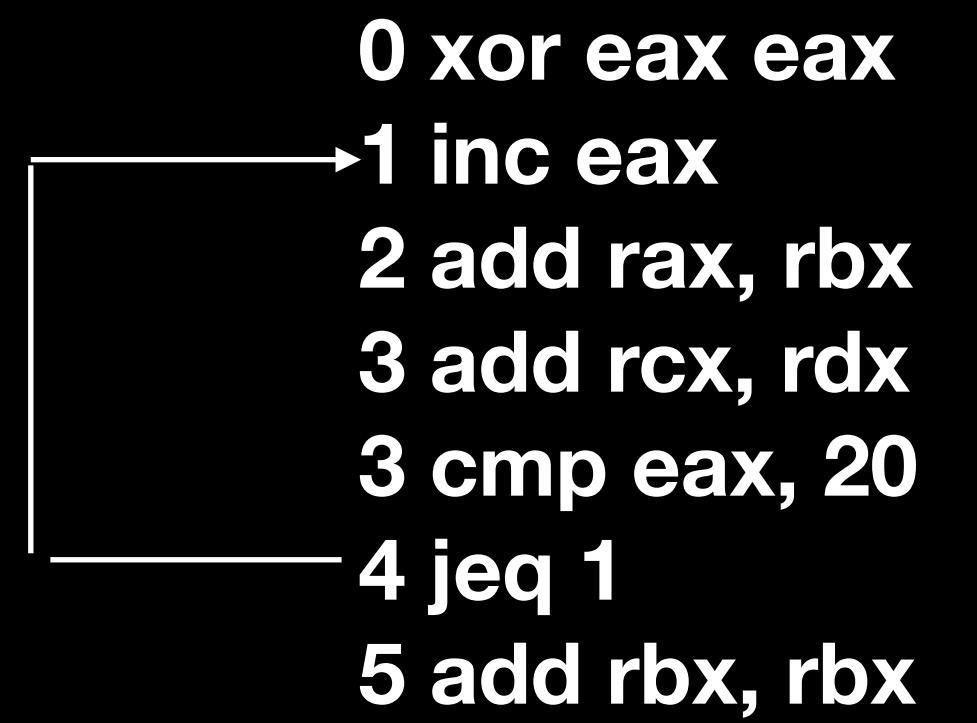
# convert to this data?

- the trace has memory writes interleaved with register traces, so preprocess the entire log into a database
- $idx$  is the instruction index,  $lastidx$  is the previous instruction that a memory state is related to. so when we step, execute all memory diffs where  $lastidx < targetidx$



# dst [count]

- naive approach
- registers are always easy, just load the previous one from the database
- for memory, we don't store the previous memory set, so... how
  - set "breakpoint" -> execute from 0 until breakpoint is hit
  - this doesn't work imagine a loop



0 xor eax eax  
1 inc eax  
2 add rax, rbx  
3 add rcx, rdx  
3 cmp eax, 20  
4 jeq 1  
5 add rbx, rbx

# dst [count]

```
stm = db.prepare "SELECT * FROM memory where lastidx = #{idx}"
rs = stm.execute
ncnt = db.prepare("SELECT COUNT(*) from memory where lastidx = #{idx}").execute.next[0]
unless ncnt > too_many_mems
  while (row = rs.next) do
    addr = row[1]
    s2 = db.prepare "SELECT * FROM memory where lastidx < #{idx} and address = #{addr}"
    rs2 = s2.execute
    while (r2 = rs2.next) do
      addr2 = row[1]
      val2 = bin_to_hex([row[2]].pack("Q"))
      call2 = "http://localhost:9090/cmd/wx #{val} @ #{addr}"
      cu2 = URI::encode(call2)
      HTTParty.get(cu2)
    end
  end
end
```

# dst [count]

- a little more clever i think
- on each step backwards, just write the previous value of all memory writes to the address that was written in this "rewinded" instruction.
- there can be "stale" writes laying around, but this is pretty good i think. that can also be worked around (not yet done)

demo time

# so what do we have/not have

- have
  - trace replayer
  - step forward and backwards
  - r2 integration, inspect memory/print structs just how you normally would
  - support for all intel x86\_64 platforms supported by PIN
  - macOS, Linux, Windows

# so what do we have/not have

- DON'T have
  - "true" debugger
  - must have trace recorded to be useful (eg. have to have caught a rare crash under the tracer to load into r2).
  - ability to see stack frames (no backtrace for example)
  - thread race/interaction usability
    - possible in the future, if multiple traces are done, one for each thread (pin supports this, i don't)

# //TODO

- cutter plugin
- coalesce memory writes for efficiency
- rm ruby, rewrite in c (it's slow as is)
- add tunables for what parts to trace/not trace
- dwarfdump integration for source level debugging
- release (soon, need to cleanup/rm trash parts, sorry intended to do this earlier)

# QUESTIONS?

